ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Second Faculty of Engineering
Curriculum in Electronics, Informatics, and Telecommunications Engineering

REUSE MECHANISMS AND CONCURRENCY: FROM ACTORS TO AGENT-ORIENTED PROGRAMMING

Candidate:
ROBERTO CASADEI

Supervisor:
Prof. ALESSANDRO RICCI

Co-supervisor:
Prof. ANTONIO NATALI

ACADEMIC YEAR 2011-2012
SESSION III

KEYWORDS

Software Reuse

Objects

Concurrency

Actors

Agents

Contents

Introduction **ix**

1 Software Reuse: An Introduction **1**
1.1 Software reuse: the what, the why, the how 1
1.1.1 Software reuse defined 1
1.1.2 Why software reuse? 3
1.1.3 Success or failure? Why is it so difficult? 4
1.2 More on reuse 6
1.2.1 Different levels of reuse 6
Source code level 6
Design level 6
Architecture level 8
System, infrastructure, and project level 10
1.2.2 Effective reuse 11
1.2.3 The impact of programming languages on reuse . . . 12
1.3 Basics of reuse in the object-oriented paradigm 13
1.3.1 Encapsulation 14
1.3.2 Inheritance 17
Multiple inheritance 18
1.3.3 Composition 19
White-box vs black-box reuse 21
1.4 More on reuse in object-oriented software 21
1.4.1 Interfaces, substitutability, types 21
1.4.2 Mixins and traits 24
1.4.3 Genericity 27
1.4.4 Other reuse techniques: a quick glance 29
Inversion of Control 29

Prototypes and delegation 30
Aspect-Oriented Programming 31
Summary 32

2 Concurrency and Reuse **33**
2.1 Concurrency 33
2.1.1 Basics of concurrency 33
Execution models 34
Coordination 35
2.1.2 The meaning of reuse in concurrent settings 36
2.2 Multi-threaded programming 36
2.2.1 Basics of multi-threaded programming 36
2.2.2 Synchronizing access to shared data 37
2.2.3 Threads and inheritance 38
2.2.4 Major drawbacks 41
2.3 Tasks in Ada 41
2.3.1 Tasks, task type, task body 41
2.3.2 Task creation and execution 42
2.3.3 Task communication 43
2.3.4 Tasks and reuse 45
2.4 The SCOOP model 46
2.4.1 Overview of the SCOOP model 46
Basic concepts 46
Design by Contract 47
The Eiffel programming language and SCOOP 47
2.4.2 Contracts and concurrency 48
2.4.3 Synchronization 49
2.5 The Actor model 53
2.5.1 Overview of the Actor model 53
Semantic properties 54
2.5.2 Synchronization and actor coordination 55
RPC-like messaging 55
Local synchronization constraints 56
Synchronizers 56
2.5.3 Case study: Actors in Scala/Akka 56
Quick glance at Scala/Akka API 57
Scala/Akka and the standard Actor model semantics 58

Extending actors 59
A code example: the Producer-Consumer problem . . 61
2.6 Inheritance anomaly 64
2.6.1 What and why 64
2.6.2 Kinds of anomalies 65
History-sensitiveness of acceptable states 65
Partitioning of states 66
Modification of acceptable states 67
2.6.3 Language-level mechanisms and inheritance anomaly 67
Bodies 67
Explicit message reception 68
Guards 68
2.6.4 Case study: Inheritance anomaly in Scala/Akka . . . 68
History-sensitiveness of states: gget 69
Partitioning of states: get2 70
Modification of acceptable states: Lock mixin 70
2.6.5 Solving the inheritance anomaly 72
Summary 73

3 Agent-Oriented Programming and Reuse 75
3.1 Agents, agent-oriented programming and `simpAL` 75
3.1.1 Agents: definition and reason 76
3.1.2 The simpAL model: a high-level overview 77
3.1.3 Agent behavior 79
3.1.4 Environment, workspaces and artifacts 80
3.2 The simpAL language and platform 83
3.2.1 The simpAL language: an overview 83
Agent interface (agent types) 83
Agent implementation 86
Artifacts 88
Organization 90
3.3 Agents and reuse 91
3.3.1 Reusing artifacts 91
3.3.2 Reusing agents 92
1) More plans: specialization 93
2) More tasks: role extension 94
3) More tasks: multi-role implementation 95

4) Task composition 96
5) Learning via script loading 96
6) Plan variation/specialization 97
Summary 98

4 Conclusion **101**

Introduction

Reuse is important for men. As the availability of energy and matter is not unlimited, waste should be contained. One way to limit waste is to take advantage of what has already been built, i.e. through the reuse of resources. Such observations apply to anything that can be created, from material things to intangible stuff such as knowledge or software. Another resource that is extremely scarce is time. Reuse also contributes to the lowering of the waste of time by reducing the time demands for building things.

A huge amount of software has been and is being developed. Building software within the deadline while ensuring good quality is a challenge and requires en effective software engineering approach. Enabling software reuse may provide significant long-term advantage but it needs to be supported with conscious effort.

Management support is helpful, but opportune mechanisms and tools are necessary. Both the platform and the programming languages should provide assistance for reuse. In particular, it is interesting to understand how language features and abstractions affect reuse.

Modular programming and the object-oriented paradigm do represent significant improvements for software construction and the ability to leverage on existing software components. Other techniques that foster the separation of concerns (e.g. dependency injection, aspect-oriented programming etc.) may also be valuable.

However, things get complicated when concurrency and distribution need to be accounted for. First of all, the essential point of what means reusing concurrent entities and how to do it effectively has not been considered yet with adequate attention. Such a lack does reflect itself in programming languages, which typically provide chance for extending concurrent behavior by redirecting the concern to object inheritance and composition.

Moreover, almost all the concurrent object-oriented programming languages are affected, at different degree, by some forms of inheritance anomaly, caused by a semantic conflict between inheritance and synchronization constraints. Again, a better separation (of concerns) between synchronization code and object behavior may provide relevant results.

The raise of concurrency and distribution demands for concurrent paradigms. The Agent-Oriented Programming (AOP) paradigm embraces these issues by providing human-inspired abstractions that help to model an extremely-interactive, concurrent world. The central abstraction is that of agent, a situated, autonomous and pro-active entity built around the notion of task. The inheritance anomaly shows that reuse of both passive and active entities in concurrent settings is difficult and not-well understood. Pro-activity may be sources of additional puzzles, ultimately pointing out to the question "What means reusing a behavior?"

This thesis aims to provide an overview on software reuse in the context of programming languages, programming techniques, and concurrency models. No works in the literature shares the same point of view. My intention is to highlight this lack and to suggest the possibility of performing further investigation on the subject.

Chapter 1

Software Reuse: An Introduction

This chapter provides an organic overview of software reuse, with emphasis on object-orientation and language-level reuse mechanisms. It consists of two parts.

The aim of the first part is to introduce the concept of *software reuse*. We will see why it represents a crucial software engineering practice and we will try to analyse the reasons for which it may be difficult to implement in all the situations. We will also see that software reuse can happen at many different levels.

The second part of the chapter focuses on the reuse-enabling mechanisms provided by object-oriented programming languages. We will look at the facets of inheritance and composition, with code examples. Finally, a quick survey of other reuse techniques is given.

1.1 Software reuse: the what, the why, the how

1.1.1 Software reuse defined

Before diving into the details of *how* reuse in software can be achieved, a precise definition of *what* we are talking about is needed. So, the first important question to be answered is: "What is software reuse?"

Software reuse can be defined as the process of creating software systems from existing software rather than building software systems from scratch [1]. From this definition, a number of consequences can be drawn about software reuse:

- It is a *process*, so it consists of a number of interrelated tasks which are aimed at making reuse happen
- It addresses the problem of the *construction* of software, thus excluding repeated execution of software, porting and source-code distribution as forms of reuse (at least, for the scope of this thesis)
- In order to be worth, all the activities involved in software reuse must require less efforts than those needed for creating the system ex novo

Note that the definition is general and don't specify what actually are the "software system" to be created and the "existing software" to be reused. In general, it is possible to substitute these expressions with "software artifact", indicating a generic element that is stored in some form in a computer and belongs to a software system. Examples of software artifact are: code, object files, executables, requirement specifications, architectural designs, and so on.

This consideration, in addition to pointing out that software reuse is not just limited to source code reuse, is propaedeutic for a concept that is prominent in the context of software reuse, that of **abstraction**, which consists of the selection or exclusion of certain parts of an idea with the aim to make it more manageable. The next section discusses more on abstraction.

Another point on the expression "existing software", i.e. the software artifact to be reused, is that it completely hides a huge part of the overall process of software reuse: the creation of reusable software. In fact, **reusability** (the ability of being reused) is a property of a software artifact that is determined by how the artifact is built, and measures the degree at which it can be reused. Reusability is not an automatic achievement, it should be promoted by the developers through intentional design decisions. The mechanisms provided by the tools used for building a software artifact can impact on reusability. Specifically, the focus of this thesis is on the reuse-enabling mechanisms provided by programming languages.

1.1.2 Why software reuse?

Building software is difficult. Building software with limited resources (time and money) is far more difficult. The approaches to improve or simplify the software development process include:

- dilating time constraints in order to be able to focus more on quality
- increasing budget for team and/or process strengthening
- adoption of a more effective process framework
- approaches for improving productivity
- approaches for improving software quality

These are not well-defined families, they might intersect based on the point of view. For example, the adoption of a more effective process framework may involve better approaches for increasing software quality. Moreover, it must be considered that software development is complex and potential solutions may not produce the attended results. For example, adding new developers to the team does not typically yield a linear growth in productivity and, in some cases, it may even hinder or reduce it (see Brooke's Law [2]).

Software has been developed for fifty years, this means that for nearly every kind of problem a solution has been already built. So, the new problems that need to be solved can be thought as a combination of variants of "old" solutions within a new context. If these "old" solutons were available, configurable to the specific needs, and able to be connected with one another, we could develop new software by just selecting, configuring and interconnecting pieces of software that were built in the past. If these operations were effort-less, this scenario would be the utopia of software reuse.

Productivity does matter. Not only it allows to stay on budget and to meet deadlines, which are key factors for a project's success and sources of series of advantages and gains, from stakeholder's satisfaction to potential new business opportunities, but it can make the difference through shorter Time-To-Market (TTM), which is especially important in the current competitive world scenario and may directly impact on the Return on Investment (ROI).

However, it should be emphasized that it is not just matter of productivity, but also of quality. Yes, it is possible to build software solutions from scratch, but who guarantees that these solutions are better than those developed in the past?

Thus, this vision of software reuse as a way for overcoming the "software crisis", i.e. the problem of building large software systems in a predictable, efficient and sustainable way, realizes itself through two big achievements:

1. productivity improvement (which also implies shorter time-to-market)
2. quality improvement

While the former is primarily concerned with the *construction* of new software, the latter's biggest consequence is the reduction of the efforts related to the *maintenance* of the system. However, these contributions do intersect as the building process is more effective if it is possible to leverage on very good components. Also, productivity enhancements can indirectly influence quality through time savings, allowing for more quality-related activities to take place. Moreover, the availability of high-quality assets has shown to be considered one factor affecting reuse, together with other aspects such as a reuse education, a common process, the perceived economic feasibility, and the type of industry [3]. In other words, an insufficient quality of reusable artifacts can limit their chance of reuse.

Empirical evidence to the widely recognized advantages of reuse presented above is provided by studies, such as [4], which found "significant benefits from reuse in terms of reduced defect density and rework as well as increased productivity". However, the gains should always be considered together with their associated costs in terms of complexity and resources.

1.1.3 Success or failure? Why is it so difficult?

What is crucial to understand is that software reuse is not just a technical problem [5]. Experiences such as [6] have shown that for large-scale reuse to work the problems to overcome are mostly non-technical, with management playing a crucial role as an enabler. This is mainly due to the fact that software reuse is actually a long-term investment [7] and does require significant changes in the software development process. Change is neither easy nor immediate, so it should be supported with adequate resources and right decisions.

Reusability, as a quality attribute of software, needs planning and conscious effort. In practice, this may means different things:

- defining the expected applications (i.e. the context) for reusable assets, as universal reusability may not probably be convenient
- considering the lack of reusability as a "technical debt"
- considering refactoring cycles to enforce this quality
- taking appropriate design decisions
- packaging the asset in a way to be easily reused

Similarly, software reuse should be pursued *continuously* (it does not just happen) and the development process should include both technical and non-technical support for it.

The two big achievements of reuse – higher productivity and less maintenance effort – should be considered in perspective with the costs associated to the setup and execution of such a software reuse process:

- the identification and definition of good abstractions requires bigger effort on problem analysis
- the creation of reusable software involves spending hours of work on artifacts' reusability, documentation and quality in general
- the creation of new software based on reusable artifacts involves searching them, learning to use them, adapting them to the specific needs, and integrating them with other software components
- in order to effectively execute these activities, tools and process support may be needed

All these activities demand resources. Are they worthwhile? This kind of decisions are taken by the organization. It is essential that the entire organization is pervaded by a software reuse education, with a shared product vision. The single projects should not be considered isolated efforts, but the idea is to capitalize the work done in the perspective of value for the organization.

In summary, the most significant benefits are afforded by a *systematic approach* to software reuse.

1.2 More on reuse

1.2.1 Different levels of reuse

Source code level

At this level, the developer reuses pieces of code that are composed and integrated for defining a certain behavior. In practice, this may mean:

- using specific functions or classes from a library to implement an algorithm
- reusing the implementation of a class by specializing it through inheritance
- copying a snippet of code (e.g. an algorithm or a data structure) and tailoring it to the needs

All these examples involve a single software element that typically will be low-level in terms of abstraction. Consequenty, they allow for low-levels *of* reuse; it means that the amount of reused behavior is little with respect to the behavior of the system under development.

How is it possible to reach higher-levels of reuse? For example, by raising the level of abstraction and dealing with a set of cohesive software elements.

Design level

The term *design* implies two things: elements and relationships. An example of software at the design level is a Java package such as java.io (see Figure 1.1). It consists of a set of related classes that model files, streams and input/output operations.

It is unusual to reuse designs in their code representation. Instead, it is common and very important to reuse design principles or **patterns** of design, such as the well-known GOF's design patterns [8]. The idea of design patterns is to distil experience through guiding principles (or abstract solutions) that are aimed to solve recurrent design problems in an effective and elegant manner.

GOF's design patterns are described by specifying the following elements:

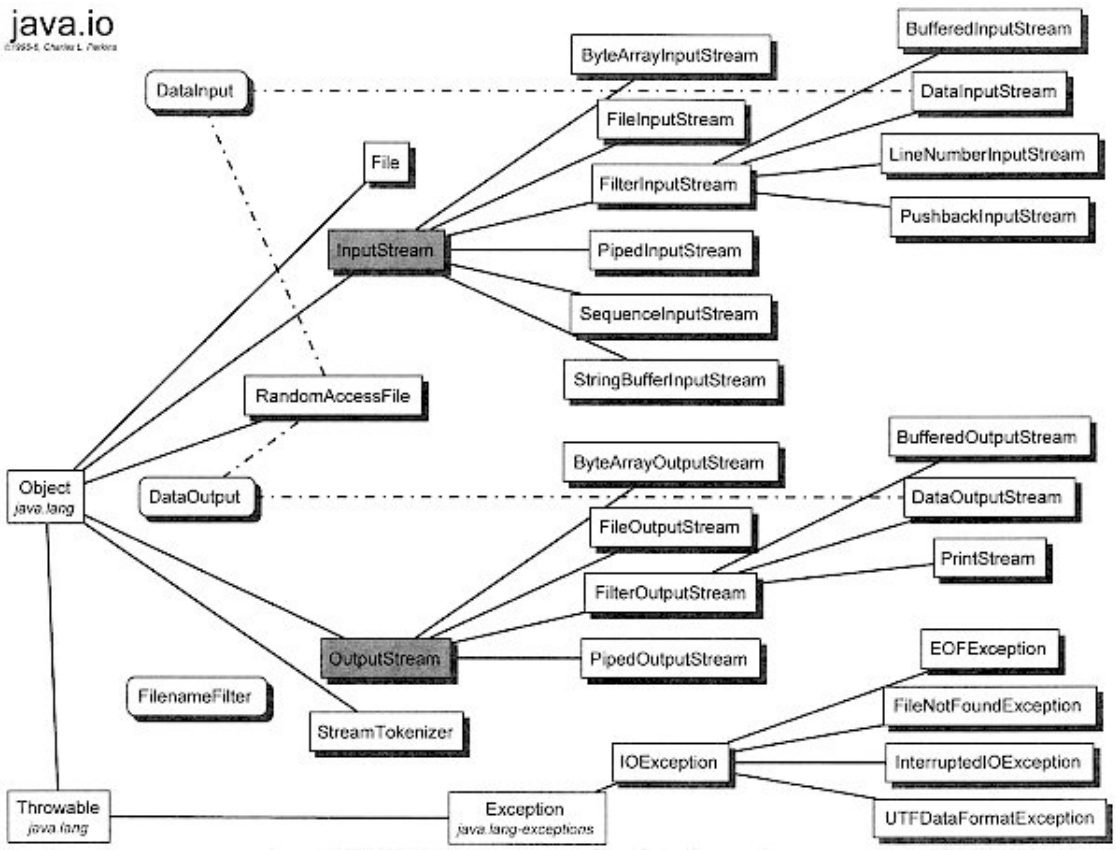

Figure 1.1: The java.io package.Taken from http://101.lv/learn/Java/ch30.htm

- *name, synonims, classification* – these elements allow for pattern referencing when discussing about design; for reuse, they help for pattern search and cataloguing
- *intent, motivation, applicability* – for reuse, they simplify the selection of the pattern given a certain design issue
- *partecipants, collaborations* – these are the elements of design and help to understand the pattern
- *consequences* – they describe dependencies and use-constraints that may directly affect the reusability of the design

- *structure, implementation, sample code* – these descriptions give a sense of how the pattern can be implemented
- *known uses*
- *related patterns* – by listing patterns that solve similar problems or that are sometimes used in conjunction with the pattern, it promotes search and selection of patterns

Confusing patterns with their implementation is a common misunderstanding. The pattern realizations may assume different forms depending upon:

- the context where the patterns are applied
- the programming language constructs and mechanisms

Design patterns are important for reuse because, in addition to their application (which is itself reuse of good design concepts), they provide well-proven software solutions that exhibit high internal quality (which usually include reusability).

Just to mention, a similar concept for the most basic abstractions in programming has been introduced by Jason McC. Smith in [9]. The idea of *Elemental Design Patterns (EDPs)* is to describe the low-level concepts that are used to model solutions for the smallest design issues, thus providing a common language (supplied with a graphical notation, the *Pattern Instance Notation, PIN*) that allows the reasoning, discussion and description of software from its fundamental aspects. These EDPs are composable and can be seen as building blocks for larger patterns and abstractions.

Architecture level

The architecture can be defined as the "fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution" (*IEEE 15288-2008, Systems and software engineering – System life cycle processes*).

