ImperialRJ: Exploring First-Class Relations in Object-Oriented Languages

Raoul-Gabriel Urma
ru107@doc.ic.ac.uk

Supervisor: Prof. Sophia Drossopoulou
scd@doc.ic.ac.uk

Department of Computing
Imperial College London

June 20, 2011

Acknowledgements

First and foremost, I would like to thank my supervisor, *Professor Sophia Drossopoulou*, for her amazing support. You have inspired me and I am very grateful you had the patience to teach me. I hope to have the chance to work with you on other interesting projects in the future!

My great thanks to *Dr Stephanie Balzer*, *Dr Michael Huth* and *Professor Alan Mycroft* for the useful discussions and help with this project.

I would also like to thank *Dr Nathaniel Nystrom* and *Professor Andrew Myers* for answering my emails about Polyglot and for the discussions about programming languages.

Finally, I would like to thank *my parents* and *friends* for their continuous encouragements and support throughout the year.

Abstract

The concept of relation is central to object-oriented development. It is explicitly defined in object-oriented modelling; however, it is not part of mainstream object-oriented programming languages. This lack of support leaves programmers to use language primitives to express them.

In this report, we provide a detailed study of first-class relationships in object-oriented languages. We investigate earlier work linking object-oriented techniques and relationships as well as explore the available design space. We discuss issues with aliasing, sets of tuples, types in the presence of covariant overriding and come up with extents, a novel feature which allows for sets of tuples as first-class entities.

In addition, we present *ImperialRJ*, a programming language we developed, which extends Java with first-class relationships. *ImperialRJ* introduces new constructs to work with relationships in a simple and safe way. We provide its formal definition and discuss its implementation.

We conclude by analysing the problems with implicit relationships in popular Java applications and explain how first-class relationships in *ImperialRJ* tackle these issues.

Contents

Chapter 1

Introduction

Object-oriented languages have existed for decades but the fundamental concepts defining the object-oriented paradigm are still not completely understood [1]. According to Alan Kay, the meaning of object-oriented programming is all about messaging between different objects [2]. To this end, there exist several modelling techniques [3, 4, 5] to design object-oriented systems and help represent explicitly relationships between objects.

However, most object-oriented languages lack support for explicit relationships between objects and this is why programmers are left to use language primitives such as references to express them. In other words, the semantics of relationships from the software design are not preserved to the software implementation. As a result, the information about object relationships is scattered across the code, the coupling of the system is increased and the code is more complex and less maintainable.

Research tracing back to 1987 already identified this problem and described the benefits for first-class relationship support in object-oriented languages [6]. Lately, there has been a new wave of interest for tackling object relationships implementation [7, 8, 9, 10, 11, 12, 13].

1.1 Motivation

As of today, there still isn't an unified model for supporting first-class relationships. However, there exists several collections libraries [14, 15], which bring sophisticated data structures that help make up for the lack of built-in relationship constructs. Nevertheless, these libraries were not built with supporting relationships in mind but rather for providing more collections to the Java Collection Library. As a result, these libraries aren't complete, bring different features and aren't intuitive to deal with relationships. For example, they don't provide consistency constraints like multiplicity, simple navigation of relationships and support for fields of relationships. Consequently, there isn't a unified model for the programmers to develop relationships, which leaves them with the overhead of implementing different solutions using various libraries and language constructs. For this reason, the code becomes less readable, less maintainable and prone to unexpected behaviours.

First-class relationships can tackle this issue and bring additional benefits to

the programmer. First, they bring a unified and complete model to deal with relationships within the language while at the same time help bridge the gap between design and implementation. Consequently, they ensure correct relationships implementation, help code readability and maintainability. In addition, they enable more maintainable code by promoting re-usability of relationships through relationship inheritance and covariant overriding of participants of a relationship. Furthermore, built-in relationship language constructs leave the opportunity to the compiler to optimise the internal implementation of how a programmer uses a relationship in the code. Moreover, first-class relationships can make refactoring of software easier in certain cases. For example, when changing requirements affect the multiplicity of the relationship between two entities from one to one to one to many. An implementation in the language would require rewriting code using a new data structure whereas first-class relationship are build to provide flexible multiplicity changes. Finally, first-class relationships could also help program verification by setting built-in design by contract style invariants on relationships and this way automatically ensuring these constraints are valid at runtime [13, 16].

1.2 Contributions

In this report we detail the following five contributions.

Design Space of First-Class Relationships

An extended design space for developing languages supporting relationships along with the issues and possible solutions involving certain design choices.

Formal Definition

A formal definition of *ImperialRJ* which extends an object-oriented language with first-class relationships in order to provide a reference for further work.

ImperialRJ

The first implementation of a pilot language *extending* Java with first-class relationships.

Evaluation of First-Class Relationships

A study of problems with the implementation of relationships in three popular Java applications and validation of the benefits of first-class relationships using *ImperialRJ*.

Polyglot

A tutorial written for researchers and students getting started with using Polyglot to modify or extend Java [17].

1.3 Report Structure

The report is structured as follows. In Chapter 2, we give a background to relations in mathematics, modelling techniques and programming languages. We follow by analysing earlier work linking object-oriented techniques with relationships. This section is optional and is not compulsory for the readers. Next, we give an overview of the tools available to create extensions to Java and also give a tutorial on how to use Polyglot for students and researchers. In Chapter 3, we explore the design space available to implement first-class relationships in an object-oriented language together with the issues involving certain design choices. In Chapter 4, we describe the formal definition of *ImperialRJ*. In Chapter 5, we present the implementation of *ImperialRJ*. In Chapter 6, we validate the benefits of first-class relationships by showing the problems of implicit relationships implementation in three popular Java applications and refactor them to *ImperialRJ*. We finish the report with some concluding remarks in Chapter 7.

Chapter 2

Background

This chapter presents all the information necessary to understand the context and implementations details of this project. Most of the second section analyses earlier work in the field and is not required to read to understand most of our work. The reader can come back to it later if necessary.

2.1 Relations

In this section we examine the concept of relations from a mathematical and modelling point of view.

2.1.1 Mathematical Representation

We introduce the concept of relations [18] by providing a familiar example within the university environment: a *Student* attends a *Course*. Given a set of students $S = \{Raoul, Sophia, Noble\}$ and a set of courses C = {*Programming, Logic*} we can represent the "attends" relation as a tuple *(Student, Course)*. For example: *Raoul* attends *Programming* is represented as *(Raoul, Programming)*, *Noble* attends *OO* as *(Stephanie, OO)*, *Sophia* attends *Programming* as *(Sophia, Programming)*.

Note that often we use the logical notation *R(x,y)* to express $(x,y) \in R$. In this case we could write *attends(Sophia,Programming)* instead of *(Sophia,Programming)* $\in$ *attends*. In general, a relation R is defined over the sets $S_1, ..., S_n$ as a subset of their cartesian product written $R \subseteq S_1 \times ... \times S_n$.

Relations can be defined over an arbitrary number of sets and can therefore be classified. For example, a relation over three sets is called a ternary relation, a relation over four sets denotes a quaternary relation. The most common type of relation though is the binary relation and is defined over the cartesian product of two sets. The example we gave earlier about a *Student* "attends" a *Course* is a binary relation. Binary relations on the same set are particularly interesting because we can define several properties and operations on them as we see in the next section.

