3. SYSTEMS ENGINEERING PRINCIPLES

Systems engineering and its perspective on information system design form the central themes of this chapter. Systems engineering is distinguished from other fields of engineering in the first section of this chapter. We continue with the principles of designing system architectures in section 3.2. *Object-orientation* is introduced as a preferred modeling paradigm of systems engineers in section 3.3. Principles for object-oriented modeling are discussed in section 3.4.

Systems engineering and its perspective on information system design form the central themes of this chapter.

3.1 Systems engineering

Since the 1950s a number of interrelated intellectual areas such as general systems theory, information theory, cybernetics, control theory and mathematical systems theory have emerged (Ashby, 1956; Klir, 1985). These areas can be identified by the general term *systems sciences* of which the engineering subset is called *systems engineering*. Simon (1976); Klir (1985) use three basic components of scientific inquiry to compare system sciences and traditional sciences; these are:

- the *domain* of inquiry
- the body of *knowledge* regarding the domain
- a *methodology* linking activities in the process of problem solving, or knowledge acquisition

We start with a definition of a system as a way to introduce the domain of inquiry of systems engineering, or science.

Definition 3.1.1 *A system is a part of the world we choose to regard as a whole, separated from the rest during a period of consideration, which contains a collection of objects, each characterized by a selected set of attributes, operations and relations (Holbaek-Hansen, 1975).*

A system is a part of the world we choose to regard as a whole, which contains a collection of objects.

The value of this definition for our research is that it is rooted in an activistic philosophy; system boundaries, objects and attributes are all subjectively chosen and selected. Systems engineering is thus considered to be a subjective, procedural rational activity.

An *information system* in juxta position to a *real system* is introduced in figure 3.1. Brussaard and Tas (1980) define this *real system* as those parts or aspects of reality we want to investigate as a whole, with the intent to know, or eventually to control.

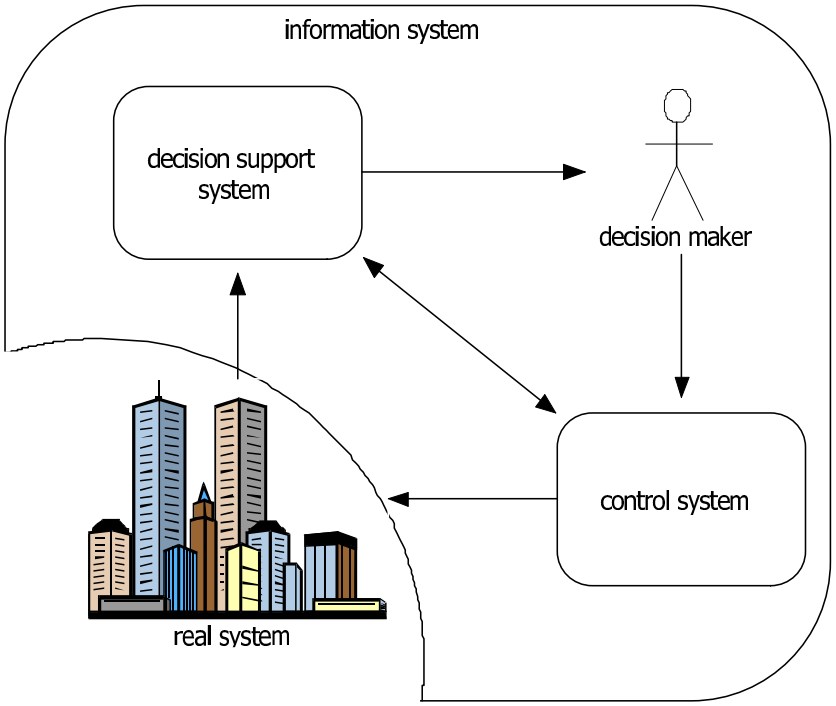

Fig. 3.1: Information System

Brussaard and Tas (1980) continue by defining the functions of *information systems* as: the collection, storage, processing, retrieval, transmission and distribution of data by human beings and machines.
We started this section by introducing the three components by which Simon (1976); Klir (1985) compare different sciences. The first component, i.e. the domain of inquiry, of *systems engineering* can now be presented. According to Klir (1985); Zeigler et al. (2000), systems engineering deals with the design of systems; systems engineers focus on the design and the specification of system structure: the objects, and relations, i.e. the behavior.
The second component by which Simon (1976); Klir (1985) compare different sciences comprises the *body of knowledge* of systems engineering. How, in other words, does one obtain knowledge about the relations within a system? As argued in chapter 1, one can obtain knowledge using a formal mathematical deductive

approach or by using an inductive approach. A consecutive multidisciplinary inductive approach is prescribed by Churchman (1971); Bosman (1977); Sol (1982); Keen and Sol (2005) in the context of decision support for ill-structured problems, in this approach the computer becomes the *laboratory* for the systems engineer.
As in any science, modeling paradigms and languages exist to make the body of knowledge communicatable. Object-orientation has emerged as the de-facto modeling paradigm in systems engineering (Booch et al., 1999). Since information system development has been influenced so heavily by this paradigm, the rest of this chapter is used to introduce its history, principles and consequences.