At this level, the idea is to tackle complexity through increased **modularity**, which involves breaking a system into separate physical entities [10], according to the *divide et impera* principle. The unit of modularity (and

reuse) is the *component*. There is a shift from lines of code, functions, and classes to coarse-grained components which conform to a component model and can be independently deployed and composed. Also, solutions such as the OSGi[1] Service Platform provide services and a runtime environment to support modularity.

Components allow for high-levels of reuse in the contexts where they can be reused. Coarse-grained components can be created by composition of finer-grained components. This approach finds its maximum expression in the so-called *Component-based Software Engineering*.

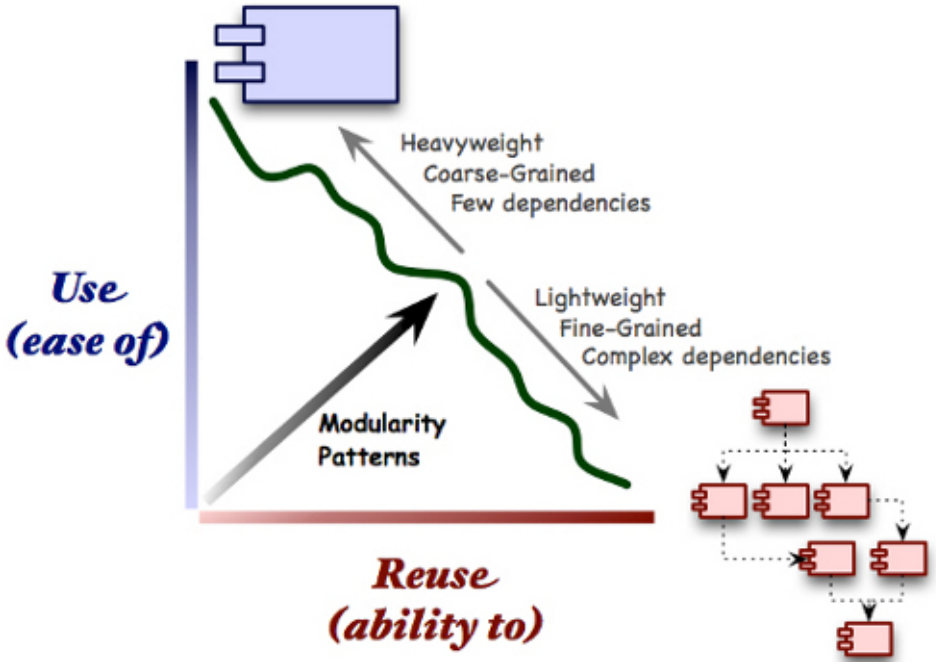

Figure 1.2: Use vs. reuse.Taken from [10]

In addition, **frameworks** allow for reuse from architectural (or process level as well) to source-code level. A framework provide libraries, services, and tools with the aim to simplify the creation of an application for a given context.

[1]OSGi commonly refers to both the platform and the specification, which describes a modular architecture for JVM-based systems: http://www.osgi.org/

Even tough frameworks generally allow for high levels of reuse, there are some drawbacks that may seriously hinder their use:

- time has to be spent in learning how the framework works and what it provides
- they may be too restrictive with respect to certain quality attributes or requirements (e.g. efficiency, as the flexibility that justify the existence of a framework often comes at the expense of performance)
- they may excessively constraint the application's architecture

System, infrastructure, and project level

An additional need may be that of reusing an entire system or parts of it (subsystems or system components at different levels of granularity). A particular case is when the object of reuse is not just a single component but two or more components *together with their relations*. In the context of the object-oriented paradigm, a similar issue has been approached by *family polymorphism* [11], aimed at the expression and management of multi-object relations.

A particular case of system reuse is the following. Suppose that you have to build a program with a simple logic and that it must work on different platforms. Due to distribution and heterogeneity, it is likely that the resulting source code is mostly infrastructural code and only in minimal part representative of business logic.

Would not be great if it were possible to write the infrastructural code just once for every target platform? Would not be nice if we could, for example, abstract the platform-dependent communication details by providing a very high-level language aimed at the specification of the communication semantics – that can subsequently be mapped on anyone of the supported platforms?

Such prospect is feasible and can be addressed by using custom software factories with a combination of Model-Driven Software Development (MDSD) and pattern languages.

The goal is to have an almost complete working system at the end of the analysis stage (not intended as a waterfall process) by raising the execution platform with an additional horizontal layer of platform-independent metamodels that capture the essential elements of their target platform.

In practice, it may turn to:

1. the ability to express the model of a software system by using a meta-model (Domain-Specific Language) that capture the concepts of problem domain (e.g. semantics of interaction)
2. the mapping of the instances of such meta-model into another meta-model which refers to the target platform
3. the executing of the resulting model to generate platform-specific code

See Figure 1.3 as an alternate description of such an idea.

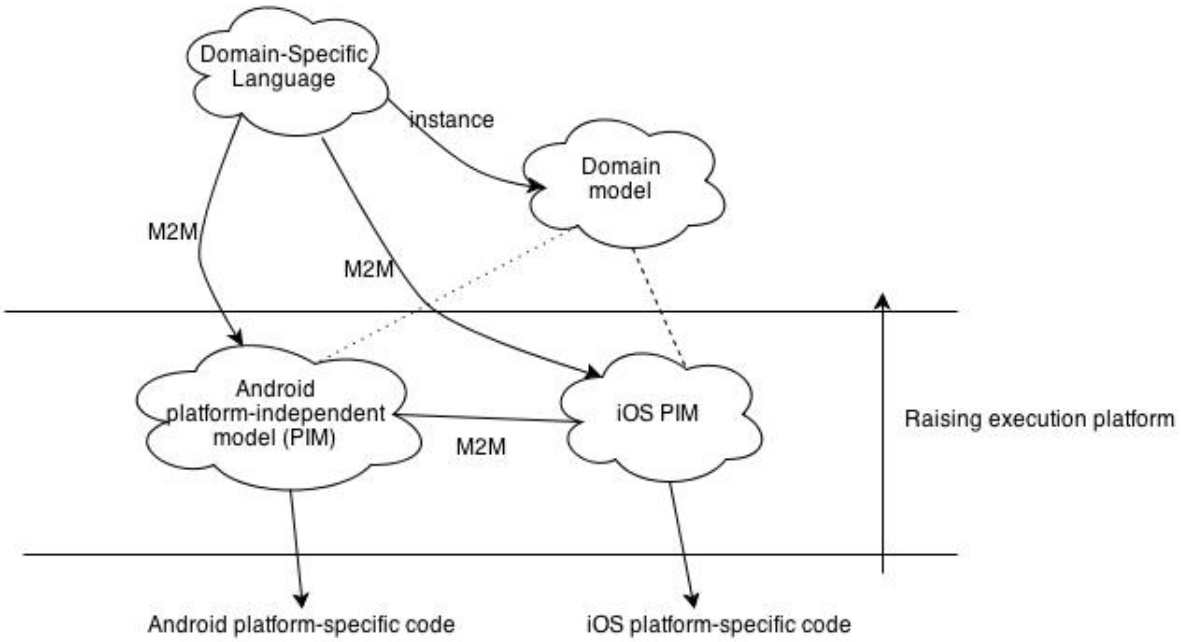

Figure 1.3: Towards software factories.

1.2.2 Effective reuse

We've seen that software reuse can happen at different levels. From this fact, one question follows: "Which is the appropriate level of reuse?" It is very similar to asking when reuse is effective.

Considering that the aim of reuse is to reduce the time and effort needed to build and maintain a software system, Krueger in [1] evaluates the effectiveness of reuse techniques by how much they help to reduce the *cognitive distance* (or conceptual gap), which can be defined as the distance between the concepts informally describing the system and the platform where the system will be executed. This can be done in two ways:

1. through effective abstractions that can be used to model the system
2. through mechanisms that help to obtain an executable system from its specification

1.2.3 The impact of programming languages on reuse

This thesis covers reuse at the language level. The problem of reuse has been presented by considering two separate steps of the process:

1. the creation of *reusable* software, i.e. how to **increase reusability** of software artifacts
2. the actual process of **reusing** reusable artifacts

The distinction between these two aspects is subtle because the definition of reusability is based on the available reuse mechanisms. Programming languages provide language-level reuse mechanisms which affect the reusability of software artifacts and their consequent chance of reuse.

Biddle and Tempero in [12] distinguish between *context reuse*, where the same context can be reused with different components (where the terms "context" and "component" are intended in their general sense), and *component reuse*, where the same component can be reused in different contexts (see Figure 1.4). Consequently, reusability can be seen as a direct expression of:

- how many contexts can invoke a component; and
- how many components can be invoked by a context

Languages' mechanisms and features can impact on these aspects. For example, we see that context reusability is related to the notion of *substitutability*, which in turn is connected to the notion of *type*.

Another important concept that need to take part in this discussion is that of **dependency**, which is a constraint relationship between a context and a component. For example, a function invocation introduces a dependency of the caller on the calling; this means that changes in the name or behavior of the function directly affect the caller, which in turn must be changed. Excessive dependencies reduce the chance of reuse, so they should be kept at the minimum.

In summary, programming languages contribute to software reuse [13] through:

- abstraction mechanisms
- dependency management features

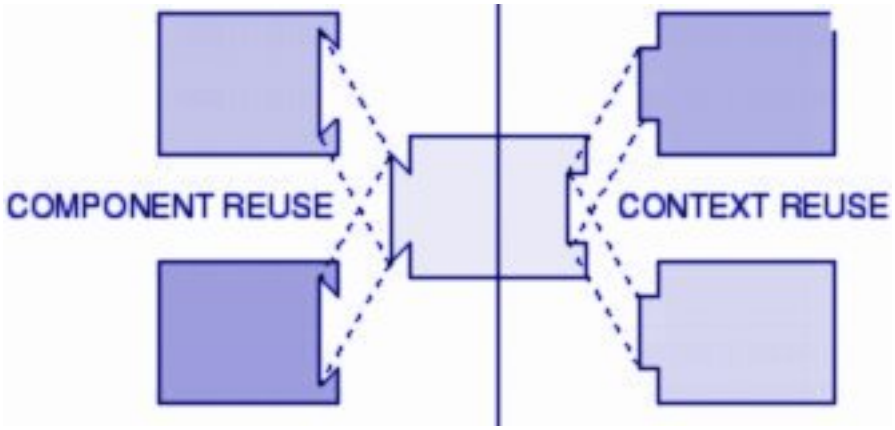

Figure 1.4: Component reuse vs. context reuse.
Taken from http://www.mcs.vuw.ac.nz/research/design1/1998/submissions/biddle/

1.3 Basics of reuse in the object-oriented paradigm

The object-oriented paradigm is a way of developing software by modelling systems as a set of interacting **objects**. It is built upon three big pillars:

1. encapsulation

2. inheritance
3. polymorphism

The unit of modularity is the object, which encapsulates state and behavior. The language may also provide **classes** to define similar objects. In the following subsections we will look at the main mechanisms provided by object-oriented programming languages to support software reuse.

1.3.1 Encapsulation

The term **encapsulation** is used to refer to two things:

1. the ability to pack some data together with the behavior that acts upon it
2. the ability to have boundaries for objects through *information hiding*, which allows to hide implementation details behind a well-defined external *interface*

The ability to isolate the implementation logic behind an interface is important because clients can minimize the dependencies on the component by programming to the interface; this way, changes to the implementation of the component do not affect the clients. For this reason, encapsulation is said to support context reusability [12].

Languages typically provide *visibility modifiers* as a way to enforce information hiding. For example, Java offers four visibility levels for class members:

- private: only the class can access its members
- protected: only the class and its subclasses can access the member
- package-private: only the class and its subclasses in the package can access the member
- public: the member is accessible from the outside

As an example showing (lack of) encapsulation, consider the following source code:

```
// Person.java
public class Person{
  public int age;
  ...
}

// context code
form.fill( person.age );
...
```

Listing 1.1: Dependencies on components' implementation are dangerous.

Suppose we would like to change `Person`'s implementation by substituting the `age` field with a `date_of_birth` field. Now, the context code need to be changed as well because it relied on `Person`'s internal details. Instead, by enforcing encapsulation, changes in implementation do not affect client code:

```
// Person.java
public class Person{
  public Date date_of_birth;

  public int getAge(){
    return Calendar.getInstance().get(Calendar.YEAR) -
          date_of_birth.getYear();
  }
  ...
}

// context code
form.fill( person.getAge() );
...
```

Listing 1.2: Encapsulation allows context code to be independent from the details of the implementation of the components it uses.

An additional example shows how programming to interfaces can help to write context code which can be reused with different components.

```
// IntOperation.java
public interface IntOperation{
  int op(int elem);
}
```

```
5
6  // Multiplier.java
7  public class Multiplier implements IntOperation {
8
9    int factor;
10
11   public Multiplier(int m){ this.factor = m; }
12
13   public int op(int elem){ return elem*factor; }
14
15 }
16
17 // Utils.java
18 public class Utils {
19
20   // perform an operation on each item of the input array
21   // and return a new array with the produced elements
22   public static int[] map(int[] lst, IntOperation op){
23     int[] res = new int[lst.length];
24     for(int i=0; i<lst.length; i++){
25       res[i] = op.op(lst[i]);
26     }
27     return res;
28   }
29
30   public static void main(String[] args){
31     int[] arr = new int[]{ 1,5,10 };
32     IntOperation doubler = new Multiplier(2);
33     int[] res = Utils.map(arr, doubler);
34     for(int i=0; i<arr.length; i++)
35       System.out.println(arr[i]+" => " + res[i]);
36   }
37
38 }
```

Listing 1.3: Interfaces represent a contract between a service provider and a service consumer.

Here the `map()` function can be seen as "context code," that can be reused with different `IntOperations`.

1.3.2 Inheritance

The taxonomy of **inheritance** is broad [14]. It is a mechanism that can serve multiple purposes and that may differ from one programming language to the other.

Basically, it allows for implementation reuse: the state and behavior from the base class is said to be inherited by the derived class. Moreover, through *specialization* it is possible to create specialized child classes. Specialization is a combination of:

- variation: where changes are made to the behavior of the base class
- extension: where the base class is augmented through additional behavior and state

As an example of implementation and specialization inheritance, consider the following Scala code:

```
1  abstract class Shape(var x:Int, var y:Int){
2
3    def position = (x,y)
4
5    def area():Int
6
7    override def toString() = "Shape␣at␣position␣" + position
8
9  }
10
11 class Square(x:Int, y:Int, var side:Int) extends Shape(x,y){
12
13   def area():Int = return side*side
14
15 }
16
17 val s = new Square(2,2,6)
18 println(s)                          // Shape at position (2,2)
19 println("Area␣=␣" + s.area)         // Area = 36
```

Listing 1.4: Implementation reuse, variation and augmentation.

The code is simple, however, multiple things happen:

- `Shape` modifies the inherited `toString()` implementation (variation)

- `Square` inherits the implementation of `Shape`'s `position()` method (implementation reuse)
- `Square` augments `Shape` by defining a new integer `side` field and realizing the abstract `area()` method (extension)

Sometimes inheritance includes **subtype inheritance**, resulting in the child class being a subtype of the base class. Some languages such as Java merge these two concepts, even tough they are actually separate [15]. Other languages keep this separation; for example, C++ provides public inheritance, which include subtyping (interface conformance), and private/protected inheritance, which is just implementation inheritance.

```
1  class Person{
2
3  protected:
4    int age;
5
6  public:
7    Person();
8    int getAge(void);
9
10 };
11
12 // inheritance WITH subtyping (interface conformance)
13 class Child : public Person { }
14 Person* p = new Child(); // OK
15
16 // inheritance WITHOUT subtyping
17 class Child : private Person { }
18 Person* p =
19   new Child(); // error: Person is an inaccessible base of Child
```

Listing 1.5: Inheritance with and without subtyping.

If inheritance does not imply subtyping, objects of the `Child` class cannot be used where objects of the (type of) parent class `Person` are expected.

Multiple inheritance

Some languages such as C++ allow classes to have multiple (more than one) base classes. While it can be seen as a natural and powerful mechanism to

model (real-world) hierarchies, it can also introduce levels of complexity that may outweight its benefits.

The main practical problem is concerned with what happens when the same operation is inherited multiple times. This issue comes in two forms. The first form is when the same operation is inherited by two or more parents: a *name clash* occurs. Dealing with it may involve renaming the member or requiring full member qualification. The second form, also known as the *diamond problem*, occurs in the situation depicted by the Figure 1.5, where there is a class (`StudentWorker`) with two or more base classes (`Student` and `Worker`) which share a common parent (`Person`). This situation of repeated inheritance raises issues about how to deal with the inherited members. Two options are possible:

1. *replication* of the members (it is the case for the `profession` field), if they refer to coexisting variants

2. *merge* (or *sharing*) of the members (it is the case for the `age` field), if they are actually the same thing

For these reasons and the additional complexity, multiple inheritance is considered dangerous when not used properly. As a consequence, many object-oriented programming languages provides single-inheritance for classes, often with the addition of other mechanisms in order to overcome the resulting limitations in modelling power. For example, languages such as Java and C# maintain multiple inheritance only for interfaces; other languages such as Scala and Ruby provide mixins as a way to reuse multiple implementations.

1.3.3 Composition

Whereas subtyping is said to be an *IS-A* relationship, composition is said to be an *HAS-A* relationship. With **composition**, an object consists of other object which provide functionality that contributes to the behavior of the containing object. A similar notion is that of *aggregation*; while in composition the lifecycle of the contained objects is bound to the containing object's lifecycle, it does not hold in aggregation.

```
1 import scala.math._
```

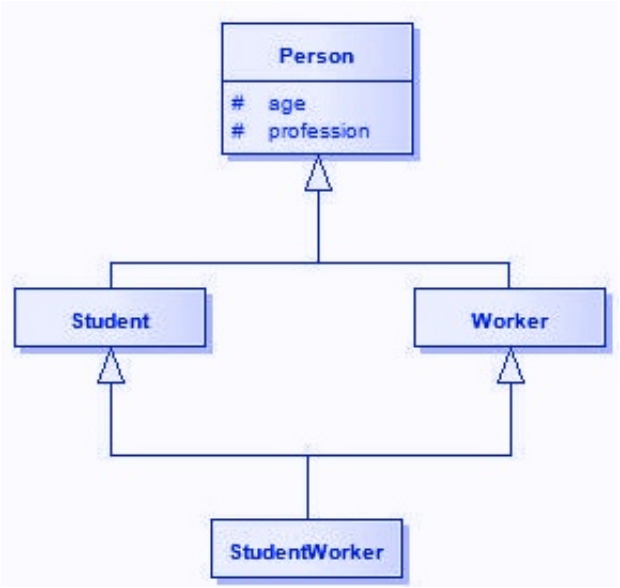

Figure 1.5: The diamond problem

```
2
3  class Point(var x:Int, var y:Int){
4
5    def distance(other: Point) : Double =
6        sqrt(pow(x-other.x,2)+pow(y-other.y,2))
7
8  }
9
10 class Rectangle(val p1:Point, val p2:Point){
11
12   def area():Int = abs(p1.x-p2.x)*abs(p1.y-p2.y)
13
14   def diagonal():Double = p1.distance(p2)
15
16 }
```

Listing 1.6: Composition.