Binary Relations Properties

We describe the most common properties of binary relations.

Let R be a binary relation over A. Then

- R is **reflexive** if and only if $\forall x \in A.$ *R(x,x).*
- R is **irreflexive** if and only if $\forall x \in A.\ \neg R(x, x)$.
- R is **symmetric** if and only if $\forall x, y \in A.$ *R(x,y)* $\Leftrightarrow$ *R(y,x).*
- R is **asymmetric** if and only if $\forall x, y \in A.$ *R(x,y)* $\Rightarrow \neg R(y, x)$.
- R is **antisymmetric** if and only if $\forall x, y \in A.\ R(x, y) \wedge x \neq y \Rightarrow \neg R(y, x)$.
- R is **transitive** if and only if $\forall x, y, z \in A.$ *R(x,y)* $\wedge$ *R(y,z)* $\Rightarrow$ *R(x,z).*

Additionally, a binary relation that is reflexive, antisymmetric and transitive is called a **partial order**. A binary relation that is reflexive, symmetric and transitive is called an **equivalence relation**.

Let R be a binary relation over sets A and B. Then

- R is **functional** if and only if $\forall x \in A,\ \forall y, z \in B.$ *R(x,y)* $\wedge R(x, z) \Rightarrow y = z$.
- R is **injective** if and only if $\forall x, y \in A,\ \forall z \in B.$ *R(x,z)* $\wedge R(y, z) \Rightarrow x = y$.
- R is **surjective** if and only if $\forall y \in B.\ \exists x \in A.$ *R(x,y).*

Binary Relations Operations

We describe the most useful operations on binary relations [18].

- **Composition**
  The composition of two binary relations $R \subseteq A \times B$ *and* $S \subseteq B \times C$ is written $S \circ R$ and is defined as $S \circ R = \{(a, c) | (a, b) \in R \wedge (b, c) \in S\}$
- **Union**
  The union of two binary relations $R \subseteq A \times B$ and $S \subseteq A \times B$ is written $R \cup S$ and is defined as $R \cup S = \{(a, b) | (a, b) \in B \vee (a, b) \in S\}$
- **Intersection**
  The intersection of two binary relations $R \subseteq A \times B$ and $S \subseteq A \times B$ is written $R \cap S$ and is defined as $R \cap S = \{(a, b) | (a, b) \in B \wedge (a, b) \in S\}$
- **Inverse**
  The inverse of the binary relation $R \subseteq A \times B$ is written R^{-1} and is defined as $R^{-1} = \{$*(b,a)* $\in B \times A \mid$ *(a,b)* $\in R\}$. It holds that $R = (R^{-1})^{-1}$.
  For example, the inverse of the attends relation that we gave earlier is read as "is attended by" and is defined as $R^{-1} \subseteq Course \times Student : R^{-1} = \{(Programming, Sophia), (Logic, Noble), (Programming, Raoul)\}$.

- **Reflexive Closure**
  The reflexive closure S of the binary relation R on $A \times A$ is defined as $S = R \cup \{(a, a)|a \in R\}$. For example, the reflexive closure of the relation "greater than" is "greater or equal than".

- **Symmetric Closure**
  The symmetric closure S of the binary relation R on $A \times A$ is defined as $S = R \cup \{(b, a)|(a, b) \in R\}$. In other words, the symmetric closure of R is the union of R with its inverse.

- **Transitive Closure**
  The transitive closure of the binary relation R on $A\times\ A$ is written as R^+ and is defined informally as the smallest transitive relation on $A \times A$ that contains R. The transitive closure is defined formally as $R^+ = \bigcup R^i$ where $R^1 = R$ and $R^{i+1} = R \cup \{(a, c)|(a, b) \in R^i \wedge (b, c) \in R^i\}$.
  The transitive closure is typically used in graph theory in order to find out if a node is reachable from a starting node. [19]

2.1.2 Modelling Representation

Relations exist in different contexts and they can be modelled graphically in order to be more expressive to people.

In this section we will examine relations modelling in the context of database (*Entity-Relationship Diagram*) and software (*Unified Modelling Language*). We also look at an extension language to *UML* called *Object Constraint Language* that enables the modeller to express extensive constraints on *UML* models.

Entity-Relationship Diagram [5]

Entity-Relationship (ER) diagrams are used to design databases and are made of entities, attributes and relationships. An *entity* is a category that a group of object may belongs to and that can be uniquely identified. Entities can have attributes to specify a property to keep in the database about them. For example, a `Student` entity might have a `Name` attribute.

Relationships are the equivalent of the mathematical concept of relations, they express an association between two or more entities. Just like entities, they can also have attributes. For example, a relationship `Attends` between a `Student` entity and a `Course` entity might have a `mark` attribute recording the mark of the student for a specific course. In addition, relationships can also be assigned *cardinalities* in order to specify the number of entities related to one another. Such cardinalities include: *many-to-many* (M:N), *many-to-one* (M:1), *one-to-many* (1:M) and *one-to-one* (1:1).

Entities are represented in Entity-Relationship diagrams by rectangles, attributes by ellipses and relationships by diamonds.

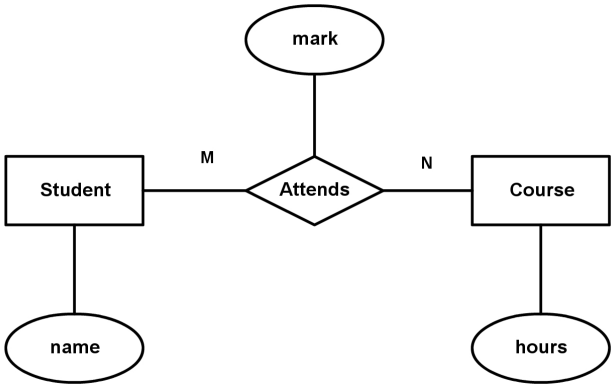


Figure 2.1: Example of Entity-Relationship Diagram
This Entity-Relationship Diagram depicts a relationship `Attends` between the `Student` entity and the `Course` entity. The relationship `Attends` has an attribute `mark`, the `Student` an attribute `name` and the `Course` an attribute `hours`.

Unified Modelling Language [4, 3]

Unified Modelling Language (*UML*) is a standard language that enables the specification and visualisation of a software system. *UML* supports a wide variety of elements used for specific needs. For example, *UML Use Case diagrams* model interactions between a system and users, *UML Sequence Diagrams* model communications between objects where the order of communication is important, *Package diagrams* are used to describe the interaction between components at a high-level in a software. In this section, we focus on *UML Class diagrams* which describe the structure of a software and also the underlying relationships between classes.

Classes are represented by rectangles and contain information about their fields and methods. Classes are linked together by relationships. Such relationships include *associations* to express a relation, *aggregations* to express a "has a" association, *composition* to express a "is part of" association, *realization* to express implementation of interfaces and *generalization* to express inheritance.

Associations are related to the concept of relationships in *Entity-Relationship Diagrams* and also to the mathematical concept of relations. They basically relate entities together through a link. Similarly to relationships in Entity-Relationship Diagrams, constraints can also be set on the associations with *multiplicity* and *navigability*. *Multiplicity* allows to specify the number of objects participating in the association. Such multiplicities include:

- **0..1** : optional instance
- **1** : exactly one instance
- 0..* or * : any number of instances
- 1..* : 1 or any number of instances

Navigability enables to specify the direction of the relationship, this is specified by an arrow head. For example in Figure 2.2, the association `Attends` can only be traversed in one direction: from an instance of `Student` to an instance of `Course`. However, `Course` to `Student` is not allowed.

