Software Reuse: Principles, Patterns, Prospects*

Mária Smolárová and Pavol Návrat
Slovak University of Technology, Bratislava, Slovakia

A closer look is presented at reusability in software development. In particular, object-oriented design is discussed. However, before turning to reviewing the recent results in object-oriented software reuse, such as design patterns, a survey of general principles and methods in software reuse takes place. Prospects of software reuse and its impact on software development in becoming an engineering discipline are stressed. Prospects connected and implied by using the Java language for a possible world wide reuse are shortly discussed.

Keywords: software reuse, design pattern, software engineering, software architecture, object-oriented methodology, Java, world wide reuse

1. Introduction

Promises of software reuse to significantly increase the productivity of software developers and to enhance software quality have not been fulfilled yet. There is still a lot of work to be done in order to overcome technical and non-technical obstacles hindering software reuse.

In this paper, a closer look is presented at reusability in software development. In particular, object-oriented development is discussed. There are different opinions about how object-orientation influences software reuse. On one side, there are beliefs that object-orientation improves reuse significantly [Mey88b, Pre91, LHKS91, Coa92, Huf92, LdC93, LW93, Kar95, GR95]. On the other side, many examples of non object-oriented reusable systems show that objects are neither necessary nor sufficient for software reuse [Tra95, Gri95]. Unfortunately, only a few empirical studies about the impact of object-orientation on software reuse have been conducted [LHKS91, BZ95]. The assumptions are based more on intuition than on scientific experimental evidence.

Our aim is to review the current trends in the area of software reuse[SN96]. We shall be concerned with principles, as well as with methods for software reuse. After reviewing them in general, we shall concentrate on questions related to object-oriented software development.

2. Software reuse principles

In software development, similar problems are solved again and again. The basic principle of reuse i.e., not to repeat solving of what has already been solved, seems to be straightforwardly applicable in software development. Many libraries of reusable components, methods for their retrieval and adaptation, and methods for building new software systems based on previously developed components were proposed. There has been a great expectation that practice of software engineering can be improved through systematic development of reusable software components. Estimated potential for software reuse is enormous, the rates ranging from 15% to 85%, depending on the application area.

Despite the wide-spread interest and reasonable progress in reuse, it is hardly, if ever, practised as a standard method for software development. Reusing software does not automatically mean

* The work reported here was partially supported by Slovak Science Grant Agency, project *Methods and Tools for Development of Software Systems*.

the task has become easier: reuse is an extra work. To build reusable units requires identifying, extracting, organising and representing reusable information in a way that it is easy to understand and manipulate. To find the software which is close to the requirements, to adjust it and to integrate previously built assets into a new system can become more difficult than to develop new software from scratch. A software, in order to be reusable and reused, must be designed, documented and implemented for reuse [Tra88]. Moreover, it must be written and stored in a way that allows easy comprehension, indexing, and retrieving.

2.1. Software reuse scope

At the NATO Software Engineering Conference in 1968, the concept of systematic, formal reuse was proposed. McIlroy who first introduced the term software reuse wanted to change the craft of software developers to an industry of software engineering and so overcome a software crisis. He proposed the factory of off-the-shelf source code components and envisioned the construction of complex target systems from small building blocks available through catalogues of reusable components. But his vision has not been fulfilled yet.

Subsequently, the concept of software reuse became broader. Later, the term "reusable asset" was introduced. A reusable asset is any item of interest that is stored in a reuse library, such as design documentation, specifications, source code, test suites, etc., or any other unit of potential value to a reuser. Reusable asset encompasses all the resources used and produced during the development of software, including knowledge. In the broadest sense software reuse involves organising and encapsulating experience and setting all mechanisms and organisational structures to support it. However, it should be noted that for a standard engineering discipline, reuse is a typical practice. Undoubtedly, software engineering has been developing - consciously or not - along similar lines.

2.2. Reuse potential

Software reuse is believed to have the potential to improve the practice of software engineering significantly. Benefits expected from reuse are mostly of an economic nature and fall into two categories:

- Software development productivity increase. The productivity will increase because [CJ92] the development time will be shortened when software does not have to be built from scratch, and development cost will be reduced because less software has to be developed.
- Software quality enhancement. The software quality will be enhanced because the reused components have already been tested and presumably contain fewer errors than the new components. Also, maintenance costs will be reduced.

The economic impact of reuse on the software development process is estimated intuitively rather than on the basis of cost evaluation. Economic models that quantify productivity increase as well as quality enhancement are needed.

2.3. Required characteristics

According to [MMM95], reusable software must be:

- useful i.e., it addresses a common need;
- (re)usable i.e., it is of sufficiently good quality, easy enough to understand and use in new software development.

In [Kar95], a reusability model was proposed. In this model, reusability was divided into:

- Adaptability: It is sometimes referred to as flexibility; it is the ease with which a component can be adapted to fulfil a requirement which differs from that for which it was originally constructed. Adaptability can be divided to modularity, i.e. how well it is possible to split the solution into disjoint subfunctions, and to generality, i.e. a component's degree of independence from the rest of software.

- Understandability: Attributes of software bearing on the user's effort to recognising the logical concept and its applicability.
- Portability: The ease with which a software can be transferred from one computer system or environment to another. This indicates environmental independence of the software, i.e. the degree of its dependence on environment-specific features.
- Confidence: The probability that a module, program or system also performs its defined purpose without failure in a different environment from that for which it was originally constructed and/or tested.

Also, the fact that reusable software must be of good quality has been pointed out and a quality model was developed as well. Software quality involves reliability and maintainability.

2.4. Restricting factors

There are non-technical as well as technical reasons for only a limited success of software reuse in practice. No agreement exists among the authors which of the factors affects software reusability more significantly. Some authors believe in the significance of non-technical reasons [Tra95, PD91], the others accept the existence of non-technical problems but they believe these can be removed in the future and the only problems playing a significant role in the failure of software reuse are the technical ones [Mey88a].

