Architectural Abstractions and Language Mechanisms *

Bent Bruun Kristensen
Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK-9220 Aalborg Ø, Denmark
E-mail: bbkristensen@cs.auc.dk
WWW: http://www.cs.auc.dk/~bbk/

Abstract

When we apply the mechanisms of an object-oriented language we form concrete architectures over some domain. Over time we identify recurring patterns and transform the concrete architectures into more general architectural abstractions. Such abstractions are then used in combination with the language mechanisms to form other kinds of concrete architectures in diagrams and programs: we apply a combination of language mechanisms and architectural abstractions in the modeling and programming process.

Our knowledge concerning the needs and possibilities in the modeling process, especially those captured in the architectural abstractions, enables us to invent new language mechanisms, typically abstraction mechanisms. Such abstraction mechanisms replace several architectural abstractions because of the generality of the mechanisms, and because of their integration with other mechanisms of the language: we invent abstraction mechanisms to replace architectural abstractions.

Software architecture is the different styles and manners of building software in terms of the choice and combination of language mechanisms and architectural abstractions. The mutual influence between object-oriented language mechanisms and architectural abstractions is the basis for the further development of both languages and software architecture.

1. Introduction

Our notation/language for analysis, design and implementation is a combination of language mechanisms. Some of these mechanisms are abstraction mechanisms. In object-oriented programming languages the most central mechanisms are the `class` and the `object`. By the `class` mechanism we name and describe the structure of objects. By the `object` mechanism we instantiate actual objects from a `class` description. Language mechanisms enable us to express ourselves in the notation/language — to construct diagrams/programs. The language also delimits our expressional power by the actual combination of mechanisms available in the language. The mechanisms are an important factor in our understanding and creative thinking concerning problems and solutions. We model an application domain by describing, for example, classification hierarchies and related structures of cooperating objects.

We form abstractions in the form of partial diagrams/programs by applying the notation/language. The abstractions are either abstractions over the application domain or the solution domain. By the `class` mechanisms we describe the concept `person`, and `employee` is another concept, a subclass of `person`. The abstractions in a diagram/program are typically formed by the use of several applications of the abstraction mechanisms. We describe the concept `company` and a relation `works_for/employed_by` between `company` and `employee` to model the actual employment organization. By relation we mean that any `company` is associated with a number of `employees`, and vice versa. We have identified and abstracted the employment facts between companies and employees as some kind of RELATION pattern. The RELATION pattern is a general abstraction, because the situation where two or more concepts are related appears again and again in our concrete architectures. The Relation is recognized as a general architectural abstraction, and when its characteristics are clarified we can apply this abstraction repeatedly without reinventing it each time. The RELATION pattern is described by means of the basic `class/object` and `reference` mechanisms. We simulate the more complicated structure by means of the more basic mechanism. Alternatively, we introduce an abstraction mechanism to model relations explicitly in our language/notation, for example by adding a special `relation_class`, and allow for the instantiation of `relation_objects` [25]. This new `relation` mechanism is integrated with other mechanisms already available in the

*This research was supported in part by the Danish Natural Science Research Council, No. 9400911.

language/notation, for example parameter mechanisms of various kinds, specialization and aggregation mechanisms, etc.

By building new concrete architectures from the new mechanisms this process is iterated and the languages/notations are further developed. The new mechanisms enable us to express not only new abstractions of existing categories, but possibly also new categories of architectural abstractions. The *design patterns* [7] and *frameworks* [13] are two existing categories of architectural abstractions. We see these as architectural abstractions — different in nature, because they serve different purposes. We identify the dimensions in the universe of such categories of architectural abstractions. A given category of architectural abstraction has its own universe — defining for example the dimensions *purpose* and *scope* for the category of design patterns. The design patterns and the frameworks — seen as two different categories of architectural abstractions — are characterized differently by the dimensions in the universe of architectural abstractions. Similarly the language mechanisms are characterized by the extent to which they address themselves towards the support of the dimensions in this universe.

The development process of object-oriented language mechanisms and architectural abstractions is the basis for the further development of the understanding and support for software architecture. The selection and combination of such architectural abstractions and their supporting mechanisms form a software architecture.

Paper Organization. In section 2 we discuss architectural abstractions, especially design patterns, and we outline a preliminary model of the universe of categories of architectural abstractions. In section 3 we discuss the existing abstraction mechanisms in object-oriented languages, and we exemplify a possible further development of languages/notations by additional abstraction mechanisms, that have counterparts among design patterns. In section 4 we identify and illustrate the interplay process between architectural abstractions and language mechanisms. In section 5 we describe some basic requirements to improve and support the architectural aspect of software and we summarize existing related mechanisms. In section 6 we summarize our understanding of the current situation and the requirements for the future in terms of the relationship between language mechanisms and architectural abstractions presented in the paper.

2. Architectural Abstractions

2.1. Design patterns.

A design pattern [7] is a general idea of a design solution to a general basic problem. The structure of a design pattern can be illustrated by a graphical representation of the classes in the pattern. We exemplify design patterns by the MEDIATOR and the DECORATOR.