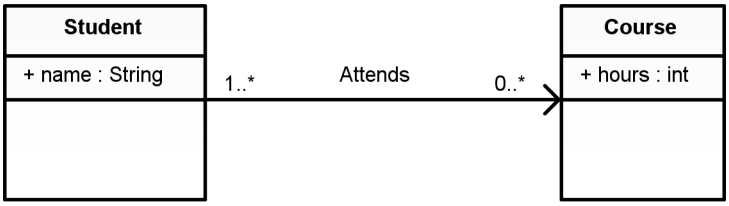


Figure 2.2: Example of UML Class Diagram: simple association.
This Class diagram describes an association between `Student` instances and `Course` instances. This association is directed and `Students` can attend 0 or more `Courses`, while a `Course` is attended by at least one `Student`.

In addition, associations can be extended with an*association class* when the relationship between classes is not only a simple logical connection. An *association class* can have attributes and is represented by a rectangle box connected with a dashed line to the association it describes as show in Figure 2.3.

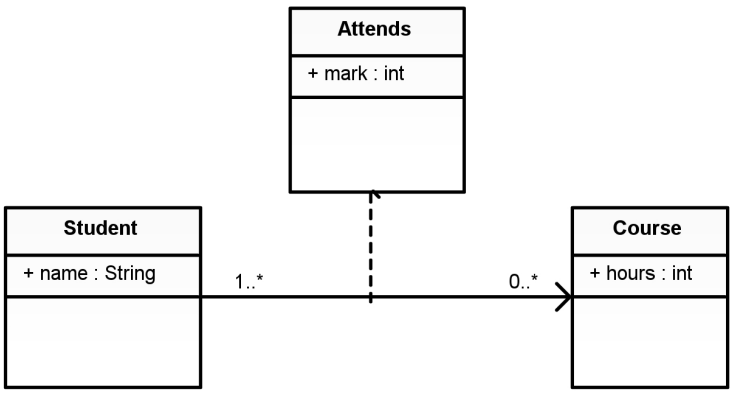


Figure 2.3: Example of UML Class Diagram: association class.
This Class diagram describes a complex association which records the `marks` between `Student` instances and Course `instances`.

Object Constraint Language [20]

The Object Constraint Language (*OCL*) enhance the *UML* specification by enabling modellers to set extensive constraints on *UML* models. For example, it can be used to set preconditions, postconditions and invariants. In the context

of relations, *OCL* is useful because it brings the possibility to reason about and to set constraints on relationships between classes.

At its simplest, *OCL* can be used to set conditions on fields and operations as illustrated in figure 2.4 However, *OCL* allows the modeller to set more evolved constraints and logic as shown in figure 2.5

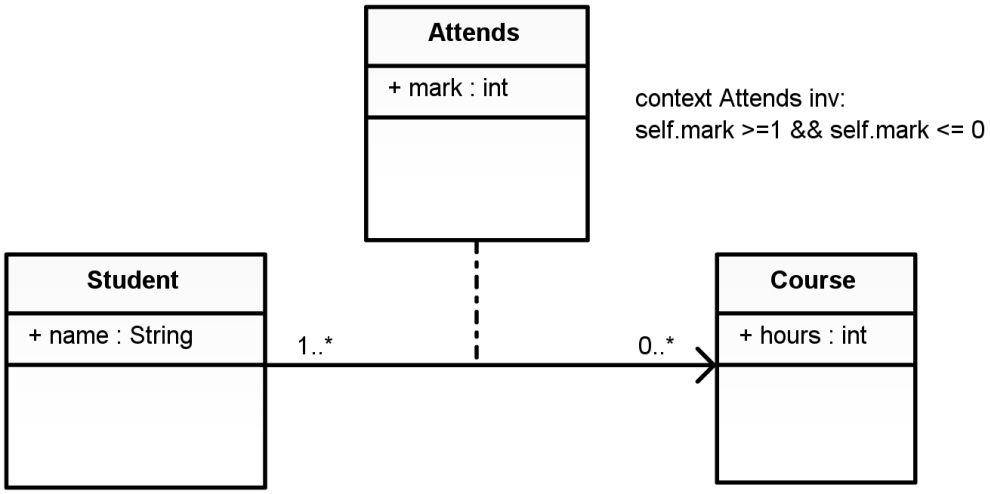


Figure 2.4: Example of basic OCL invariant: field invariant.
The `Attends` association class's `marks` must be in the range 1 to 6.

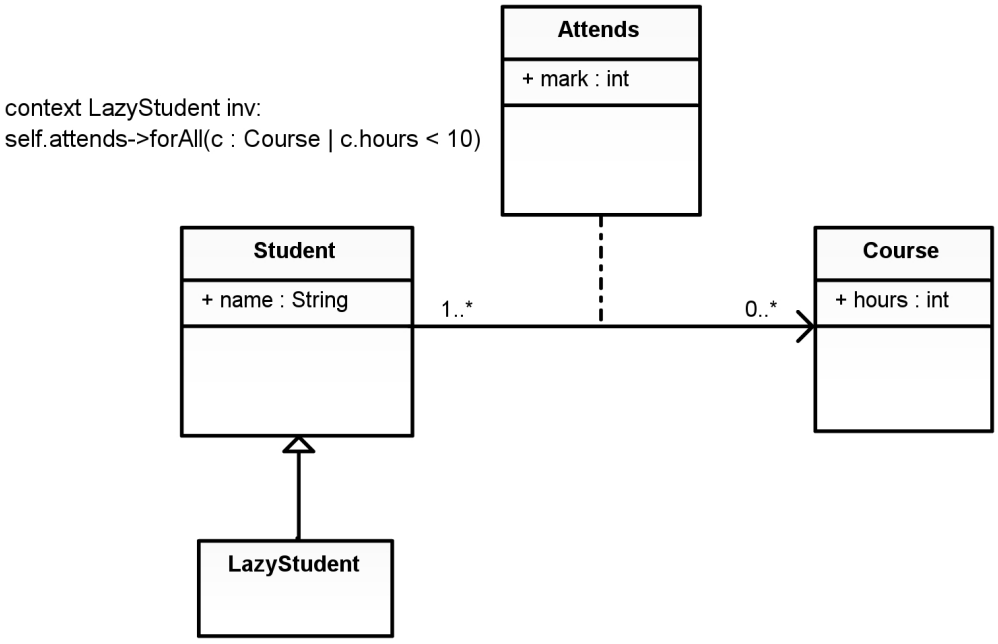


Figure 2.5: Example of complex OCL invariant
All the courses attended by a `LazyStudent` have less than 10 `hours` of lectures.

2.2 Relationships in Programming Languages

In this section, we examine how relations apply to object-oriented programming languages and why first-class relationships are important. We come back to this with further details in Chapter 6.

We demonstrate the need of first-class relationship by implementing the simple example of a `Student` attending a `Course` in listing 2.1. The *UML* diagram is illustrated in figure 2.2.

