Software Architecture and
Object-Oriented Modeling, and UML

Henrik Bærbak Christensen
Version 1.6
Status: Draft
Nov 2009

Change History:
Version 1.5: Introduced discussion of model versus responsibility perspectives. Introduced more discussion of the role concept; introduced initial description of protocol. Introduced definitions of 'view' by Buschmann and Bass.
Contents

1 Software Architecture 4
  1.1 Views ........................................ 5

2 The object-oriented paradigm 6
  2.1 Definition .................................... 7

3 Modeling the real world 8
  3.1 Phenomena and concepts ...................... 9
  3.2 Properties .................................... 10
  3.3 Relations ..................................... 11
    3.3.1 Classification ............................ 11
    3.3.2 Generalisation ........................... 12
    3.3.3 Association ............................... 12
    3.3.4 Aggregation ............................... 12

4 Roles, Responsibility and Behaviour 13
  4.1 Behaviour .................................... 14
  4.2 Responsibility ............................... 15
  4.3 Role .......................................... 17
  4.4 Roles at the design level ..................... 18
  4.5 Assigning responsibilities ................... 21
    4.5.1 Design patterns .......................... 21
    4.5.2 Keep the number of responsibilities low .. 21
    4.5.3 Do not spread responsibilities .......... 22
    4.5.4 Contractual responsibilities and sub responsibilities . 22
    4.5.5 Objects from responsibilities? .......... 23

5 UML: Documenting Architecture 23
  5.1 Static View ................................... 24
    5.1.1 A simple class diagram .................. 25
    5.1.2 Perspectives .............................. 26
CONTENTS

5.1.3 Class diagram summary ........................................ 27
5.1.4 Package Diagram .............................................. 28
5.2 Dynamic View ...................................................... 29
  5.2.1 Synchronous and asynchronous calls ....................... 30
  5.2.2 Discussion ..................................................... 31
In this chapter we will discuss software architecture with a focus on its relationship with the object-oriented paradigm.

1 Software Architecture

What is software architecture? To many software professionals it is something quite vague like “design at a more abstract level” or “the overall structure of the system.” These descriptions, however, do not provide us with much information, and the latter one is easy to dismiss by asking “what structure are you talking about?”

We here use the definition that has been put forward by Bass, Clements, and Kazman in the book “Software Architecture in Practice, 2nd edition” [1].

**Software architecture** The software architecture of a computing system is the structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

There are several implications in the above definition.

First, note that it talks about “elements” and “relationships” but not about what the actual semantics of these are. This is enforced by the second point we can observe, namely that the definition talks about “structures” in plural and not simply about “structure” in singular. This is because any software system can be looked upon from many different perspectives. We can look at the static structure—what classes, methods, modules, functions are there, and how are they related? We can look at the dynamic structure—what objects exist at run-time, what method invocations are on the stack, how do they interact, how is the flow of control and data? In the rest of this chapter, we will use the term **view** instead of structure. Though there is an academic difference between the two, we will not consider it here, and the term view is a more common term in the software architecture literature.

These two points show that an architecture/system does not only have a single view but many different views; and that the “elements” and “relationships” to consider depend on exactly what kind of view you are interested in understanding, describing, or documenting. For instance, the static view can be described by elements like “class” and “interface” and relationships like “association” and “generalisation.” The dynamic view can be described by elements like “object” and relationships like “invokes method on”, “creates”, and “depends upon”.

The third point is the statement of “externally visible properties.” Software
architecture is an abstraction of system. Elements are considered “black boxes”—we are not interested in “what goes on inside” as long as we know its externally visible properties. Abstraction is a hallmark of software engineering as in many other disciplines. Many important contributions indeed deals with abstraction: the invention of the procedure/subroutine allowed a complex algorithm to be encapsulated by a single method call; local variables and scoping allowed us to abstract global aspects away and reason locally instead, etc.

What constitute important externally visible properties depends on what aspect of the architecture we are interested in and what view we are studying. A well-known example is the interface provided by a class. The interface tells which functionality the class provides, the expected parameters, method names, etc., while encapsulating the actual algorithms and business logic used. However, a class and its methods exhibits many other important external properties that may be important to know in order to correctly reason about the architecture’s abilities. If high performance is important we may need to state the time complexity as an external property of certain central methods; if security is important we must assess the potential security aspects of the class; if we are writing embedded software for a platform with limited RAM we must consider the memory footprint of the class, and so on.

A final point is whether this definition is operational? What does it tell about designing architectures and building software?

We find that the definition provides a checklist to consider when building software systems:

1. What views are relevant to design and document? Static, dynamic, deployment?
2. Document elements and relationships for each relevant view.
3. Be sure to consider and document externally visible properties.

1.1 Views

Buschmann has defined the term view:

**View (Buschmann)**

A view represents a partial aspect of a software architecture that shows specific properties of a software system.

Views are well known from building architecture. Architects and building
engineers draws many different blueprints of a building. Some diagrams detail the plumbing, others the view from the outside, 3D models, cardboard models, electric wiring, etc. They all show “particular aspects” of the same building. Software also have many different aspects that we want to document.

Another definition of view is given by Bass et al. [1]:

**View (Bass)** A view is a representation of a coherent set of architectural elements.

In our context, we will primarily be interested in two views:

- **Static View** (often called “module view”) describes the design time view of our architecture. That is, what is it you consider at design time, what is it you see in your editor when coding. Elements are compilation units: classes, modules, packages; and relationships are dependencies, generalization, aggregation, etc. We will use UML class diagrams and package diagrams to describe this view.

- **Dynamic View** (often called “component and connector view”) describes the execution time view—“what happens at run-time?” Elements are functional components, objects, etc.; and relationships are control and data flows. We will use UML sequence diagrams and object diagrams to illustrate this view.