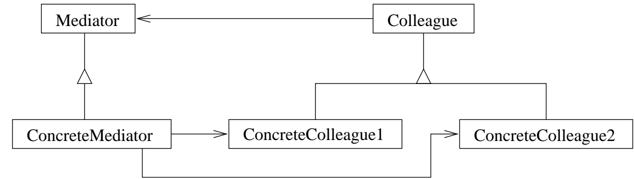

Figure 1. The MEDIATOR **Pattern**

The intent of the MEDIATOR pattern (its structure is reproduced in Figure 1) is to "define an object that encapsulates how a set of objects interact. MEDIATOR promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently". The `Mediator` class defines an interface for communicating with `Colleague` objects. The `ConcreteMediator` implements cooperative behavior by coordinating `Colleague` objects, and knows and maintains its colleagues. Each `Colleague` class knows its `Mediator` object, and each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

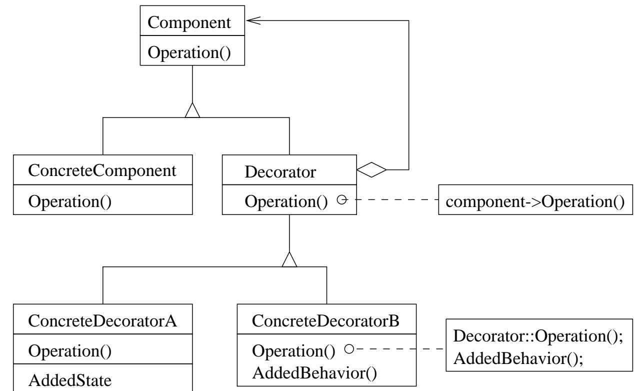

Figure 2. The DECORATOR **Pattern**

The intent of the DECORATOR pattern (its structure is reproduced in Figure 2) is to "attach additional responsibilities to an object dynamically. DECORATORs provide a

flexible alternative to subclassing for extending functionality". The `Component` class defines the interface for objects that can have responsibilities added to them dynamically. The `ConcreteComponent` class defines an object to which additional responsibilities can be attached. The `Decorator` class maintains a reference to a `Component` object and defines an interface that conforms to `Component`'s interface. The `ConcreteDecorator` class adds responsibilities to the `Component`.

Design patterns such as MEDIATOR and DECORATOR are abstract, general design solutions for general classes of problems. Some characteristics of design patterns are:

(1) They are informally defined in terms of problem, solution and consequences; they need to be because they describe a general arrangement/scheme.

(2) They always require instantiation; a pattern needs to be mapped into a concrete implementation space.

(3) They are not atomic; patterns represent interacting parts, systematic structures and relationships.

(4) Because patterns are not atomic, their use in design is at a larger level of granularity than classes/objects; usually they can be expressed in a notation or formal language.

2.2. Categories of Architectural Abstractions.

Design patterns and frameworks are examples of *architectural abstractions* [1]. Design patterns were introduced in the previous subsection. Frameworks are only introduced here by the following quotation from [7]: "a set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by sub-classing and composing instances of framework classes". Each category of architectural abstraction has its own characteristics. Design patterns and frameworks have certain (different and similar) characteristics. We characterize the categories of architectural abstractions according to the following *dimensions* in the *universe* of architectural abstractions — exemplified by the categories design patterns and frameworks:

(1) Level of Abstraction: How abstract is the category? Design patterns are at a higher level of abstraction than frameworks.

(2) Degree of Domain specificity: To what degree is the category specific to some application domain — and to which degree is the category generic? Frameworks are domain specific — design patterns are generic, domain independent.

(3) Level of Granularity: What is the level of granularity of the category? How often and how many times is it typically used in a system? The granularity of frameworks is very large, whereas the granularity of design patterns is very small.

(4) Degree of Completeness: Is the category a complete "application-like" abstract program/concept/idea — or is it "library-like" with a collection of related abstract concepts? Frameworks can be either of these possibilities, whereas a design pattern is either one independent abstraction or part of a collection.

This list of dimensions is not exhaustive — the intention is only to exemplify the universe (in [5] Buschmann & Meunier discuss a classification scheme with three categories granularity, functionality and structural principles). Other aspects of the universe include the collection of related patterns as exemplified by "transaction patterns" (able to interconnect, still without being domain specific) in [4], Meta Patterns [24], the presence of structural and behavioral elements in architectural abstractions, different qualities of architectural abstractions needed in the design, analysis and implementation phases of the software development process [12], the organization and composition of larger, more independent parts of software, like subsystems/components.

3. Language Mechanisms

3.1. Mechanisms: Class & Object.

In this section we discuss the `class` and `object` notation/language mechanisms. Since the appearance of Simula [6], a variety of extensions, elaborations and unifications have appeared in object-oriented programming language design — for example metaclasses in Smalltalk [9], multiple inheritance in C++ [28], assertions/invariants in Eiffel [23], multi-methods in Clos [14], prototypical objects in [87], and unification of class, method and type in BETA [22]. The `class`, `object` etc. mechanisms are available in the notation of object-oriented analysis and design methodologies. In the very natural interplay these methodologies have simply borrowed the object-oriented mechanisms from the programming languages. The notation part of the methodologies has become more rich, and interesting additions have appeared — for example *associations* [26], *contracts* [30], *physical/logical view* [2], and *use cases* [11] — as inspiration for programming language design. The methodologies have all added the method part (a process part in addition to the notation part) that was more or less absent at the programming language level. By this mutual influence, object-oriented programming has become an intimate integration of analysis, design and implementation. However, so far new basic mechanisms have not succeeded in being

[1] We consider usual class libraries, for example container libraries and graphical user-interface libraries, to be architectural abstractions too.

integrated with the `class/object` mechanisms and added to actual programming languages.

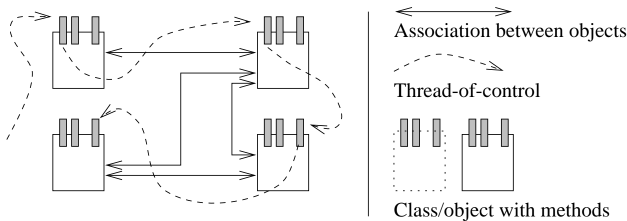

Figure 3. Object-Centric Behavior

In Figure 3, we illustrate the usual organization of an object-oriented structure in terms of (classes and) objects. The static aspect is organized in terms of generalization and aggregation hierarchies of classes (not shown) as well as client-server relations between the classes. The dynamic aspect is organized in terms of actual client-server relations between the object (illustrated as associations) as well as the method invocation structure, where a method of one object invokes a method of another, etc.