Object-orientation has emerged as the de-facto modeling paradigm of systems engineers.

This is not to forget the third component by which Simon (1976); Klir (1985) compare different sciences: the *methodology*. The methodology of systems engineering forms the basis of chapter 4 on *simulation*. It involves the activities of conceptualization, specification, verification, validation and experimentation.

3.2 Principles for system design

We present a variety of strategies to divide systems into modular sub-systems in this section. We introduce the value of this concept by taking a look at the size of a typical information system: between 10^1 and 10^5 objects are related (Eckel, 2000). If we cannot find a strategy that we can use to divide such design into subsystems, (re)use, validity and future development becomes at least questionable. The concept of subsystems is introduced to help us understand the variety of strategies that might be used.

Definition 3.2.1 *A subsystem is a system that is part of some larger system (Gove, 2002).*

This implies a recursive definition of a system S as a set of systems $\{s\}$. The usefulness of this concept is entwined with the concept of *modularity*. The following principle reflects strategies for dividing systems into subsystems.

System design requires decomposition into either vertically partitioned or into horizontally layered subsystems.

Principle 3.2.1 *System decomposition results into either vertically partitioned or into horizontally layered subsystems.*

Both approaches are illustrated in figure 3.2. A *horizontally layered* system is an ordered set of subsystems in which each of the subsystems is built in terms of the ones below it. A *vertically partitioned* system divides a system into multiple autonomous, and therefore more loosely coupled subsystems, each providing a particular service. The orthogonal decomposition of systems into either vertical partitions or horizontal layers is not exclusive. Systems can be successfully decomposed into various combinations according to both approaches; partitions can be layered and layers can be partitioned.

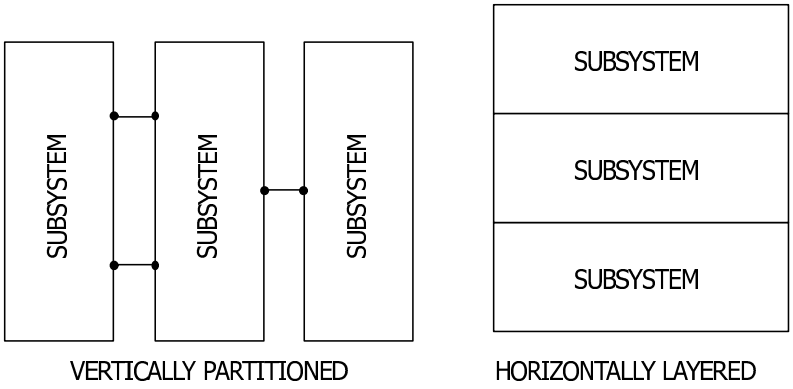

Fig. 3.2: Decomposition of systems into subsystems

This orthogonal decomposition is not exclusive: partitions can be layered and layers can be partitioned.

The value of dividing a system into horizontal layers and vertical partitions is entwined with the concept of *separation of concerns*, and thus with the concept of a subjective choice for a part of a larger system under investigation (Holbaek-Hansen, 1975; Sol, 1982). Separation of concerns is at the core of systems engineering. It refers to the ability to identify, encapsulate, and manipulate those parts of a system that are relevant to a particular concept, goal, task, or purpose (Tarr and Ossher, 2001). Guided by this principle of systems decomposition, we can describe the objects and underlying relations of these decomposed systems: we introduce object-oriented systems description.

3.3 Object-oriented system description

Nygaard and Dahl launched a project in 1962 to develop a discrete event simulation language, to be called *Simula* (Dahl, 2002). The resulting language was called *Simula 1* and was, partly because of European patriotism, based on the *Algol 60* language.
In 1967, Hoare (1968) proposed the concept of *record handling*, consisting of record classes and subclasses. Based on this proposal, Nygaard and Dahl stripped Simula 1 of all references to simulated time to give us the general purpose *Simula 67* programming language. Nygaard and Dahl used the terms *class* and *object*, and thus object-orientation was officially born.