In the previous example, a `Rectangle` is composed of two `Points` representing two opposite corners. As a consequence, it can (re)use the functionality provided by its components to implement new behavior. The

`diagonal()` method has been defined this way.

White-box vs black-box reuse

Reuse can be considered of two types:

1. *white-box* – if reuse exploits knowledge upon the internal details of the reusable component
2. *black-box* – if reuse does not exploit it (so the reusable component is seen "from the outside" as a black-box)

Inheritance is considered a mechanisms for white-box reuse, whereas composition is considered a black-box reuse technique. White-box reuse can be dangerous because it can create dependencies on implementation details, which are more unstable than interfaces. For this reason, a common advice is to favor composition over inheritance [8], because the latter can break encapsulation by exposing the internals of the base class to its subclasses.

It should be noted that inheritance can be used in a black-box manner, by avoiding the use of `protected` members. However, this may preclude opportunities for specialization. It is also possible to use composition in a white-box manner, if implementation details are exposed through the public interface.

1.4 More on reuse in object-oriented software

1.4.1 Interfaces, substitutability, types

We have seen that subtyping, through interface conformance, allows objects of subclasses to be used where objects of the type of the base class are expected. Polymorphism is what make it possible to dispatch the correct method implementation.

In languages such as Java and C#, interfaces can be defined using language constructs and can be used as types. An interface represents a contract between providers and consumers; it makes the syntactic dependencies explicit. Programming to interfaces has shown to be an effective practice for writing reusable code. The notion of *type* is important because the higher

the substitutability of the components used in a piece of code is, the higher is its potential reusability because it can be reused with a higher number of different components.

In the aforementioned languages, interfaces need to be implemented explicitly by providers, resulting in what is referred to as *nominal subtyping*, which is based on names and explicit declarations. However, other kinds of type systems exist.

For example, **duck typing** is a dynamic type system which is often informally described by the expression "if it walks like a duck and quacks like a duck, it must be a duck." It means that the type of an object is defined by what the object is able to do. Consider the following example:

```
class Adder
  def self.add(x,y)
    x+y                          # equals to x.+(y)
  end
end

Adder.add(1, 5)                  # -> 6
Adder.add("duck","typing")       # -> "ducktyping"

class Person
  attr_accessor :age

  def initialize
    @age = 18
  end

  def +(years){
    @age = @age + years
  }
end

bob = Person.new
Adder.add(bob, 2) == bob.age     # 20 == 20 -> true

Adder.add("code",404)            # TypeError
```

Listing 1.7: Duck typing in Ruby.

It is immediate to see the flexibility given by such a dynamic behavior: the `Adder`'s static `add()` method can be used with any object `x` that supports a `+()` method. However, the disadvantage is also clear: errors

can be caught only at runtime (see the "TypeError: can't convert Fixnum into String" on the last line). In particular, the problem is that to be able to use correctly `Adder::add()`, we should know parts of the implementation of both `Adder::add()` and `x.+()` (dependencies are implicit, so they should be documented!). The consequences are serious and impact directly on the ability to write robust code for large systems.

A static approach to achieve a sort of "type-safe duck typing" is based on the notion of **structural typing**, where type compatibility depends on how objects are defined (i.e. what is their structure), conversely to the nominal approach where names and explicit declarations are used. The following example in Go source code shows the things work as if interfaces were implicitly implemented:

```
1  package main
2  import "fmt"
3
4  type Printable interface {
5    toString() string
6  }
7
8  type Printer interface {
9    print(string)
10 }
11
12 func printThemAll(p Printer, lst ...Printable) {
13   for _, item := range lst { p.print(item.toString()) }
14 }
15
16 // implicit implementation of Printable interface
17 type Person struct {
18   name string
19   age int
20 }
21 func (p Person) toString() string {
22   return fmt.Sprintf("Name: %s - Age: %d", p.name, p.age)
23 }
24
25 // implicit implementation of Printer interface
26 type BasicPrinter struct { }
27 func (BasicPrinter) print(s string){
28   fmt.Println(s)
29 }
30
```

```
31 func main(){
32   printer := BasicPrinter{}
33   bob     := Person{"Bob", 30}
34   john    := Person{"John", 35}
35   printThemAll(printer, bob, john)
36 }
37 // Output:
38 //   Name: Bob - Age: 30
39 //   Name: John - Age: 35
```

Listing 1.8: Structural typing in Go.

Here, the `printThemAll()` function can be called with any `Printer` and any number of `Printables`. The conformance to these interfaces is checked by the compiler. The code is type-safe but extremely flexible. For example, it is possible to write an interface for existing objects, which implicitly implement the interface without any need to be adapted on purpose. In languages such as Java, it is not possible: when an interface is introduced, classes are required to be changed in order to create the desired subtype relation.

1.4.2 Mixins and traits

So far, we have seen that implementation reuse is supported by composition and inheritance. However, these techniques do introduce strong dependencies, especially inheritance which may also involve subtyping and dependencies on implementation details.

Mixins are, substantially, abstract classes: they cannot be instantiated and provide implemented methods which can be inherited by other classes. However, mixins are more a way to collect functionality, rather than representing concepts that can be specialized. Mixins have more sense in multiple-inheritance languages because in single-inheritance languages there is place for just one base class.

Instead, **traits** [16] offer different composition operators. They are like interfaces (thus, with no state) with implemented methods, representing purely unit of reuse, distinct from classes which serve different purposes. Traits can be also parametrized with methods, requiring classes implementing the traits to provide them.

Some languages such as Scala and Ruby provide trait-like mechanisms. As a simple example, consider the following Scala code:

```
trait Person {

  def getName():String

  def presentMyself() = println("Hi, my name is " + getName())

}

class Engineer(name:String){

  def getName():String = return name

}

bob = new Engineer("Bob") with Person
bob.presentMyself()     // "Hi, my name is Bob"
```

Listing 1.9: Scala traits.

Here, `Engineer` class is augmented by the functionality provided by the `Person` trait, but in order to achieve this, it has needed to fulfill the contract with the mixing in module by providing an implementation for the abstract `getName()` method (which represents a parameter for the `Person`'s trait).

The Ruby programming language provides *modules* that can be used similarly to traits. They can be used as a simple extension mechanism, but they also serve to support **metaprogramming** techniques. Metaprogramming allows for the creation of extremely expressive code (and *domain specific languages* as well) and great code flexibility; in this sense, it can be seen as a powerful reuse technique. The next example is not trivial and shows in action the use of modules and metaprogramming:

```
module Utils

  def self.included(klass)
    klass.extend(ClassMethods)
  end

  module ClassMethods
    def accessor(attrname)
      define_method "#{attrname}=" do |value|
        instance_variable_set("@#{attrname}", value)
      end
```

```
12        define_method "#{attrname}" do
13          instance_variable_get("@#{attrname}")
14        end
15      end
16
17      def accessors(*attrlst)
18        attrlst.each do |attr|
19          accessor(attr)
20        end
21      end
22    end
23
24    def presentMyself
25      puts "Hi, my name is " + @name
26    end
27
28  end
29
30  class Person
31
32    def initialize(name, age)
33      @name = name
34      @age = age
35    end
36
37    include Utils
38    accessors 'name', 'age'
39
40  end
41
42  bob = Person.new("Bob", 30)
43  bob.presentMyself                # Hi, my name is Bob
44  bob.name                         # "Bob"
45  bob.age = 31                     # 31
```

Listing 1.10: Mixin modules and metaprogramming in Ruby.

Here the `accessors()` utility method is used to define a getter and a setter for each of its arguments; in Ruby classes the state is private and can be accessed from the outside only through accessors, as in the last two lines in the listing. Other things to note include:

- the `Utils` module defines the `presentMyself()` method which, through module inclusion, will be an instance method of `Person`

- the definition of `presentMyself()` is based on the `@name` instance variable, which must be defined by the including class in order to avoid runtime errors
- `self.included()` is a hook method that is called when the module is mixed-into by the class
- `self.included()` calls the `extend()` method, which includes the module provided as argument into the receiver's metaclass (its effect is to have the `ClassMethods` module's methods to become static methods in the `Person` class)
- the methods `define_method()`, `instance_variable_get()`, and `instance_variable_set()` allows to define class members dynamically

It is very dynamic code which provides powerful features. The module includes method implementations that can be easily reused: just one call to `accessors()` is sufficient to generate getters and setters for any arbitrary number of fields!

1.4.3 Genericity

Genericity (or generic programming) consists of creating generic code schemes that accept type parameters. Next, they can be instantiated to specific types. Such code schemes are called in various ways, such as *templates* in C++, *generics* in Java, or *parametrized types*. Usually they are statically checked, so to be type-safe.

The positive impact of genericity on reuse is crystal-clear because it allows to implement type-safe functionality independently from the specific types. Otherwise, it would be extremely unsatisfying to write the same code for each type it works with.

Static type checks allow for more robust code, without the need of casts. The following code snippets shows Java generics in action:

```
1 // Iterable.java
2 public interface Iterable<T> {
3   Iterator<T> getIterator();
4 }
5
```

```
6
7  // Iterator.java
8  public interface Iterator<T>{
9    boolean hasNext();
10   T next();
11   void reset();
12 }
13
14
15 // Pair.java
16 public class Pair<T> implements Iterable<T>{
17   private T first;
18   private T second;
19
20   public Pair(T first, T second){
21     this.first = first;
22     this.second = second;
23   }
24
25   public Iterator<T> getIterator(){
26     return new PairIterator<T>(this);
27   }
28
29   public T getFirst(){ return first; }
30
31   public T getSecond(){ return second; }
32
33 }
34
35
36 // PairIterator.java
37 public class PairIterator<T> implements Iterator<T>{
38   private Pair<T> pair;
39   private int counter;
40
41   public PairIterator(Pair<T> p){
42     this.pair = p;
43     this.counter = 0;
44   }
45
46   public boolean hasNext(){ return counter<2; }
47
48   public T next(){
49     if(counter==0){
50       counter++;
```

```
51        return pair.getFirst();
52      } else if(counter==1){
53        counter++;
54        return pair.getSecond();
55      }
56      return null;
57    }
58
59    public void reset(){ counter = 0; }
60
61  }
62
63
64  // main program somewhere
65  public class MainProgram {
66    public static void main(String[] args){
67      Iterable<Integer> p = new Pair<Integer>(1,5);
68      Iterator<Integer> it = p.getIterator();
69      while(it.hasNext()){
70        System.out.println(it.next());
71      }
72    }
73  }
74  // Output:
75  //   1
76  //   5
```

Listing 1.11: Java generics: iterable and iterator.

Generics are commonly used for defining *container classes* as well as the generic algorithms that act upon them.

1.4.4 Other reuse techniques: a quick glance

Inversion of Control

Inversion of control is a technique (which can be implemented through the *Dependency Injection* pattern) where an object's dependencies are not retrieved by the object itself but they are determined and set by other components.

Thus, inversion of control can be seen as a way to promote context reusability because an object does not need to be changed to work with another component. Dependencies can be configured externally but no change

in code is required.

Prototypes and delegation

Prototype-based languages are object-oriented but they do not provide classes. Instead of using inheritance, these languages allow for implementation reuse through **delegation**. Objects are not created by instantiating a class; instead, they are *cloned* from another object (that is called their *prototype*).

A different approach is taken by the Go programming language which can not be defined neither as a class-based language (as it does not have classes/inheritance) nor as a prototype-based language (as prototypes and cloning are not present).

```
1  package main
2  import "fmt"
3
4  type Pair struct{
5    a int          // composition
6    b string       // composition
7  }
8
9  func (p Pair) getKey() int { return p.a }
10 func (p Pair) getValue() string { return p.b }
11
12 type OnOffPair struct {
13   Pair           // embedding, delegation
14   on bool        // composition
15 }
16
17 func main() {
18   a := OnOffPair{Pair{1, "hello"}, true}
19   fmt.Println(a.getKey())       //  delegation
20   fmt.Println(a.getValue())     //  delegation
21 }
22
23 // output:
24 //   1
25 //   hello
```

Listing 1.12: Delegation in Go.

Note that `getKey()` and `getValue()` methods are called on an `OnOffPair` object even though they are defined by accepting a `Pair` object as receiver.

This is possible because the `OnOffPair` object delegates the execution of these methods (implementation reuse) to the embedded `Pair` object.

Aspect-Oriented Programming

Aspect-Oriented Programming is a programming technique that promotes the *separation of concerns* by supporting the implementation of the so-called *crosscutting concerns*, which are functionality that spans the entire application, into **aspects**.

Normally, crosscutting concerns would produce code that is both *scattered* and *tangled*. The result is code that is difficult to understand and change. Moreover, if a new class require the same functionality, the same code has to be written again, given that the class does not belong to a hierarchy where that logic has been factored in the base class.

Aspects come to solve these issues. They promote reuse and flexibility by encapsulating functionality that otherwise would be scattered across the system.

Moreover, aspect-oriented approaches have been considered in the attempt of solving the inheritance anomaly problem - which will be covered in the next Chapter -, producing better results with respect to more traditional approaches.

Summary

The key points of this chapter are:

- software reuse is a process that consists of developing new software from existing software
- its aim is to increase productivity and quality in the software development process while reducing the maintainance effort
- it is not just a technical issue, it requires an appropriate process and management support in order to pursue its long-term benefits
- reusability is a quality of software that requires calculated design decisions and supporting activities
- reuse can happen at many different levels
- in OO software, inheritance and composition (with the support of encapsulation) are the most common mechanisms for reuse
- many different techniques and constructs such as interfaces, types, generics, traits, aspects etc.. can impact on reusability and reuse

Chapter 2

Concurrency and Reuse

The first chapter was about software reuse and reuse mechanisms, with special emphasis on the object-oriented paradigm and language-level aspects.

This chapter looks at concurrency and explores the topic of software reuse in concurrent settings. This topic has not been tackled yet in its full scope by research. Here, some considerations are reported, together with the realization of the importance of the question.

An overview of some of the most important models for concurrent programming is provided. We will see what their approaches are and how they relate with reuse and inheritance.

Finally, a digression about the inheritance anomaly in object-oriented concurrent programming languages is presented.

2.1 Concurrency

2.1.1 Basics of concurrency

The success of the object-oriented paradigm has been dictated by its contributes in terms of modularity and data abstraction [17], which are decisive aspects for the construction and the maintenance of large and robust software systems. However, the object oriented approach is mainly structural and, while representing a valid foundation for the development of sequential programs, it lacks of adequate support for **concurrency** and **distribution**, which represent two of the major issues of software development at the present time.

The world is inherently concurrent and nowadays computers are everywhere, pervasively accessing all the aspects of the life and where all is interconnected, bringing us to a notion of *ubiquitous computing*. This is why we need adequate tools aimed at reducing the complexity involved in these increasingly important issues.

The term "concurrency" comes from the verb "to concur", which in turn comes from the Latin "concurrere" (CUM+CURRERE, "to run with"). The etymology suggests that, in concurrency, software entities (or pieces of code) progress simultaneously. Thus, they *may* run together, that is, they *may* be executed **at the same time**. When there is only one processor, such an execution can be only virtually simultaneous. In the case of two or more processors, the concurrent execution can be actually contemporaneous, or *parallel*.

With respect to sequential contexts, where the execution path of a program is predictable, *concurrency entails unpredictability upon the execution order*. This results in greater complexity.

Concurrent programming refers to all the aspects that are involved in the implementation of concurrent programs. Concurrent programming is performed using a language which may adhere, in general, to one or more concurrency models.

A **concurrency model** consists of abstractions that support the reasoning and the description of concurrent programs. It defines a set of concepts, relations and semantic properties that can be used to model a concurrent scenario. A concurrency model is effective when it reduces the conceptual gap between the problem to be solved (at the human level) and the underlying platform; thus, such a model should provide high-level abstractions and easy-to-use realization mechanisms for them.

Execution models

As said before, the execution order of the steps of a **sequential** program is predictable, as the next step (determined by the program's logic) cannot change accidentally: with a given set of inputs, the same output will always be provided. It is an easy way of thinking, but also limited because such a **deterministic execution model** makes it impossible for the program to be parallelized.

By contrast, **concurrent executions** may, in general, produce different orders in which the steps of a program are carried out. In fact, when multiple pieces of software run concurrently, it is impossible to guarantee which is the next statement to be executed. This is particularly evident when the execution is actually parallel, for example when part A of a program is run by processor 1 and part B is run by processor 2. However, also in a scenario with a single processor where the concurrent parts are interleaved in time, it is not possible to predict when such parts will be paused or started.

Coordination

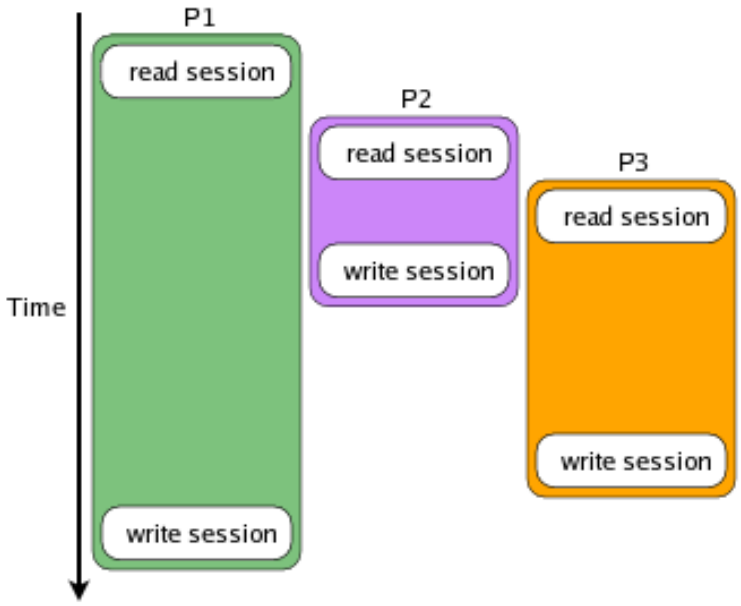

Figure 2.1: Example of race condition. In this case, P2's update is overwritten by P3, and P3's update is overwritten by P1.
Taken from http://thwartedefforts.org/2006/11/11/race-conditions-with-ajax-and-php-sessions/

As concurrency implies uncertainty upon the execution order of the steps of a program, a number of issues arise. In fact, not all the execution paths may be correct, with respect to the desired behavior. For example, when

multiple threads of execution access some share data (see Figure 2.1), **race conditions** are possible. Thus, **synchronization constraints** are required in order to exclude the execution paths that lead to incorrect behavior.

The solution to the issue in Figure 2.1 is to have the concurrent parts `P1`, `P2`, and `P3` to **coordinate** themselves so that the two-step read-write operations are *serialized* in time.

Programming languages provide different *synchronization mechanisms* or *primitives*, depending upon the concurrency model they embrace, that can be used to implement the coordination logic.

2.1.2 The meaning of reuse in concurrent settings

Reuse targets *state* and *behavior*. Objects encapsulate both these elements, and we have seen in Chapter 1 how reuse takes place in the object-oriented paradigm. Specifically, we have seen how inheritance can be used to modify or extend base classes.