Various possible sources of inspiration and motivation exist for the further development of object-oriented languages — or next generations of programming language paradigms. We exemplify one such possibility for research in abstraction mechanisms, namely *conceptual programming* [16]: "Programming is regarded as a modeling process of some referent system where the phenomena and abstractions from the referent system are expressed in a programming language supporting abstractions, which are based on a general understanding of phenomena and concepts". Concept formation is based on phenomena and concepts. A concept has a *denomination* by which it is known, an *extension*, which is the set of all phenomena covered by the concept, and an *intention*, which is a description of the properties of the phenomena from the extension. An example of a concept is "car" where we use the notation `car` as its denomination. All the Toyota Corolla cars in the world form a subset of the extension of `car`. The properties `color` and `drive` are in the intention of `car`. The *abstraction processes* available for concepts/phenomena include for example *classification* and *specialization*. The invention of new language abstraction mechanisms based on the conceptual programming perspective includes at least two elements (where we outline two examples of the last type, namely the *activity* and the *role*):

(1) to use the theory of concept formation as the definition in the mapping to language constructs — for example to define class/object/inheritance to correspond strictly to concept/phenomenon/specialization, and

(2) to explore subconcepts of (the concept) `concept` as candidates for defining more rich and powerful abstraction mechanisms.

3.2. Mechanisms: Activity & Role.

The *activity* concept ([15], [19]) is a natural element in our daily concept formation. We classify phenomena of meetings, sports events, etc as activities. In addition to being a usual concept, the activity is characterized by the fact that it has a duration in time, it includes a number of participating phenomena, and it has some kind of specification for the interplay of actions between the participants ("who is doing what to whom, and when").

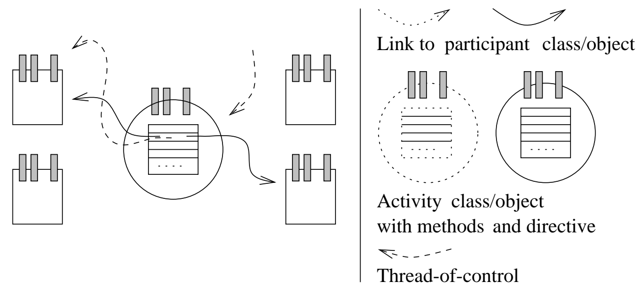

Figure 4. Collective Behavior

In Figure 4, we illustrate the `activity` as an alternative to the usual object-centric method invocation (Figure 3). The scheme employs collective behavior, where the `activity` class can have methods on its own, and prescribes a sequence of interplays each including a caller object and a method call for the (callee) object. The collective behavior is modeled by the `activity` mechanisms — similar to the `class` mechanism, but reflecting the characteristics of the activity as a concept.

As an illustrating example, we consider the activity of reviewing papers for an upcoming conference which can be referred to as a `paper_review`. This activity requires a certain degree of interaction/interplay between those who are involved in it. For instance, an `author` will submit a paper for review, while the `chairman` will distribute papers to each `reviewer` who must report back. There will usually be some sort of specification that describes how the activity should be carried out. Figure 5 illustrates `paper_review`, its participants and how its specification might be carried out (`paper_selection` is a smaller activity that takes place inside `paper_review`).

The *role* concept ([18], [21]) is also a natural element in our daily concept formation. We classify phenomena such as tennis player, lecturer, etc. as roles of the person concept. In addition to being a usual concept, a role is

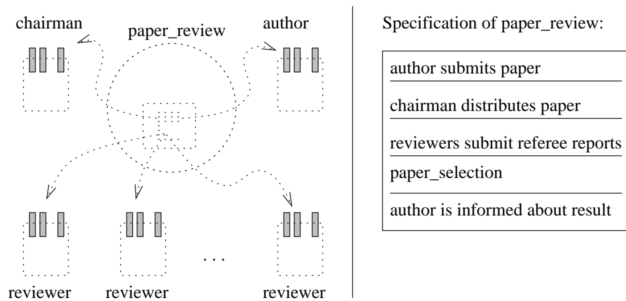

Figure 5. The Activity `paper_review`

characterized by the fact that it is always dependent on an ordinary (intrinsic) concept and has no identity alone, it can be added and removed dynamically, and the properties of a role are visible for certain clients only (including the properties of the intrinsic concept, but excluding those of other roles for the concept).

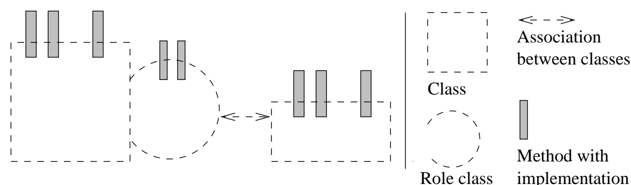

Figure 6. Roleification

In Figure 6, we illustrate the `role` as an alternative to the usual simulation of roles (either by aggregating the roles as usual objects or by roles as specializations of the intrinsic concept). The scheme employs *roleification*, where a `role` class is bound to a class, is associated with other classes, and can have methods on its own. The roleification is modeled by the `role` mechanism — similar to the `class` mechanism, but reflecting the characteristics of the role as a concept.

As an illustrating example, we consider a `Person` involved in the preparation and/or holding of a conference by adding the role `Conference-Associate`. This includes several roles, for example as a `Participant`, an `Author`, and a `Reviewer`. These roles then become roles of `Conference-Associate` — roles of a role — as illustrated in Figure 7. A method of for example `Participant` can utilize the methods of `Conference-Associate` in its description, but not vice versa. We model `Participant`, `Author`, and `Reviewer` as roles of `Conference-Associate` because the various relations to a given conference are related to these specific roles only (and not to `Conference-Associate` as a whole): A `Participant` is related to the `Conference`; an `Author` and a `Reviewer` are related to a `Paper`.