Non-technical reasons include [Mey88a, PD93]:

- Economical issues. Because an extra effort is needed to achieve software reuse, economic models that would help measure reuse investment cost and return on that investment must be developed and validated.
- Organisational issues. The ways reusable assets are to be distributed, searched for and sold must be found. Even the best reusable component is useless if nobody knows about it, if it takes a long time to obtain or if it is too expensive to buy it.
- Educational issues. Not only does the way to build reusable software have to be taught, but training in building software from an existing one is also necessary.
- Psychological issues. The reuser must trust reusable software assets. The famous "Not Invented Here" syndrome is the most common psychological barrier in reuse acceptance.
- Managerial issues. Management structures and practices are mostly inadequate. Management support at least has to identify different roles in the reuse project and to fill them with suitable people.

Software reuse is significantly restricted by technical factors. Among them the most important ones are:

- the immaturity of software development as an engineering discipline:
  - no standards for representing reusable software components exist,
  - standard architectures are not sufficiently available,
- the lack of reusable component libraries with acceptable retrieval support,
- the lack of methodologies incorporating reuse as a standard development practice,
- the lack of appropriate tools supporting software reuse.

2.5. Diversity

Software reuse suffers from the lack of standard terminology. No standard definition of it exists. Different authors have given their own intuitive definition of software reuse, if any definition is given at all. This diversity has led to different streams in software reuse. Different approaches to software reuse are categorised along five more or less orthogonal axes [Dus92]:

- Transformational (sometimes also called generative) versus compositional reuse. In transformational systems, the result is obtained via transformations from a given high level specification of the desired system. In the compositional approach, choosing components and their combining is done by a software engineer.

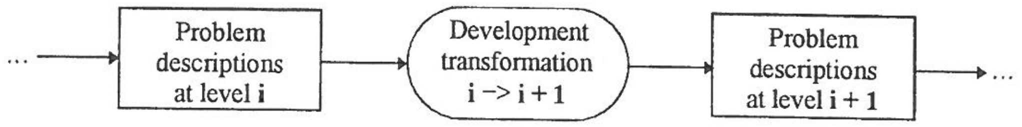
Problem descriptions at level i
Development transformation i -> i + 1
Problem descriptions at level i + 1

Fig. 1. One step of software development process

- Black box versus white box reuse. In the black box approach, reusable products are used as-is, whereas in the white box approach products are modified and adapted to the specific application.
- Abstraction level. Reuse can be applied at the level of code, design, tests, requirements, etc., although it is generally concerned with code reuse.
- Product versus process reuse. Is just the product reused or is the knowledge why and how to do things in a certain way also reused?
- Development of reusable things versus the application of reuse.

In [PD93], another dimension of software reuse is pointed out:

- Vertical versus horizontal reuse. Vertical reuse takes place within the same application area or domain while horizontal reuse means that reusable parts produced in one application area are reused in a different one.

Reuse always takes place somewhere in such a multidimensional space. Typically, it is not placed at an extremal point on an axis. Rather, reuse is a mixture of various approaches. So, an unambiguous classification of different software reuse approaches is difficult, if possible at all.

3. Software reuse methods

Software development has been traditionally divided into stages such as analysis, design, implementation, testing and maintenance, so different life cycle models have been proposed in order to help manage the difficulty of software development processes.

In this paper, we have adopted a process-centered view on software development as it helps us survey software reuse methods. Software development is a process consisting of a sequence of partially ordered steps, starting from more or less formal specifications and leading eventually to an executable code [Che94, DvK95]. Each step in this process is a transformation of a problem description at level i into the problem description at level i + 1 (Fig. 1).

With such a view on software development, reuse methods can be divided into two groups [RGP88, BR89]:

- **generative** methods which concentrate on reusing the transformations of problem descriptions, and
- **compositional** methods which concentrate on reusing the problem descriptions as building blocks for current software development.

3.1. Generative reuse methods

Generative reuse methods automate one part of the software development process, i.e. some of the transformations designed and performed during the software development. The idea of generative approaches is very similar to that of automatic programming - a high-level formal specification should be compiled into an implementation. However, while automatic programming tries to automate the software development from specification to implementation, generative reuse either gives up the goal to automate the whole sequence of transformations during the software development process or the application domain is narrowed.

There are three reuse approaches of generative nature [BR89, Kru92]:

- language-based systems;
- application generators;
- transformational systems.

LANGUAGE-BASED SYSTEMS. Language-based systems are also known as executable specification languages or very-high level languages (VHLLs). They typically use mathematical model such as a set theory or constraint equations that embody reusable patterns in a high level language. Automation of the mappings from VHLL constructs to high level language implementation patterns is possible usually only at the expense of code efficiency and design quality. The problem with VHLLs is in the fact that they have the advantage of being generally applicable to software development but the semantic gap between mathematical abstraction used in model and software development is still too large.

SETL [DFCS89] is a wide spectrum VHLL based on the set theory. Data types in the language are either atomic types (integer, real or Boolean values) or composite data types (sets or tuples). The set operations union, intersection, difference and power set as well as the set iterators are defined in the language. Maps in SETL correspond to functions in the set theory. A map binds the elements in the domain to those in the range.

MODEL [PL89] is a declarative constraint VHLL based on simultaneously solving a collection of constraint equations. The language contains constructs for data declarations and consistency equations. A compiler performs an analysis to determine if the consistency equations can be satisfied and to find the order of computations that will assign consistent values to the data objects. The MODEL compiler checks for the completeness and consistency of the specifications.

APPLICATION GENERATORS. Application generators automate the whole development process in a very narrow application domain. An application generator is a tool that takes input specifications and produces executable programs as output. Specifications are typically high-level abstractions in a specific application domain.

To build an application generator is difficult, as it requires [Cle88] identifying appropriate domains, to define domain boundaries and underlying computational model for the application domain, to define variant and invariant parts of an application family, to define the specification input method and to define products generated by the application generator.

To be able to build an application generator requires not only an intimate knowledge of the application domain [Cle88], but also, the domain must be narrow, well-understood, with slowly changing technology [Big92].

TRANSFORMATIONAL SYSTEMS. Transformational systems do not fully automate the software development process. Software developer's assistance is needed in selecting among applicable transformations. With transformational systems, software is developed in two phases. First, the semantic behaviour of a software system is described in a specification language and second, the user-guided transformations are applied to the specification.

Paddle [Bal89] is a transformational system that stores a development history as a sequence of applied transformations. Paddle provides the program, i.e. the structure of the sequence of transformations needed to implement some application. Design decisions that lead to writing the program are not saved, so the transformations are difficult to understand and modify.