The use of objects distinguishes object-orientation.

The use of objects distinguishes object-orientation from techniques such as traditional structured methods, i.e. process-based methods in which data and function are separated, or other techniques such as knowledge based systems, i.e. logic programming, or mathematical methods, i.e. functional programming.

Several more object-oriented modeling languages appeared in the mid 1970s as systems engineers began to experiment with this alternative approach to analysis and design. New generations of tools and techniques emerged among which were Fusion, Coad-Yourdon, OMT and OOSE (Meyer, 1997).
A critical mass of ideas emerged in the early 1990s when the designers of these tools and techniques began to adopt ideas from other workers. With this came the advent of the *Unified Modeling Language* (Booch et al., 1999).
Dahl (2002) argues that the importance of the object-oriented paradigm today is such that one must assume that something similar would have come about with or without the Simula effort. The fact remains however, that the object-oriented paradigm was introduced in the mid 60s through *Simula 67*.
The basic theory of object-orientation is to divide a system into objects and relations. As described, an object is characterized by a selected set of attributes, operations and relations. Objects are instances of a *class*, which is a description of a set of objects that share the same attributes, operations, relationships and semantics (Booch et al., 1999). Object-orientation distinguishes the following two types of relationships:

The basic theory of object-orientation is to divide a system into objects and relations, i.e. ...

- *generalization* $\leftrightarrow$ *specialization*; class A is a generalization of class B if, and only if, every instance of class B is also an instance of class A, and there are instances of class A which are not instances of class B. Equivalently, class A is a generalization of B if B is a specialization of A. Formally we speak of a generalization between classes whenever $\forall x(Bx \rightarrow Ax), \exists x(Ax\neg \rightarrow Bx)$.

...generalization and association.

- *association*; where generalization specifies a relation between classes, association refers to the structural relation between objects, or instances. A special form of association that specifies a *whole-part* relationship between the aggregate, i.e. the whole, and the object, i.e. the part, is called *aggregation*, or *decomposition*. An aggregation relation is an association relation with the exception that instances cannot have cyclic aggregation relationships, i.e. a part cannot contain its whole.

Both types of relations are presented in figure 3.3. The arrow connecting the `Manager` and the `Employee` illustrates a specialization relation. A manager is thus a special employee and its class inherits both name and contract from the `Employee` class.
The association between the `Manager` class and the `Employee` class distinguishes the `Manager` from the `Employee` and therefore justifies the existence of the `Manager` class. This relation is equivalent to the relation between an employee and his or her name and contract. They all express association relations.
A number of principles, or guidelines, have evolved to improve the quality of object-oriented modeling and design since its birth in 1967.

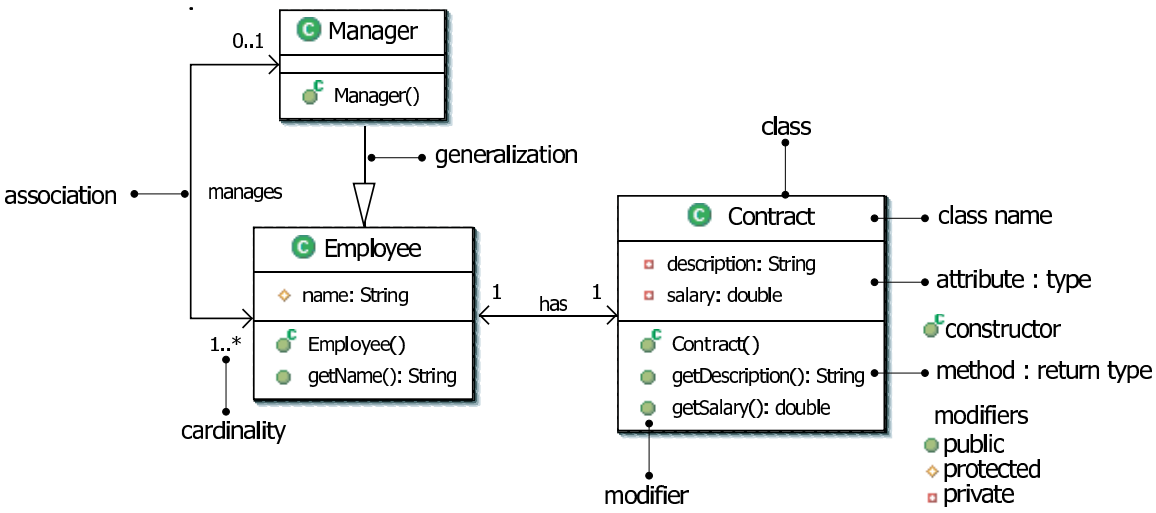

Fig. 3.3: Relations in object-orientation

3.4 Principles for object-oriented modeling

A number of principles, or guidelines, have evolved to improve the quality of object-oriented modeling.