Listing 2.1: Naive implementation of Attends relationship

```
1  public class Student {
2
3    String studentName;
4    HashSet<Course> attends = new HashSet<Course>() ;
5
6    public Student(String studentName) {
7      this.studentName = studentName;
8    }
9
10   public void add(Course c)
11   {
12    attends.add(c);
13    c.attendees.add(this);
```

```
14    }
15  }
16
17  public class Course {
18
19    private String courseName;
20    HashSet<Student> attendees = new HashSet<Student>() ;
21
22    public Course(String courseName) {
23      this.courseName = courseName;
24    }
25
26    @Override
27    public String toString() {
28      return courseName;
29    }
30
31  }
32
33  public class Main {
34
35    public static void main(String[] args) {
36
37      Student stephanie = new Student("Stephanie");
38
39      Course oo = new Course("Object-Oriented Languages");
40      Course compilers = new Course("Compilers");
41
42      stephanie.add(oo);
43      stephanie.add(compilers);
44
45      for(Course c : stephanie.attends)
46        System.out.println(c);
47    }
48  }
```

Courses can be added through the add() method in the Student class.

Listing 2.1 shows that a lot of boiler plate code is needed to implement a simple relationship. In addition, there are two undesired issues with this implementation.

- The semantic of the UML diagram representing a `Student` attends a `Course` is lost in the implementation because it is made implicit using references and collections. In fact, the relationship had to be implemented using a `HashSet` in both classes.
- The `Student` and `Course` class are now tightly coupled to one another. In fact, changing the implementation for recording the `mark` that a `Student` gets by attending a `Course` would require changes in both classes.

The goal of first-class relationship is to eliminate these problems and add a first class construct which encapsulates the relationship implementation details. In addition, it enables the classes participating in the relationship to be entirely decoupled. Moreover, the semantics of the relationship from the design would be preserved to the implementation.

In the next sections, we describe the terminology used in the literature for dealing with first-class relationship. In addition, we present some of the main

existing literature in the object relationships field. This section is not compulsory to read to understand the main concepts in the next chapters.

2.2.1 Terminology in Literature

- **Relationship**
  A relationship abstracts the implementation of object collaborations. Relationships are the implementation abstraction of the *UML* association concept.

- **Participants**
  The participants of a relationships are the objects' classes involved in the relationships.

- **Relationship instance**
  A relationship instance represents a possible association between the participants of the relationship. For example *(Raoul, OO)* is one possible relationship instance of the relationship *Attends(Student, Course)*.

2.2.2 Existing Work

Relations as Semantic Constructs in an Object-Oriented Language

James Rumbaugh, one of the creators of *UML*, was the first to identify the need for a built-in relation construct in object oriented languages [6]. He states that even though object-oriented languages express classification and generalisation by supporting built-in syntax and semantics, they do not have any semantics for expressing relations directly.

Although relations can be implemented by distributing the information about the relation among the different participant classes or by defining a collection class "relation" to represent values of particular relations, he argues that relations should be considered a semantic construct and not simply an implementation construct. The reason is that this will further abstract the high-level structure of the system and help decouple classes. Furthermore, he states that relations are more important to the design of large systems than generalisation and should therefore have built-in syntax and semantics. In fact, relations affect globally the way a system is partitioned whereas a generalisation hierarchy is usually bound to a single module within a system.

The rest of his paper provides an overview of relations as logical constructs and how to extend object-oriented language to implement relations. In relation to our work, the paper introduces several useful concepts: updating a relation, testing membership in a relation, cardinality, encapsulation etc. In addition, the author discusses a possible syntax for dealing with relations. He argues that the syntax of declaring relations should be parallel to the syntax for declaring classes because relations are a first-class semantic construct. However, importantly, it is not necessary for the user to declare new methods on particular relations, because they are not actual classes and do not describe instances. The paper concludes by describing a few implementation optimisations for efficient access and also by introducing *Data Structure Manager* (DSM), an object-oriented language developed by the author which fully supports relations. However, it does not support relations inheritance, constraints on relations and further relation

operations like transitive closure, inverse and composition. Unfortunately, we couldn't find an updated reference link to the *DSM* project.

First-Class Relationships in an Object-Oriented Language

Gavin Bierman and Alisdair Wren developed a formal specification of a language *RelJ* which is a subset of Java and which has first-class relationship support [8, 7]. *RelJ* enables the definition of relationships between objects and also the specification of attributes on relationships. Furthermore it introduces a new concept to any previous work: *relationship specialisation*. In other words, a relationship can be derived from another relationship. In addition, the authors explore the idea of *multiplicity* constraints on relationships. However, they do not provide any implementation of their formal work.

The authors declare relationships similarly to class declaration; they contain a number of field declarations and methods declarations: `relationship r extends r' (n, n') FieldDecl* MethDecl*` This defines a global relationship r with a number of fields and methods declarations. This relationship is between `n` and `n'` which range over classes as well as further relationships. Consequently, this enables relationship aggregation.

Listing 2.2 shows an example of declaring an `Attends` relationship over `Student` and `Course`. It has also an attribute `mark` and a method `getCertificate()`. Listing 2.3 shows how to access and set fields of relationship in *RelJ*.

Listing 2.2: Example relationship declaration and manipulation in RelJ

```
1  relationship Attends extends Relation(Student, Course){
2    int mark;
3    Certificate getCertificate(Academic signatory){
4    ...
5    }
6  }
7
8  Student stephanie = new Student();
9  Course oo = new Course();
10 Attends attnds = Attends.add(stephanie,oo);
11 Attends.rem(stephanie, oo);
```

This example shows how to add and remove a relationship instance `(stephanie, oo)` from the relationship `Attends`.

Listing 2.3: Example relationship fields access in RelJ

```
1  relationship Attends extends Relation(Student, Course){
2    int mark;
3    Certificate getCertificate(Academic signatory){
4    ...
5    }
6  }
7
8  Student stephanie = new Student();
9  Course oo = new Course();
10 Attends attnds = Attends.add(stephanie,oo);
11 attnds.mark = 10;
```

This example shows how to access and set the value of relationship's fields. In this case we set the `mark` for the relationship instance `(stephanie,oo)` to 10.

Listing 2.3 shows that *RelJ* works by managing relationship instances. In fact, the add method on the static relationship `Attends` returns an `Attends` instance which enables the developer to access the fields and methods defined in the `Attends` relationship. However, the returned instance is strictly bound to `(stephanie,oo)`. This approach can lead to problems. In fact, if one removes a relationship instance that was stored in a variable from the relationship then the variable has now an instance that no longer exist. This problem is illustrated in Listing 2.4

Listing 2.4: Removal of relationship problems

```
1 Attends attnds = Attends.add(stephanie,oo);
2 attnds.mark = 10;
3 Attends.rem(stephanie,oo);
4 int mark = attnds.mark; // problem
```

The authors suggest three solutions to deal with this issue: taking no action, deleting instances from the heap and nullifying references. However, the first option is not an accurate solution because it basically says we don't deal with the issue. The last two add further complexity to the language implementation but also raise security risks when dealing with dangling pointers or is prone to unexpected null pointer exceptions since we access a null relationship instance.

The paper also introduces *relationship inheritance*. First the authors define relationship inheritance exactly as for classes. Namely if relationship r_2 extends r_1 then:

- r_2 is a subtype of r_1
- r_2 inherits all fields and methods of r_1

However this approach raise a few issues. First, the participants types of the child relationship must be sub types of the parent relationship's participants. Listing 2.5 shows an example of relationship inheritance. In this case it is necessary that `LazyStudent` extends `Student` and `HardCourse` extends `Course`.