Glitter [Fic85] is a transformational system that encodes into transformational rules the decisions that a software developer has made during their applications. To select from a collection of reusable transformations, expert system technology has been used. The rules embody expert knowledge about how to accomplish goals during program transformations. Each rule consists of a goal, strategies and a selection rationale part. Glitter automates as much of the transformation process as possible. If the strategies are incomplete or selection rationale fails, the system asks the software developer for guidance.

3.2. Compositional reuse methods

Compositional reuse is the most common form of software reuse. It is based on reusing components from earlier software development as the building blocks for new software development. Components stored in reusable libraries may contain any units that are of potential value for a reuser, such as source code, subsystems, test data, user documentation, design, development plan, architecture, installation instruction, etc.

When considering a transformational model of software development (Fig. 1), reused components are equivalent to problem descriptions at one of the levels during the software development process. The main idea behind this approach - reuse of any previously developed software components - is straightforward, but there is a lot of potential difficulties with its application. To discuss crucial parts of compositional reuse more precisely, the following aspects of this approach to software reuse are taken into account:

- **identification** of reusable components,
- component **description**,
- **retrieval** of reusable components,
- **adaptation** of components to specific needs,
- component **integration** into current software developed.

IDENTIFICATION. Identification of reusable parts is the first step towards systematic and effective software reuse [PD90]. To foresee the future reuse opportunities is difficult. It requires development experience. Also, thorough domain knowledge gained on the basis of domain analysis is necessary. Many aspects must be considered when reusable units are identified. Among them are:

- **Granularity.** The granularity (also called size) of a component is important. The larger the reusable component, the greater reuse productivity improvement. But a larger reusable unit involves many functions that make such a component more difficult to reuse because its complex functionality may not exactly fit into the required one.
- **Categories.** A reusable component may be any concrete intermediate product of the software development process, i.e. not only the source code but also design, specifications, requirements, tests, documentation. The earlier in the software development process, the higher the reusability leverage.
- **Generality.** Generally, application-independent components that can be applied to a broad range of application domains are presumed to be reused more frequently, but they tend to be more difficult to reuse because of the generality they offer [BR89].

It is still far from being clear what a reusable component should be and how to identify it. Rather than to be able to recognise reusable components from the first principle, their incremental identification and building seems to be the way the software reuse must be conducted.

DESCRIPTION. Component description facilitates understanding of component functionality. A good component description should express what a component does without knowing how it does that [WOZ91]. Besides being abstract, a component description should be clear, unambiguous and understandable. Also, it should permit a wide variety of reusable components to be described [RW89].

In the software reuse community, two approaches to component description can be seen:

- **Component models.** A component model is an abstract description of the components in a given domain [Whi95] that depicts all attributes a reusable component should have. Efforts are made to find a generally acceptable model.

  Up to now, the *3C reference model* [Edw92, Whi95] received most acceptance. The 3C model distinguishes different aspects of a reusable component: its concept i.e. an abstracted description of what functionality the software component provides; its content, i.e. an implementation that says how the component achieves the functionality described in its concept; and its context that describes the component relationship to other components on which it depends.

  The *REBOOT* (Reuse Based on Object-Oriented Techniques) component model [FM92, Kar95] is based on a facet classification. Four facets particularly relevant

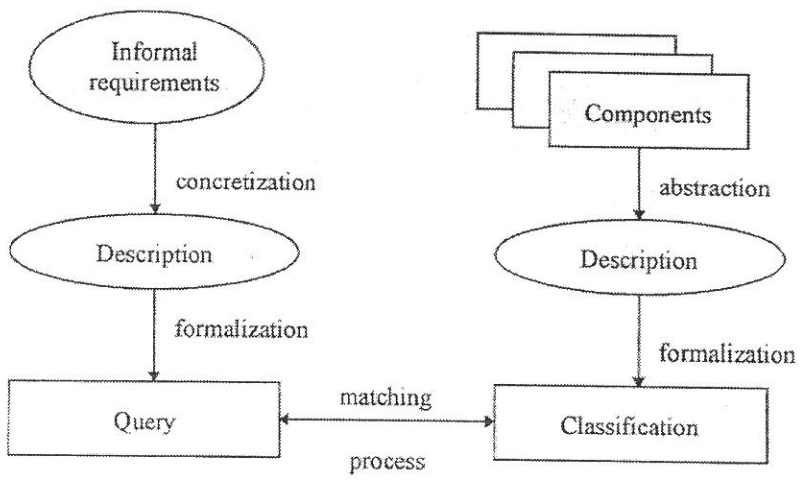

Fig. 2. Component classification and retrieval

to reuse have been proposed: Abstraction, Operations, Operates On and Dependencies. Each facet can have an arbitrary number of terms in its restricted vocabulary. A component can be considered as a combination of the terms within its facets.

- **Component description languages.** Either at the implementation level or at the design level, these languages try to capture the essential attributes of components. A survey of languages aimed at describing reusable components in the design stages of development is given in [Whi95].

  The goal of a component description language is to provide semantics of the component functionality. In fact, there are two main approaches to formal specifications: an algebraic and a model-based approach [Som92]. An algebraic formal specification describes syntactic interface by names and signatures for all of the component's operations and states the relations among the operations. A model-based formal specification defines postconditions and preconditions for each operation.

RETRIEVAL. Reusable libraries may grow to a large collection of components. In order not to become a write-only medium, they must be well organised. In particular, methods for component retrieval that would encode abstract component description and match it against reuser's requirements must be added to the library (2). Major approaches used for component retrieval include [FP94]:

- **Library and information science.** There are two groups of methods used in library and information science: one that uses a controlled vocabulary i.e. enumerated, facet and attribute-value classification, and the other one that does not restrict vocabulary, also called free-text retrieval.

  Enumerated classification uses short, usually one word metaphorical descriptions to break a subject area into mutually exclusive classes that form a hierarchical structure. This is possible only in well understood, narrow application domains with one word abstractions that embody a big amount of domain knowledge so that they are universally understood. Classification provides a natural way for searching when domain is well analysed and exclusive hierarchical categories exist. A disadvantage of enumerated classification is the difficulty connected with changing it.