Another view that must also be considered for distributed computing is:

- **Deployment View** (often called “allocation view”) describes deployment aspects: what software is running on which machines, and how are they connected. Elements are computing nodes (computers) and relationships are connections (network, protocols). UML deployment diagrams are a candidate notation here.

## 2 The object-oriented paradigm

Humans have been writing programs for computers for a long time. During this time different ideas of “good” ways of writing programs have emerged. These paradigms define certain ways or perspectives on how to structure and organize programs and how to regard the resulting computations when the programs are run. A program to solve a given problem can be structured in many different ways and still solve the problem correctly.
However, comparing these programs will reveal that some have superior properties compared to others.

For instance, the functional paradigm views programs as sets of functions that transform data by taking data as input, perform calculations upon the data, and return data as output. This paradigm has the advantage that it is well understood from a mathematical point of view. Thus, programs written using this paradigm for structuring it will have properties that make it easier to prove correctness of the program by mathematical means.

Another important perspective is the procedural paradigm that views a computation as a set of instructions that alter data i.e. as a large programmable calculator. This paradigm has the property that it quite closely resemble the architecture of the underlying hardware.

The fundamental perspective in this note is the object-oriented paradigm which over the last couple of decades has shown its strength in structuring large scale programs.

It should be noted in passing that even the object-oriented paradigm has been interpreted in various ways. Christensen has identified three different perspectives: The language centric, the model centric and the responsibility centric perspective [3]. Our discussion will start by a model centric perspective (section 2 and 3) but shift into a responsibility centric perspective (section 4) that can be viewed as an extension of the former.

2.1 Definition

In the scandinavian tradition, object-orientation is summed up as

Object Orientation A program execution is viewed as a physical model simulating the behaviour of either a real or imaginary part of the world.

This simple statement has many implications.

First, the OO paradigm takes its starting point, not in mathematics or in computer hardware, but in a physical model … of part of the world. Thus from having computation as something disjoint and remote from the everyday life that we humans are involved in, computation is brought much closer to human reality: making physical models. This means that instead of programming being a process that tediously transforms human problems and endeavours into mathematical or machine level terms, it becomes the easier process of describing computation in natural terms. (Note that we wrote “easier” and not “easy”—there are still at lot of challenges in writing sound
object-oriented programs.)

Second, the terms *simulating* and *model* are central to the statement. Simulations are dynamic and unfold in time. Simulations exhibit behaviour that is visible external to the simulation—if not then the simulation does not have any purpose. It is of course the behaviour that is the primary purpose of the program—to add value for the user.

Simulation also means that our computation is *not* the real world but a mirror of it. This mirroring is not a perfect match of the part of the world that we are simulating—simulation means simplifying and leaving irrelevant details out of the simulation. This is also evident from the statement *part of the world*—not all the world. This means that we must decide on which properties to put into the simulation and which to leave out. Simulating every aspect of the world is ridiculous (except as theme for science fiction dramas) because of the overwhelming complexity of the physical world. Every simulation has a purpose and it is this stated purpose that guides us when we decide on the parts to put into the simulation, and what parts to leave out. It also guides us in the process of simplification; deciding on what parts can be simplified and in what respect, compared to the real world counterpart.

As an example of a physical model that simulates a real part of the world, consider a model plane. A model plane has many properties that it shares with a real plane for two reasons: to look like an airplane and to behave like an airplane due to the physical properties of the shaped wings and rudders. A model plane, however, has many simplifications compared to a real plane. It has no flaps, no room for cargo, etc. These properties of real planes have been ignored in our model because they do not really add value to the purpose of the model: namely to have the fun of steering a model airplane.

### 3 Modeling the real world

In order to simulate a part of the world, we need a way to describe the world as a model. One of the great benefits of OO is that OO programming languages allow us to describe a part of the world in terms that are close to the terms we use in everyday language. In this section we will look at how humans organize our knowledge of the world, and how we transfer this understanding into structures that can be expressed in an object oriented programming language. This transferal process is accordingly called *modeling*.

We are must be careful not to confuse the things that exists in the real part of the world and in the model. When modeling we will denote the part
of the world that serves as template for our model the **domain** (also called the **referent system**) and the model itself for the **model system** (or simply the **model**).

### 3.1 Phenomena and concepts

Our physical world is inhibited by myriads of **phenomena**.

**Phenomenon**

A phenomenon is anything recognizable or evidenced by senses. A phenomenon has individual existence in reality or in the mind.

For instance, at my home there is a beautiful but noisy phenomena that I recognize as my daughter “Mathilde.” Phenomena have individual existence and thereby individual **identity**. Even though a pair of twins look the same, wear identical clothes, have identical height and weight, they are not identical.

Understanding the world and communicating about it purely in terms of phenomena is infeasible due to the enormous amount of distinct phenomena in the world, therefore language has evolved the concept of a **concept**.

**Concept**

A concept is an abstract, generalized, idea of a collection of phenomena with similar properties and characteristics.

Concepts allow us to collectively refer to a large group of phenomena in a simple way; and characterize phenomenon quickly based on a common knowledge of properties shared by all instances of the concept. For instance, “child” is a concept and Mathilde is a phenomenon that is covered by the concept of a child. Or, less tricky for the tongue, “Mathilde is a child”, which is a simple statement that conveys a lot of information to any listener about the behaviour and physical properties of Mathilde.

A concept is characterized by three properties:

1. **Designation** The name(s) by which the concept is known.
2. **Extension** The collection of all phenomena that is covered by the concept.
3. **Intension** A collection of properties and characteristics that characterizes the phenomena in the extension.