The principles for object-oriented modeling as described in (Lee and Tepfenhart, 2002; Booch et al., 1999; Eckel, 2000) are presented in this section. Where these principles require examples to illuminate their inner-works or importance, they are specified in the Java programming language.

Principle 3.4.1 *Class design: the ability to specify classes.*

A class can be viewed from different perspectives: modeling, design, implementation and compilation. From a modeling perspective, a class is a template for a *category* of objects. It defines the attributes, operations and relations of the category and thus of all objects belonging to the category. From an implementation perspective a class is a *global* object with *globally accessible* attributes, relations and operations.

Information hiding denotes that an object explicitly describes which attributes and methods are publicly visible.

Principle 3.4.2 *Information hiding: an object explicitly describes which attributes and methods are publicly visible and therefore accessible to other objects.*

Object-orientation provides modifiers to define to what extent attributes and operations are hidden from other objects. Besides a public modifier, an object may declare its attributes to be *protected* or *private*. Where the private modifier is used to encapsulate the attributes of an object, the protected modifier grants access to the object and all its subclasses (see principle 3.4.5).

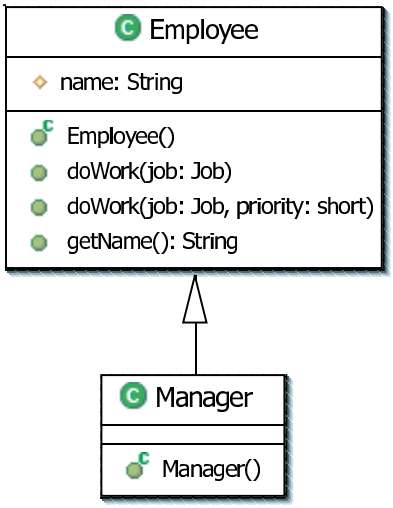

Fig. 3.4: Overloading

Principle 3.4.3 *Encapsulation: attributes and operations uniquely belong to an object.*

Encapsulation denotes that attributes and operations uniquely belong to an object.

Encapsulation is a concept of which the value might not be immediately apparent. It's importance results from the elimination of the need to check that attributes are manipulated by an appropriate operation. The manipulation of an attribute is explicitly granted to the object that owns the attribute.

Principle 3.4.4 *Polymorphism: the word polymorphism comes from the Greek for "many forms" and denotes an object's capacity to have multiple forms.*

Polymorphism denotes an object's capacity to have multiple forms.

Cardelli and Wegner (1985) divide polymorphism into two categories, i.e. runtime and universal polymorphism. Runtime polymorphism can be further categorized into *coercion* and *overloading*. Universal polymorphism includes *parametric polymorphism* and *inclusion*. These four variants can be defined as follows:

- *coercion* represents an implicit parameter type conversion to the type expected by a method or an operator, thereby avoiding type errors. A good example is `2.0+2.0` versus `2.0+"2.0"`. Where two double values are added in the first examples, coercion converts the `2.0` double value of the second example into a string and the result will be the concatenated string `"2.02.0"`.
- *overloading* permits the use of the same operator or method name to denote multiple, distinct program meanings. An example of overloading is presented in figure 3.4. In this example we define two methods, both are

named `doWork()`. The method, and thus the implementation to be used, depends on whether a priority is given as argument to the invocation.

- *parametric polymorphism*, also referred to as *type parameterization* or the use of *generics*. Lisp and Simula 67 were the first languages to support parametric polymorphism (Eckel, 2004b). One may define *latent types*, or *latent classes*, in these languages, i.e. a type that is implied by how it is used, but that is never explicitly specified. That is, the latent class is implied by the methods that one may call on it. If one calls methods `f()` and `g()` on a latent class, then one implies that a class has methods `f()` and `g()`, even though that class is never actually defined anywhere (Eckel, 2004b,a).

- *inclusion* achieves polymorphic behavior via an inclusion relation between classes. The inclusion relation is a subtype relation in most object-oriented languages; inclusion is therefore often called *subtype polymorphism*.