Listing 2.5: Example of relationship inheritance

```
1 relationship Attends extends Relation(Student, Course){
2   int mark;
3   ...
4 }
5
6 relationship RelunctantlyAttends extends Attends(LazyStudent, HardCourse){
7   int missedLectures;
8   ...
9 }
```

Second, this opens several ways to interpret relationship access. Consider Listing 2.6 where a `LazyStudent` raoul attends two courses: `programming` and `compilers`. According to the authors, if we consider relationship inheritance as a reuse and subtyping mechanism then only the course `programming` is available in the `Attends` relationship. However, if inheritance implies that `ReluctantlyAttends` relationship is included in the `Attends` relationship then both `programming` and `compilers` are accessible through the `Attends` relationship because even if `Raoul` lazily attends `programming` he's still however attending `programming`.

Listing 2.6: Relationship inheritance issues

```
1 LazyStudent raoul = new LazyStudent();
2 HardCourse compilers = new HardCourse();
3
4 Attends.add(raoul, programming);
5 ReluctantlyAttends.add(raoul, compilers);
```

Furthermore, this raise another issue: if one adds the relationship instance `(raoul,compilers)` to the `Attends` relationship, we would end up with two same relationship instances in `Attends` and `ReluctantlyAttends`. It is unclear which relationship instance has more importance if we ask for the `(raoul,compilers)` relationship instance. To address this issue the authors came up with a different idea: "inherited fields should be implemented by an instance of the relationship in which they were declared, so that this instance may be share among instances of sub-relationships". This is why, the authors impose the following two invariants:

1. Consider a relationship r_2 which extends r_1. For every instance of relationship r_2 between objects o_1 and o_2, there is an instance of r_1, also between o_1 and o_2, to which the r_2 instance delegates requests for r_1's fields.

2. For every relationship r and pair of objects o_1 and o_2, there is at most one instance of r between o_1 and o_2.

Finally, the paper also describes a formalisation for introducing *multiplicity* constraints on relationships. The author suggests two annotations: *one* and *many* to respectively imply '0..1' and '0..*' in *UML*. The introduction of multiplicity adds further restriction on relationship inheritance: a *many-to-one* relationship may only inherit from a *many-to-one* or *many-to-many* relationship. Similarly a *one-to-many* relationship may only inherit from a *many-to-many* relationship or *one-to-many* relationship. The authors introduce a new invariant to enforce this restriction:

3. For a relationship r, declared `relationship r (`n_1`,`n_2`)`, where n_1 is annotated with one, there is at most one n_1-instance related through r to every n_2-instance. The converse is true where n_2 is annotated with one.

In comparison to our work, we believe *RelJ* introduces novel concepts like inheritance and multiplicity which we explore further. However, the main issue is that *RelJ* deals with relationship as a global static entity. Consequently, implementing relationship specialisation is different to object-oriented specialisation because a relationship can not be instantiated several times just like a class. *RelJ* makes uses of complex delegations to solve this issue. With *ImperialRJ* we show how several OO concepts can be elegantly ported to relationships if we don't force relationships to be global entities. In addition, we explore a different alternative for accessing relationship instances in order to avoid these problems: we provide a safe interface to access and update relationship instances through the relationship entity itself. This way we don't need to store relationship instances to variables at all.

Basic Relationship Patterns

James Noble describes five basic patterns to implement relationships [10].

- **Relationship as Attribute**
  How to design a very simple relationship?
  Attributes are ideal for modelling simple one way, one-to-one relationships. Because these types of relationships are very common they need to be easy to write and understood but also implemented with minimal overhead. For instance, a class A has an attribute of type B to represent its relationship with class B.

- **Relationship Object**
  How to design a big, important, or common relationship?
  Complex relationships can be implemented directly using attributes however, this distributes the implementation of the relationship across the participants resulting in tight coupling and low cohesion. The author suggests introducing an additional relationship object containing all the information about the relationship. For example, a relationship between a class A and a class B can be managed through a new class C. The objects involved in the relationship are now independent of it and therefore have lower coupling.

- **Collection Object**
  How to design a one-to-many relationship?
  This pattern is used for describing one-to-many relationships. The author suggests using a container or a collection object to represent this relationship; for example, a List or an Array. In addition, if necessary, a more specialised collection can be implemented to add any constraints or behaviour required by the relationship.

- **Active Value**
  How to design an important one-to-one relationship?
  An active value object is an object representing a single variable and provides an interface to retrieve the value of the variable and a setter to change the variable's value. It is used to associate two objects that are both dependent on this particular value. To use an active value, the author advices to make it an attribute of the source object of the relationship and access the relationship through the active value, rather than the source object. The active value will detect when its value changes and update any dependent objects.

- **Mutual Friends**
  How to represent a two-way relationship?
  This patterns describes how to implement a two-way relationship where all the participants are equally important. In this type of relationship, a change in any participant may affect all other objects in the relationship. To implement this pattern the author suggests two steps: firstly, split the relationship into two-one way relationship. Secondly, keep these consistent through an interface in one of the mutual friend which is defined as a leader and manages the other objects as its followers in the relationship.

In comparison to our work, he doesn't discuss first-class order relationship but rather uses object-oriented languages primitive constructs to describe how objects can be used to model relationships in an ideal way.

Relationship Aspects

In [21], David Pierce and James Noble argue that because most object oriented programming languages provide little support for dealing with relationships, the developers have to implement them using languages primitives. As a result, the participant classes of a relationship are highly coupled and this leads to poor maintainability and the software becomes difficult to extend. The authors suggest a different solutions to first-class order relationships, they explore using available techniques and library to implement relationship by using *Aspect Oriented Programming*. They model relationships as a separable cross-cutting concern and therefore decouple the relationship responsibility from the participant classes to external and centralised aspects. The authors developed the *Relationship Aspect Library* (RAL) [22] that includes a set of aspects implementing various types of relationships. All relationship aspects implement the base Relationship interface that is shown in listing 2.7

Listing 2.7: Aspect Relationship interface

```
1  interface Relationship<FROM,TO, P extends Pair<FROM,TO>> {
2  public void add(P);
3  public void remove(P);
4  public Set<P> toPairs(TO t);
5  public Set<P> fromPairs(FROM f);
6  public Set<FROM> to(TO t);
7  public Set<TO> from(FROM f);
8  ...
9  }
```

The `Relationship` interface contains methods to add and remove relationship instances. In addition it comes with various methods to access the participants of the relationship.

For example, the `Attends` relationship described previously 2.2 can be easily implemented as an aspect using the `SimpleStaticRel` aspect which implements the `Relationship` interface from the Relationship Aspect Library. Listing 2.8 illustrates the `Attends` relationship implementation.

Listing 2.8: Attends relationship using RAL

```
1  public aspect Attends extends SimpleStaticRel<Student,Course>
2  {
3
4  }
5
6  Course compilers = new Course();
7  Student raoul = new Student();
8
9  // create relationship between Student and Course
10 Attends.aspectOf().add(raoul,compilers);
```

The aspect `Attends` extends the `SimpleStaticRel` aspect which provide methods to access and manipulate a relationship in accordance with the `Relationship` interface.