For instance, the concept “child” has the designations “child” and “kid”,

---

---
the extension is the rather large group of all children in the world, and the intention is list of characteristics of children: they are human beings, that their behaviour is sometimes rather unpredictable, and their age is less than, say, 13 years. The last vagueness about age is typical for many everyday concepts. The concepts are usable and understandable even though it is difficult or even impossible to unambiguously define the intention of the concept.

Human understanding of the world and communication about it is structured in terms of concepts and phenomena. For instance, when I ask my wife “Has Mathilde had a good time in kindergarten?” I refer to both phenomena and concepts. (Incidently, it also shows how humans are adept at handling ambiguities; the statement refers to “kindergarten” which is a concept but my wife will interpret it as a phenomenon, namely the specific kindergarten that Mathilde attends. Object-oriented programs must be precise about whether a phenomenon or concept is refered.)

Object-oriented languages have developed language constructs that mirror phenomena and concepts: the object and the class.

Objects are computational equivalents to phenomena: they have individual existence and an object identity. Even though two objects may have the exact same state, they are still two distinct objects.

Classes are equivalent to concepts: they are abstract ideas that express commonality between phenomena. Often we want to emphasise that an object belongs to certain class, in that case we will use the term instance instead of object.

Phenomena have individual existence while concepts are abstract ideas. This also closely matches OO languages where we do the programming at the abstract level in terms of classes while objects exist at runtime during program execution.

3.2 Properties

Phenomena have individual properties. For instance, Mathilde has a certain mass. Properties are usually not directly observable, but requires some kind of measurement. A measurement results in a value. For instance, Mathilde has a mass but her mass can only be assessed by putting her on a weight, only then we will have her weight expressed in kilograms.

In a similar vein, objects have individual state. The space of possible states an object can have is defined by the object’s class. The class defines the statespace by a collection of attributes, each with a name and type.

An object’s state can only be determined by measuring it. Most OO lan-
3 MODELING THE REAL WORLD

Languages allow measurements by directly reading the value of individual attributes. However, time has shown that accessing the state of an object through method calls has many nice properties. These measurement methods are called **accessor methods** or often “get” methods.

### 3.3 Relations

The world is not a chaotic mess of individual and unrelated phenomena and concepts. Instead the world and human society are highly organized as concepts and thus phenomena are related in many different ways. For example, I am the father of Mathilde. Thus, there is a father relation between me and Mathilde. Generally, humans have children and we speak of this relation between the concept of parents and children as a parent-child relationship.

Relationships are important to structure and organize our knowledge about the world. Humans use a number of strategies for organizing knowledge about concepts and phenomena and specific kinds of relationships emerge based on which strategy is applied. Below we describe the archetypical strategies.

#### 3.3.1 Classification

**Classification** is the process of grouping a collection of phenomena with similar properties.

For instance Mathilde is classified as a child.

The opposite process is that of **examplification**: Mathilde is an example of the concept child.

Thus classification/examplification defines a relationship between a phenomenon and a concept.

In OO languages, this corresponds to the relationship between an object and a class. We denote this relation the **instance-of** relation, so taking the above real-world example to the world of OO programming we would say that ‘Mathilde is an instance-of child’.
3.3.2 Generalisation

Generalisation is the process of forming a concept that covers a number of more special concepts by focusing on similarities and ignoring peculiarities.

For instance, the concept of a mammal is a generalisation of concepts such as human beings, monkeys, etc.

The opposite process is that of specialization, i.e. form a more special concept from a general one. The concept of a monkey is a specialization of the concept of a mammal.

Thus generalisation/specialization defines a relationships between concepts. It is a hierarchical relation.

One of the main things that make OO languages different from non-OO languages is that generalisation is supported at the language level. Often, generalization is also called the is-a relation.

3.3.3 Association

Association is the process of defining a phenomenon in terms of what other phenomena it has relationships with.

For instance, a reservation on a flight is defined in terms of the identity of a person, a specific seat at a specific flight, departure- and arrival time, and possibly references to other information.

Association is a “weak” relationship. A flight reservation does not “own” or “contain” the person it is issued to, it merely refers to it. Thus the involved phenomena are peers and the relation is non-hierarchical. This is in contrast to the stronger aggregation relation described next.

In OO languages association is usually expressed by object references or collections of object references, depending on multiplicity.

3.3.4 Aggregation

Aggregation is the process of defining a phenomenon in terms of what other phenomena it is made of or consists of.

For instance a human being consists of a torso, a head, two legs, two arms,
and so forth. We say that a human being is an aggregation of torso, head, legs, etc.

The aggregation relation is a relation between phenomena.

The aggregation relation is often also denoted the whole-part relation. The human is the “whole” and head, torso, etc., form the “parts”.

Aggregation is usually implemented in the same manner as association: by object references. However, Java and some older research languages like BETA allow aggregation to be defined at the language level by allowing scoped class definitions. That is, classes may be defined within classes. The Java inner class concept is an example. However, in most cases you would use references between classes even to express aggregation. Thus, often you will see no difference at the code level between association and aggregation even though they conceptually are quite different.

4 Roles, Responsibility and Behaviour

The previous section dealt with phenomena and concepts / objects and classes, and their relations. This is a classic understanding and also links very well to the definition of software architecture—an architecture consists of elements and their relations. In object-orientation, objects and their relations are the constituents, the parts, of a model or simulation. However, what makes a simulation interesting and relevant is the behaviour that these parts have, individually and jointly.

We can also state this more pragmatically: The customers of our software are quite indifferent about classes and relations; they care about the software’s functionality—what does it do to make my work easier, more fun, more efficient? Thus, software behaviour is the most important aspect, as it is really what pays the bills!

At present, there is a good deal of research going on that seems to indicate that the two fundamental concepts; object and relation; have to be supplemented by a third equally fundamental concept: role [6]. The role concept is less accepted as fundamental which you can easily convince yourself of by reading most books covering object oriented programming, analysis and design.