However, whereas in sequential settings there is just one thread of execution, in concurrent settings two or more threads of execution exists. Moreover, the behavior may contain coordination logic as well, which typically should be preserved by reuse (especially if it is accompanied by a subtype relation). Later in this Chapter we will see how such a synchronization logic may result in issues when using inheritance.

In general, reuse means leveraging on existing software artifacts in order to produce new artifacts. For example, a set of objects can be interconnected to form a new system; likewise, new tasks can be created by reusing behaviors.

When it comes to concurrency, additional semantic constraints (e.g. upon the order of some computational steps, or upon the set of messages that an object can accept) need to be accounted for, and should be guaranteed *in the new context* of reuse.

2.2 Multi-threaded programming

2.2.1 Basics of multi-threaded programming

Multi-threaded programming consists of creating applications using multiple threads that execute concurrently. A **thread** (of execution), or *kernel*

thread (also known as *native thread*), is the smallest entity that can be scheduled by the operating system. It differs from the notion of *process* because, in general, processes may include multiple threads which share the process' resources. Also, typically threads share the same address space, resulting in cheaper context-switch overhead with respect to process context-switch. Threads can be think as lightweight processes that share memory.

In addition to kernel threads, *user threads* (also known as *green threads*) exist. User threads are threads at application level. They are not managed by the kernel; instead, support for them is given by the language runtime. User threads can be mapped to kernel threads in different ways. One possibility is N-to-1 mapping, where N user threads are mapped to one kernel thread; in this case, context switch overhead may be sensibly lower, however no parallelization is possible as the operating system scheduler sees only one (kernel) thread.

2.2.2 Synchronizing access to shared data

Threads typically **share data**. This solution is very efficient, but it is also source for issues: when an object is accessed concurrently, its state may potentially be corrupted. For this reason, a form of synchronization must be applied in order to serialize access to it. *Critical section* is the term used for referring to the part of code wherein a shared resource is accessed; their execution must be *mutually exclusive*.

The two main forms of synchronization exist:

- *lock-based* synchronization
- *non-blocking* (or *lock-free*) synchronization

The former kind of synchronization involves the use of *locks*, i.e. mechanisms that allow exclusive access to resources or parts of code. Before being granted the right to access a shared resource, the corresponding lock must be acquired; after processing has been performed, the lock must be released. Situations when more than one thread try to acquire a lock must be managed with appropriate *lock contention* policies. Multiple lock-based mechanisms exist:

- locks (mutexes)

- condition variables
- monitors
- semaphores

The mutex is the lock mechanism used to achieve **mut**ual **ex**clusion:

```
1  int shared_int = 0;
2
3  std::mutex gmutex;
4
5  void increment(){
6    // using lock_guard<>, lock on gmutex is acquired
7    //    at construction and released at destruction
8    std::lock_guard<std::mutex> lg(gmutex);
9
10   // the increment operation needs atomic semantics
11   //    in order to avoid race conditions
12   shared_int++;
13 }
```

Listing 2.1: Use of mutexes in C++11.

Conversely, lock-free synchronization supports access to resources along with ensuring system or thread progress. This is mainly achieved through *atomic operations*, which must be provided by the hardware or emulated at the operating system level.

2.2.3 Threads and inheritance

Threads have an associated *thread body* containing the instructions that will be executed when the thread is running. The thread body contains the behavior scheme or processing logic of the thread. Threads are, in this sense, similar to functions executed concurrently. Therefore, threads are usable, but *not* reusable.

When threads and object-oriented programming merge, one possibility for reuse is to apply inheritance and the *Template Method* design pattern. Consider the example in Listing 2.6.

```
/***** Work.java *****/
public class Work {
  private int value;

  public Work(int value){ this.value = value; }

  public int getValue(){ return value; }
  public void setValue(int value){ this.value = value; }
}

/***** WorkQueue.java *****/
public class WorkQueue<Work>
    extends java.util.concurrent.ArrayBlockingQueue<Work> {

  public WorkQueue(int capacity){ super(capacity); }

  public Work nextWork(){ return this.poll(); }
}

/***** Worker.java *****/
public class Worker implements Runnable{
  protected WorkQueue<Work> queue;

  public Worker(WorkQueue<Work> wq){ this.queue = wq; }

  /* THREAD BODY */
  public void run(){
    Work work;
    while( (work=nextWork()) !=null){
      boolean ok = preProcess(work);
      if(ok){
        process(work);
      }
    }
  }

  protected Work nextWork(){ return queue.nextWork(); }

  /* TEMPLATE METHODS */
  protected boolean preProcess(Work w){
    return (w.getValue()%2==0) ? true : false;
  }

  protected void process(Work w){
```

```
45      System.out.print(w.getValue() + "␣");
46    }
47  }
48
49  /***** SpecializedWorker.Java *****/
50  public class SpecializedWorker extends Worker {
51    public SpecializedWorker(WorkQueue<Work> wq){ super(wq); }
52
53    public void process(Work w){
54      System.out.print("["+w.getValue()+"]␣");
55    }
56  }
57
58  /***** Executor.java *****/
59  public class Executor {
60    public static void main(String[] args)
61        throws InterruptedException {
62      WorkQueue<Work> wq = new WorkQueue<Work>(200);
63      for(int i=0; i<100; i++){
64        wq.put(new Work(i));
65      }
66
67      Worker wk = new Worker(wq);
68      Worker wk2 = new SpecializedWorker(wq);
69
70      Thread t = new Thread(wk);
71      Thread t2 = new Thread(wk2);
72
73      t.start();
74      t2.start();
75    }
76  }
```

Listing 2.2: Threads, inheritance, and the Template Method pattern.

First of all, note that synchronization is required for `WorkQueue` objects; in fact, as its operations (such as polling an element out of the queue) are not atomic, race conditions can occur and leave the objects in an inconsistent state. The required protection is achieved by extending the thread-safe `ArrayBlockingQueue<T>` class in the `java.util.concurrent`.

Here, subclasses of `Worker` can provide their implementations for methods `preProcess()` and `process()`. Of course, chance of reuse are limited to the template methods.

Finally, note that similar results may be achieved by using composition.

Thread classes may depend upon utility classes which can be extended to provide specialized behavior.

2.2.4 Major drawbacks

Threads represent a commonly-used mechanism for implementing concurrent programs. However, some disadvantages are serious and demand for better approaches to concurrency.

First and foremost, the thread is **inadequate as an abstraction mechanism**. The concept of "thread" is at the level of either the language runtime or the operating system, too low with respect to the concepts of the domain problem. Also, the association of threads with objects (as it can be found in Java, for example) does not represent a solution.

Moreover, **access to shared data must be synchronized**, resulting in additional complexity in the code and in the risk of incurring in deadlocks or inconsistent system states. In particular, it is not easy to write thread-safe software while keeping low the overhead associated with the protection of shared data structures. Alternative forms of communication (other than *data-based communication*) exists, as we will see when talking about the Actor model.

Finally, it can be affirmed that, as a model of computation, the contribute of threads for handling **nondeterminism** has shown to be far from being satisfying [18].

2.3 Tasks in Ada

This section briefly describes Ada's [19] approach to concurrency [20] and points out the most peculiar traits of the model.

2.3.1 Tasks, task type, task body

In Ada, programs consist of one or more **tasks** that execute concurrently. Tasks are similar to threads, but, whereas a thread is a mechanism for achieving concurrency, namely, an entity encapsulating an autonomous thread of control, a *task* is more a conceptual entity representing some *work* to be done.

Ada is an object-oriented language. So, *passive entities* can be modelled through objects. *Active entities*, instead, can be modelled by means of tasks.

A *task unit* consists of a *task declaration* and a **task body**. A task declaration is either the definition of a **task type** or the declaration of a single task. The declaration of a task specifies its communication interface, that is a list of *entries* which represent the access points for other tasks to communicate.

```
1  task type MyServer(port: Integer) is
2    entry Service1(in_arg: in T1);
3    entry Service2(arg: T2, out_arg: out T3);
4  end MyServer;
5
6  task body MyServer is
7    -- imports..
8    -- local declarations..
9  begin
10   -- TASK BODY
11   -- ...
12   accept Service1(in_arg: in T1) do
13     -- Service1 ENTRY BODY
14   end Service1;
15   -- ...
16 end MyServer;
```

Listing 2.3: Task type and task body.

Thus, an *explicit separation between communication interface and implementation* exists.

2.3.2 Task creation and execution

A task is `created` when an object of the relative task type is declared.

```
1  declare
2    type TaskType_A_Ptr is access TaskType_A;
3    task_a1, task_a2 : TaskType_A; -- these tasks are created
4  begin
5    -- exec is blocked until task_a1 and task_a2 are activated
6    -- this context, task_a1, and task_a2 run concurrently
7    dynamic_task_a : TaskType_A_Ptr := new TaskType_A;
8    -- here, exec is blocked until dynamic_task_a is activated
9    -- dynamic_task_a runs concurrently
10 end;
```

Listing 2.4: Task type and task body.

Before being able to be executed, tasks need to be `activated`. Next, they execute (concurrently) until completion.

Ada is a block-structured language. Blocks can be nested, and tasks can be created within any block. Thus, a **task hierarchy** arises, where we have *master tasks* and *dependend tasks*. Before a task is said to be terminated, it must be finalized (so that its memory is reclaimed); however, a dependant task may need to access to its master's local data. Due to this reason, a task is not finalized until all its dependent tasks have terminated.

Figure 2.2 provides a summary of state transitions for a task.

2.3.3 Task communication

Tasks can communicate between one another

- *directly*, using task-to-task communication facilities, or
- *indirectly*, using shared data

Direct communication is achieved through **rendez-vous**. It is a form of synchronous communication where the calling task and the called task meet at one entry point:

- the calling task performs an *entry call* and waits for it to be accepted and completed
- the called task is suspended until it receives an entry call for the entry it is accepting

Once they meet the accept body for the entry is executed; when it is finished, the rendez-vous terminates and both the tasks proceed with their flow of execution (see the bottom part of Figure 2.2). During rendez-vous, data can flow in both directions through `in`, `out` or `inout` parameters.

Indirect communication is achieved through **protected objects**. Protected objects are defined by a *protected type* and a *protected body*; alternatively, a single protected object can be specified. The protected type consists

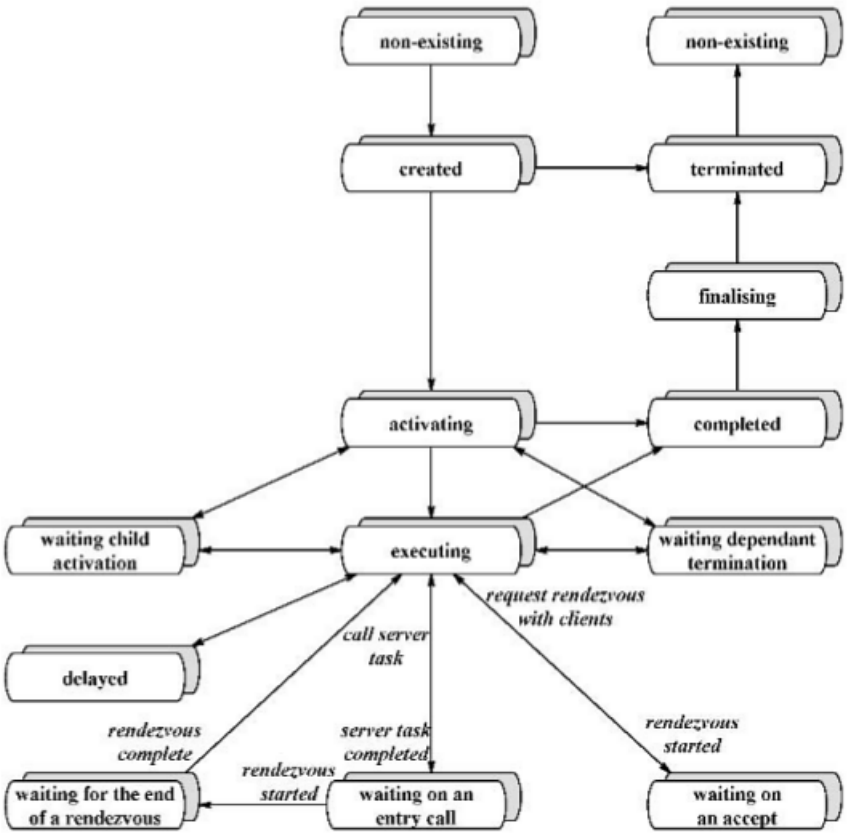

Figure 2.2: Task state diagram.Taken from [20]

of the declaration of the private data (i.e. the data to be protected) and a set of operations. Three types of operation are possible:

- *protected functions*: allow for concurrent read-only access to the protected data
- *protected procedure*: are executed in a mutually exclusive fashion
- *protected entries*: are like protected procedure but can also be guarded

As an example, consider the following code snippet:

```
1 protected type Bounded_Buffer(capacity: Positive) is
```

```
2     entry Get(item: out Integer);
3     entry Put(item: in Integer);
4
5     function Empty return Boolean;
6     function Full return Boolean;
7   private
8     buffer : array (1..capacity) of Integer;
9     nitems : Integer;
10    -- ...
11  end Bounded_Buffer;
12
13  protected body Bounded_Buffer(capacity: Positive) is
14
15    entry Get(item: out Integer) when nitems>0 is
16    begin
17      -- ...
18    end Get;
19
20    -- ...
21    -- ...
22  end
```

Listing 2.5: Definition of a protected object: protected type and protected body.

2.3.4 Tasks and reuse

The possibility of reusing tasks has not been sufficiently considered by the designers of Ada.

No mechanism has been provided for defining a new task body by reusing an existing task body. In fact, only tagged types support type extension and polymorphism. Consequently, tasks can be extended only indirectly and with considerable effort [21].

This lack is an additional demonstration of what poor emphasis has been given to the reuse in concurrent settings.

2.4 The SCOOP model

2.4.1 Overview of the SCOOP model

The goal of the **SCOOP (Simple Concurrent OOP)** model [22] is to simplify the development of concurrent system by providing a minimal set of extensions able to add concurrency and distribution support to object-oriented programming.

The object-oriented paradigm has shown through the years to be a valid approach for software construction. It represents a consolidated way to reason, analyse, design, and implement software-based solutions. Unfortunately, distribution and concurrency still constitute complex issues to deal with in object-oriented languages. Traditional approaches for integrating concurrency with object-orientation work by explicitating concurrency (at increasingly higher levels). Instead, a different approach (embraced by SCOOP) consists of hiding concurrency, or at least introducing it in the most unobtrusive manner.

Basic concepts

The SCOOP model of concurrency is based on two concepts: processors and separateness.

A SCOOP **processor** represents a *logical* thread of control where instructions are carried out sequentially. Every object is handled by one and only one processor. SCOOP processors are abstract concepts and should not be confused with physycal processors. In fact, the idea is to decouple the high-level notion of thread of execution from the actual mapping to physical resources (often referred to as *computational vehicles*).

Within the context of the same SCOOP processor, client objects call methods on provider objects with the traditional synchronous invokation semantics. If, instead, the client object is handled by a different SCOOP processor with respect to the provider object (also called *separate object*), calls should be asynchronous and can progress at the same time with the calling context.

A **separate entity** is an entity that is defined as being potentially handled by SCOOP processor different to the processor handling the current context. As a result, method calls to that entity (also called *separate calls*) should be executed concurrently.

Design by Contract

The SCOOP model builds on the **Design by Contract (DbC)** [14] approach to software development, which promotes the use of "contracts" between software components. Contracts formally specify the obligations and benefits resulting from such an agreement. They are expressed in terms of:

- *preconditions* are required by providers and should be respected by clients
- *postconditions* should be guaranteed by providers, so that clients are assured that nothing unexpected happens once they had respected the contract preconditions
- *invariants* should always be mantained and represent a guard against incorrect states

These in turn consist of *assertions*, i.e. predicates that must be true.

The idea of the Design by Contract method is to have contracts to guide design. The main advantages of such an approach include:

- an effective documentation of software components through the explicitation of their constraints
- a systematic approach for designing correct software, together with a framework supporting this activity
- a way for dealing with errors, together with the mechanism of exceptions

The Eiffel programming language and SCOOP

SCOOP has been designed for the Eiffel programming language. The extension to Eiffel (i.e. the implementation of the SCOOP model in the language) merely consists of the addition of a keyword, `separate`.

```
1 -- Separate entity declaration
2 separate_obj : separate SomeType
3
```

```
4 -- Separate call
5 separate_obj.some_feature
```

Listing 2.6: Separate entity and separate call.

In addition, the semantics of contracts must be adjusted for concurrency.

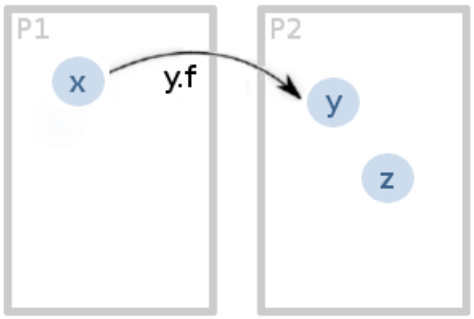

Figure 2.3: Separate call.

2.4.2 Contracts and concurrency

Preconditions The preconditions that are *not* expressed in terms of separate entities maintain the usual semantics, i.e. when a precondition is not met, an exception is raised.

Conversely, if a precondition expressed as a separate expression is not met, it becomes a *wait condition*, forcing the client to wait until the condition is satisfied, at which time the feature application is able to proceed.

Postconditions The benefit of postconditions deteriorates in concurrent settings. In fact, even though postconditions holds at the end of the call, they are not guaranteed to be satisfied when, later on, the client may need to count on them; this is because, in the meantime, other clients, by interacting with the server, can potentially invalidate the original postconditions.

Invariants Expressions with separate calls are not allowed in invariants. Thus, their semantics remain unchanged.

2.4.3 Synchronization

The approach of SCOOP for what concerns the protection of the access to shared resources is peculiar. First of all, in order to be valid a separate call requires the target to be an argument of the enclosing routine. Then, it is established that when a routine is called with separate arguments, it will be executed only when exclusive locks on all the separate objects are acquired. These two statements express SCOOP's **access control policy**. With such a scheme, for separate calls to be correct a lock for the associated separate object have been acquired. As a result, no race condition are possible.

Once the locks for the separate objects are acquired, preconditions for the routine are evaluated. If any of the separate preconditions is not met, all the locks must be released and the process must be restarted.

It has been said that a call on a separate entity should be handled asynchronously. However, in some cases the client may need to wait for the feature call to complete. As no explicit mechanisms are introduced, how is it possible to discriminate between asynchronous and synchronous calls? The idea of **wait-by-necessity** has been introduced. It states that the client should wait only if it actually needs to.

Now, a discrimination (also known as *Command-Query Separation*) has to be made between:

- *query*: attribute read or function call which returns data from the called object, and
- *command*: procedure call which modifies the state of the called object

Returning to wait-by-necessity, the result is that, when a call is a query on a separate object, then it must wait for the previous calls (be they queries or commands) on the same separate object to be completed, and will be executed with synchronous semantics. Conversely, commands which are invoked without separate arguments can proceed asynchronously.