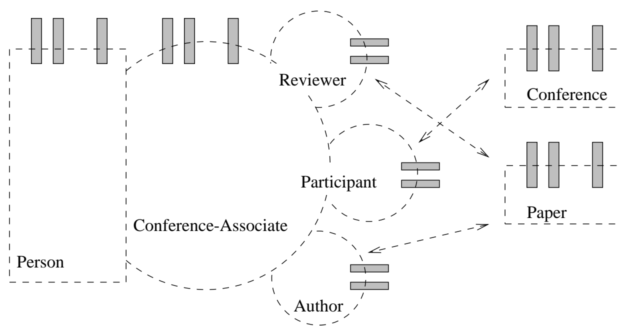

Figure 7. Roles of `Conference-Associate`

Abstraction mechanisms such as the `activity` and `role` are potential elements of languages/notations with important characteristics:

(1) They are well-defined and they are basic building blocks. As a consequence of their nature, the mechanisms are at a more elementary level.

(2) They can be integrated with other mechanisms (for example inheritance and polymorphism) in the language/notation to be even more powerful.

(3) They can be used in the creative thinking about a problem. They form our basic understanding and thinking about problems as well as the initial outlining of a model.

(4) Abstractivity — the abstraction processes available for these mechanisms such as:

(a) *Classification*, being the process of going from a phenomena to a concept, and the inverse *exemplification*, going from a concept to a concrete phenomena covered by that concept.

(b) *Aggregation*, being the process of combining several concepts into a new concept.

(c) *Specialization*, being the process of creating a more special concept on the basis of a general one, so that the extension of the specialized concept is a subset of the extension of the general concept.

4. Evolution Spiral

The development process of object-oriented language mechanisms and architectural abstractions can be seen as an evolution spiral. This development is exemplified by the design patterns MEDIATOR and DECORATOR and the abstraction mechanisms `activity` and `role`. The MEDIATOR/DECORATOR pattern is an architectural abstraction, the purpose of which is similar to the language mechanism `activity/role`. The point is not which these was invented before the other, but only that these are intended to support the same modeling situation, and that there is an

obvious mutual influence between the two approaches for these examples[2].

They are examples of the outcome of the abstraction evolution spiral: from language mechanisms we form concrete architectures in programs — from which we form more general architectural abstractions. Inspired from architectural abstractions and other sources as for example concept formation, we invent new language mechanisms, typically abstraction mechanisms. Before the `procedure` was invented as an abstraction mechanism in imperative programming a CODE-SEQUENCE pattern was identified as important in assembly programming. A CODE-SEQUENCE is reusable by introducing an entry point for the sequence, and return jumps to varying places (by saving a return address). By this pattern we can simulate the meaning of the abstraction mechanism which later has become known as the `procedure`. The `procedure` has been integrated with various parameter mechanisms to make it an even more powerful mechanism. Recursion is another pattern captured by the introduction of recursive `procedures`. The `record` is an example of an abstraction mechanism based on a DATA-SEQUENCE pattern. A certain sequence of dependent data is identified as important, because it is always instantiated and used as a unit. The `record` abstraction mechanism captures these properties and is integrated with for example the `pointer` mechanism to make it an even more powerful mechanism. Instead of recursive data structures we use general patterns of various forms to model linked lists, binary trees etc.

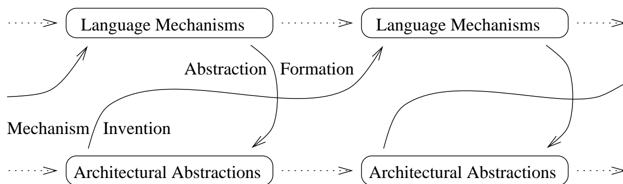

Figure 8. Evolution of Abstractions

In Figure 8, we illustrate the result of the evolution spiral in terms of the development of abstraction mechanisms and architectural abstractions over time. There are several observations and possible consequences relative to this spiral:

(1) The design pattern and the language mechanism are both developed because some support for a modeling situation is needed. A design pattern can easily be added to the collection of architectural abstractions, whereas in general it is very costly to add a new language mechanism to a programming language.

(2) The abstraction mechanism can easily be added to the object-oriented analysis and design methodologies and at least be used in these phases of the development process. If a programming language contains the abstraction mechanism then a design pattern is not needed — else the design pattern can be used as an implementation strategy for the mechanism.

(3) An abstraction mechanism is more general than a design pattern, and it will cover more situations, but there can be several possible ways of applying it. A design pattern can be more straightforward to apply because it prescribes very explicitly an abstract solution to a specific group of problems in some given context and with known consequences.

(4) The design pattern is an abstract idea, and it prescribes an implementation of the idea. When we use a design pattern we go through the refinement process[3]: design pattern (abstract pattern) to instantiated pattern (concrete design) to implemented pattern (code). The design pattern is maintained by the user and its organization is visible in the program/diagram; the implementation is done manually for each application. The language mechanisms hide the implementation entirely and only the abstraction is visible in the program/diagram; the implementation of an application of a mechanism is done automatically without user involvement.

(5) By the `activity` and `role` mechanisms we can create libraries of activity and role classes for reuse. We can develop libraries of useful abstractions organized in generalization and part hierarchies. In using the MEDIATOR or the DECORATOR pattern we do not create libraries. We use the abstract pattern as a solution and we implement it, but this is really not practical for reuse. It is a concrete implementation built into some system. The abstract idea of a pattern is for reuse of design, not the instantiated ones (only for reuse of code).

Mechanism Invention & Abstraction Formation. In the evolution spiral two evolution processes have mutual influence on each other as illustrated in Figure 9:

Mechanism Invention: In this process we invent an abstraction mechanism from architectural abstractions, seen as general ideas and solutions to problems (including examples of abstract pieces of programs). We apply architectural

[2] By means of delegation one object can offer some behavior, and supply additional behavior by delegating responsibility of another object. This organization is the basis for the DECORATOR pattern, and also partial inspiration for the invention of a role language mechanism. Other sources of inspiration include conceptual programming as well as the notion of *subjectivity* from [10]. The invention of the role mechanism motivates the formation of new abstractions — patterns of different kinds (dependent on which phase in the development process they support) are described in [20].