The role concept helps us to break the rigid ties between object and functionality. However, we will approach the role concept gradually by studying another definition of object orientation with a stronger emphasis on functionality.

Timothy Budd [2] has given another description of object orientation:
Object Orientation

An object-oriented program is structured as a community of interacting agents called objects. Each object has a role to play. Each object provides a service or performs an action that is used by other members of the community.

This statement emphasises that the overall behaviour and functionality of a program execution is defined by lots of individual objects working together.

This view is a fruitful one and again is an example of how OO has taken the real world of human endeavour as a model for how to view computation. It resembles how much of human society gets work done: by organizing a lot of individuals and defining responsibilities among them and ways of collaboration for these individuals. The overall behaviour of an organization or company is the sum of many individual but concerted tasks being accomplished.

Let us illustrate this by a simple example from real life of how collective behaviour results in a desired outcome. Let us say I want to send flowers to a good friend who lives in a town far away. Bringing the flowers to my friend myself is too costly and tedious. Instead I walk to my local florist and ask her to make sure that flowers are brought to my friend. I need to specify some further details such as the address of my friend and the variety and quantity of flowers, but besides that I can rest assured that my friend will receive the flowers. The actual process of making sure that flowers are brought to my friend, involves many other persons such as a florist in the home town of my friend and a delivery person. What makes the delivery process possible is that the process is detailed and known and each person knows his or her responsibility and how to perform it.

4.1 Behaviour

Abstractly, behaviour may be defined as:

**Behaviour**

Acting in a particular and observable way.

In OO languages behaviour is defined by objects having methods that are executed at runtime. Methods are templates for behaviour, algorithms, in the sense that parameters and the state of the object itself and associated objects influence the particular and observable acting.

Collective behaviour arises when objects interact by message passing. A

---

1This example is again given by Timothy Budd
message passing occurs when one object requests the behaviour of one of its associated objects. We also call this a **method invocation**.

Method names and parameter lists, however, convey little information about the actual behaviour. Often an object’s behaviour is the result of a concerted effort from a number of collaborating objects. For instance, when a flight reservation object is requested to print out, it will request the associated person object to return the person’s name. UML sequence diagrams are well suited for describing object interaction.

Behaviour is the ‘nuts-and-bolts’ of a program execution in the sense that what actually gets done at runtime is the sum of the behaviour defined by the methods that have been invoked. However, time has shown that the concept of behaviour is usually too low-level to be really useful when designing systems—we need a more abstract concept.

### 4.2 Responsibility

**Responsibility**  The state of being accountable and dependable to answer a request.

The florist example shows the difference between behaviour and responsibility. The florist is accountable for satisfying my request: to have flowers sent to my friend. How she will satisfy my request I do not know, and usually, I do not want to know; it is her responsibility to have some method to use to ensure that my request gets carried out. Actually she may use different methods for a number of reasons that I do not know; each different method will result in different behaviour on her part.

Responsibility is a more abstract way of organizing and describing object behaviour at the design level. As an example, we could define responsibilities of a florist in a high-level manner like:

**Florist**

*Responsibilities:*

- To arrange flowers in a (beautiful) bouquet.
- To ensure delivery of a bouquet to a certain address.

Responsibility is often best communicated in a design/implementation team in these short and broad statements. This technique has been elaborated in the **CRC Card** technique, where classes are described at the design level using three properties: **Class name**, **Responsibilities**, and **Collaborators**. If
the above description of a florist was elaborated with collaborating objects, like customer and delivery person, it would constitute a CRC card for the Florist class.

Behaviour is defined by methods on objects at the programming language level—an object behaves in a certain way when a method is called upon it. What about responsibilities?

The language construct that comes closest is the interface. An interface is a description of a set of (related) method signatures but no method body (i.e. implementation) is allowed. One may envision encoding the Florist in a Java interface like:

```java
public interface Florist {
    /** arrange a bouquet at the specified cost */
    public Bouquet arrange(Money cost);
    /** ensure that a bouquet is delivered to the specified address */
    public void deliver(Address address, Money cost);
}
```

Thus, an interface is much more specific than the above high level description, however an interface only specifies an obligation of implementing objects to exhibit behaviour that conforms to the method signatures, not any specific algorithm to use. We say “conforms to the method signatures” but in practical programming an object that implements an particular method in an interface must also conform to the underlying contract—that “it does what it is supposed to do” according to the method documentation. For instance, the Florist interface’s method `arrange` can only specify that a bouquet is returned, but the underlying contract is that it should be a beautiful bouquet.

The reason we argue that the language construct of interface is better for expressing responsibility is that the developer is explicitly forced not to describe any algorithm—and thus keep focused on the contract instead of getting hooked on details too early. Objects are free to implement the methods in any way as long as the behaviour adheres to the specification.

In the florist example above, there is a one-to-one relation between stated responsibilities and methods in the interface. Generally this is not the case—a high level statement of a responsibility is often associated with a set of methods and a particular ordering of method calls in order for the collaboration to be correct (a protocol). Consider a responsibility of a database to store patient records; this would at least consist of methods to store and fetch patient records and probably also transaction handling, roll-back etc.
4.3 Role

Responsibilities are only interesting in the context of collaboration: If no-one ever wants to send flowers, there is no need for a “deliver bouquet” responsibility of the florist. Thus, what makes the responsibility interesting and viable is the fact that there are customers wanting this service.

In a program executing, we likewise have objects collaborating, each object having their specific responsibilities. To collaborate they must agree upon their mutual responsibilities, the way to collaborate, and the order in which the collaborate must proceed.

As the term responsibility is more abstract than behaviour, we also want a more abstract way of expressing mutual responsibilities and collaboration patterns than in terms of object message passing.