As an example, consider the following implementation of the producer-consumer problem:

```
1 -- Bounded buffer: array-based, generic implementation
2 ---------------------- buffer.e ----------------------
3 class BUFFER[T] -- (1)
```

```
4
5  create init
6
7  feature
8
9    buffer : ARRAY [T]
10   capacity : INTEGER
11   cursor : INTEGER
12   nitems : INTEGER
13
14   init(max_size: INTEGER)
15   require
16     size_is_positive: max_size>0;
17   do
18     create buffer.make(0,max_size)
19     capacity := max_size
20     cursor := 0; nitems := 0
21   end
22
23   put(value: T)
24   require
25     buffer_not_full: nitems < capacity
26   do
27     buffer.put(value, next_pos(cursor,nitems))
28     nitems := nitems + 1;
29   end
30
31   get() : T
32   require
33     non_empty: nitems>0
34   do
35     Result := buffer.item(cursor)
36     cursor := next_pos(cursor,1)
37     nitems := nitems-1
38   end
39
40   is_full() : BOOLEAN do
41     Result := (nitems=capacity)
42   end
43
44   is_empty() : BOOLEAN do
45     Result := (nitems=0)
46   end
47
48   next_pos(k: INTEGER; i: INTEGER) : INTEGER do
```

```
49      if (k+i)>=capacity then
50        Result:= (k+i-capacity)
51      else
52        Result:=k+i
53      end
54    end
55  end
56
57  ------------------------------------------------------
58  -------------------- producer.e --------------------
59  class PRODUCER
60
61  create init
62
63  feature
64
65  buffer: separate BUFFER [INTEGER]
66
67  init(buf: separate BUFFER [INTEGER]) do
68      buffer := buf
69    end
70
71  put(buf: separate BUFFER [INTEGER]; value: INTEGER) -- (2)
72    require
73      buffer_not_full: not buf.is_full
74    do
75      buf.put(value) -- (3) (5)
76    end
77
78  produce_first_n_nums (n: INTEGER) -- (4)
79    require
80      positive_num: n>0
81    local
82      i:INTEGER
83    do
84      from i := 0
85      until i = n
86      loop
87        put(buffer, i)
88        i := i+1
89      end
90    end
91  end
92
93  ------------------------------------------------------
```

```
94  -------------------- consumer.e --------------------
95  class CONSUMER
96
97  create init
98
99  feature
100
101 buffer: separate BUFFER [INTEGER]
102
103 init(buf: separate BUFFER [INTEGER]) do
104     buffer := buf
105   end
106
107 consume(buf: separate BUFFER [INTEGER]) -- (2)
108   require
109     buffer_not_empty: not buf.is_empty
110   local
111     value:INTEGER
112     format: FORMAT_INTEGER
113   do
114     value := buf.get -- (3) (6)
115     create format.make(1)
116     print("I'm␣consuming␣elem␣"+ format.formatted(value) + "%N")
117   end
118
119 consume_n(n: INTEGER) -- (4)
120   local
121     i:INTEGER
122   do
123     from i := 0
124     until i=n
125     loop
126       consume(buffer)
127       i := i+1
128     end
129   end
130 end
```

Listing 2.7: Producer-consumer implementation in Eiffel-SCOOP.

The following facts should be noted:

1. no critical sections are defined; we can say that class `BUFFER` is unaware of concurrency (actually, classes can be defined as being

`separate`, thus defining a *separate type*, expressing a sort of "concurrency awareness")

2. in order for `PRODUCER.put` and `CONSUMER.consume` to be executed, a lock must be acquired on their actual (separate) argument and preconditions must be satisfied (i.e. the buffer must not be full and empty, respectively)

3. the call `buf.put(value)` and `buf.get` are valid because `buf` is a separate argument of the enclosing routine

4. no separate calls can be made inside `produce_first_n_nums()` and `consume_n`

5. `buf.put(value)` is a command, so it would be carried out asynchronously

6. `buf.get` is a query call, so it is carried out synchronously

2.5 The Actor model

2.5.1 Overview of the Actor model

The **Actor model** [23] is a model of concurrent computation that is built around the notion of **actor**, an *autonomous* object that acts **concurrently** and **asynchronously**, with a globally unique name and a behavior that is based on three primitives:

- `send`, that allows to send messages to other actors
- `create`, that allows the creation of new actors
- `become`, that allows the actor to update its state into another state

Within this model, software systems consists of autonomous actors that interact with one another by sending **messages** in an asynchronous fashion, following certain *patterns of interaction*. The only way for actors to communicate is through message exchange, as they do *not* share state.

Every actor owns a *mailbox* that contains incoming messages. The behavior of an actor can be considered as a loop where the actor:

1. wait for incoming messages (if the mailbox is empty)
2. remove a message from the mailbox
3. execute the behavior associated to such message

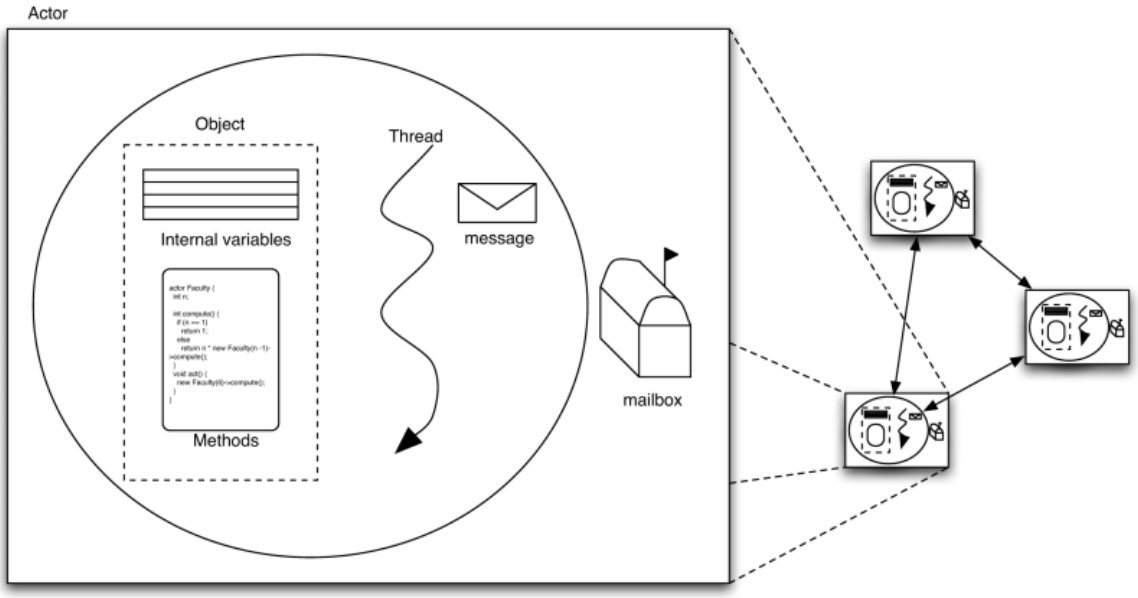

Figure 2.4: Actors.
Taken from http://werner.yellowcouch.org/Papers/chubiqcomp/index.html

Semantic properties

The standard Actor model defines a few semantic properties for actors:

1. *encapsulation*: actors do not share state; one consequence is that message passing should have call-by-value semantics
2. *atomicity of message processing*: a macro-step semantics apply, resulting in the target actors to process one message at a time
3. *fairness of execution*: guarantee of actor execution exists, so that actors do not starve

4. *fairness of delivery*: guarantee of message delivery exists, unless the recipient actor is permanently unreachable

5. *location transparency*: the actual location of an actor does not influence its name and thus the way through which messages are sent to him

    - *mobility*, which is simplified by location transparency, is essential for supporting scalability approaches (e.g. load balancing) and fault-tolerance

These properties may or may not be enforced by implementations. It has been noted that faithful but naive implementations can be highly inefficient [24]. For example, it makes sense to allow call-by-reference semantics for immutable types, as deep copying of message data can result in considerable overhead.

2.5.2 Synchronization and actor coordination

Actors do not share data, so there is no risk for data corruption due to concurrent access on write (or read/write). However, there is still the need for actor *coordination*, as not all the interactions may be correct with respect to the desired behavior of the system.

Communication is the way to achieve synchronization. Therefore, constraints upon the way in which messages are exchanged are necessary in order to exclude incorrect execution orderings. For example, consider the following synchronization mechanisms.

RPC-like messaging

Here, the actor which initiated the communication by sending a message waits for the reply of the recipient actor. It is a sort of synchronous communication, similar to classical function call or RPC.

If the sender receives a message that is not the reply it is waiting for, the processing of such message is postponed. As actors should not block indefinitely, the possibility of not receiving the reply must be taken into account, for example, by associating a timeout with the interaction.

This kind of communication is useful when the behavior of the sending actor depends upon the reply by the receiving actor, or when holding *conversations*.

Local synchronization constraints

An actor may not be able to handle all the possible messages that can be sent to it. Also, the messages that can be handled in a given situation may depend upon the current state of the actor.

The local synchronization constraints provide a way to specify which messages can be processed by an actor in a given state.

For example, consider an actor implementing a bounded buffer. It cannot accept a `put` message when the actor is in the `full` state, or a `get` message when it is `empty`.

When a message cannot be handled when it is received, the actor must decide if it has to be postponed (deferring the processing of the message when the actor is able to do it) or discarded. In the case when the latter policy is adopted, one possibility for client actors is to poll the server actor in busy waiting, which is very expensive and typically not satisfying as an approach.

Synchronizers

One limit of the local synchronization constraints is that they are, indeed, *local*; that is, they are based on the local state of a single actor. However, they are not effective for expressing the coordination constraints of a group of actors.

One proposal for multiactor coordination consists of the use of *synchronizers*, that are special actors that determine which messages can be accepted or deferred by a certain group of actors. Synchronizers define conditions that need to be met for messages in order to be accepted for processing by the group.

2.5.3 Case study: Actors in Scala/Akka

The *Scala Actors API* is defined under the `scala.actors` package. Since Scala 2.10 (released in January, 2013) such library is deprecated in favor of Akka, an actor framework which have been integrated in Scala's standard

library under the `akka.actor` package. This section briefly describes how actors work in Scala/Akka [25].

Quick glance at Scala/Akka API

Originally, actors were defined by extending the `scala.actors.Actor` trait and overriding the `act` method. It is a body-based approach, which makes reusing actors very difficult.

In Akka, **defining an actor** consists of extending the `Actor` class in the `akka.actor` package and implementing the `receive` method (**message handler** or **message loop**), which is declared as returning an object of type `PartialFunction[Any,Unit]` and typically contains multiple `case` statements that express the mapping between received messages and associated processing logic. Whereas `act` needed to loop explicitly, in Akka the loop is implemented in the library code.
*Note that both these approaches are different with respect to the standard Actor model which is based on an **implicit** reception and dispatch of messages (messages are automatically routed to the corresponding method).*

In order to **create an actor**, the method `actorOf` has to be called on an `ActorSystem` or `ActorContext` instance. In the former case, the actor is a top-level actor and is supervised by the system. In the latter case, the actor is the child of the current actor, which is also its supervisor. In fact, a **supervisor hierarchy** exists and allows the supervisors to apply a `SupervisorStrategy` to handle child termination and failure.
`actorOf` is passed a `Props` instance, which contains the options for the creation of the actor, and returns an `ActorRef` instance, which is a **reference** to the newly created actor. Such reference can be used to send messages to the actor; it can be serialized and passed through the network, always pointing to the actor in the node where it resides. Inside an actor, two `ActorRef` instances are available:

- `self`, pointing to the actor itself
- `sender`, pointing to the actor which sent the current message

Moreover, inside an actor the object `context` (of type `ActorContext`) is available and refers to the current context.

Actors communicate by **sending messages**. Two forms of communication are provided by Akka:

- Tell (Fire-And-Forget): `destActor ! msg`
- Ask (Send-And-Receive-Future): `destActor ? msg`

`ask` returns a `Future`, which is a container for the destination actor's reply. The use of `Futures` allows for avoiding blocking, for example by registering a callback through `Future.onComplete`.

The `receive` method, as we have seen, defines the initial behavior of an actor. Moreover, an actor can also **update its behavior**. This can be done by calling `context.become`, which accepts a `PartialFunction[Any,Unit]` object as an argument. Conversely, `unbecome` rewinds to the previously defined behavior.

Scala/Akka and the standard Actor model semantics

As we have seen in Section 2.5.1, the standard Actor model specifies a few semantic properties for actors. Are they enforced in Scala/Akka? Well, it turns out that:

- encapsulation is *not* enforced for actors: actors may share data and messages can be of mutable types
- the macro-step semantics for message processing is ensured
- fairness can be promoted through an adequate configuration of the system
- location transparency is supported through `ActorRef`
  - actors are movable from one node to another: both the actor itself and its reference can be serialized and moved around the distributed system

Extending actors

The Akka documentation [25] suggests two ways for specializing actors via inheritance. Both consist of extending the original `receive` implementation with one or more specific message handling clauses.

```
// the class need to be abstract because
//  'specificReceive' is unimplemented
abstract class ParentActor extends Actor {
  // 'Receive' is an alias for 'PartialFunction[Any,Unit]' type
  // the following need to be redefined in child actors
  def specificReceive: Receive

  def genericReceive: Receive = { /* generic rcv impl */ }

  // when 'specificReceive' is not defined, receive impl
  //   fallbacks to genericReceive
  def receive = specificReceive orElse genericReceive
}

class ChildActor extends ParentActor {
  override def specificReceive: Receive =
    { /* additional rcv impl */ }
}
```

Listing 2.8: Extending actors: providing a specific 'receive' implementation.

A similar approach, slightly more flexible and still based on `PartialFunction` chaining via `onElse`, is shown in the following Listing:

```
abstract class ParentActor {
  var rcvs: List[Receive] = List()

  def registerReceive(newrcv:Receive) = { rcvs = newrcv :: rcvs }
  def unregisterLastReceive() = { rcvs = rcvs.tail }

  // the following produces the 'union' of the
  //  message handling 'parts';
  // the last added part takes higher priority as parts
  //  are added at the head of the list, thus having precedence
  def receive = rcvs reduceLeft { _ orElse _ }
}

class ChildActor extends ParentActor {
  registerReceive( { case _ => print("Sw reuse thesis") } )

```

```
17    registerReceive( { case _ => print("It'll be unsubscribed") } )
18    unregisterLastReceive()
19  }
```

Listing 2.9: Extending actors: message handler subscription.

It makes use of list reduction. We have used `reduceLeft` instead of `reduce` because in the documentation for the latter is specified that "the order in which operations are performed on elements is unspecified and may be nondeterministic."

Note that nothing prohibits from providing a message handler implementation through a construction argument.

In addition to what we have just seen about actor extension, consider that:

- the Template Method design pattern naturally applies: the `receive` message handler is responsible for selecting the appropriate processing logic upon a certain message reception; such processing logic can be "templated" and easily specialized in subclasses
- `become` can be used to update an actor's behavior
- the new behavior for a becoming-actor can also be provided from the outside (it can be seen as a sort of learning or actor's mind programming) as in Listing 2.10; such a technique can be accompanied by mixing traits into objects at runtime

```
1  class ExtendMsg(val rcv: PartialFunction[Any,Unit]){ }
2
3  class MyActor extends Actor {
4    var rcvs: List[Receive] = List()
5
6    def registerReceive(newrcv: Receive) = { rcvs = newrcv::rcvs }
7
8    def unregisterReceive() = { rcvs = rcvs.tail }
9
10   def genericMsgHandler: Receive = {
11     case ex: ExtendMsg => {
12       registerReceive(ex.rcv) // register new behavior
13       context.become(receive) // extend current actor behavior
14     }
```

```
15      /* other cases */
16    }
17
18    registerReceive(genericMsgHandler)
19
20    def receive = rcvs reduceLeft { _ orElse _ }
21  }
22
23  /****** SYSTEM EXECUTION ******/
24  val system = ActorSystem("mysystem")
25
26  val myactor = system.actorOf(Props(new MyActor), "myactor")
27
28  def newbehavior: Actor.Receive = {
29    case "newmsg" => print("new behavior")
30    /* other cases */
31  }
32  myactor ! new ExtendMsg( newbehavior )
33  myactor ! "newmsg"
```

Listing 2.10: Extending an actor message loop from the outside.

A code example: the Producer-Consumer problem

The following listing represents a basic implementation of the Producer-Consumer problem and shows Scala/Akka actors in action.

```
1   import akka.actor._
2   import akka.pattern.ask
3   import akka.util.Timeout
4   import scala.concurrent.duration._ // for implicit timeout
5   import scala.concurrent.Future
6   import scala.language.postfixOps // for postfix ops, e.g. seconds
7   import system.dispatcher // for execution context
8   import java.util.concurrent.Executors // for futures
9
10  /****** MESSAGES ******/
11  class PutMsg(val item: Int) { }
12  class ProduceMsg(val value: Int){ }
13  class ConsumeNMsg(val n: Int){ }
14
15  /****** BOUNDED BUFFER ACTOR ******/
16  class BoundedBuffer(capacity:Int) extends Actor{
17    val buffer  = new Array[Int](capacity)
```

```
18    var start    = 0
19    var end      = 0
20    var nitems   = 0
21
22    def receive = {
23      case p:PutMsg  if nitems<capacity => put(p.item)
24      case "get"     if nitems>0 => sender ! get()
25      case "isEmpty" => sender ! (nitems==0)
26      case "isFull"  => sender ! (nitems==capacity)
27    }
28
29    def get():Any = {
30      val postoget  = start
31      start         = nextPos(start)
32      nitems        = nitems-1
33      return buffer(postoget)
34    }
35
36    def put(value:Int) = {
37      buffer(end) = value
38      nitems      = nitems+1
39      end         = nextPos(end)
40    }
41
42    def nextPos(i:Int):Int = {
43      return if (i+1 == capacity) 0 else i+1
44    }
45  }
46
47  /****** PRODUCER ACTOR ******/
48  class Producer(val buffer: ActorRef) extends Actor {
49    val random = new scala.util.Random(System.currentTimeMillis)
50    val MAX_RANDOM = 9999
51
52    def produce(value: Int) = {
53      buffer ! new PutMsg(value)
54    }
55
56    def produce_random(max: Int) = {
57      buffer ! new PutMsg(random.nextInt(max))
58    }
59
60    def receive = {
61      case p:ProduceMsg => produce(p.value)
62      case "produce_random" => produce_random(MAX_RANDOM)
```

```
63    }
64  }
65
66  /****** CONSUMER ACTOR ******/
67  class Consumer(val buffer: ActorRef) extends Actor {
68    def consume() = {
69      val gotval:Future[Int] =
70        buffer.ask("get")(5 seconds).mapTo[Int]
71      gotval onSuccess {
72        case res => processing_logic(res)
73      }
74      gotval onFailure {
75        case e => print("\nCannot␣process␣it.␣Exception:␣" + e)
76      }
77    }
78
79    def consume_n(n: Int){
80      for(i <- 1 to n) consume()
81    }
82
83    def processing_logic(value: Int) = {
84      print("{Processing␣" + value + "}")
85    }
86
87    def receive = {
88      case "consume" => consume()
89      case c:ConsumeNMsg => consume_n(c.n)
90    }
91  }
92
93  /****** SYSTEM EXECUTION ******/
94  val system = ActorSystem("mysystem")
95
96  val CAPACITY = 100
97
98  val mybuffer =
99    system.actorOf(Props(new BoundedBuffer(CAPACITY)), "mybbuffer")
100
101 val myproducer =
102   system.actorOf(Props(new Producer(mybuffer)), "myproducer")
103
104 val myconsumer =
105   system.actorOf(Props(new Consumer(mybuffer)), "myconsumer")
106
107 myproducer ! "produce_random"
```

```
108 myproducer ! new ProduceMsg(100)
109
110 myconsumer ! new ConsumeNMsg(2)
111
112 system.shutdown
```

Listing 2.11: The Producer-Consumer problem in Scala/Akka.