[3] To identify and decide to use some abstract pattern is different from the use of a language mechanism. Both involve recognition and creativity, but to instantiate and implement a pattern is more like a construction or engineering process. When a mechanism is used, the result of the process is an abstraction (and nothing more needs to be described/implemented). When a pattern is used, the result of the process is an implementation in the form of code/diagrams (because we see the classes, their relations, how the messages are delegated around).

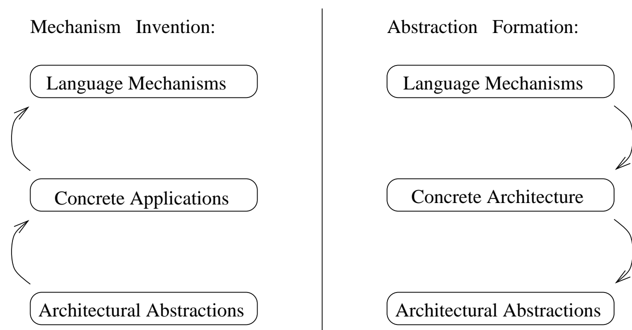

Figure 9. Evolution Processes

abstractions in actual concrete programs — we identify the use of similar architectural abstractions to form similar fundamental and central parts of our programs. Over time we conceive a language mechanism to replace the use of architectural abstractions for the identified purpose. We invent the abstraction mechanism — we define it, we integrate it in our language, and we implement it. A mechanism is designed and defined, including its syntax and semantics as part of a language/notation. The mechanism is supposed to subsume one or more architectural abstractions, and is intended to capture a natural element in the abstraction process. At the programming language level the abstraction mechanism has a general implementation, hidden by the language implementation. The success of a mechanism depends upon whether it truly replaces a number of architectural abstractions, whether it is integrated with the rest of the language/notation (especially the other abstraction mechanisms), its naturalness and usefulness in the abstraction process, how comfortable the user is with its implementation, and the efficiency of the implementation.

Abstraction Formation: In this process, we first form concrete architectures, followed by more abstract versions of these. Using language mechanisms we design and construct formations and organizations — as concrete architectures — including both structure and behavior, at a higher level than what we can express directly by the given mechanisms. From experience with several of such concrete architectures we form more general and abstract formations and organizations — architectural abstractions. The success of an architectural abstraction depends among others on how general an idea is captured by the abstraction, how simple and powerful the abstraction is (not cluttered by complicated structure and behavior), the number of important applications, it's ability to be integrated with other abstractions, and if it is still identifiable in the code after it has been applied[4].

[4]The *generative patterns* [1] have the added advantage of documenting not only how they were applied, but also why.

5. Elements of Architecture

Architectures are built by the application of architectural abstractions and language mechanisms. Using existing system design methodologies and object-oriented programming languages, some support for software architecture is available. However, the description and execution support for architecture is primitive and insufficient. We describe some basic requirements to improve the support of architecture. The requirements include the notion of a subsystem/component, some interaction relationships between these, some basic organizations of these, and a principle of application independent description of these. The requirements are related to existing architectural abstraction and language mechanisms, that are intended to support architectural purposes. The claim is that the potential of the evolution spiral discussed in the previous sections for abstractions and mechanisms in general provides the means for the evolution of advanced and adequate abstractions and mechanisms for software architecture in particular.

5.1. Basics of Subsystems/Components.

Architecture means "the style or manner of building". For a software system we interpret *software architecture*[5] to mean

— the actual choice of architectural abstractions and language mechanisms, especially abstraction mechanisms, and

— the actual use of the elements chosen, i.e. how they are combined, their iterations, variations etc.

In principle there is always a choice between applying a language mechanism and an architectural abstraction. In practice the given language/notation gives little choice: either the organization can be supported by means of an abstraction mechanism, in which case this is the obvious solution, or it cannot, in which case an architectural abstraction is the solution (if a suitable one exists). In both case there can be several candidate abstractions, each fitting more or less appropriately.

Our claim is that some kind of explicit subsystem/component concept is needed — both for the description and for the instantiation at execution time — to better support software architecture. The FACADE pattern [7] is a simulation of an explicit subsystem/component concept — the applicability and consequence aspects of the FACADE pattern are relevant for such concepts. In Figure 10, we illustrate a notation for subsystems/components. We have descriptions

[5]In [8] the following characterization is given: "Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among those components. Such a system may in turn be used as a (composite) element in a larger system design".

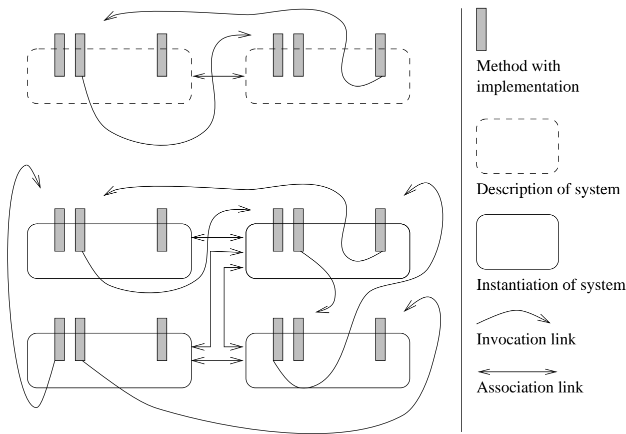

Figure 10. Lateral Component Organization

as well as instantiations of subsystems/components, and we have different kinds of relations between these. We do not define the characteristics of subsystems/components in any detail, but we briefly outline some properties of these. Given this preliminary understanding of the subsystem/component concept we can gradually develop architectural abstractions and language mechanisms to support and explore such a concept.

The *interaction relationships* between subsystems/ components are indicated by:

— *association links* (indicating that one subsystem/component has access to another, and probably vice versa) and

— *invocation links* (indicating that one method can invoke another method).