  Multifaceted classification organises terms of a subject area into facets. The development of facets is accomplished by identifying important vocabulary in a domain and grouping terms together into facets.

  Attribute-value classification uses a set of attributes and their values to describe a component in the library. It is similar to facet classification because it uses attributes and values as facet classification uses facets and values. The differences are that attribute values are not restricted to having predefined values as facets and there is no limitation as to the number of attributes being used for component description.

Controlled vocabulary approaches have a disadvantage as the vocabularies of reuser and developer can diverge. Usually, a thesaurus of synonyms is provided in order to eliminate this diversity.

The *free-text retrieval* is based on natural language. The textual representation of the component is used as a component description. The advantage of the free-text retrieval is that no encoding is required and queries in natural language are easy to formulate because of the closeness to the reuser. But the natural-language ambiguity, incompleteness and inconsistency make the component retrieval less precise. Natural language descriptions for reusable library have been used in e.g., [WS88].

- **Knowledge-based retrieval.** Knowledge-based retrieval [KV95, DBSB91, Pol87] uses various kinds of reasoning that relate the new query to an old one or match the query with the components. These methods need a knowledge base for the application domain and for the decisions taken. They require more human resources, especially for development of the knowledge base, but they have a potential to be more powerful because of capturing query and component semantics.

- **Hypertext based retrieval.** Hypertext retrieval [GS89] organises information nonlinearly into a network of nodes and links. A reuser can access the stored information by navigating along the links. To be able to design a hypertext network, one should first carefully study the application domain and find an optimal set of relations. It can also be difficult to add a new component, because it requires studying exhaustively the links between the new component and the old ones. Hypertexts are easy to use and may lead to the right component quickly. But the user may also get lost during the navigation or some information may even be inaccessible because the path he has chosen to follow has no link to the required information.

- **Specification-based retrieval.** Specification-based approaches use formal component descriptions as the basis for partial ordering of the components in the library. These methods differ mostly in the expressiveness of the specification language and also in how fully they take advantage of the specification language. Some make use of component signatures only, i.e. syntactic interfaces; the others also handle the semantics of specifications. Examples of retrieval methods based on formal specifications are [CJ92, JC93, ZW93].

No agreement exists about which retrieval method is the most appropriate. [FP94] compared four different representation methods for the component libraries (attribute-value, enumerated, faceted, and keyword). They conclude that a single retrieval method is not sufficient for finding all the relevant components for a given search query. Different individual users prefer and are more successful with different methods. There were no significant differences between the compared methods in search effectiveness measured in terms of recall (the number of relevant components retrieved over the number of relevant components in the library) and precision (the number of relevant components retrieved over the total number of components retrieved). However, the differences in effectiveness measured by search time were significant; the search with enumerated classification took 60% less time than with the keyword method. None of the methods supports the component understanding more than moderately. To devise retrieval methods best suited for reusable components remains a challenge.

Query formulation is another important issue in the retrieval method. Here, the unwillingness of a reuser to formulate long and precise queries should be taken into account. A reuser would prefer an interactive query formulation where a system assists to define the query. In [Hen94], the fact that a reuser is often not able to completely formulate his query at the beginning of the retrieval process is stressed out and the incremental query construction is proposed. Also, a relaxed or approximate retrieval should be considered. Whenever an exact match fails, components that are closest to the requirements should be retrieved.

ADAPTATION. One of the important basic characteristics of a reusable component is its generality [Kar95], i.e. its applicability is not limited to one specific use. Generality supposes that the component is open [Mey88a], i.e. it is still available for extension. Each general component has a variable part that must be replaced according to specific needs [Kru92]. This act is called component adaptation or specialisation.

Component adaptation may be done using different techniques [Hal88]:

- **Parameter substitution.** Parametrization allows to build generic components, i.e. components with parameters that are adapted to specific needs by setting values to its parameters.
- **Inheritance.** Inheritance is a mechanism that allows adding new properties to existing classes (substantial feature of the object-oriented paradigm). It is an example of incremental programming when subclasses are defined by stating how they differ from the existing ones.
- **Modifications.** Modifications of component code is a white-box reuse that requires understanding all the implementation details. Also, this technique can invalidate correctness of the original component [Kru92].

INTEGRATION. In order to develop a component for reuse and to subsequently reuse the component, not only the component, but also its relationship to other components must be well understood [Whi95]. Effectiveness of reuse highly depends on the way those components are combined [Sha95]. There are not only a number of diverse composable software entities such as functions, classes, templates, modules, processes, but, consequently, also a number of component styles for combining the software systems from these components. Different styles expect different ways of component packaging and different kinds of interactions between them [NM95].

The most common way to describe component integration is to use module interconnection languages [PDN86]. Module interconnection languages describe modules in terms of exported operations that a module implements and imported operations that a model uses. Modules are assembled into a system by interconnecting them through appropriate exports and imports [Kru92].

In [Bea92], declarative languages are suggested to be used for describing the component interconnection. By standardising certain mechanisms for component interaction, declarative languages can be used to describe the ways components are connected. The main advantage is that the interactions of components need not be described in details and only relationships among components are stated. Also, declarative languages may be invertible so the same relationships may be interpreted in different ways.

So far, most of the emphasis has been put on components and only very little of it on how they are composed [NM95]. More investigation is needed on how to compose software from computational elements. A lot of recent work has been done in areas such as module interface languages, domain-specific frameworks, and software architectures [GP95]. In particular, software architecture as a high-level description of how specific systems are composed from its components has been recognised as a critical aspect of the design for any larger software system and it has gained significant attention recently [SG96, BMSS96].

4. Reuse within object-oriented methodology

Recently, many researchers and practitioners in the software reuse area devoted their attention to object-oriented software development. It is believed that object-oriented (OO) languages facilitate development of software that can be reused. Here, a strong impetus could recently be observed in development and rapid spreading of Java programming language.

There are, in fact, different reuse levels offered in OO software development. Objects and classes are basic reusable units. But they alone would not be enough for an effective reuse. Not the components themselves, but the way those components are combined, affects software reuse significantly. In fact, OO methodologies gave birth to application frameworks and, more recently, design patterns for building OO applications. Application frameworks provide generic designs for building applications [Pre91]. Design patterns communicate solutions to recurring design problems in OO development. Both frameworks and patterns elevate reuse to a higher level because of supporting especially the design reuse.