Although this paper introduces good ideas to provide a general abstraction for relationships, the RAL [22] is not first-class. It requires the user to install the aspectJ [23] library. Nonetheless, the authors did a good work on coming up with a skeleton of relationship interface that is fairly intuitive to use and learn. We intend to bring some of these concepts to *ImperialRJ* when designing our relationship abstraction.

A Relational Model of Object Collaboration

In [13], Balzer, Gross and Eugster link relationships to discrete mathematics and introduce the concept of relationship invariants to maintain the consistency of constraints on relationships as well as introduce *member interposition* which allows the specification of relationship-dependent members of classes.

Member interposition differentiates from relationship attributes as defined in previous research [6, 7] because they define properties on the particular role that a class plays in a relationship. However, attributes on relationships (referred to as *non-interposed members*) describe properties on the relationship itself. The example given in the paper to demonstrate the use of member interposition is described in Figure 2.9: a `Student` assists a `Course` as a teaching assistant and the language of instruction must be recorded.

Listing 2.9: Example of member interposition.

```
1 relationship Assists
2   participants(Student ta, Course course){
3     String >ta instructionLanguage
4   }
```

The attribute instructionLanguage is interposed into ta.

The paper then extensively explores the concept of relationship invariants, an idea mentioned previously by Rumbaugh [6]. The authors define relationship invariants as a way to express consistency constraints on relationship. They distinguish between two categories: *structural invariants* and *value-based invariants*. *Structural invariants* express the scope of the invariant; in other words they restrict the participation of objects in the relationship. For example, such structural invariants include defining a relationship to be symmetric, irreflexive or even defining two relationships to be disjoints. *Value-based invariants* are predicates on the attributes and participants of the relationship in the same way as OCL does [20]. For example, a value-based invariant could be added to the example in Listing 2.9 to restrict the `instructionLanguage` to be in a specific set of languages. Listing 2.10 shows an example which defines an invariant on the `Attends` relationship. We define the relationship `Attends` to be surjective; i.e all the students must be attending at least a `Course` and we also set a restriction on the range of the `marks`.

Listing 2.10: Example of relationship invariant.

```
1 relationship Attends
2   participants(Student ta, Course course){
3     int mark;
4
5     invariant
6       surjective(Attends) && mark >=1 && mark <= 6;
7   }
```

The `Attends` relationship is surjective (structural invariant) and the `mark` is in the range 1 to 6 (value based invariant).

Relationship Detector

While most work was concerned with the introduction of relationships from a language design perspective[6, 7, 8, 13, 10], little work was done on whether first class relationship support is actually justified and useful in practice for developers. In [11, 12], Balzer, Burns and Gross investigated empirically the frequency of collaborations between objects in order to assess the need for first class relationship support. To conduct this experiment the authors develop a fully automated tool, RelDJ (Relationship Detector for Java) that takes Java classes as input and which rely on a categorisation of possible implementation of object collaborations called *collaboration code skeletons*. The authors describe five main collaboration code skeletons representative of real world collaboration implementations: `Direct Binary Unidirectional, Direct Binary Bidirectional, Indirect Indexed Extent, Indirect Pair Extent and Indirect Triple Extent`. These skeletons make use of Java raw features: references to classes' instances through fields, Collections and Maps.

RelDJ identifies collaborations between classes by detecting such skeletons in the input classes. However, the identification of collaborations when a program uses non-generic collections is still a work in progress. In fact, the collection elements' types are used to infer the participants of a collaboration. However, since the element type of a non-generic collection is unknown it is difficult to infer the participant of a collaboration without complex data-flow analysis to deduce the element type of the collection.

After using RelDJ on a portfolio of 25 different Java programs to conduct the experiment, the authors concluded that collaboration occurs frequently and this cast further evidence on the benefits of a first class relationship construct.

Other Papers

There exist other papers linking Relationships to Databases. Current research is looking into creating an optimal language to query information in tables representing various relationship. One paper describes *Cw* which comes with an Xpath [24] like queries [25]. Another paper describes a language extension for Java (JQL) which permits queries over object collections [26]. We intend to further investigate this area and bring these concepts to enable efficient querying on first class relationships.

Reflection

The existing research in relationships applied to object-oriented language is still at an embryonic stage. People are more and more aware of the lack of first-class support for relationships and research describes the benefits of having relationship constructs build in current object-oriented languages [6, 9, 7, 8, 21]. However, there's still a lack of in depth research to design such a language. The only prototype language available to us at the moment is *RelJ* [7, 8]. However, RelJ addresses relationship as global static entities. There is still a big issue that needs to be looked at: whether multiple instances of relationships have benefits. With *ImperialRJ*, we further explore this route and show the benefits

of this model. We believe this approach enables new opportunities to bring known concepts like generalisation, aggregation, delegation, interfaces to the relationships world.

2.3 Language Extensions

In this section we present the two most popular extensible compiler frameworks that enable the creation of compilers for Java language modifications: *JastAddJ* and *Polyglot*.

For our project, we chose Polyglot because it has a wider community and lots of projects are available opensource online [27], which makes it easier to understand the internals of the Polyglot framework. However, there's little documentation and this is why we provide a tutorial for students and describe a comprehensive example of how to build a simple language modification using Polyglot. This tutorial has been made official and is available on the Polyglot website [17].

2.3.1 JastAddJ

Introduction

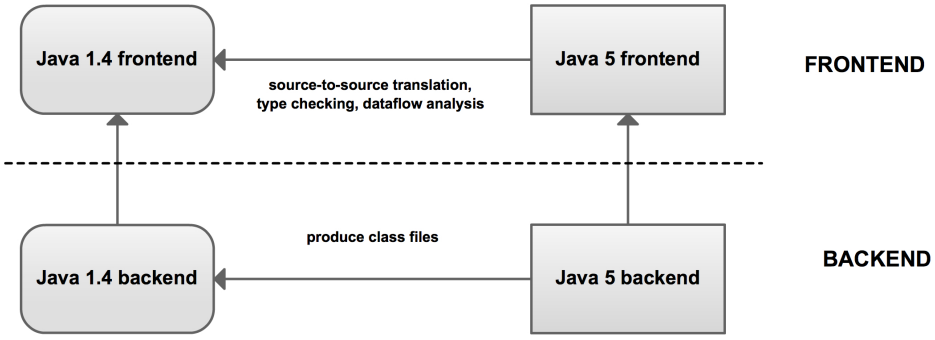
the JastAdd Extensible Java Compiler (JastaddJ) is a Java compiler that can be extended in order to build new languages on top of Java. JastAddJ [28] was developed using the metacompiler JastAdd [29] which enables the construction of customised extensible compilers.

JastAddJ consists of four main components:

- A Java 1.4 **frontend** and **backend** defining the constructs, semantics and translation of Java 1.4 features
- A Java 5 **frontend** and **backend** which are extension of the Java 1.4 components and extend the language with Java 5 features including generics, enums and the extended for loop on collections.

Each main component is a directory of JastAdd files, parser generator input files and programs in Java. The frontend programs are responsible for the source-to-source translation of the language modification to Java by parsing the input source files. On the other side, the programs in the backend are responsible for producing class files.

Figure 2.6: Components architecture of JastAddJ



Extending the language

Language extensions can be specified as new components within the 4 main components. As an example, we describe the outline of extending Java 1.4 with the Java 5 enhanced for loop.