Role (General)  A function or part performed especially in a particular operation or process

This definition (from Merriam-Webster’s on-line dictionary) embody the dual requirement: both “function performed” (responsibilities) as well as “in a particular process” (collaboration pattern/protocol).

The role concept allows us to express a set of responsibilities and a specification of a collaboration pattern without tying it to a particular object or particular class. In the discussion of the florist above there is actually nothing that force us to view the Florist concept as a class—it is a role. It may be that I prefer my local florist but actually any florist would be able to satisfy my request to have flowers sent to my friend.

Roles are well known from our everyday. Shakespeare’s Hamlet is not a person, it is a role that many different actors have played over the years. The Hamlet role is an extreme example of a role where the ‘obligation to perform a specific task’ is detailed down to movements and speech. Another place where roles are important is the organization of companies and institutions in terms of roles with more or less well-defined associated responsibilities: project leader, software architect, coder, tester, etc. In smaller companies, often the same person must serve many different roles: sometimes he acts as the project leader and sometimes as a software developer; developers often also acts as testers of their own code. In a hospital nurses and physicians are again roles where the “players” are constantly changing.

As is evident from the discussion above the relation between role and player is a many-to-many relation. A single role can be played by many different players/objects while a single object can also play multiple roles.
Sometimes roles are invented to make an organisation work better. As a simple example a pre-school kindergarten had the problem that the teachers were constantly interrupted in their activities with the children to answer the phone, deliver messages from parents, fetch meals, etc. They responded by defining a new role, the flyer, whose responsibility it was to answer all phone calls, bring all the meals, etc. Thus all other teachers were relieved of these tasks and allowed to pay attention to the children without interruptions. The teachers then made a schedule taking turns on having the role as flyer.

Roles express mutual expectations. Students expect the lecturer to raise his voice and start presenting material. It will not work if he instead just sat down on a seat at the back row and kept silent. Likewise the lecturer expects the students to keep silent while lecturing. A Shakespeare play will not work if the actor playing Hamlet begins to do all sorts of things unrelated to the role. Thus roles relies on more or less well-defined protocols.

**Protocol**

A convention detailing the expected sequence of interactions or actions expected by a set of roles.

Remember that an old interpretation of protocol is indeed “diplomatic etiquette”, that is the accepted way to do things.

### 4.4 Roles at the design level

The role concept loosens the tie between a systems functionality and the objects that make up the system as we will see.

Another definition, more software oriented, could be:

**Role (Software)**

A set of responsibilities and associated protocol with associated roles

Rebecca-Wirfsbrock simply defines role as “set of responsibilities” [7, p. 3] but I find the protocol aspect important.

Roles do not have any direct counterpart in main stream programming languages. The programming language construct that comes closest is again the “interface” that supports aspects of the concept of a role. As a role can be viewed as a set of responsibilities along with a understanding of the collaboration patterns with partners an interface similarly defines a set of methods and some assumptions about the proper ordering and sequencing in which they can be invoked. However, main-stream languages have no way of forcing specific protocols, that is no way of enforcing that method
A on object X is invoked before method B on object Y, etc.
However, if we use interfaces to express roles then we see that the same many-to-many relation exists between objects and interfaces as between people and roles.

The one-to-many relation between interface and objects manifests itself in that many different objects may implement a particular interface. To take a good example from the Java library then the only thing the sorting algorithm in the Collections class expects is that all objects in a given list implements the java.util.Comparable interface. Thus if we make a class representing apples then we can sort apple objects simply by implementing this interface:

```java
public class Apple implements Comparable {
    private int size;
    // [other Apple implementation]
    public int compareTo(Object o) {
        // [apple comparison algorithm]
    }
}
```

Thus, in the context of the sorting algorithm it is irrelevant what the objects are (here apples), the only interesting aspect is that the objects can play the Comparable role. And - the comparable role simply specifies that objects must have the responsibility to tell whether it is greater than, equal to, or less than some given object; and the protocol is quite simple: it must return this value upon request.

In another application the sorting algorithm is reused as Carrot objects can also implement the Comparable interface.

The many-to-one relation between interface and object is possible in Java and C# as these languages support multiple interface inheritance. That is, a class may implement multiple interfaces. To rephrase this, we can assign several roles to a single object.

As an example from an implementation of a Backgammon game there are a set of roles defined:

- **Game.** This role/interface is a facade pattern encapsulating the backgammon domain from the graphical user interface.
- **Board.** This role/interface models the physical board and is viewed as a collection class containing checkers that can be moved around.
- **GameAdapter.** This role/interface defines an adapter pattern that encapsulates inspecting and movement on the backgammon domain.
from a game AI (artificial intelligence). This way the AI becomes independent of what ever representation and interface one has decided for the game.

All these interfaces express well-defined roles in a full backgammon domain. From the AI’s point of view all that is needed is some object fulfilling the GameAdapter role in order for it to do its job. The GameAdapter defines some responsibilities to allow inspection and performing moves; and the protocol states that the game must be inspected before any movement is made and that the movement should be valid according to the inspected game state.