These relationships between subsystems/components are supported by existing mechanisms and abstractions. However, there is almost no support for

— the description and organization of the *interplay* (the sequence of mutual access/invocation, longer lasting, tightly coupled, committed interaction, implicit or explicit acceptance of interaction, etc.).

In the *basic organization* of subsystems/components we distinguish between *lateral* (Figure 10) and *nested* (Figure 11) organization:

In the lateral organization the actual organization is entirely explicit with the subsystems/components enjoying the same status. The architectural aspects — the choice and use of mechanisms and abstractions in the lateral organization — cover both the type of the organization as such and the type of the elements in the organization.

In the nested organization the local subsystems/ components are implicitly dependent on the enclosing subsystem/component (the dependence involves for example life-

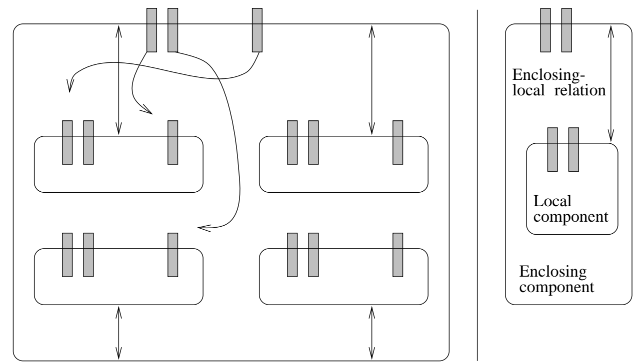

Figure 11. Nested Component Organization

time, visibility, etc.). The architectural aspects — the choice and use of mechanisms and abstractions in the nested organization — cover both the types of the enclosing/local subsystems/components and the additional (except from being nested) organization between the enclosing element and the local ones.

In both lateral and nested organizations all forms of interaction relationships are suitable for the support of the description of the architecture. Layers and partitions [26] are both lateral organizations, with specific access disciplines enforced. Complex associations [17] support a specific and advanced kind of nested organization.

A subsystem/component must be describable *independently* of its actual use. The Subsystem/component must express its interface, interplay and organization requirements towards other subsystems/components (including for example graphical user-interface or database systems). Independently of its description it must be possible later to describe a coupling of a subsystem/component by additional descriptions (not only of the mutual access relations, but specifically of the interplay organization). In the lateral organization a subsystem/component can be added and removed. In the nested organization the enclosing subsystem/component itself is described in terms of its local subsystems/components, but still independently of its use.

In summary the requirements to subsystems/components include the explicit support of:

(1) A subsystem/component concept.

(2) Interaction relationships, not only association and invocation, but also interplay.

(3) Both lateral and nested organization.

(4) Description independently of use.

5.2. The Facade Pattern.

The intent of the FACADE pattern [7] is to "provide a unified interface to a set of interfaces in a subsystem. FACADE

defines a higher-level interface that makes the subsystem easier to use". The FACADE pattern is applicable when we want to provide a simple interface to a complex subsystem, when there are many dependencies between clients and the classes of the subsystem, and when we want to layer our subsystems. The intent of the FACADE pattern is illustrated by Figure 12 (reproduced from [7]).

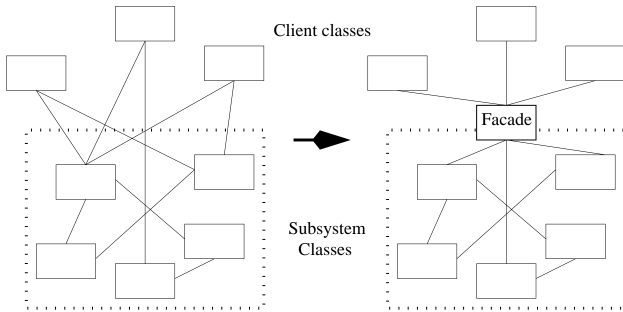

Figure 12. The FACADE Pattern

The `Facade` class knows which subsystem classes are responsible for a request, and delegates client requests to appropriate subsystem objects. The subsystem classes implement subsystem functionality, handle work assigned by the `Facade` object, and have no knowledge of the `Facade` object. Some of the characteristics of a subsystem/component are captured by the FACADE pattern, and a partial implementation strategy is given. The consequences of the FACADE pattern are that the clients are shielded from the subsystem classes, a weak coupling between the subsystem and the clients (possibly involving a strong coupling between the classes of the subsystem), and the clients can still use classes of the subsystem directly.

5.3. Layers and Partitions.

In system design a system is usually broken into a number of smaller components (or subsystems). A subsystem encompasses aspects of the system that share some common property — for example functionality or physical location. The relationship between two subsystems can be *client-server* or *peer-to-peer* with the client calling the server for the former or the subsystems calling each other for the latter. These forms of relationship are usually supported by interface descriptions only. The organization of the subsystems can be organized as a sequence of horizontal *layers* or vertical *partitions* (Figure 13). The partitions in a layer can be independent or weakly-coupled subsystems. A subsystem at one layer knows about the layers below it, but not about the layers above it. These forms of organization are supported by means of simple version control and configuration management systems or by file systems alone (for example the `include` directive in C++).

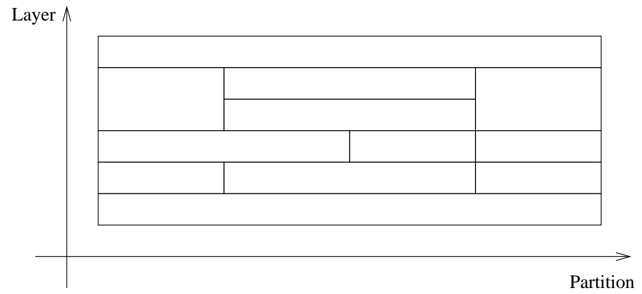

Figure 13. Layers and Partitions

5.4. Complex Associations.

The associations in OMT [26] are object-external abstractions and are useful for designing and partitioning systems of interrelated objects. *Complex* and *implicit associations* [17] are more powerful object-external abstraction mechanisms. Implicit associations are described by means of enclosing components (or classes), the instances (or objects) of which have methods, attributes, etc. Components can be local to enclosing components. Complex associations are described by means of association components, the instances of which have the usual properties of association objects. Associations can be enclosing components also. Figure 14 is a schematic illustration of the typical flat model with objects and simple associations and a nested, structured model with complex and implicit associations.