Three principal parts need to be specified for any language extension:

1. *an abstract grammar* defining the structure of the Abstract Syntax Tree (AST). This is specified in a **.ast** file.

2. *a context-free grammar*, defining how text is parsed into an AST. This is specified in a **.parser** file.

3. *behaviour specifications* defining the behaviour of an AST. These are defined as aspects using JastAdd constructs in a **.jrag** file.

To extend Java 1.4 with the enhanced for loop (easy navigation of collections) we need to provide the definition of three files: *EnhancedFor.parser*, *EnhancedFor.ast* and *EnhancedFor.jrag*.

The enhanced for loop grammar is defined as a `Statement` and is specified as follows in *EnhancedFor.parser*:

Listing 2.11: Enhanced For Loop grammar in JastAddJ

```
1  Stmt statement =
2    enhanced_for_statement.f                                    {: return f;
         :}
3    ;
4
5  Stmt statement_no_short_if =
6    enhanced_for_statement_no_short_if.f                        {: return f;
         :}
7    ;
8
9  Stmt enhanced_for_statement =
10   FOR LPAREN enhanced_for_parameter.p COLON expression.e RPAREN
         statement.s
11   {:  return new EnhancedForStmt(p, e, s);  :}
12   ;
```

```
13
14 Stmt enhanced_for_statement_no_short_if =
15   FOR LPAREN enhanced_for_parameter.p COLON expression.e RPAREN
       statement_no_short_if.s
16   {: return new EnhancedForStmt(p, e, s);  :}
17   ;
18
19 VariableDeclaration enhanced_for_parameter =
20   modifiers.m? type.t IDENTIFIER dims.d?     {: return new
       VariableDeclaration(new Modifiers(m), t.addArrayDims(d),
       IDENTIFIER, new Opt()); :}
21   ;
```

If the parsing is correct, a new AST node *EnhancedForStmt* is created which is specified as follows in *EnhancedFor.ast*:

Listing 2.12: Enhanced For Loop AST specification

```
1 EnhancedForStmt : BranchTargetStmt ::= VariableDeclaration Expr
    Stmt;
```

The final step is to define the behaviour of the EnhancedFor AST. This can be defined by an aspect plugged in in different *attributes* of a AST JastAddJ node. For example translation is handled by the toString attribute of an AST node. The translation of EnhancedFor is specified as follows within the *EnhancedFor.jrag* file:

Listing 2.13: Enhanced For Loop grammar in JastAddJ

```
1
2 aspect EnhancedFor{
3
4   // pretty printing
5   public void EnhancedForStmt.toString(StringBuffer s) {
6     s.append(indent());
7     s.append("for (");
8     getVariableDeclaration().getModifiers().toString(s);
9     getVariableDeclaration().getTypeAccess().toString(s);
10    s.append(" " + getVariableDeclaration().name());
11    s.append(" : ");
12    getExpr().toString(s);
13    s.append(") ");
14    getStmt().toString(s);
15  }
16 }
```

Conclusion

According to the authors of JastAddJ, it compares well with existing extensible Java compiler framework like Polyglot [27] and Jaco [30]. In fact, JastAddj passes more tests as defined by the Java Language Specification and new compiler extension require a smaller implementation size than its competitor. However, as JastAddJ is relatively new, there aren't many projects available yet which use it to implement new language modifications on top of Java. The only extension available on the JastAddJ website are an implementation of the *JSR308*, which extends the Java annotation syntax to permit annotations on any occurrence of a type [31] and an implementation of pluggable non-null types for Java [32].

2.3.2 Polyglot

Introduction

Polyglot is a highly extensible compiler construction framework developed by Nystrom, Clarkson and Myers at Cornell University [27]. It performs parsing and semantic checking on a language extension and the compiler outputs Java source code. It is implemented as a Java class framework using design patterns to promote extensibility.

Several projects have successfully used Polyglot to extend the Java programming language; they range from large-to middle-scale projects. For example, *Jif* [33] is a language modification that extends Java with support for information flow control and access control, *SessionJ* introduces session-based Distributed Programming in Java [34] and J_0 is a subset of Java used for education [35].

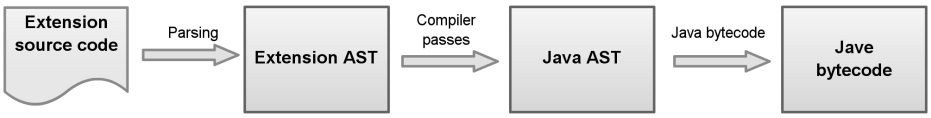
Language modifications follow the same pattern: they are implemented by extending the base grammar, type system and defining new code transformations on top of the base Polyglot framework. The result is a compiler that outputs Java source code. We can then compile the output with the standard Java compiler javac to generate bytecode runnable by the JVM.

Currently, Polyglot only supports Java version 1.4, but a language extension supporting most Java version 5.0 features has been developed [36].

Polyglot comes with the Polyglot Parser Generator *(PPG)*, a customised Look-Ahead LR Parser based on *CUP* [37, 38]. It is specially developed for the language extension developer to easily extend the Java base grammar defined with *CUP* by specifying the required set of changes [27]. In fact, *PPG* provides grammar inheritance, which enables the language extension developer to augment, modify or delete symbols and production rules from the Java base grammar.

The architecture of Polyglot follows standard compiler construction techniques. First of all, it first uses *JFlex*, a lexical analyser generator [39], to perform lexical analysis on the source language. This step transforms the sequence of characters from the source code file into tokens. This chain of tokens is then parsed by *PPG*, which creates an abstract syntax tree (AST). An AST is a tree data structure that reflects the syntactic structure of a program. During the Polyglot process, this data structure is visited by several passes; the default set of passes include for example type checking, ambiguities removing and translation. In addition, Polyglot enables the introduction of extra passes in order to perform operations related to the compiler purposes: for example, optimising the AST. Finally, if no exceptions are thrown during the Polyglot process, the created compiler is expected to produce valid Java source code that can be compiled into Java bytecode.

Figure 2.7: Polyglot high-level process



In Details

In practice, implementing new language modification requires some knowledge about the Polyglot structure and internals. In this section, we explore Polyglot at a deeper level.

The latest revision of Polyglot can be fetched from the project SVN [40]. The Polyglot distribution contains several directories:

- **/bin/** : contains Polyglot base compiler and script *newext.sh* that generates the skeleton for a new language extension
- **/doc/** : various documentation about Polyglot
- **/examples/** : source codes of sample language extensions using Polyglot
- **/skel/** : skeleton directory hierarchy and files used for building new language extensions
- **/src/** : complete source code of Polyglot framework
- **/tools/java_cup/** : source code of tweaked version of Java CUP
- **/tools/ppg/** : source code of PPG

To create a language extension called [*extname*], the first step is to run **/bin/nexext.sh**, which creates the necessary skeleton package hierarchy and files:

- **[extname]/bin/[extname]c** : compiler for [extname]
- **[extname]/compiler/src/[extname]/** : source code for language extension
  - **ast** : AST nodes specific to [extname]
  - **extension** : Extension and delegate objects specific to [extname]
  - **types** : type objects and typing logic specific to [extname]
  - **visit** : visitors specific to [extname]
  - **parse** : symbols table, lexer and parser for [extname], *PPG* grammar file ([extname].ppg), lexer grammar file([extname].flex) The newext.sh script takes several parameter:

Listing 2.14: newext.sh Parameters

```
1 Usage: ./newext.sh dir package LanguageName ext
2   where dir          - name to use for the top-level
        directory
3                        and for the compiler script
4         package      - name to use for the Java package
5         LanguageName - full name of the language
6         ext          - file extension for source files
```

In addition, a class *ExtensionInfo.java* defines how the language extension will be parsed and type-checked. A file *Version.java* specifies the version of the language extension. Finally, the class *Main.java* brings all the parts together and performs the compilation.