While it is probably fair to say that whether the claim that object-orientation fosters reuse is justified remains, at least for some time, to be seen, it can be stated already now that has been a rather substantial progress achieved in seeking concepts and forms that allow expressing standard software design within the object-oriented methodology.

4.1. Reusability of objects

The fundamental unit in object-oriented development is the object. Basic concepts involved in the notion of object are data abstraction, information hiding and encapsulation. Objects are defined in terms of abstract data types where each type defines also a set of methods. An object has its internal state (data) and its operations (methods). The only way other objects interact with an object is through sending messages (requests) to it.

The following characteristics of an object foster its reusability [GR95]:

- Objects separate interface from the implementation.
- Objects often closely map the real world.
- Objects come in different sizes and abstraction levels.
- Objects live all the way down during software development.

4.2. Reuse of classes

Class is a common template for similar objects. Class defines interface as well as behaviour of a set of objects. Objects instantiated from a class perform common methods and share common structure while each object within a class retains its own states.

The OO programming languages usually offer a library of predefined classes. Classes in the library are related through inheritance relationship. Inheritance, a built-in mechanism, has two possible uses:

- **Instantiation** creates a specific object from a class template by setting values to variables defined in the class. That is why objects are sometimes called instances of the class. When an object is created, its internal data are allocated according to the class structure and the class operations are associated with these data. Many objects may be created by instantiating the same class.
- **Subclassing** allows the derivation of new classes by modification of existing classes i.e. by stating how they differ from them. Subclassing involves two types of relations between class and subclass:
  - **Specialisation.** Specialisation adds new functionality to subclasses. It retains interface semantics i.e., a subclass interface is a subtype of its superclass interface.
  - **Overriding.** Overriding may refine, redefine, or even hide the parent's functionality. Redefinition and hiding breaks interface semantics.

Benefits of class reuse include less code to develop, less code to maintain, and not having to redesign the same items repeatedly. Inheritance makes it easier to include existing code by extending the original classes and adding new ones. But inheritance allows also redefinition and hiding that may have harmful effects on reuse. Such diverse uses of the same inheritance mechanism cause difficulties in class reuse.

One useful mechanism that inheritance permits is to define **abstract classes.** An abstract class has no implementation for its methods, so it cannot be instantiated. An abstract class defines a common interface for its subclasses whilst the concrete definitions are left to the subclass itself. This helps define families of interchangeable classes with common interface.

Abstract classes make object-oriented class libraries more reusable and easier to understand.

The mechanism that abstract classes provide is one of the most important ones: classes should be derived from abstract superclasses as often as possible and all subclasses of an abstract class should only add or refine but not override operations of the parent class [GHJV95]. Then, all subclasses can respond to the identical interface i.e., they all are subtypes of the same abstract class.

The different views of an implementor and a reuser that may cause diversity in class hierarchy should be pointed out. An implementor is mostly concerned with quick derivation of a new class implementation from the existing ones. A reuser is interested more in the interface inheritance that says when a class can be used in place of another one [Edw92].

Another source of potential difficulties is that class libraries may become vast. Reusing classes from such a hierarchy may be a time consuming laborious task. It is necessary to look at class names, at names of methods, or in the worst case at implementation of methods in order to investigate reuse opportunities. Inheritance is a white-box reuse that requires all the internals of a parent class to be visible to its subclass [GHJV95]. OO environments usually facilitate reuse by visualisation of the structure of class hierarchies and by supporting easy class browsing.

4.3. World wide reuse

The Internet, a rapidly growing collection of networks, and the World Wide Web, a hypertext information and communication system, gave birth not only to new ways of people's communication but they also changed the software industry significantly.

Internet and Web ultimately require new approaches to software development. The ability to run on heterogeneous and distributed platforms is the necessity for all future software systems. With having such a platform-independent software, the reuse increases by a significant rate. No more environment specific code rewriting is needed. The same piece of software is reused across a wide variety of computer systems, hardware platforms, and operating systems.

Java, a new programming language gaining a considerable popularity recently, has been designed to make full use of advances in distributed networking. Java is a language that brings a new dimension to software reuse, too. Undoubtedly, Java's key feature that aids to the code reuse is its genuine object-oriented nature. But Java introduces additional capabilities that support reuse [NS96]:

- Java is architecture-neutral. Java bytecode is able to run on any platform that has the Java runtime environment. With Java, the capability of reuse is enhanced by having a single application that is immediately usable on multiple platforms.
- Java is portable. Java language specification defines standard behaviour applied to the data types across heterogeneous platforms. Reuse is enhanced because implementation is no more hardware-dependent.
- Java is dynamic and robust. Dynamic linking of classes at runtime and checking data structures at both compile time and runtime cause that Java code can be reused without recompilation even if the environment has changed.

There is another Java feature that should be focused on - Java code can be an executable part of the Web document. Java applets i.e., small applications that are downloaded directly from Web pages, bring richness, interactivity and enhanced information delivery to Web pages. Through the Internet, the reuse is fostered world wide.

4.4. Application frameworks

A framework is a reusable design for solutions to problems in some particular problem domain. An object-oriented application framework is a set of classes that, taken together, represent an abstraction or parameterized skeleton of an architecture [Big92]. Frameworks group classes, objects and relations together in order to build a specific application [Coa92]. A framework is a generic software architecture together with a set of generic software components that may be used to realise specific software architecture [NM95].

An OO framework consists of a set of abstract and concrete classes that extensively communicate through messages. An ideal framework would provide all concrete classes from which new application would be composed in its class library. In real software development based on frameworks, some of the application-specific classes must be constructed. They are typically derived by subclassing from abstract classes provided in the framework. Here, abstract classes depict variable parts of the framework that are configured according to the specific needs of an application [Kar95].

There are many different OO frameworks available. Among them, one of the best known is the Model-View-Controller (MVC) framework built in the Smalltalk-80 system. MVC is a user interface framework that provides a uniform architecture for interactive applications [Deu89]. An abstract class view converts some interesting aspects of the model to visible form. A controller knows how to interpret commands inserted by a user. A model itself accesses and updates the application.