Game AI’s must consider a lot of moves and evaluate what is the best move. This means that the underlying implementation must be efficient. A radical, high performance, proposal is simply to make a class that implements all these roles.

```java
public class GerryGame implements Game, Board, GameAdapter {

By merging, object of this class can serve all the roles. This way the implementation of each role’s responsibilities (read: each interface’s methods) has direct access to instance variables. The implementation of Board and Game can be dictated by the GameAdapter method signatures as it is the latter that needs to be high performance. Note of course, that the design becomes less flexible! A GerryGame cannot change the board implementation without a lot of coding.

This way of making a single object play different roles is an excellent way of “glueing” systems together. An example is from the JHotDraw framework. JHotDraw is a 2D editor framework and thus obviously have to have a graphical user interface. However, the main abstractions of the framework are expressed without any reference to a specific 2D GUI toolkit. This makes it easy to port JHotDraw from Java AWT to Java Swing, for instance.

But, of course you must couple JHotDraw to the concrete GUI toolkit. JHotDraw solves this problem elegantly by defining classes that simply play roles in both the JHotDraw framework as well as the AWT framework. An example is the drawing area roles that is defined by the `DrawingView` interface. To couple it with AWT, JHotDraw defines the `StandardDrawingView` class:

```java
public class StandardDrawingView extends Panel implements DrawingView,
   MouseListener,
   MouseMotionListener,
   KeyListener {
```
This way the concrete AWT drawing area Panel and the JHotDraw role can communicate directly within the StandardDrawingView object.

4.5 Assigning responsibilities

If an object-oriented program can be defined as the collective behaviour of a (large) set of collaborating objects, then the question arise how to partition behaviour on the objects? I.e. what responsibility should each individual object have? While this may seem a simple question, finding a “good” answer is far from trivial. Finding an answer is at the core of what makes object-oriented programming intellectually challenging—and difficult.

Peter Deutsch, the author of the GhostScript system, has stated this as “Interface design and functional factoring constitutes the key intellectual content of software and are far more difficult to create or re-create than code.”

The software engineering community has worked a lot on this and several aspects of “the answer” is available. Below I will outline a number of techniques, observations, and “rules of thumb”. The list is by no means meant to be complete but consider it as a list of ideas.

4.5.1 Design patterns

Design patterns are solutions to problems in a context. There are different classes but one of these is behavioural patterns. The patterns specifically address recurring problems about how to partition responsibility such that certain qualities of the solution is met, typically a wish for reusability and flexibility.

While design patterns are often shown as class diagrams, patterns are not structures of objects; they are descriptions of how to divide the responsibility in the design.

Take the observer pattern as an example. Basically, the observer pattern dictate that there should be one object having the responsibility of the subject (information holder) and a set of objects having the role of observers (information viewers). The pattern then defines the interaction pattern between these two roles in order solve the problem of keeping multiple views synchronized with an underlying information.

4.5.2 Keep the number of responsibilities low

Do not let any object take on too many responsibilities! This tendency (an anti-pattern named “The Blob”) is often seen in aging software, where a
few objects begin to accumulate responsibilities. In one project, I did, I found one class with 65 methods and 16 identified responsibilities. The object-oriented design is slowing becoming a procedural design.

Blob classes are difficult to understand and maintain and becomes inflexible to change.

4.5.3 Do not spread responsibilities

A keen focus on responsibilities is the best way to avoid falling into the trap of assigning responsibilities as you go. If you do, some responsibilities tend to become thinly smeared out on a large set of objects leading to tricky bugs, code that is difficult to understand and maintain, and inflexible to change.

This problem often occurs in “classical” designs where the designers have focused only on the “elements in the real world”. Consider a chess program. The only real artifacts of the domain is chess pieces and a chess board which are quickly and easily translated into two classes. If the designers are not careful they easily wind up having some of the move validation code (verifying that a given chess move is legal) distributed in both classes. Next, this quickly leads to duplicated behaviour (both the chess piece and chess board class can check if a location is free), leading to inconsistent behaviour (the implementation in the piece is somewhat different from that in the board). And, when the costumer demands that the game can use alternative chess rules (more often seen in other domains than chess :) there is a bit problem because the program basically has to be rewritten.

The way out is to clearly identify the roles and clearly know which objects serve which roles.

4.5.4 Contractual responsibilities and sub responsibilities

Some responsibilities requires a lot of hard work. Use divide and conquer here also. Define subresponsibilities; just as a project leader delegates work to team members and coordinate and collect the result, let “manager objects” request objects to help out.

This leads to a concept that we may term contractual responsibility which is the entry point from the outside; and sub responsibilities which are the simpler tasks. The “deliver” responsibility of the florist is actually the contractual responsibility relying on subresponsibilities of another florist and a delivery person.
4.5.5 Objects from responsibilities?

Objects/classes does not necessarily come first in a design phase and are to be remained fixed when responsibilities are assigned. As an example, consider the flyer role that teachers simply invented and took turns to have. In a similar vein, it is often convenient to define particular roles in software and then make special objects that only fulfil this role. (This is what Larman calls “Pure Fabrication” [5].) Consider the chess game case again. Is it the game board object or the individual chess piece objects that are “the best” to handle validation of chess moves? Neither are obvious—the board is involved through the location and the chess pieces through their types that define movement patterns. So—maybe the right proposal is to make a special role, “Validator”, and make a special purpose object to implement this role. Concerning flexibility the case of changing the movement rules will then be simply a matter of changing the concrete Validator object fulfilling this role in the game.

5 UML: Documenting Architecture

UML is short for “Unified Modeling Language.” The reason for the “unified” in the term is that it is really three different persons that after a couple of years of “notational war” decided to combine their efforts instead in order to define a standard visual notation. The three persons (that became known as “The three amigos”) were:

- James Rumbaugh, who with some colleagues invented the Object Modeling Technique, OMT, and published it in 1991. Rumbaugh had a background in database and Entity Relation modelling whose influence on the UML class diagram is still visible.