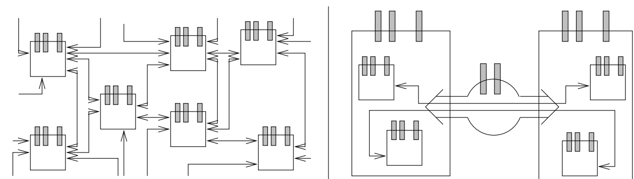

Figure 14. Simple & Complex Associations

As an illustrating example we consider a system to support a computerized banking network including both human cashiers and automatic teller machines (ATM's) (the object diagram in [26] page 161 is a flat model for this example). The classes are related in more complex structures — components and association components. At the top level a relation `Banking` is an association between `Consortium` and `Customer`. The `Consortium` and `Customer` are the

domains of the Banking association. At the next level the Consortium is organized as Bank and ATM. Similarly, the Customer is organized as Cash-Card and Account. The Consortium is an enclosing component but it also consists of a number of components local to it (similarly for Customer). The associations between the components of this next level can exist only because of the association between the enclosing components at the top level, and they are local to the association component on the top level. In the example, Account is associated with Bank by Holds, and Cash-Card is associated with ATM by Authorized-By. Therefore associations such as Holds and Authorized-By are local to the association Banking. The domains of these local associations are domains local to the domains of Banking.

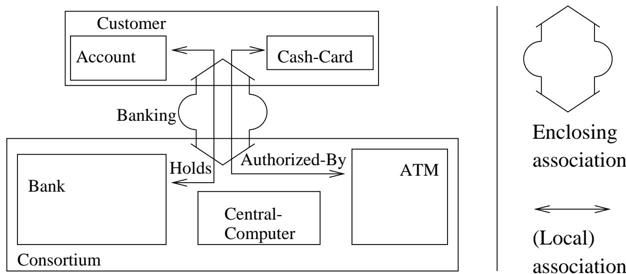

Figure 15. The Banking Association

The Bank, ATM and Centralized-Computer components are local to the enclosing component Consortium (in Figure 15 the *local to* relation is illustrated by nesting one component into another). The model has an implicit association between the enclosing component and each of its components. Similarly, there are implicit associations between each of the associations Holds and Authorized-By being local to the association Banking.

5.5. Common Architectural Styles

In [8] Garlan and Shaw examine some representative, broadly-use architectural styles[6]. In addition to *object-oriented organization* and *layered systems* the following styles are illustrated and evaluated: *pipes and filters*, *event-based, implicit invocation*, *repositories* and *table driven interpreters*. Through a series of case studies it is shown how architectural principles can be used to increase our understanding of software systems.

Similar styles and case studies are included in [27]. This book also considers the need for new higher-level languages specifically oriented to the problem of describing software architecture. It is shown how ideas from programming language design apply to software architecture, the requirements to the characteristics of such architectural languages are presented, and it is shown how existing approaches fail to satisfy these requirements. The idea is not to propose a particular language, but rather to establish a framework to support architectural language design. In Chapter 7, the general linguistic nature of architectural description is considered. The general requirements for architecture-description languages are followed by a specific focus on the composition of components in terms the interactions among the components, denoted by *first-class connectors*. The purpose with having connectors as first-class entities along with components in system composition, is to avoid to leave the description of the interactions among the components implicit, distributed, and difficult to identify.

[6]"We are still far from having a well-accepted taxonomy of such architectural paradigms, let alone a fully-developed theory of software architecture. But we can now clearly identify a number of architectural patterns, or styles, that currently form the basic repertoire of a software architect"

5.6. Prototypical Architectures.

In [26], Section 9.10, six common architectural frameworks are described: batch transformation, continuous transformation, interactive interface, dynamic simulation, real-time system and transaction manager. Each of these categories of systems is characterized according to the importance and usefulness of the three views of modeling systems in OMT, namely the object model, the dynamic model and the functional model. The overall architecture of the prototypical systems is not described in any notation, diagram or figure.

In [26], Section 9.11, the architecture of an ATM system (characterized as a hybrid between an interactive interface and a transaction management system) is shown by Figure 9.4. The illustration is informal, primitive and little informative, and shows the three major subsystems as physical units. These subsystems are connected by phone lines, and each has some internal organization. The entire system is characterized as a simple architecture — still very insufficiently and informally described.

5.7. Other Architectural Organizations.

Abstraction mechanisms can support the *logical* and *physical* view of a system [2]. The logical mechanisms express the meaning of the description and the physical mechanisms express the organization of the description in manageable pieces for different purposes. Often the two views are mixed in a mechanism — then its properties must be defined clearly according to this distinction.

The *module diagram* in [2] is part of the physical design of a system and describes the allocation of classes and objects in software modules as a concrete implementation of the logical design. *Subsystems* are introduced to represent

clusters of logically related modules. The *class category* is an abstraction mechanism supporting the understanding of the logical architecture of a system, but it has no effect on the execution of a system and supports program organization only.

The concepts *subsystems* and *contracts* of [30], building on the concepts *responsibilities* and *collaborations*, are powerful mechanisms for understanding and expressing the relationships between classes and groups of these. The mechanisms have no semantic influence and give only additional information in relation to the organization and cooperation of objects.

The *subject* in [3] is an organizational structure intended to guide the reader through a large complex model. In OMT [26] the *module* is a logical construct for grouping classes and associations; a *sheet* is the mechanism for breaking a large model down into a series of pages and a module consists of several sheets; the *system architecture* is part of the physical model.