Designing a framework requires thorough domain analysis as well as experience from distinct projects [EG92]. The most critical issue in designing a framework is to achieve its flexibility i.e. easy adaptation to all applications in the domain, and its extensibility i.e., the ability to cover all applications in the domain [GHJV95]. Also, low sensitivity to domain evolution is necessary.

New applications can be built faster when frameworks are reused. Another important fact is that all applications in the domain then have similar structures and are easier to understand and maintain [GHJV95].

However, frameworks may introduce additional complexity and constraints to OO software development. Each framework achieves its functionality by a co-operation of single components. Understanding and mastering of a single component may become more difficult because it depends on its relationships with other components as well [SSP95]. Besides this, frameworks provide "frame" solutions to the problem that may not necessarily fit to the current problem.

4.5. Design patterns

Design patterns proposed by [GHJV95] (collected in a catalogue referred to as G4 hence on) introduce a new mechanism for expressing design knowledge that experienced developers use over distinct applications. Each design pattern systematically identifies, names and explains a recurring design problem and presents good and elegant solution to it. Design pattern is a micro architecture, a small grouping of classes describing responsibilities of elementary parts as well as the relationships and collaborations among them.

Patterns are built by observation and by gathering experience during the development of many OO applications. A pattern template involves:

- *Pattern:* name, synonyms, one sentence description, title, nicknames.
- *Intent:* goal, motivation.
- *Applicability:* context, examples.
- *Consequences:* the end-situation, results.
- *Constraints:* interdependences, forces, affecting factors.
- *Resolution:* structures, actions.
- *Implementation:* code fragments, practical concerns to be aware of.
- *Applications:* known uses.
- *Risks:* indications of potential difficulties in use.
- *Related patterns:* references to other patterns.

The uniform template used for pattern description makes them easier to learn, compare and use.

Each pattern explains a solution to a problem comprehensively so that no essential information is lost to the readers. The pattern description uses a template which is a structure of attributes. The template contains four essential elements: a pattern name, a problem, a solution, and consequences. Class diagrams, object diagrams and interaction diagrams are used for problem illustrations. Beside graphical notations, natural language describes decisions, alternatives, examples, and trade-offs that led to the proposed design pattern.

Patterns divide solutions into their elementary parts which can be later recombined and reused [Vil95]. Patterns thoroughly describe a single atomic solution. Descriptions of the combinations may become simplified because they can refer to the already described and stored patterns.

Application frameworks and design patterns are often confused. The main differences between frameworks and patterns are [GHJV95]:

- Design patterns are more abstract than frameworks. Frameworks can be embodied in code, but only examples of patterns can be embodied in code.
- Design patterns are smaller architectural elements than frameworks. A typical framework contains several design patterns, but the reverse is never true.
- Design patterns are less specialised than frameworks. Frameworks always have a particular application domain. In contrast, design patterns can be used independently from an application.

Patterns occur outside OO methodologies, too. Design patterns vary in their granularity and level of abstraction. Patterns may abstract an analysis, a design or a process [FLM95].

Patterns could be organised into families of related patterns in order for the reuser to be able to learn and find them more easily. When collecting patterns, the way to categorise patterns in a collection should be proposed as well.

The G4 catalogue of reusable design patterns records 23 design patterns that were recognised by expert object-oriented designers. Two dimensions for pattern categorising are proposed: scope and purpose. The scope specifies whether the pattern applies primarily to classes or to objects. The purpose reflects what a pattern does. Creational patterns concern the process of object creation, structural patterns deal with the composition of classes and objects, and behavioural patterns characterise the ways in which classes or objects interact and distribute responsibility.

5. Conclusions and future work

We have presented an overview of the important issues related to software reuse. We give special stress to the key concepts among principles as well as methods of reuse. Those aspects of object-oriented software development that can support reusability as well as those that make reuse difficult have been outlined. Among the factors having positive influence on reuse are:

- objects separate interface from implementation;
- classes are reusable building blocks the OO software systems are constructed from;
- abstract classes can be used to define families of classes with common interfaces;
- inheritance makes code reuse an integrated part of OO programming;
- classes are reusable building blocks the OO software systems are constructed from;
- application frameworks provide design reuse that is embodied in domain specific generic architectures;
- design patterns offer approved solutions for recurring OO design problems.

The factors hindering reuse include:

- inheritance is a white-box reuse;
- inheritance exposes a subclass to know about all internal details of its parents;
- class hierarchy reflects implementation inheritance, not interface inheritance.

Software development based on object composition makes the system's behaviour dependent on interrelationships among objects. Inheritance is a built in mechanism that is not at all sufficient to model different interconnections between objects. In OO design, some stereotypical problems for object composition have been recognised. Good solutions of recurring problems have been recorded into design patterns. When they are applied, design knowledge is reused.

The situation today is such, that first catalogues of OO design patterns already exist [Coa95, GHJV95]. To use them and apply them does not seem to be an easy task at all. In their book [GHJV95], the authors point out that patterns can hardly be understood completely on the first reading. A designer will have to refer to them again and again. Successful application of a pattern requires knowing about the pattern, comprehending it and examining it. After that, design patterns can be applied repeatedly by analogy. The statement also hints what the open problems are in standardising software design, which in turn is the core of software reuse as a major vehicle for turning software development into a standard engineering. As we have discussed in this paper, the identification, description, retrieval, adaptation and integration of software assets are all important issues and achieving progress in any of them will be a step towards effective reuse. For a future work, we wish to single out at least these three subgoals:

- To propose a formalism for representation of design patterns, frameworks and software architectures that would primarily serve to retrieve and apply them in software design.
- To propose a tool that would support the user in retrieving and applying design patterns, frameworks and software architectures in software design.
- To investigate possibilities of incorporating design patterns, frameworks and software architectures into a design/implementation language.

Obviously, these are only a few of the open problems. The whole area of software reuse is much broader: it does not encompass only design, but the whole life cycle process, and it does not relate only to design with reuse, but also design for reuse.