2.6 Inheritance anomaly

2.6.1 What and why

The term **inheritance anomaly** [26] refers to a class of issues that emerge from the use of inheritance in object-oriented concurrent programming languages (OOCPLs).

A language may keep the notion of object and thread separate. Alternatively, the process of unifying objects and concurrency brings to the notion of *active object*. Objects become concurrency units encapsulating a thread of control. This is what happens, for example, by merging objects with actors.

Be the objects active or separated from the concurrency building blocks, synchronization is needed in order to enforce coordination and safe access to shared data.

However, usually that synchronization code cannot be kept completely separate from the code expressing the logic or behavior of objects. Such an imperfect separation is what allows inheritance anomalies to happen. In fact, it turns out that, in these cases, **synchronization code cannot be effectively inherited without requiring extensive class redefinitions**.

The inheritance mechanism works in a (usually) static, structural perspective. Conversely, the synchronization code expresses dynamic, functional constraints. This difference may justify (in part) the **semantic conflict** between inheritance and concurrency that have been noted through the years. Such interference is what makes it so difficult to inherit synchronization code and preserve the inherited behavior without breaking encapsulation.

2.6.2 Kinds of anomalies

A specific inheritance anomaly may or may not happen depending upon which synchronization primitives are used and how they are applied collectively (which is referred to as a *synchronization scheme*). So, every language exhibits an inclination to suffer of inheritance anomaly that is determined by its concurrency facilities.

The occurrences of inheritance anomaly can be divided into three main categories [27]:

History-sensitiveness of acceptable states

The inheritance anomaly may occur when the extension consists of one or more methods guarded against history-sensitive constraints. A history-sensitive constraint is a condition that regulates if a method is enabled or not which depends on the past history of the object state. A typical example is the addition in the child class of a `BoundedBuffer` class of a method `gget` which "cannot be called after `get`". Suppose that `get` and `put` contains the state transition logic as below in Listing 2.12. Now, the new constraint is clearly history-sensitive and it can be shown that it requires `get` and `put` to be redefined in order to keep trace of that "history" (for example, by using a boolean `after_get` state variable).

```
1  class BoundedBuffer(capacity:Int) extends Actor {
2    /* ... IMPL cut for space ... */
3
4    def empty_state : PartialFunction[Any,Unit] = {
5      case p:PutMsg   => put(p.item)
6      case "isEmpty"  => sender ! true
7      case "isFull"   => sender ! false
8    }
9
10   def full_state : PartialFunction[Any,Unit] = {
11     case "get"      => sender ! get()
12     case "isEmpty"  => sender ! false
13     case "isFull"   => sender ! true
14   }
15
16   def partial_state : PartialFunction[Any,Unit] = {
```

```
17      case p:PutMsg   => put(p.item)
18      case "get"      => sender ! get()
19      case "isEmpty"  => sender ! false
20      case "isFull"   => sender ! false
21    }
22
23    def receive = { empty_state }
24
25    def get():Any = {
26      /* impl */
27
28      if(nitems==0) context.become(empty_state)
29      if(nitems==capacity-1) context.become(partial_state)
30
31      return result
32    }
33
34    def put(value:Int) = {
35      /* impl */
36
37      if(nitems==capacity) context.become(full_state)
38      if(nitems==1) context.become(partial_state)
39    }
40  }
```

Listing 2.12: Alternative (naive) bounded-buffer implementation in Scala/Akka.

Partitioning of states

The need for redefinitions may take place when the extension forces the original set of states to be further partitioned. A characteristic example consists of specializing `BoundedBuffer` with a `get2` method that behaves like `get` but returns two items of the buffer. This addition causes a partition of the buffer state set. In fact, in the `partial` state the buffer can contain from 1 to N-1 items; however, `get2` must not be enabled in the case that only a single element exists in the buffer. So, a `one_element` state should be considered, but it requires both `get` and `put` to be redefined in order to adjust the state transitions.

Modification of acceptable states

Suppose of extending the `BoundedBuffer` by mixing into it a `Lock` trait that allows the buffer to be locked or unlocked by calling `lock` and `unlock`, respectively. Now, you see that it is not possible to use `put` when the the buffer is `locked`, even if the buffer is `empty`. In fact, the `locked` and `unlocked` states are orthogonal with respect the other states. Thus, such a mix-in has modified the acceptable states. Depending on how the method enabling logic is expressed, different types of actions may be required. Using the approach of Listing 2.12, for each of the original states, a locked/unlocked version should be provided. So, we can have the `locked_empty` and `unlocked_empty` states. Such changes must be reflected in `put` and `get` implementation. Ideally, these methods should contain only the behavior needed to put/get an element into/from the buffer; instead, we see that the coexistence with the state transition logic results into the necessity of redefinitions.

2.6.3 Language-level mechanisms and inheritance anomaly

We have seen the three main classes of inheritance anomalies. A programming language, depending on the constructs it provides, may be immune to one or more kinds of anomalies. For example, Eiffel (and SCOOP-based approaches) do not suffer from anomalies related to active objects, however they are not totally resistant (e.g. they do suffer from anomalies due to history-sensitiveness of states [27]). Here, we consider a few synchronization schemes and analyze their position with respect to the aforementioned issues.

Bodies

The *body* is an object method with its own thread of control. Examples of bodies are the `run` method in Java `Threads`, the `act` method of the `Actor` trait in the (old) Scala Actors Library, and the `task body` in Ada. The problem with bodies is that they contain both the concurrent object behavior and the synchronization code. The only chance of clean reuse is to specialize the methods that are called inside the body. In all the other cases, it is evident that the body needs to be totally rewritten. The impedance to reuse is extreme.

Explicit message reception

While the standard Actor model establishes that message reception is implicit as well as the corresponding method invokation, other Actor-based implementations may support a different semantics. For example, the Scala Actors Library supports the explicit reception of messages via the `receive` construct (that may be called `select/accept` in other languages), which works similarly to the `receive` message handler method in Scala/Akka.

In many cases, adding new methods on the class would force the redefinition of the entire message handler in order to take them into account [26]. Actors written in Scala/Akka, however, are not susceptible to such a catastrophic eventuality because new message handling clauses can be aggregated (as in Listing 2.9), and because methods can be overridden, so the original processing logic (if expressed via the Template Method pattern) can be easily modified.

Guards

Guards can be used to control the method enabling, postponing the method execution until a certain condition holds. Guards may be specified on methods (e.g. when message reception is implicit) or together with `receive` clauses as in Scala/Akka.

Of course, when the acceptable states are modified (e.g. in the case of the `Lock` trait mixed-into the buffer), guards need to be adjusted accordingly. Thus, the question is: does the language allow to adjust the guards without requiring the method redefinition? If it is not possible, it is a serious inheritance anomaly.

Now, consider the bounded buffer and the case of the addition of `gget`, method that cannot be executed after `get`. `gget` would normally be guarded against a `afterGet` boolean variable. This variable must be updated. Consequently, `get` and `put` must be redefined (maybe not totally in fortunate circumstances).

2.6.4 Case study: Inheritance anomaly in Scala/Akka

Here, we try to implement the extensions for the bounded buffer that has been described in Section 2.6.2. The aim is to evaluate how much effort

is necessary to implement such specialized buffers with the concurrency facilities provided by Scala/Akka.

History-sensitiveness of states: gget

In Listing 3.4 we see that `get` and `put` need to be redefined. Luckily, the original implementation can be partially reused. However, it is not optimal.

```
1  abstract class BoundedBuffer extends Actor{
2    /* ... IMPL cut for space ... */
3
4    def messageHandler:Receive = { /* ... */ }
5    def moreMsgHandler : Receive
6    def receive = moreMsgHandler orElse messageHandler
7
8    def get():Int = { /* ... */  }
9    def put(value:Int) = { /* ... */   }
10 }
11
12 class HistoryBoundedBuffer extends BoundedBuffer {
13   var lastIsGet = false;
14
15   override def moreMsgHandler() : Receive = {
16     case "gget" if !lastIsGet => /* ... */
17   }
18
19   override def get():Int = {
20     val result = super.get()
21     lastIsGet = true
22     result
23   }
24
25   override def put(value:Int) = {
26     super.put(value)
27     lastIsGet = false
28   }
29 }
```

Listing 2.13: Extending the bounded buffer: implementation of 'gget' in Scala/Akka.

A different approach is possible: the addition of Template methods `afterPut` and `afterGet`. They can be redefined in the subclass in order to contain the logic needed to keep track of the history-sensitive informa-

tion. However, while it is possible, it is unrealistic: why the designers of `BoundedBuffer` would have done this? Moreover, what if such an action needed to be performed on more than two methods? In summary, this alternative is far from being satisfying. Nevertheless, it points out a suitable technique that may be useful in similar cases: aspect-oriented programming.

Partitioning of states: get2

The buffer extension with `get2` does not suffer from inheritance anomaly in Scala/Akka, as the following code demonstrate:

```
1  // the parent 'BoundedBuffer' code is the same as
2  //  in the previous listing
3  class PBoundedBuffer extends BoundedBuffer {
4    override def moreMsgHandler(): PartialFunction[Any,Unit] = {
5      case "get2" if nitems>1 => sender ! get2();
6    }
7
8    def get2(): Tuple2[Int,Int] = {
9      /* get first and second value */
10     return (first, second)
11   }
12 }
```

Listing 2.14: Extending the bounded buffer: implementation of 'get2' in Scala/Akka.

Modification of acceptable states: Lock mixin

In this case, the message handler `receive` must be entirely redefined. However, no changes to `get` and `put` are necessary.

```
1  trait Lock {
2    var isLocked = false
3    def lock = { isLocked = true }
4    def unlock = { isLocked = false }
5  }
6
7  class LockableBoundedBuffer extends BoundedBuffer with Lock {
8
9    override def receive = {
10     case p:PutMsg  if nitems<capacity && !isLocked => put(p.item)
11     case "get"     if nitems>0 && !isLocked => sender ! get()
```

```
12      case "lock"   => lock
13      case "unlock" => unlock
14    }
15
16  }
```

Listing 2.15: Extending the bounded buffer: implementation of a lockable buffer in Scala/Akka.

Another approach is to factor guard conditions into methods, as in the following Listing:

```
1  abstract class BoundedBuffer extends Actor {
2    /* ... IMPL cut for space ... */
3
4    def messageHandler:Receive = {
5      case p:PutMsg  if putGuard => put(p.item)
6      case "get"     if getGuard => sender ! get()
7      /* other cases */
8    }
9
10   def putGuard:Boolean = { nitems < capacity }
11   def getGuard:Boolean = { nitems > 0 }
12
13   def moreMsgHandler : Receive
14   def receive = moreMsgHandler orElse messageHandler
15 }
16
17 class LockableBoundedBuffer extends BoundedBuffer with Lock {
18
19   def moreMsgHandler: Receive = {
20     case "lock"   => lock
21     case "unlock" => unlock
22   }
23
24   override def putGuard:Boolean = { super.putGuard && !isLocked }
25   override def getGuard:Boolean = { super.getGuard && !isLocked }
26 }
```

Listing 2.16: Factorizing guard conditions into methods in Scala/Akka.

Such a technique may also makes sense as a general approach for developing actor classes in Scala/Akka.

2.6.5 Solving the inheritance anomaly

The inheritance anomaly has a huge impact when the synchronization scheme of a language does not allow for a sufficient separation between the synchronization logic and the object behavior. This consideration shows that the **separation of concerns** can be crucial for avoiding anomalies related to the conflict between concurrency and inheritance. As a result, all the techniques the foster the separation of concerns can potentially contribute to solve these issues. For example, aspect-oriented programming can be valuable and some proposals based on it have already been advanced [27].

Summary

The key points of this chapter are:

- concurrency and distribution are two key issues in software design and development
- concurrency, by removing the constraint of fixed execution order that regulates sequential programs, entails an unpredictability that need to be managed through opportune synchronization logic
- reusing concurrent units should ensure that both their original behavior and semantic (concurrency-related) constraints are preserved in the new context
- the thread is the smallest schedulable unit in a system; it shares the same address space with the other threads in the process
- threads typically share data, which - if it is not read-only - must be accessed in a mutually exclusive fashion; programming languages provide synchronization primitives both for thread coordination and for avoiding race conditions on shared resources
- threads do not contribute enough to the lowering of the conceptual gap, and issues such as race conditions and deadlocks make it difficult to write both reliable (thread-safe) and efficient concurrent code
- tasks in Ada provide a conceptual separation between the work to be done and the underlying concurrency mechanism; task definition requires an explicit separation between communication interface (entries) and task implementation (task body)
- tasks communicate directly with rendez-vous or indirectly via (shared) protected objects
- SCOOP is a concurrency model which aims to empower the OOP with a minimal extension for concurrency, according to its motto "concurrent software development made easy"
- the SCOOP's model builts on the concepts of processors and separateness and is based upon the Design by Contract technique

- SCOOP's realization consists of: an extended semantics for contracts in concurrent settings, the definition of what a valid separate call is (Separateness Consistency Rules), mutual exclusion through an appropriate access control policy on separate objects, the determination of the synchronous or asynchronous nature of a separate call based on wait-by-necessity
- actors are autonomous objects executing concurrently and asynchronously, with an associated mailbox, which communicate between one another only by exchanging messages (no shared data)
- each actor can send/receive messages, create another actor, and update its local state
- the standard Actor model defines some semantic properties: encapsulation, atomicity, fairness, location transparency
- the term "inheritance anomaly" refers to a class of issues originated by a semantic conflict between inheritance and concurrency and by an imperfect separation between object behavior and synchronization logic
- the result of the inheritance anomaly is that a class cannot be inherited without causing the need of considerable redefinitions; the synchronization logic as expressed does not hold, thus breaking encapsulation
- the occurrence of the inheritance anomaly depends on the synchronization schemes which are used; three broad classes of issues are commonly recognized: history-sensitiveness of acceptable states, partitioning of states, modification of acceptable states

Chapter 3

Agent-Oriented Programming and Reuse

The last chapter covered concurrency and reuse. We have come across different concurrency models, each with its own set of concurrency mechanisms and abstractions. We can see a trend of increasingly higher abstractions finally aimed at reducing the conceptual gap inherent to the construction of concurrent applications: from threads to tasks, to end with SCOOP separateness and actors.

This chapter includes the Agent-Oriented Programming paradigm to the discussion, taking the simpAL programming language as a reference implementation. Agents can be seen as abstractions which conceptually extend the notion of actor and concurrent object. After an overview of agents and related abstractions as they are provided by the simpAL programming model, the aim is to discuss what is the meaning of reuse in the context of multi-agent systems.

3.1 Agents, agent-oriented programming and `simpAL`

Agents [28] are subject of study in multiple areas of computer science research. Two important areas are:

- *(Distributed) Artificial Intelligence*, where agents were introduced and are primarily studied with respect to their "intelligent capabilities,"

with focus on machine learning and reasoning

- *Software engineering*, which sees agents as entities that can contribute to the modelling of modern software systems where concurrency, distribution, and interaction represent increasingly important issues

Within the latter point of view, the *Agent-Oriented Programming (AOP)* paradigm has been proposed as an approach for (general-purpose) software development where software systems consist of a set of interacting agents. Such systems can be called *multi-agent systems*.

In this thesis, we take the simpAL programming language [29] as a representative of the agent-oriented paradigm and as a reference of the concepts that will be described hereafter.

3.1.1 Agents: definition and reason

No single authoritative definition exists for the term "agent."

As we intend it, an **agent** is both a software abstraction and a computational entity which is characterized by specific *human-inspired* properties and capabilities. In particular, an agent is:

- **situated** in an *environment* which can be perceived and manipulated by the agent (at the same time with other agents situated in the same environment)
- **reactive**, i.e. he is able to react to changes in the environment and to the messages that are sent to him by the other agents
- **autonomous** and **pro-active**, i.e. he does not only react to perception but also exhibits an autonomous behavior aimed at the accomplishment of certain tasks that have been assigned to it
- possibly **social**, i.e. he may have a social ability which makes he able to collaborate with other agents in order to meet his objectives; collaboration is based on communication, which can be direct (by sending messages) or indirect (by manipulation of shared artifacts)

The emphasis is on the attitude of agents to continuously pro-act so as to fulfill their assigned duties.

Agents can be seen as an extension of the Actor model, which has been covered in Chapter 2. Actors were defined as autonomous objects that act concurrently and that communicate between one another by exchanging messages in an asynchronous fashion. Thus, actors are both autonomous (as they encapsulate a thread of control) and reactive (as they act on the basis of the received messages), but they are neither pro-active nor specified as being situated in an environment.

Moreover, agents are defined upon an *abstraction layer that is inspired by the way in which human societies work*. So, not just objects or actors as in traditional application models, but is provided an entire set of concepts aimed at the description of a human-like "agent world."

In summary, the agent-oriented paradigm introduces a new model, with new abstractions, that embraces concurrency and decentralization of control from the beginning; that is, multi-agent systems are *inherently* concurrent and distributed. Therefore, the agent-oriented programming represents an attempt to provide a solid foundation for developing modern applications, where "modernity" is characterized by an evolution of computational infrastructure which results in increasingly higher levels of parallelism, interconnection, and ubiquitousness.

3.1.2 The simpAL model: a high-level overview

In simpAL, programs are *organizations* of agents. A program, or multi-agent system, consists of multiple agents – playing certain *roles* –, which autonomously act in order to accomplish the *tasks* that have been assigned to them. The agents live in a common *environment* that is organized into one or more *workspaces*. Agents can interact between one another by sending messages, or requests. They can also make use of tools and resources, which are called *artifacts*.

Such a scenario does resemble real-world work organizations. Note that the similarity is not only about the structure, but common elements exist also in the dynamics. In fact, for example:

- our world is by nature concurrent and asynchronous, and the same is perceived by agents
- certain objects (e.g. a spoon, a pen..) cannot be effectively used by

two or more people, and the same is for artifacts

- when people work together, they typically reside in the same place, where all the equipment they need is available; such a container for work groups and tools is represented by the workspace

- companies and industries have one or more branches and/or offices which allow for work distribution; the organization, environment, and workspace concepts reflect such an arrangement

Moreover, agents are very similar to human workers. While an agent may not get bored at work, he have a job to do which consists of completing the tasks that are assigned to him. In order to do so, he cooperates with other agents and, while doing one or more tasks (likewise to people, which support multi-tasking!), he perceives changes that occur in the environment and acts accordingly.