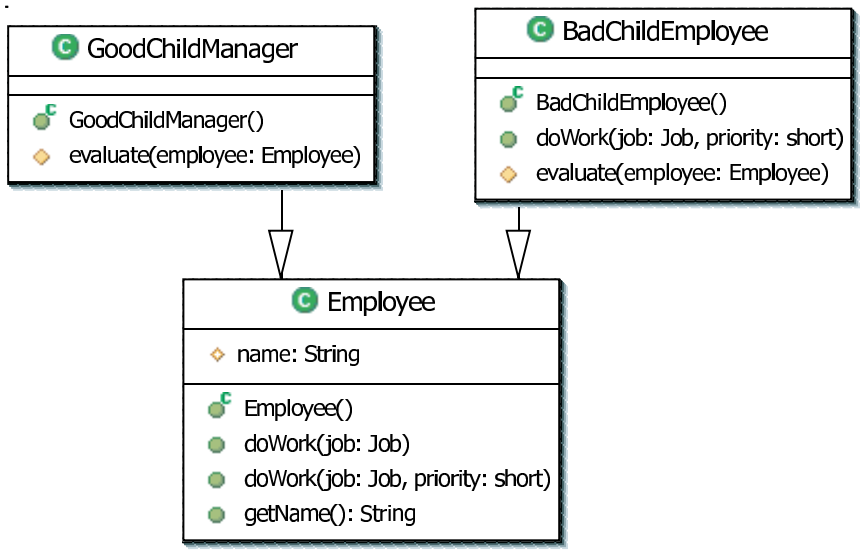

Fig. 3.5: Good child versus bad child

Inheritance denotes that classes can be organized in a hierarchical structure.

Principle 3.4.5 *Inheritance: classes can be organized in a hierarchical inheritance structure. In such a structure, the subclass inherits the protected and public attributes and operations from the superclass.*

The value of inheritance is far from trivial. Where some argue that inheritance should be the leading principle in systems design, others argue that its value is

overrated. As presented in section 3.3, inheritance embodies the specialization relation between classes. An *abstract class* is used to create only subclasses; therefore there may be no direct instances of such class. The principle of inheritance has the following properties:

- object instances of the descending class have values for all the attributes and relations of the ancestor class.
- all operations provided by the ancestor class must also be provided by the descendent class. *Visibility* and accessibility of operations may not be limited by subclasses.
- inheritance distinguishes *good children* versus *bad children* (Eckel, 2000). A good child is a descending class without polymorphism; all operations provided by the ancestor are used by the descendent class. A bad child supplies its own customized implementation for some of the operations provided by the ancestor class. The difference between a good child and a bad child is illustrated in figure 3.5: where the good child inherits the `doWork` methods of the `Employee`, the bad child overwrites its implementation
- inheritance is *antisymmetric*. If class A is a subclass of class B, then class B cannot be a subclass of class A. Inheritance is furthermore *transitive*. If class A is a subclass of class B and class B is a subclass of class C, class A is also a subclass of class C.

Principle 3.4.6 *Delegation: an object passes the invocation of an operation on to another object which actually fulfills the invoked operation.*

Delegation denotes that an object has passed the invocation of an operation on to another object.

Delegation is also referred to as the *perfect bureaucratic principle*; an invoked operation is delegated from an object to another object that has the attributes and operations to fulfill the required operation. Delegation is closely related to the *design by contract* principle (see principle 3.4.9). Delegation implies that it is the authority that is delegated; not the responsibility.
Since the early 1980s there has been much debate over which principle embodies a more powerful concept for implementing the specialization relation presented in section 3.3: the principle of inheritance or the principle of delegation. Stein, Lieberman and Unger came together in 1987 to discuss their differences and this resulted in a statement reflecting the need for both principles (Lieberman et al., 1988). This treatment became known as *The Orlando Treaty*.

Asynchronous communication is the principle of invoking an operation on another object where the requesting object does not expect an immediate result.

Principle 3.4.7 *Asynchronous communication: an object invokes an operation on another object where it does not expect an immediate result (Booch et al., 1999).*

This principle introduces the pattern of *subscription*. Instead of polling an object for a state change in which one is interested, an object provides operations for asynchronous subscriptions. Whenever the state change occurs, the object that subscribed is notified. The object interested in potential state changes is referred to as *Listener*, the object which accepts asynchronous subscriptions is called *Producer*.

Late binding is the support for the ability to determine the specific class, and thus the specific specification of an operation, at runtime.