Seen from a more general perspective, the recent endeavours should be considered as steps in the right direction. If the discipline of software development is to turn from art into engineering, then software reuse is definitely one of the major factors influencing the change. With catalogues of typical solutions, standardised methodologies, languages, etc. we witness an emerging engineering discipline; and the prospects of it are rather promising, despite the length of the road that is still ahead. One of the promises originates in the potential of the object-oriented methodology, which offers better expressive and structuring capabilities when compared to other programming paradigms [Náv96]. The OO paradigm itself, in turn, is developing with one of the main motivations being increasing reusability. In the future, new paradigms may emerge e.g., the proposed generative programming [Eis97].

References

[Bal89] R. Balzer. A fifteen-year perspective on automatic programming. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability. Volume II - Applications and Experience*, pages 289–312. Addison-Wesley, 1989.

[Bea92] B. W. Beach. Declarative programming for component interconnection. In *5th Annual Workshop On Software Reuse WISR'92*, 1992.

[Big92] T. J. Biggerstaff. An assessment and analysis of software reuse. In *Advances of Computers*, volume 34, pages 1–57. Academic Press, New York, 1992.

[BMSS96] F. Buschmann, R. Meunier, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture: A System Of Patterns*. John Wiley & Sons, 1996.

[BR89] T. J. Biggerstaff and Ch. Richter. Reusability framework, assessment, and directions. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability. Volume I - Concepts and Models*, pages 1–18. Addison-Wesley, 1989.

[BZ95] J. M. Bieman and J. X. Zhao. Reuse through inheritance: A quantitative study of C++ software. *ACM SIGSOFT, Proc. of the Symposium on Software Reusability SSR'95*, 20(special issue August):47–52, 1995.

[Che94] J. Cheng. A reusability-based software development environment. *ACM SIGSOFT Software Engineering Notes*, 19(2):57–62, 1994.

[CJ92] B. H. C. Cheng and J. J. Jeng. Formal methods applied to reuse. In *5th Annual Workshop On Software Reuse WISR'92*, 1992.

[Cle88] J. C. Cleveland. Building application generators. *IEEE Software*, 5(4):25–38, 1988.

[Coa92] P. Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–158, 1992.

[Coa95] P. Coad. *Object Models. Strategies, Patterns, and Applications*. Yourdon Press, 1995.

[DBSB91] P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. A knowledge-based software information system. *Communications of ACM*, 34(5):34–39, 1991.

[Deu89] L. P. Deutsch. Design reuse and frameworks in the Smalltalk-80 system. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability. Volume II - Applications and Experience*, pages 57–71. Addison-Wesley, 1989.

[DFCS89] E. Dubinsky, S. Freudenberger, E. Chonberg, and T. Schwartz. Reusability of design for large software systems: An experiment with the SETL optimizer. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability. Volume II - Applications and Experience*, pages 275–294. Addison-Wesley, 1989.

[Dus92] E. M. Dusink. Reuse is not done in vacuum. In *5th Annual Workshop On Software Reuse WISR'92*, 1992.

[DvK95] E. M. Dusink and J. van Katwijk. Reuse dimensions. *ACM SIGSOFT, Proc. of the Symposium on Software Reusability SSR'95*, 20(special issue August):137–149, 1995.

[Edw92] S. H. Edwards. Toward a model of reusable software subsystems. In *5th Annual Workshop On Software Reuse WISR'92*, 1992.

[EG92] T. Eggenschwiler and E. Gamma. ET++ Swapsmanager: Using object technology in the financial engineering domain. *SIGPLAN Notices, Proc. OOPSLA'92*, 27(10):166–177, 1992.

[Eis97] U. W. Eisenecker. Generative programming (GP) with C++. In H. P. Mössenböck, editor, *Modular programming languages*, volume 1204 of *LNCS*, pages 351–365. Springer Verlag, 1997.

[Fic85] S. F. Fickas. Automating the transformational development of software. *IEEE Trans. on Software Engineering*, 11(11):1268–1277, 1985.

[FLM95] S. Fraser, D. Leishman, and R. McLellan. Patterns, teams and domain engineering. *ACM SIGSOFT, Proc. of the Symposium on Software Reusability SSR'95*, 20(special issue August):222–224, 1995.

[FM92] J. Faget and J. M. Morel. The REBOOT approach to the concept of a reusable component. In *5th Annual Workshop On Software Reuse WISR'92*, 1992.

[FP94] W. B.. Frakes and T. Pole. An empirical study of representation methods for reusable software components. *IEEE Trans. On Software Engineering*, 20(8):617–630, 1994.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements Of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GP95] D. Garlan and D. E. Perry. Introduction to the Special Issue on Software Architecture. *IEEE Trans. on Software Engineering*, 21(4):269–274, 1995.

[GR95] A. Goldberg and K. S. Rubin. *Succeeding With Objects. Decision Frameworks For Project Management*. Addison-Wesley, 1995.

[Gri95] M. Griss. Systematic software reuse: Objects and frameworks are not enough. *ACM SIGSOFT, Proc. of the Symposium on Software Reusability SSR'95*, 20(special issue August):17–18, 1995.

[GS89] P. K. Garg and W. Scacchi. Ishys, Designing an intelligent software hypertext system. *IEEE Expert*, 4(3):52–63, 1989.

[Hal88] P. A. V. Hall. Software components and re-use. *Software Engineering Journal*, 3(10):171, 1988.

[Hen94] S. Henninger. Using iterative refinement to find reusable software. *IEEE Software*, 11(5):48–59, 1994.

[Huf92] S. Hufnagel. Formally specified object-oriented approach to reuse. In *5th Annual Workshop On Software Reuse WISR'92*, 1992.

[JC93] J. Jeng and B. H. C. Cheng. Using formal methods to construct a software component library. In I. Sommerville and M. Paul, editors, *Proceedings of 4th Software Engineering Conference*, volume 717 of *LNSC 717*, pages 397–417, Garmisch-Partenkirchen, 1993. Springer Verlag.

[Kar95] E. A. Karlsson. *Software Reuse. A Holistic Approach*. John Wiley & Sons, 1995.

[Kru92] Ch. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[KV95] P. Katalagarianos and Y. Vassiliou. On the reuse of software: A CASE-based approach employing a repository. *Automated Software Engineering*, 1(2):55–86, 1995.

[LdC93] D. Lea and D. de Champeux. Object-oriented software reuse technical opportunities. In *6th Annual Workshop On Software Reuse WISR'93*, 1993.