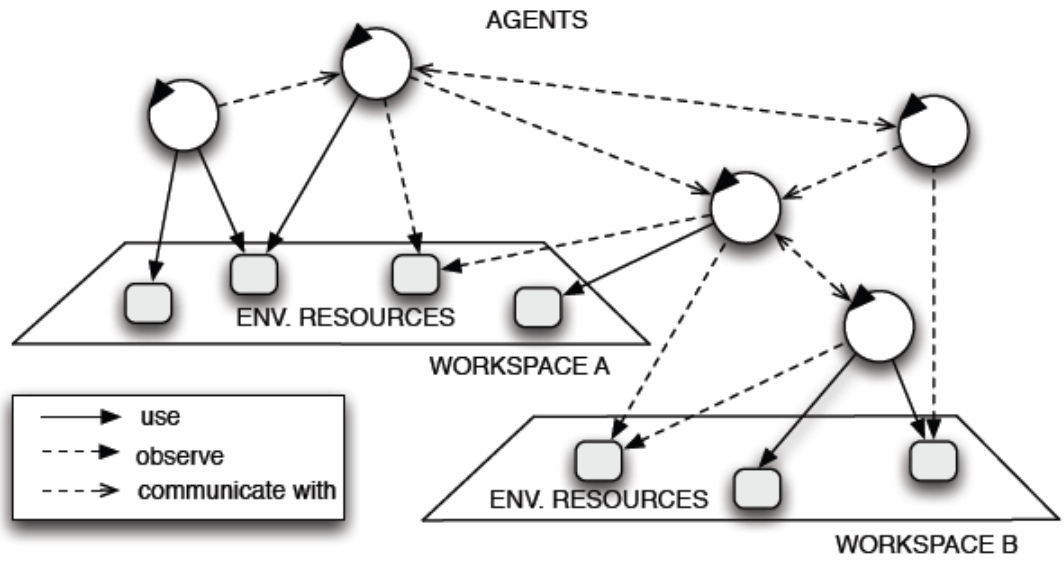

Figure 3.1: A world with agents.

3.1.3 Agent behavior

Agents have an autonomous **goal-oriented** behavior. It means that, for an agent, all the actions are finally directed towards the accomplishment of some objectives. So, agents own an implicit notion of **task**, which is the primary element upon which their behavior can be described and defined. The knowledge about how a certain task can be fulfilled is encapsulated by the concept of **plan**. A plan is a module that specifies the logic – in terms of actions and interactions to be performed – that allow the agent to progress in the task and ultimately finish it (hopefully).

simpAL is based on the *Belief-Desire-Intention* model [30], which provide a foundation for the reasoning capabilities of intelligent agents.

The BDI model, in addition to defining how the rational process works in agents, ultimately allows the separation of two activities:

- the decision of what to do (the creation of an intention)
- the actual execution of the action (the expression of an intention)

Such separation is crucial for the implementation of agent behavior. According to this model, in order for an action to be executed it must first be established – based on the current knowledge (**belief**) and the desired consequences (**desire**) – and become an **intention**.

The beliefs for an agent can be compared, for humans, to the facts that reside in memory, even though – of course – agents are not affected by issues related to memory retention and retrieval. The summa of all the beliefs represents the "conscious" knowledge of the agent, which may be learned at some point in the past (with respect to the "birth" of the system) or be the result of recent experiences, for example drawn by speaking with other agents or by looking at the environment.

Belief, desire, and intention can be seen as the *mental states* of an agent while he is reasoning:

- belief – consider all your knowledge (*information state*)
- desire – consider what state of affairs you want (*motivational state*)
- intention – commit yourself for doing an action that allows to reach your desired state of affairs (*deliberative state*)

There may be a virtually infinite number of "hows", of ways for a desire to be satisfied. The actual process of choosing the actions that are aimed at turning the desire into facts is known as *means-end reasoning*. In the context of simpAL, it just turns to the application of appropriate criteria for the selection of a plan from a set of available plans.

On this basis, the behavior of an agent can can be thought as a *non-blocking* execution cycle (also known as **control loop**) where the following stages are carried on in a sequential fashion:

- **sense** – The agent perceives the world around him: a change in the environment, an update on the state of an artifact of interest, a message from another agent. Such perception is automatically reflected on the knowledge of the agent (i.e. its beliefs). All the (new) perceptible information can be seen as external events that are enqueued on a queue associated to the agent; such events are processed in this stage, and the internal state of the agent is updated accordingly.

- **plan** – Based on local state and intentions, the actions to be performed are selected; moreover, if new tasks have been assigned to the agent since the last plan stage, a plan (i.e. the actions to be executed in order to accomplish the task) must be chosen for each of them and must be added to the current set of intentions.

- **act** – The actions that were selected in the plan stage are performed.

Again, note that the human behavior (decision making process followed by action) can be reasonably represented with *SENSE-PLAN-ACT* cycles: the five senses perceive the world around us and the brain elaborates an "execution strategy" that is carried on through the capabilities of the human body. Also, such an approach has been used in robotics; for example, a robot may work with sensors providing inputs, a computational unit which elaborates these inputs and produces a plan of actions and movements, and actuators that operate on the environment.

3.1.4 Environment, workspaces and artifacts

Agents live in an **environment**. They are *situated* entities that encapsulate control and act in a working environment that supports them for the

satisfaction of their goals.

At the beginning of the studies in the field of multi-agent systems, the environment actually corresponded to the deployment context (as it is typically seen in traditional software development). It was not considered yet as a significant abstraction for the design of systems of agents. At the present time, there is a general agreement about such view which suggests to treat the environment as a first-class concept in multi-agent systems [31], accompanied by the realization of the its importance as an infrastructural pillar [32] where several system-wide functions can be provided.

The environment serves two main related purposes:

1. encapsulation of a significant part of the system's functionality, such as core services, basic mechanisms, distribution support, and so on
2. provision of environment abstractions representing functions and services for agents

Thus, by factoring a relevant part of the system logic in the environment, agents and the other abstractions can be accounted for less responsibilities. Moreover, the environment has the role of a common *context* for agents, which can be used to model aspects that span the entire organization.

Agents, in order to be able to complete the tasks that have been assigned to them, often require a *working* environment, i.e. an environment that enables and supports their work. This is especially true for cooperative work, where two or more agents collaborate – each with its own set of capabilities and tasks to do – in order to reach a common goal (of which they may also be unaware).

As a craftsman requires both raw matter and utensils in order to produce its creation, the same may be for agents. For this purpose, simpAL is based on the *Agents&Artifacts (A&A)* model [33], which provides a conceptual foundation for the modelling and definition of working environments.

According to the A&A model, a working environment is organized into workspaces which contains agents and artifacts. The main concept that has been introduced is that of **artifact**. The artifact abstraction is voluntarily general so that it could be used to represent different types of artifacts. Two main classes of artifact exist:

1. *tools* – provide functions, extended capabilities that enable agents to do some actions; for example a megaphone or a calculator

2. *resources* – encapsulate information that may be updated and observed by agents; for example a blackboard or a thermometer

As it would be a very chaotic world if people did not *organize* themshelf and their resources using spatial critera, agents and artifacts should not be sparse in the MAS "world". **Workspaces** come to the rescue. They are logical containers for agents and artifacts, and allow to model the notion of locality together with the topology of the environment.

The set of agents and artifacts in a workspace is *dynamic*. The same happens in real-world workplaces: employees may change job or be fired, new ones may be hired, the machinery may be enlarged with new tools, and so on.

Artifacts are not autonomous, they are passive and can only be *used* by agents. For this purpose, artifacts provide a **usage interface** which specifies the manner in which agents can make use of them. The usage interface also defines the external quality af an artifact. Artifacts should be highly usable and ergonomic, in parallel with the need of human workers to have instruments which satisfy (at least basic) interaction design principles.

Ergonomic arguments also apply to workspaces and environment, and refer to the ease with which tasks can be fulfilled through an efficient communication and an effective and coordinated use of artifacts.

The usage interface of an artifact specifies:

- a set of *operations*, aimed at the execution of some functions (e.g. writing on a blackboard with a piece of chalk)

- a set of *observable properties*, so that an agent can keep track of states (e.g. the temperature of a thermometer) and events (e.g. a mail is arrived at the post office)

Definitely, the actions that an agent can do are of three types. He can update its internal state (*internal action*), send a message or a request to another actor (*communicative action*), and perform an action on artifacts (*practical/usage action*) including the possibility to create and destroy them.

3.2 The simpAL language and platform

simpAL is a platform that includes both a programming language and tools which provide support for the creation of multi-agent systems. At the present, the simpAL platform equips the developers with:

- an Eclipse-based IDE with a simpAL perspective
- a compiler, based on the open-source Xtext[1] framework
- a runtime for execution of agent-based programs, with concurrency and distribution support coherently with the semantics of the agent conceptual framework

The simpAL programming language embeds a subset of Java for the object-oriented support, in order to be able to use objects as values (e.g. for beliefs) and reusing existing functionality that does not conflict with the new agent paradigm.

Certain principles are embraced by the simpAL language:

- separation of concern – in particular, the separation between interface and implementation is extensively fostered
- easy of use – through broad use of syntactic sugar to make common patterns and interaction quick and effortless to express
- consistency between conceptual and programming model

simpAL is currently under definition and development.

3.2.1 The simpAL language: an overview

Agent interface (agent types)

An agent, as we have seen, is a computational entity that encapsulates control and is characterized by a pro-active goal-oriented behavior. As a consequence, the computational model of agents must build upon the notions of belief, task, and plan. In simpAL, each agent is internally composed by the following elements:

[1]It is a framework that allows developers to build (domain-specific) languages: http://www.eclipse.org/Xtext/

- *belief-base* – It logically contains all the beliefs, thus representing the knowledge base of the agent.
- *plan-library* – Here reside all the plans, i.e. the practical competence that specifies how tasks can be done.
- *intentions* – It is the set with all the on-going plans that will be executed in the `act` stage of control loop. During the `plan` stage, the plans chosen for new tasks are added to this data structure.
- *event queue* – It is where the incoming external events (such as messages from other actors or state changes of observed artifacts) are enqueued. This queue is checked for new events during the `sense` stage.

The definition of an agent's behavior includes actions that implicitly manipulate these data structures, so it is useful to know them.

Now, how can agents be defined? How can they be created?
In simpAL, agents can be created by instantiating the associated *agent type*. An agent type is identified by a **role**, which consists of the definition of any number of **task type**s. So, the type of an agent depends on the tasks that the agent is able to do inside the organization.

```
1  role Producer {
2
3    task Booting { }
4
5    task Producing {
6      input-params {
7        numInitialItemsToProduce: int
8        bufToUse: Buffer
9      }
10
11     understands {
12       newItemsToProduce: int;
13     }
14   }
15
16 }
```

Listing 3.1: Agent type definition in simpAL.

For example, agents of the agent type `Producer` are able to perform tasks of task type `Producing`, that is, they approach the problem of producing items on a given buffer by turning into actions their encoded expertise. However, it does not mean that a producer will actually be able to produce items on the buffer.

A role specifies an interface for agents of that type, defined on the basis of the capabilities that the role itself exposes. As an abstraction, the role may be associated to the notion of profession. For example, the "teacher" occupation is characterized by the ability to teach.

Roles, similarly to Java interfaces, support inheritance through the `extends` keyword. Coherently with the specialization in professions, a role can extend another role to provide an increment in terms of supported tasks (i.e. in terms of capabilities or expertise).

Task types are defined using the `task` construct, which accepts a block to contain attributes that describe the task, for example:

- `input-params`, to specify the inputs for the task
- `output-params`, to specify the outputs returned by the task
- `understands`, to indicate which messages can be sent to the agent who is doing the task, and so on.

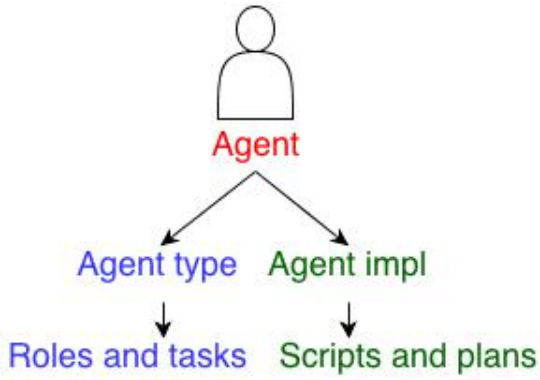

Figure 3.2: SimpAL agent.

Agent implementation

The implementation of an agent type (i.e. of a role) is encapsulated in a module that is called a **script**. Inside a script both beliefs and plans can be defined; such definitions will be available into the belief-base and plan-library data structures of the agent.

An agent implementation must contain a plan for each task declared in the implementing role. Plans contains the procedural knowledge needed to perform a certain type of task. A **plan definition** consists of:

- the target task type
- an optional context
- the body of the plan, that is, an action rule block

An **action rule block** encapsulates the behavior aimed at the accomplishment of tasks of the relative task type. It can specify local beliefs and a set of *action rules*. Such blocks can be quite complex, as they may need to execute both autonomous and reactive behavior. The language is intended to provide sufficient expressiveness to model various patterns of interaction and action-oriented behavior. Moreover, some syntactic sugar is provided to simplify common schemes of behavior.

Examples of features provided by simpAL to support the programming of plans include:

- several attributes affecting the execution of action rule blocks, for example:
  - `#hard-block` and `#soft-block` for, respectively, non-interruptible and interruptible action blocks
  - `#completed-when` to provide a condition for action block completion
  - `#to-be-repeated` for action blocks with iterated behavior
  - and so on
- sequences for serial execution of a set of actions
- action rule block nesting

- pre-defined internal actions, for example
  - actions for manipulation of tasks: `new-task`, `assign-task`, `do-task`, etc..
  - actions for communication, such as `tell`
  - actions for creating a `new-artifact` or a `new-agent`
  - and so on
- conventions for semantic defaults (e.g. reaction blocks are by default hard blocks)

Note that certain pre-defined actions (e.g. those for the creation of new artifacts or agents) are actually mapped to operations on "special" artifacts which implement a part of system functionality.

As an example of agent implementation consider the following listing:

```
1  agent-script MyProducer implements Producer in ProdConsModel {
2    itemMaker: ItemMaker // POJO
3    testing: boolean
4
5    plan-for Booting {
6      new-artifact ACMEItemMaker() ref: itemMaker
7      testing = false
8    }
9
10   plan-for Producing context: !testing {
11     #completed-when: is-done jobDone || is-done stopNotified
12     #using:  console@main, gui@main
13
14     noMoreItemsToProduce: boolean = false
15     nProduced: int = 0
16     nItemsToProd: int = numInitialItemsToProduce
17
18     println(msg: "num␣items␣to␣produce:␣"+nItemsToProd);
19     {
20       #to-be-rep-until: nProduced >= nItemsToProd || stopPressed
21       #using: itemMaker, buffer
22
23       newItem: acme.Item
24
25       makeItem(item: newItem);
26       put(item: newItem) on buffer;
```

```
27        nProduced = nProduced + 1
28      };
29      println(msg: "job␣done") #act: jobDone
30
31      when changed stopPressed in gui@main => {
32        println(msg:"stopped.")
33      } #act: stopNotified
34
35      every-time told newItemsToProduce => {
36        nItemsToProd = nItemsToProd + this-task.newItemsToProduce
37      }
38    }
39
40    plan-for Producing context: testing {
41      #using: console@main
42      println(msg: "this␣is␣a␣test")
43    } // multiple plans for same task can be provided
44  }
```

Listing 3.2: Agent type implementation in simpAL.

Finally it should be noted that, actually, scripts does not conceptually represent an agent implementation. Instead, they are modules that encapsulate the expertise needed to perform the tasks for which plans are defined. They can be seen as educational programs which endow agents with the knowledge and the capabilities that make them operative (in a given role).

Artifacts

We have seen in Section 3.1.4 that the artifact model is based on the notion of *usage interface*. Thus, the simpAL language allows the definition of both the `usage-interface`, in terms of supported operations and observable properties, and the implementation (also referred to as *artifact template*) of an artifact.

Some peculiarities of simpAL include:

- keyword-based parameters for operations, which must be explicitly indicated during invokation
- parameters marked with `#out` are returned to the invoking agent at completion of the operation

- operations are like synchronized Java methods, i.e. they are executed in a mutually exclusive fashion
- operations may use synchronization primitives – e.g. `wait`, which suspend execution until the relative condition is satisfied
- agents are notified for the completion of an requested operation through an appropriate event – this would have spared a lot of burnt food in real world kitchens!

```
1  /*** Buffer's usage interface ***/
2  usage-interface Buffer {
3    obs-prop nElems: int;
4
5    operation put (item: Item);
6
7    operation get (item: Item #out);
8  }
9
10 /*** Buffer's artifact template (impl) ***/
11 artifact BoundedBuffer implements Buffer {
12   /* declaration of local state */
13   // obs prop nElems don't need to be declared
14
15   init (maxElems: int) { /* construction */ }
16
17   operation put (item: Item) {
18     await nElems < numMaxElems;
19     /* impl */
20   }
21
22   operation get (item: Item #out) {
23     await nElems > 0;
24     /* impl */
25   }
26 }
```

Listing 3.3: Definition of an artifact in simpAL.

When an agent is observing an artifacts, the observable properties of that artifact are automatically mapped to the agent beliefs. Observability is hard-wired into the agent infrastructure. However, the agent beliefs may be different from the actual environment state in a given instant of time. In

fact, we can say that an agent's beliefs are caused by its *perception* of the world, which cannot be perfectly istantaneous.

Organization

A program, in simpAL, is an organization with a defined topology expressed by one or more workspaces which logically contain a group of agents and a set of artifacts. Workspaces may be distributed across different nodes on the network, so they can be considered as distribution units; such deployment information can be specified at configuration level.

The topology of an organization is specified through an organization model (`org-model`), which consists of a number of definitions of `workspaces` together with their initial contained entities. Then, such a model of the organization must be implemented. The implementation typically consists of the instantiation of artifacts and agents declared in the model. At creation of agents, an initial script and an initial task for it must be provided.

```
1  /*** Organization model (logical topology) ***/
2  org-model ProdConsModel {
3
4    workspace producers { manager: Manager }
5
6    workspace consumers {
7      buffer: Buffer
8      cons: Consumer
9    }
10
11   workspace main { gui: GUI }
12 }
13
14 /*** Organization "implementation" ***/
15 org ProdCons implements ProdConsModel {
16
17   workspace main {
18     gui = new-artifact SimpleGUI (title: "Simple GUI")
19   }
20
21   workspace producers {
22     manager = new-agent ManagerScript()
23       init-task: new-task Manager.SetupProducers()
24   }
25
```

```
26    workspace consumers {
27      buffer = new-artifact BoundedBuffer(maxElems: 10)
28      cons =  new-agent SimpleConsumer()
29        init-task: new-task Consuming (maxItemsToProcess: 5000)
30    }
31
32  }
33
34  /*** Physical topology - Configuration - Deployment ***/
35  org ProdCons
36  org-id my-test-app
37  workspace-addresses {
38    main = localhost
39    producers = localhost:1000
40    consumers = 137.204.107.188
41  }
```

Listing 3.4: Defining an organization in simpAL.