Nested *patterns* (block structure for classes) in Beta is similar to enclosing and local classes. The nested class of C++ (and the *friend* mechanism) is related to visibility only. The *cluster* in Eiffel (Lace only) is used for arranging classes into groups. Clusters do not require specific language support but are provided by the operating system facilities.

6. Summary

We have discussed the relationships between the architecture of software, the predefined architectural abstractions available for the program construction process, and the language mechanisms available in the language/notation, especially the abstraction mechanisms.

By understanding the architecture of some piece of software we understand the style and manner in which the software is built. This includes the choice of the abstraction mechanisms of the language and the choice of predefined architectural abstractions, and the way these are applied in the software. We need a better understanding of the needs and possibilities in the organization of the components/systems in the software, and more mechanisms and abstractions to support such organizations as well as methods prescribing how to apply such mechanisms and abstractions.

Design patterns and frameworks are examples of abstractions in the universe of categories of architectural abstractions. We need a better understanding of the dimensions of this universe, especially to what extent the dimensions are related to the various aspects of the architecture of software. The usability and understandability of the architectural abstractions depend intimately on the abstraction mechanisms available for expressing these abstractions. We need to understand what categories of mechanisms must be available in order to further explore the applicability of architectural abstractions.

The abstraction mechanisms of a language/notation are the ultimate basis for the conception and description in the modeling process. The mechanisms are also the basis for the description of general architectural abstractions as extensions of the language/notation. The further development of abstraction mechanisms is partly inspired from various similar architectural abstractions, and partly from other sources, for example conceptual modeling. We need further development of the abstraction mechanisms — also inspired from the universe of categories of architectural abstractions.

We conclude that the continuous development spiral with mutual influence between abstraction mechanisms and architectural abstractions is a basis for the development of even more advanced language mechanisms and architectural abstractions. We also conclude that this continuous development spiral is the basis for the further understanding and development language mechanisms and architectural abstractions to the support of software architecture.

Acknowledgments. We thank Kasper Østerbye and Daniel May for continuous discussions concerning the topics of this paper as well as joint work on activities and roles in conceptual programming.

References

[1] K. Beck, R. Johnson: Patterns Generate Architectures. Proceedings of the 8th European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, Vol. 821 Springer-Verlag, 1994.

[2] G. Booch: Object Oriented Analysis and Design with Applications. Benjamin/Cummings, 1994.

[3] P. Coad, E. Yourdon: Object-Oriented Analysis. 2/E, Prentice Hall, 1991.

[4] P. Coad, D. North, M. Mayfield: Object Models: Strategies, Patterns, & Applications. Prentice Hall, 1995.

[5] J. O. Coplien, D. C. Schmidt: Pattern Languages of Program Design. Addison-Wesley, 1995.

[6] O. J. Dahl, B. Myhrhaug, K. Nygaard: SIMULA 67 Common Base Language. Norwegian Computing Center, edition February 1984.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[8] D. Garlan, M. Shaw: An Introduction to Software Architecture. In: Advances in Software Engineering and Knowledge Engineering, (ed. V. Ambriola, G. Tortora), World Scientific Publishing Company, 1993.

[9] A. Goldberg, D. Robson: Smalltalk-80 - The language and its implementation. Addison-Wesley, 1983.

[10] W. Harrison, H. Ossher: Subject-Oriented Programming (A Critique of Pure Objects). Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, 1993.

[11] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard: Object-Oriented Software Engineering, A Use Case Driven Approach. Addison-Wesley, 1992.

[12] E. E. Jacobsen, B. B. Kristensen, P. Nowack. Patterns in the Analysis, Deign and Implementation of Frameworks. Department of Computer Science, Aalborg University, 1996.

[13] R. E. Johnson, B. Foote: Designing Reusable Classes. Journal of Object-Oriented Programming, 1988.

[14] S. E. Keene: Object-Oriented Programming in Common Lisp. Addison-Wesley, 1989.

[15] B. B. Kristensen: Transverse Activities: Abstractions in Object-Oriented Programming. Proceedings of International Symposium on Object Technologies for Advanced Software (ISOTAS'93), 1993.

[16] B. B. Kristensen, K. Østerbye. Conceptual Modeling and Programming Languages. SIGPLAN Notices Vol.29, No.9, 1994.

[17] B. B. Kristensen: Complex Associations: Abstractions in Object-Oriented Modeling. Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, 1994.

[18] B. B. Kristensen. Object-Oriented Modeling with Roles. Proceedings of the 2nd International Conference on Object-Oriented Information Systems, 1995.

[19] B. B. Kristensen, D. C. M. May. Activities: Abstractions for Collective Behavior. Proceedings of the European Conference on Object-Oriented Programming, 1996.

[20] B. B. Kristensen, J. Olsson. Roles & Patterns in Analysis, Deign and Implementation. To appear in: Proceedings of the 3rd International Conference on Object-Oriented Information Systems, 1996.

[21] B. B. Kristensen, K. Østerbye. Roles: Conceptual Abstraction Theory & Practical Language Issues. To appear in: Theory and Practice of Object Systems (TAPOS), 1996.

[22] O. L. Madsen, B. Møller-Pedersen, K. Nygaard: Object Oriented Programming in the Beta Programming Language. Addison-Wesley 1993.

[23] B. Meyer: Eiffel, The Language. Prentice Hall, 1992.

[24] W. Pree: Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.

[25] J. Rumbaugh: Relations as Semantic Constructs in an Object-Oriented Language. Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, 1987.

[26] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: Object-Oriented Modeling and Design. Prentice Hall 1991.

[27] M. Shaw, D. Garlan: Software Architecture — Perspectives on an Emerging Discipline. Printice-Hall, 1996.

[28] B. Stroustrup: The C++ Programming Language. 2/E, Addison-Wesley 1991.

[29] D. Ungar, R. B. Smith: Self: The Power of Simplicity. Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, 1987.

[30] R. Wirfs-Brock, B. Wilkerson, L. Wiener: Designing Object-Oriented Software. Prentice Hall, 1990.