[LHKS91] J. A. Lewis, S. M. Henry, D. G. Kafura, and R. S. Schulman. An empirical study of the object-oriented paradigm and software reuse. *SIGPLAN Notices, Proc. OOPSLA'91*, 26(11):184–196, 1991.

[LW93] H. C. Liao and F. J. Wang. Software reuse based on a large object-oriented library. *ACM SIGSOFT*, 18(1):74–80, 1993.

[Mey88a] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Mey88b] B. Meyer. Reusability: The case for object-oriented design. In W. Tracz, editor, *Software Reuse: Emerging Technology*, pages 201–215. IEEE Computer Society Press, 1988.

[MMM95] H. Milli, F. Milli, and A. Milli. Reusing software: Issues and research directions. *IEEE Trans. On Software Engineering*, 21(6):529–561, 1995.

[Náv96] P. Návrat. A closer look at programming expertise: Critical survey of some methodological issues. *Information And Software Technology*, 38(1):37–46, 1996.

[NM95] O. Nierstrasz and T. D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, 1995.

[NS96] P. Norton and W. Stanek. *Peter Norton's Guide to Java Programming*. SAMS.NET, 1996.

[PD90] R. Prieto-Díaz. Domain analysis: An introduction. *ACM Sigsoft*, 15(2):47–54, 1990.

[PD91] R. Prieto-Díaz. Making software reuse work: An implementation model. *ACM Sigsoft*, 16(3):61–68, 1991.

[PD93] R. Prieto-Díaz. Status report: Software reusability. *IEEE Software*, 10(3):61–66, 1993.

[PDN86] R. Prieto-Díaz and M. Neighbors. Module interconnection languages. *Journal Of Syst. Software*, 6(4):307–334, 1986.

[PL89] N. S. Prywes and E. D. Lock. Use of the model equational language and program generator by management professionals. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability. Volume II - Applications and Experience*, pages 103–129. Addison-Wesley, 1989.

[Pol87] S. Pollitt. Cansearch: An expert systems approach to document retrieval. *Information Processing And Management*, 23(2):119–138, 1987.

[Pre91] W. Pree. Reusability problems of object-oriented software building blocks. In A. Mrázik, editor, *EastEurOOP'91 Proceedings*, pages 15–20, Bratislava, 1991. ArtInAppleS.

[RGP88] C. V. Ramamoorthy, V. Garg, and A. Prakash. Support for reusability in Genesis. *IEEE Trans. On Software Engineering*, 14(8):1145–1154, 1988.

[RW89] Ch. Rich and R. C. Waters. Formalizing reusable software components in the Programmer's Apprentice. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability. Volume II - Applications and Experience*, pages 313–344. Addison-Wesley, 1989.

[SG96] M. Shaw and D. Garlan. *Software Architecture - Perspectives Of An Emerging Discipline*. Prentice Hall, 1996.

[Sha95] M. Shaw. Architectural issues in software reuse. *ACM SIGSOFT, Proc. of the Symposium on Software Reusability SSR'95*, 20(special issue August):3–6, 1995.

[SN96] M. Smolárová and P. Návrat. Recent directions in software development with reuse: Design patterns and beyond. In *Proc. of the Conference on Electronic Computers and Informatics*, Košice, 1996.

[Som92] I. Sommerville. *Software Engineering*. Addison-Wesley, 1992.

[SSP95] A. Schappert, P. Sommerlad, and W. Pree. Automated support for software development with frameworks. *ACM SIGSOFT, Proc. of the Symposium on Software Reusability SSR'95*, 20(special issue August):123–127, 1995.

[Tra88] W. Tracz. Software reuse myths. *ACM SIGSOFT*, 13(1):17–21, 1988.

[Tra95] W. Tracz. Third International Conference on Software Reuse. Summary. *ACM SIGSOFT*, 20(2):21–25, 1995.

[Vil95] P. Viljamaa. The patterns business: Impressions from PLoP'94. *ACM SIGSOFT*, 20(1):74–78, 1995.

[Whi95] B. Whittle. Models and languages for component description and reuse. *ACM SIGSOFT Software Engineering Notes*, 20(2):76–87, 1995.

[WOZ91] B. W. Weide, W. F. Ogden, and S. H. Zweben. Reusable software components. In *Advances of Computers*, volume 33, pages 1–65. Academic Press, New York, 1991.

[WS88] M. Wood and I. Sommerville. An information retrieval system for software components. *Software Engineering Journal*, 3(10):199–207, 1988.

[ZW93] A. M. Zaremski and J. M. Wing. Signature matching: A key to reuse. *ACM SIGSOFT*, 18(5):182–190, 1993.

Received: August, 1996
Accepted: April, 1997

Contact address:
Slovak University of Technology
Dept. of Computer Science and Engineering
Ilkovičova 3
812 19 Bratislava
Slovakia
e-mail: smolarova@dcs.elf.stuba.sk,
navrat@elf.stuba.sk.
WWW: http://www.dcs.elf.stuba.sk/~smolar,
http://www.elf.stuba.sk/~navrat.

MÁRIA SMOLÁROVÁ received her MSc in Computer Science from Slovak University of Technology in Bratislava in 1986. She is a research assistant at the Department of Computer Science and Engineering and a PhD candidate in Applied Informatics, both at Slovak University of Technology. Her research interests include software engineering, software reuse, object-oriented software development, and design patterns.

PAVOL NÁVRAT received his Ing. (MSc.) summa cum laude in 1975, and his CSc. (PhD.) degree in Computing Machinery in 1983, both from Slovak University of Technology in Bratislava. He has been with its Department of Computer Science and Engineering since 1975. Since 1996, he is a full professor of Computer Science and Engineering. His scientific interests include automated programming and software engineering. He is a member of the IEEE and its Computer Society (active for Software Engineering Standards Committee), American Association for Artificial Intelligence, Slovak Society for Informatics, and Association for Advancement of Computers in Education (co-founded its Central European Chapter). He is a member of editorial board of the international journal *Informatica*. Frequently, he has been serving in programme committees of scientific conferences. He (co-)authored two books and numerous scientific papers. He regularly publishes reviews of monographs and articles in ACM Computing Reviews and other journals.