3.3 Agents and reuse

In this section we try to discuss about the relation between agents and reuse. In particular, the following questions should find an answer:

- *Why* do we need reuse in the context of MAS (Multi-Agent Systems)?
- *What* means "reuse" in the context of MAS?
- *How* can *reusability* be promoted in the context of MAS?
- *How* can *reuse* be applied in the context of MAS?

Fundamentally, for what concerns the "why" question, the arguments explained in Chapter 1 apply: productivity, time-to-market, quality.
The remaining questions are approached in the following sections.

3.3.1 Reusing artifacts

It is about reuse of passive entities. Artifacts have a usage-interface which defines observable properties and operations, and an artifact template which implements those operations and possibly an internal state. Artifacts are

similar to object monitors in that their operations are performed under automatic mutual exclusion.

A natural way to think at artifact *extension* may be that of having a new set of operations and observable properties in addition to the original set. t It is not very different from extending classes in traditional object-oriented programming, so we will pass over it.

3.3.2 Reusing agents

It is about reuse of proactive entities. The conceptual framework from which we can start reasoning about agent reuse is the following:

- The task is the central notion of agency
- The plans are the response of an actor to its assigned tasks
- The role represents the guarantee of a task-oriented expertise, i.e. it is a contract that specifies that agents implementing the role are able to work (and hopely accomplish, but it cannot be guaranteed) for the designated tasks
- The script does not correspond exactly to an agent implementation. It is more a module encapsulating the expertise for doing one or more roles within an organization; it is similar to a formative course for giving actors the procedural knowledge they need to tackle certain types of task

Our aim is to be able to define new agents by reusing, in general, both the knowledge and the behavior of existing agents.

Moreover, the *substitutability principle* must hold; informally, the agent with extended capabilities must be able to be used in place of the reused agent. However, a rigorous definition of (behavioral) substitutability should be considered, but it is beyond the scope of this thesis.

Now, let's examine a few cases of reuse, but consider that they are more a way to start a discussion than sound proposals to be evaluated.

1) More plans: specialization

An interpretation of agent reuse may be the following: "A specialized agent provides more expertise for tackling the same set of tasks."

In this case the new agent reuses the plans defined for the parent agent and, in addition, provides its own set of plans. The real-world parallel is that of job specialization (intended as vertical learning), where the aim is not that of being able to do more tasks, but that of learning to do the already-known tasks better.

In simpAL, such an approach would turn itself into a sort of script extension which actually seems to be more a composition of scripts (see Figure 3.3).

Note that actually, in this case, we are *not* reusing agents. Instead, we are reusing scripts as modules that can be used to give agents the instruction that accumulates to form their total expertise (localized in the local `plan-library`).

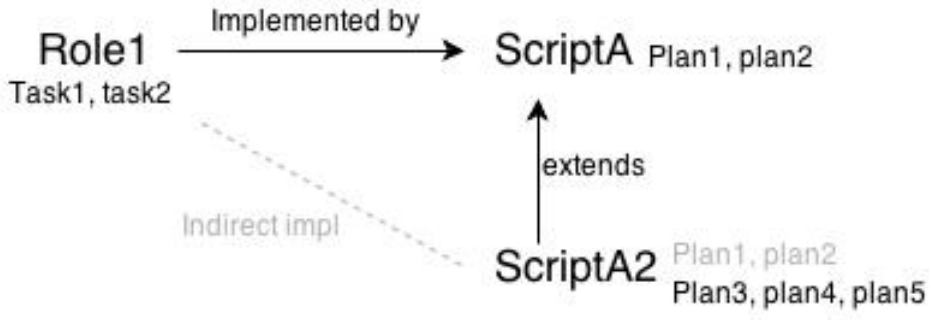

Figure 3.3: Specialization: more plans to tackle the original set of tasks.

Remember that plans can be defined to be applicable in a specific context. It is reasonable that the newly added plans could specify:

- a new context of applicability
- a context that intersect the contexts specified for other plans
- the same context as other plans or a context that include one or more contexts of existing plans

The second and third possibilities would be useful if used together with policies (which may be defined as configuration or programmatically) that specify conditions for the selection of a plan (in the plan stage) when more than one plans are applicable. In this case, it may also be useful to consider the possibility to attach annotations to plans.

2) More tasks: role extension

A different interpretation may be the following: "An extended agent is able to pro-actively act in order to accomplish more tasks within the same role."

In simpAL, it substantially reflects to:

1. extending a role from a basic role
2. defining a script that
    - reuses the plans for the basic role (i.e. a script implementing that role)
    - implements new plans only for the newly added tasks

Such an approach is illustrated in Figure 3.4.

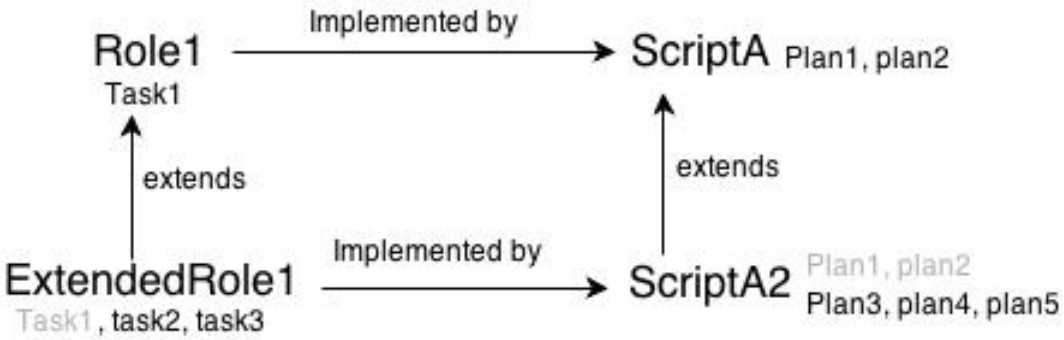

Figure 3.4: Role extension

In the previous reuse interpretation, we have seen that the addition of plans for the same set of tasks can be seen as vertical learning. In this case we have an extra set of plans that target supplementary tasks. So, the act

of loading them into an agent can be conceived as horizontal learning, i.e. a broadening of oneself's competences.

Moreover, note that not only this interpretation of agent reuse does not exclude the previous one, but it builts on it (on the ability to extend a script with other plans).

3) More tasks: multi-role implementation

Another approach for augmenting agents is that of making them implement more roles. For example, in a software project one team member may work both as a programmer *and* as a tester.

The idea is that we would like to use a mechanism for role composition and plan reuse. Suppose to have defined two roles `R1`, `R2`, and their associated scripts `S1` and `S2`. Now remember that, in simpAL, a correspondence between agent type and role exists. A new role may be created by merging the two roles. Then, two possibilities may be considered:

1. simply load the two scripts on the agent implementing the new role
2. define a new script that consists of the composition of the two initial scripts and load it into the multi-role agent

The second option is represented in Figure 3.5.

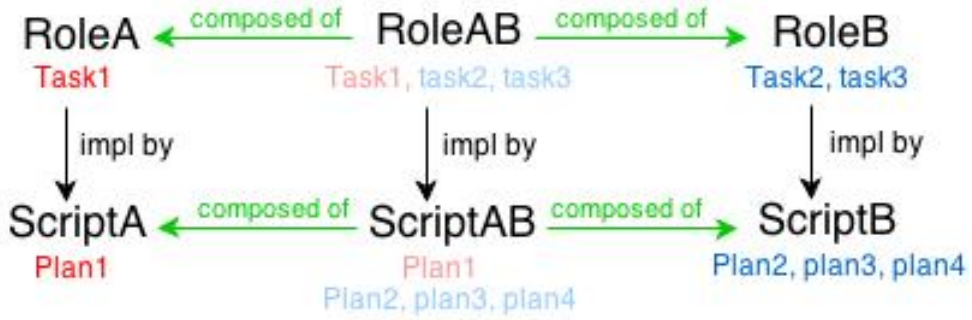

Figure 3.5: Multi-role implementation.

Conceptually, it is a sort of multiple inheritance aimed at implementation reuse. So, different issues need to be managed, primarily those due to name clashes.

4) Task composition

The reuse of agents readily turns into reuse of behavior. So, the way may be that of analyzing what means reusing a behavior. One interpretation may be the following: "If I am able to do something, I would like to be able to do something *more*."

Now, suppose you have a task `T1` and a set of plans $\texttt{P1}_i$. There are more possibilities to augment it. For example, if `T1` is "Clean your room", a composite task may be:

- "Clean your room" *and then* "Open the window"
- "Clean your room" *but, before completing it,* "Open the window"
- "Clean your room" *but, before starting it,* "Open the window"
- "Clean your room" *and in the meantime start* "Open the window"
- "Clean your room" *and finish if you have already started* "Do your homework"
- and so on

It is about tasks and relations, with new tasks that can be defined by combining existing tasks. Of course, we would like to be able to reuse already-defined plans. Such an approach allows, for example, to define tasks as sequences of sub-tasks.

Some issues exist, in particular those related to "conflicting" tasks and/or plans. For example, how to deal with tasks that have contrasting effects? How to deal with plans that affect the state of the same artifact, at the same time, and that need to retrieve such an information later on?

5) Learning via script loading

At the beginning of this section it has been said that our aim is to reuse proactive behavior. All the previous proposals are similar in that they all try to reuse existing plans. In other words, it is inside the plans the knowledge we would like to reuse. The differences are about how such a knowledge is organized into plans and roles.

Let's review how agents, roles, and scripts are related in simpAL, at the present time:

- scripts implement roles, not tasks
- a script can implement only one role
- a script must provide at least one plan for every task of the role
- an agent is defined via a role
- an agent is created by providing a script which makes the agent implement the associated role

Such constraints are the reason because, in the multi-role implementation proposal, the roles must be combined into a new role and a new script must be composed; it is because of the 1-to-1 relation between roles, scripts, and agent types.

Such constraints might be relaxed. For example, it might be possible to have agents loaded with more than one script – scripts could be seen as mixin modules and the process could be considered as a sort of learning. In this case, the 1-to-1 association between roles and scripts may be mantained, but agents would nevertheless be able to implement multiple roles by simply loading the respective scripts.

6) Plan variation/specialization

Another possibility for reuse may be that of supporting plan specialization. The idea is to have points within the code describing the plan into which a replacement or additional behavior can be injected.

Remember that plans allow to define both procedural and reactive (event-driven) behavior. Plans consist of possibly nested action rule blocks that can be serialized (action sequences) or parallelized.

Now, in order to support the specialization of plans, the following expedients might be applied:

1. provision of a mechanism for definition of custom actions
2. application of the *Template ~~Method~~ Action* design pattern to provide doorways for variation or specialization

And, of course, an extension mechanism for scripts must be provided so that actions could be overridden. It would make scripts similar to classes and actions similar to methods.

Summary

The key points of this chapter are:

- Agents are situated, autonomous (encapsulating control), reactive, pro-active, and possibly social entities built around the notion of task
- The Agent-Oriented Programming (AOP) is a paradigm that embraces concurrency and distribution by supporting the construction of software systems through agents and other human-inspired abstractions
- While agents have been traditionally studied in the context of Distributed Artificial Intelligence (DAI), the simpAL platform and language advance AOP as a general-purpose paradigm
- The simpAL model draws inspiration from the Agent&Artifact conceptual model and the Belief-Desire-Intention agent architecture
- In summary, the simpAL language provides the means for defining the following concepts and first-class abstractions:
  - *agent* – autonomous, proactive entity encapsulating control in form of a SENSE-PLAN-ACT cycle
  - *belief* – an agent's knowledge about its state or about the environment
  - *task* – something an agent may be assigned to do
  - *plan* – description of the steps an agent must apply in order to accomplish a certain task
  - *role* – it is specified as a set of tasks that the agent implementing the role must be able to do
  - *script* – module encapsulating the plans that define how to do the tasks for a certain role; it is used to provide agents with the expertise for that role
  - *action* – as the name imply, it is an action that can be done in a plan

  - *artifact* – it represents a tool or a resource; it can be used or observed through its usage-interface
  - *operation* – it represents a function provided by an artifact
  - *observable property* – information encapsulated by an artifact that can be perceived by agents who are interested at it
  - *artifact template* – module encapsulating the implementation of an artifact's operations
  - *workspace* – container that logically groups related agents and artifacts
  - *organization* – it abstracts over a set of workspaces and gives a name to the whole environment
  - *organization model* – it describes how the organization is subdivided into workspaces

- Agent reuse is about reuse of pro-active behavior

Chapter 4

Conclusion

The reuse is one of the most important issues in software engineering. We have seen that it is not just an activity, but really a process which needs both management and technical support. By a technical point of view, it is a two-step activity: design for reusability and actual reuse. Different software-related artifacts can be reused, but in this thesis we have focused on code.

Programming languages mainly contribute to reuse through abstraction and dependency-management mechanisms. Moreover, programming techniques and many language features may have a sensible impact for both component reuse and context reuse. The object-oriented programming, through a strong support for abstraction and modularity, provides a solid basis for both reuse and reusability. In particular, the inheritance is a commonly used mechanism for implementation reuse and extension of objects.

However, we have seen that porting inheritance – which works pretty well in sequential programs – in concurrent settings results in a class of issues also known as inheritance anomaly. Languages may be resistant to some kinds of anomaly while suffering from others, depending on the set of supported synchronization schemes.

Concurrency, together with distribution, is an issue that is becoming increasingly more important, due to current trends in the ICT field. So, one question was considered: how do concurrency models and programming languages tackle reuse?

We went through some significant concurrency models – multi-threaded programming, tasks in Ada, the SCOOP model, and the Actor model –

each with its own strength and weakness. Such an analysis shows that no particular attention has been given to reuse. For example, the designers of the Ada programming language did not consider the possibility of creating new task bodies from existing task bodies. For multi-threaded programming and the Actor model, no particular best-practices or features have been proposed for reuse, thus delegating the issue to the implementing language, which generally offers inheritance or extension mechanisms that are, however, effective only in sequential settings.

One may argue, for example, about the usefulness of reuse mechanisms for actors. The behavior of actor-based applications may primarily arise from (maybe complex) interactions of several simple actors; so, why would one care about overloading actors with extended behavior?

The final part of the thesis considered simpAL and the Agent-Oriented Programming paradigm. After an analysis of the main abstractions of multi-agent systems and the programming model of simpAL, some ideas about the reuse of agents were advanced.

It is about reuse of pro-active behavior, which is not very different from thinking about the reuse of human-like procedural knowledge and approaches for task-oriented work. Some possibilities were described, for example

- The ability of implementing *more plans* for the same set of tasks (Vertical learning or job specialization)
- The ability of implementing *more tasks* via role extension or multi-role realization (Horizontal learning)
- The definition of new tasks by composition of more sub-tasks tied together with specific relations

In particular, the last idea involves the concept of task composition, which cannot be faithfully transposed, for example, to objects as it is (reasonable for and) characteristic of pro-activity, maybe revealing the scope of a new paradigm.

This thesis points out a lack in research and in mainstream programming languages. It may not be considered worthwhile tackling this issue at this level. However, with the advent of concurrent paradigms, this concern need to be adequately faced.

Bibliography

[1] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.

[2] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[3] William B. Frakes and Christopher J. Fox. Sixteen questions about software reuse. *Commun. ACM*, 38(6):75–ff., June 1995.

[4] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10):104–116, October 1996.

[5] Will Tracz. Software reuse myths. *SIGSOFT Softw. Eng. Notes*, 13(1):17–21, January 1988.

[6] Martin L. Griss. Software reuse experience at hewlett-packard. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 270–, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[7] Kevin D. Wentzel. Software reuse - facts and myths. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 267–268, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[9] J.M.C. Smith. *Elemental Design Patterns*. Pearson Education, 2012.

[10] K. Knoernschild. *Java Application Architecture: Modularity Patterns With Examples Using Osgi*. Agile Software Development Series. Prentice Hall, 2012.

[11] Erik Ernst. Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 303–326, London, UK, UK, 2001. Springer-Verlag.

[12] R. L. Biddle and E. D. Tempero. Understanding the impact of language features on reusability. In *Proceedings of the 4th International Conference on Software Reuse*, ICSR '96, pages 52–, Washington, DC, USA, 1996. IEEE Computer Society.

[13] B. Stroustrup. Language-technical aspects of reuse. In *Proceedings of the 4th International Conference on Software Reuse*, ICSR '96, pages 11–, Washington, DC, USA, 1996. IEEE Computer Society.

[14] Bertrand Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[15] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 125–135, New York, NY, USA, 1990. ACM.

[16] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, March 2006.

[17] Gul Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, September 1990.

[18] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[19] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, and Erhard Ploedereder. *Consolidated ada reference manual: language and standard libraries*. Springer-Verlag, Berlin, Heidelberg, 2000.

[20] Alan Burns and Andy Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, New York, NY, USA, 3rev ed edition, 2007.

[21] Stephen Michell and Kristina Lundqvist. Extendable, dispatchable task communication mechanisms. *Ada Lett.*, XIX(2):54–59, June 1999.

[22] Volkan Arslan, Patrick Eugster, Piotr Nienaltowski, and Sebastien Vaucouleur. Dependable systems. chapter SCOOP: concurrency made easy, pages 82–102. Springer-Verlag, Berlin, Heidelberg, 2006.

[23] Rajesh K. Karmani and Gul Agha. Actors. In *Encyclopedia of Parallel Computing*, pages 1–11. 2011.

[24] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20, New York, NY, USA, 2009. ACM.

[25] TypeSafe Inc. *Akka Documentation Release 2.1.0*, 2012.

[26] Satoshi Matsuoka and Akinori Yonezawa. Research directions in concurrent object-oriented programming. chapter Analysis of inheritance anomaly in object-oriented concurrent programming languages, pages 107–150. MIT Press, Cambridge, MA, USA, 1993.

[27] Giuseppe Milicia and Vladimiro Sassone. The inheritance anomaly: ten years after. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, pages 1267–1274, New York, NY, USA, 2004. ACM.

[28] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd edition, 2009.

[29] Alessandro Ricci and Andrea Santi. From actors and concurrent objects to agent-oriented programming in simpal.

[30] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In Victor R. Lesser and Les Gasser, editors, *1st International Conference on Multi Agent Systems (ICMAS 1995)*, pages 312–319, San Francisco, CA, USA, 12-14 June 1995. The MIT Press.

[31] Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, February 2007.

[32] Mirko Viroli, Tom Holvoet, Alessandro Ricci, Kurt Schelfthout, and Franco Zambonelli. Infrastructures for the environment of multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):49–60, February 2007.

[33] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Give agents their artifacts: the a&a approach for engineering working environments in mas. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, AAMAS '07, pages 150:1–150:3, New York, NY, USA, 2007. ACM.