- Ivar Jacobson, described the Object-Oriented Software Engineering technique,OOSE, in 1992. Jacobson’s major contribution is the use case.

- Grady Booch, who described the Booch notation in 1994. His work was in object-oriented concepts and his notation focused on structural aspects especially inheritance.

OMT and Booch notations were very focused on the static view and had almost no way of expressing on dynamics. Over the years UML has absorbed existing diagram types (like state charts) as well as invented diagrams to provide a richer notation to describe dynamics.
The list of diagram types in UML 2.0 includes (note that some diagram types have changed their names in 2.0; the UML 1.x name is given in parentheses):

- class diagram
- object diagram
- use cases
- sequence diagram
- communication diagram (1.x: collaboration diagram)
- package diagram
- state machine diagram (1.x: statecharts)
- activity diagram
- component diagram
- deployment diagram

We will not cover them all here but describe only the ones that are most important for our use.

The presentation here has been heavily influenced by Martin Fowler’s “UML Distilled” that is highly recommendable [4].

5.1 Static View

As stated in the previous chapters, the static view describes a software system / an architecture from the standpoint of compilation units: what are the modules, classes, methods and how are they depending on each other, what are their relationships? This view is thus linked with “what you see in your editor and browser,” and is important for overviewsing a large software system. I often think of it as kind of roadmap that tells me how to navigate in my software system. They key for understanding the road map is of course to choose names that express the behaviour and responsibilities assigned to each interface/class.

The static view can be described by a UML class diagram. As evident from the name, it describes classes in an architecture. Thus elements are classes and interfaces, and relations are those described in Section 3.3: association, aggregation, generalization, etc.
5.1.1 A simple class diagram

As an first example the design of our simple parking machine is shown in Figure 1.

![UML class diagram for a parking machine.](image)

Figure 1: UML class diagram for a parking machine.

The gray notation is not part of UML but included to show each UML notational concept.

**Classes** are shown by boxes and relations by lines between boxes. An full arrowhead describes **generalization** (“inherits”, between a superclass and a subclass) and **realization** (“implements”, between an interface and an implementing class). The line in the former case is a full line while it is dashed for the latter. An **association** is simply a line between two class boxes.

**Interfaces** are not really something special in UML, it is a class box with a **keyword** inside. The keyword “≪interface≫” shows that the class is really an interface. You may add keywords to any class box to indicate special semantics of the class box. For instance, if you use a class only as an enumeration of objects you can indicate this by a “≪enumeration≫” keyword. Another often used keyword is “≪component≫” to indicate that binary components for component based development.

It is important to state the **multiplicity** of relations. Multiplicity is shown by numbers, and a * means “zero or many”. Thus you can read the diagram like “A parking machine is associated zero to many receipts”. And it can of course be read in the opposite direction as well “A receipt is associated a single parking machine.” Allowed values for multiplicity is N..M (lower and upper bound on multiple values). No upper bound is shown by *
(many), and as the range 0..* is used so often it is shown simply as *.
The range 1..1 is also shortened to simply 1. The 1 multiplicity often occurs so
many times that it is better simply not to show it. So an association without
multiplicity usually indicate 1.

An association can also show a role name which describes the role the ab-
straction next to it serves. So—a better description is “A parking machine
produces zero to many receipts.”

Associations may include navigability, shown by the arrowhead. It sim-
ply augments the description in the respect that for instance ParkingMa-
chineGUI knows ParkingMachine but the opposite is not the case. This is
quite near the code level, where we can guess that the gui has an object
reference to the parking machine but the opposite is not the case.

Finally, an important relation is aggregation. Aggregation is shown by a
diamond at the place of the whole part like in figure 2. Thus this diagram
indicate that ParkingMachine objects must contain a PricingStrategy. You
do not write multiplicity at the diamond end at this is always 1.

![Figure 2: Notation for aggregation](image)

5.1.2 Perspectives

Martin Fowler distinguish between two perspectives for using class dia-
grams.

With the conceptual perspective, the UML represents a description of the
concepts and relationships present in the domain itself. That is, you de-
scribe what exists without reference to any IT system. To go back to the
parking machine domain, a conceptual perspective model may be the one
shown in Figure 3.

The notation is exactly the same but the interpretation is somewhat differ-
et. Now boxes are concepts from the problem domain and the relations-
ships are the relations these concepts have.

Conceptual diagrams are very good for requirement analysis and early de-
sign as they are a way to capture how customers perceive the world. In
the early days of object-orientation the conceptual diagram was also trans-
ferred directly and untranslated into the software architecture—i.e. every
concept was translated into a class in the program. Time has shown, however, to be very careful with a direct translation. Thus, use the conceptual perspective as inspiration only. As you can see, the parking machine design has taken some of the concepts directly (parking machine, receipt), and simply avoided others (payment, customer).

In the software perspective the elements of UML map more or less directly to the elements of the software system—a class box means a class, etc. The software perspective is clearly the perspective in figure 1.

5.1.3 Class diagram summary

Class diagrams have many benefits. A major benefit is that their focus on overall structure quickly generate questions in a team. For instance, why is there a 1-1 relation between customer and receipt in figure 3? Is it because we constrain the system so that a customer can never buy another parking receipt? Is it because we only consider one session? Or is it simply wrong? These questions arise quickly when discussing the diagram. Consider how much more difficult it would be to generate such discussions if we were to look at the code instead—important decisions are simply much less visible in code.

For this reason, one should be very careful about using the UML class diagram to the detail level of showing attributes and operations. Putting too much (code) detail into the diagram draws the attention away from more abstract discussions and clutters the diagram. I prefer only to express interfaces and classes as in the parking machine diagram, and am more interested in showing responsibilities than in showing actual methods. For complex

\footnote{Formerly Fowler made a further classification of the software perspective into a specification perspective and a code perspective. The former was more abstract looking more at the contract/interface level while the latter as very code near.}