Principle 3.4.8 *Late binding: support for the ability to determine the specific class, and thus the specific specification of an operation, at runtime.*

Although the concept of late binding increases the complexity of understanding object-oriented code, it is one of the most powerful methods used in the pursuit of loosely coupled systems. The concept is illustrated by the following example:

```
26 public class Worker
27 {
..
33   public void execute(final Worker worker)
34   {
35     worker.execute(worker);
36   }
..
37 }
```

At first sight, the above specification of the `execute` operation might seem dubious and destined to produce an infinite loop; this is not the case. Late binding allows us to invoke the `execute` operation with an instance of a subclass of `Worker`. Then this overridden implementation of the `execute` operation is invoked.
Current object-oriented programming languages such as Java even allow objects to automatically download unavailable class information of such subclasses at runtime. This is called *dynamic class downloading* (Sing, 2000).

Design by contract is the ability to design a set of operations as a contract.

Principle 3.4.9 *Design by contract: the ability to design a set of operations as a contract.*

Meyer (1992) originated a design principle called design by contract: in addition to specifying the signature of a method, the designer also specifies the pre-conditions, the post-conditions and the invariants, i.e. conditions that should be true of a class

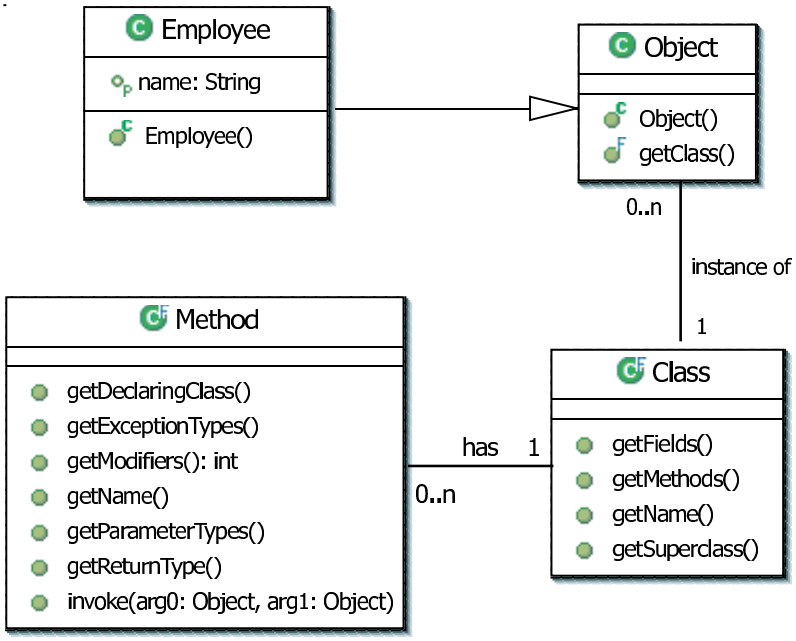

Fig. 3.6: Reflection

in general. The strength of this principle is that it gets the designer to think clearly about what service an method provides (Meyer, 1992).
A few programming languages, e.g. Eiffel, implement pre- and post-conditions in executable code so that they are checked at run time. Most programming languages do not have such support, so programmers who want to use pre- and post-conditions often write comments documenting the conditions. In these languages the principle of design by contract is embodied in the signature of a method, which is published in an *interface*. An interface describes the syntax of a service description; it specifies the signatures of the operations to be implemented by classes implementing the interface. If we recall the service oriented computing paradigm of section 1.6.2 we may conclude that an interface embodies the concept of a service description. Thus interfaces promote the conceptual strength for separating requirements from specification, and increase our ability to design loosely coupled systems.

Principle 3.4.10 *Reflection: an object knows the detailed information about the class(es) and interface(s) of which it is an instance.*

Reflection is the ability of an object to know detailed information about the class(es) and interface(s) of which it is an instance.

A consequence of reflection is that an object can, at runtime, aquire detailed information on its state and methods. The principle of reflection is presented in figure

3.6. The `Employee` class inherits a method `getClass()` which returns the class of which the object is an instance, i.e. `Employee.class`. This `Employee.class` contains methods to resolve methods and fields.

3.5 Summary

Systems engineering was presented in this chapter, where we presented a system as a part of the world we choose to regard as a whole, separated from the rest during a period of consideration, which contains a collection of objects, each characterized by a selected set of attributes, operations and relations.
The de-facto language to describe systems is the object-oriented description. Although object-orientation only specifies two orthogonal types of relations, object-oriented system design is considered to be complex. In this chapter we presented a set of design principles which will form the basis for the design presented in the remainder of this thesis.