designs that uses a lot of patterns, like frameworks, I even remove classes, and only show the interfaces and their relationships. Again, the purpose is to convey the overall picture rather than be annoyed with too much detail.

Talking perspectives, I find that adding information about attributes and operations is a code perspective, that is too close to the code. The first reason for this is that code often evolves fast. This means that methods change names and thus a code perspective UML class diagram quickly becomes out of date; much quicker than does the more abstract one that only state interfaces/classes and relationships. The second reason is that I like class diagrams to be a road map of the code for understanding it, reason about it, and for providing a guide to where I have to make modifications. For this purpose I do not need the detailed level of attributes and operations.

5.1.4 Package Diagram

Once your design grows enough classes and interfaces you begin to lose the overview. Most modern languages have language constructs that allows sets of compilation units to be grouped. In Java we have the package structure and in C# we have namespaces.

UML also have a grouping structure named packages that are described by package diagrams. The package diagram elements are packages and the relationships stated are dependencies.

![Figure 4: UML package diagram](image)

Figure 4 shows an example. You can write the package name both on the tab as well as in the middle of the box, here only the former is used. You can show hierarchy by putting classes as well as other packages inside the package box (left package) or by using a :: notation as done with the java.util package (right package).

Packages are abstractions over the interfaces they encapsulate. Thus you should only show interfaces and classes that are interesting and relevant from an outside perspective. In the parking machine case the central ab-
stractions that client classes need are ParkingMachine and Receipt; therefore only these are shown. Typically complex packages are accessed through Facade patterns and these are of course obvious to show in the package diagram.

Between the packages you shown dependency. A dependency between two package simply state that some element in one package has some dependency with an element from the other package. Dependency in UML is simply the supertype of all types of relations. Thus you cannot from the diagram see if it is a association, generalization, aggregation or any other type of relation. Again, the goal is overview and the means is abstraction.

5.2 Dynamic View

The dynamic view of an architecture is concerned with what exists at runtime, that is elements like objects and method activations and relationships like control and data flows between objects.

The UML sequence diagram is well suited to describe concrete collaboration patterns between objects. Figure 5 shows an annotated sequence diagram outlining the parking machine’s buy operation (the C# version of the parking machine code).

![Figure 5: UML sequence diagram for the buy transaction.](image)

The sequence diagram is visually organized to display time. At the top you show a set of objects (this is actually only true for UML 1.x, in UML 2.0 the boxes express something more complex but this is beyond the scope of this note). The UML notation for objects is also a box but the name in the box is
underlined and follows a syntax

name : type

name is the name of the object and optionally you may identify its class. If you do not want to name the object you may only indicate the type but you must remember to include the prefixing colon then, like “:ParkingMachineImpl”.

The objects have a timeline, a dashed line going vertically down from the object box. The timeline is a time axis with later points in time being further down.

A sequence diagram essentially describes a single scenario of interaction between a set of objects. The initial action that starts the scenario is shown by a found message, here it is an event from the .NET platform that invokes the method “OkClickEvent” in the “pm” object. That a method is executing is shown by an activation which is a box on the timeline. The box starts when the method is called and ends when the method returns. (To put it in more technical terms the activation box shows the amount of time the method’s stack frame is on the stack.)

Sequence diagrams show message invocation by arrows between the timelines including the name of the method called. Thus you can see that object “pm” invokes method “pushBuyButton” in the “parkingMachine” object. Synchronous method calls (the standard object-oriented call semantics) are shown with a full arrowhead.

When a method returns you may show it by a return arrow; it is dashed and with a stick arrow head. The return arrow may optionally show the attribute that gets assigned with the return value, as is the case with the “receipt” variable. Often return arrows simply clutters the diagram and often they are not drawn at all.

Finally, a instantiation of an object is shown by a create message, and the object created is placed aligned with the create message instead of at the top of the diagram.

5.2.1 Synchronous and asynchronous calls

The messages shown in figure 5 are all synchronous, that is the calling object is blocked until the method call has ended. In many settings you use asynchronous calls instead where the caller may continue processing without waiting for a response. In UML 2.0 you use stick arrowhead for asynchronous calls.

Unfortunately UML 2.0 revised the notation to use and have introduced
incompatible notation with UML 1.x. Figure 6 shows the two call types for 2.0 (above) and 1.x (below). Note the very unfortunate change where the stick arrowhead in 2.0 has changed semantics all together. Be sure to include in your diagrams which version of UML you are using.

Figure 6: UML 2.0 (above) and UML 1.x (below)

5.2.2 Discussion

Sequence diagrams are very important. One of the key things in understanding object-oriented programs is understanding behaviour. In contrast to the algorithms of procedural programming that are quite easy to follow just by looking in the source code, object-oriented programs define behaviour by having many objects collaborate. Thus, the behaviour is much less visible in the code itself and you often find yourself browsing numerous classes and following long chains of objects calling each other. These long chains of method calls between many objects makes OO programs much harder to understand.

The sequence diagrams tell the story: what methods are called in which objects and in which sequence?

The back side of sequence diagrams are their concreteness—they show one possible execution. That is, they show what happened when the first
branch of an if-statement was taken only.

The interaction frames introduce a notation that allows us to express algorithms in UML but my opinion is that they are a poor choice. If you rewrite the code below into a sequence diagram with interaction frames, you will get a much more complex diagram than the code (the code below corresponds to Figure 4.4 in his book which is almost one page):

```plaintext
foreach (lineitem)
  if (product.value > 10000)
    careful.dispatch
  else
    regular.dispatch
  end if
end for
if (needsConfirmation) messenger.confirm
```

Thus the sequence diagram becomes a cumbersome way of expressing algorithms instead of a good tool to overview long chains of interaction. I generally prefer writing more sequence diagrams if there are several interesting scenarios (like one for the normal behaviour, and one for a failure behaviour).

## Acknowledgements

Thanks to Søren Boll Overgaard for comments and spellchecking early versions of the object-oriented description of this report.
References