- **[extname]/tests/** : sample [extname] source code test files

The second step is to define the language modifications. This is done by modifying the [extname].ppg file, which is processed by *PPG*. It specifies the changes to the base Java grammar required to generate a parser for the new language extension. Sometime the developer has to updated the lexer grammar file [extname].flex too, if new tokens are added. The full list of available instructions for the *PPG* grammar can be found on the *PPG* Project page [37]. They include:

- **extend S ::= productions**
  the specified productions are added to the nonterminal S.
- **override S ::= productions**
  the specified productions completely replace S

The *PPG* file also specifies how the parsed information is used to create a new AST. New AST nodes are instantiated through the language extension's NodeFactory, which has factory methods to create each AST node. This NodeFactory typically extends Polyglot's Java node factory, which is defined by the class `NodeFactory_c` class and implements the `NodeFactory` interface. This interface specifies all the factory methods for each node. Figure 2.8 depicts a UML diagram explaining the different classes involved in the node factory.

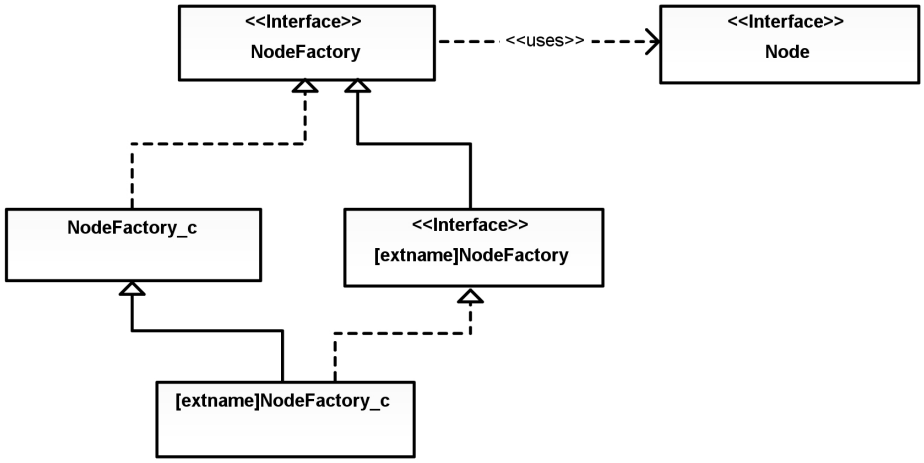


Figure 2.8: Language extension NodeFactory UML diagram
A language extension's node factory mechanism is split into two parts: an interface implementing the base node factory and a concrete class node factory extending the base concrete node factory.

The node factory can be accessed within the *PPG* file through the *parser.nf* instance. Listing 2.15 shows as an example the parsing and creation of an Assert Java base node.

Listing 2.15: Parsing and Creation of Assert AST Node

```
1 assert_statement ::=
2 ASSERT:x expression:a COLON expression:b SEMICOLON:d
3 {:
4   RESULT = parser.nf.Assert(parser.pos(x, d), a, b);
5 :};
```

This snippet defines the production rule assert_statement. It is defined by an assert symbol and an expression representing the assert condition, a colon and another expression representing the error message followed by a semicolon. For example: assert(size == 0) is a valid assertion. If the parsing is successful, a new Assert AST node is created through the parser.nf.Assert(Position,Expr) node factory method.

After defining the syntactic changes through the grammar and defining the new AST node classes for the language modification, the next step is to specify the new semantic changes. New passes can be defined by defining and including them in *Extensions.java*. Most of the time, semantic changes can be implemented directly as part of the type-checking pass. Note that each node has a method `visitChildren(NodeVisitor v)` that is called before the type-checking process in order to disambiguate the types of the Node's fields. New nodes defined for the language extension must therefore also execute the visit on each fields. This is done by overriding `visitChildren (NodeVisitor v)` and passing the instance of the visitor to each field using the method `visitChild(Node,Visitor)`. For the sake of completeness, Listing 2.16 illustrates this mechanism and shows the method `visitChildren` of the Assert node.

Listing 2.16: Assert node's visitChildren(NodeVisitor) method

```
/** Visit the children of the statement. */
public Node visitChildren(NodeVisitor v) {
  Expr cond = (Expr) visitChild(this.cond, v);
  Expr errorMessage = (Expr) visitChild(this.errorMessage, v);
  return reconstruct(cond, errorMessage);
}
```

The Swap node has two fields: the condition expression (this.cond) and the error message (this.errorMessage). Both are passed to the visitChild method. The node is then reconstructed using the returned instances.

Type checking is done in each node by the method `typeCheck(ContextVisitor tc)` of a Node. If a type error exists the method throws a `SemanticException`. To continue with our example of the Assert node, Listing 2.17 illustrates type checking for an assert statement.

Listing 2.17: Assert node's type checking

```
public Node typeCheck(ContextVisitor tc) throws SemanticException {
        if (! cond.type().isBoolean()) {
            throw new SemanticException("Condition of assert
                statement " +
                                        "must have boolean type.",
                                        cond.position());
        }

        if (errorMessage != null && errorMessage.type().isVoid()) {
            throw new SemanticException("Error message in assert
                statement " +
                                        "cannot be void.",
                                        errorMessage.position());
        }
        return this;
    }
```

The method type-checks if the expression cond is of type boolean and if the expression *errorMessage* is defined and not void.

In addition, the language developer can access the `TypeSystem` instance through the `ContextVisitor`. The `TypeSystem` instance defines the types of the language and how they are related. For example, it is used to compare the equality between two types, set new types on the expressions or check if a type can be cast to another type. Listing 2.18 shows an example of using the `TypeSystem` from the `ContextVistor`.

Listing 2.18: Switch_c's node typechecking

```
 /** Type check the statement. */
public Node typeCheck(ContextVisitor tc) throws SemanticException {
  TypeSystem ts = tc.typeSystem();
  Context context = tc.context();

  if  (!ts.isImplicitCastValid(expr.type(), ts.Int(), context)
      && !ts.isImplicitCastValid(expr.type(), ts.Char(), context))
    {
      throw new SemanticException("Switch index must be an integer
          .", position());
```

```
10      }
11          return this;
12  }
```

The index of a switch statement (switch(index)) can only be a char type or an integer type. However, any type that can be cast to an int or a char is also allowed. For example, an Integer or a short is valid but a String isn't. The Switch_c's typeCheck method gets the type system from the instance tc of the ContextVisitor and then calls the method isImplicitCastValid(Type, Type, Context) to perform the casting checks.

The final step after the semantic analysis of each AST node is to produce valid Java code. There are several options available to the language extension developer.

First, the most extensible way is to introduce a separate pass that transforms the language extensions AST nodes to valid Java AST nodes and then rely on the default Java AST translation pass to output valid Java source code. New passes can be added by extending the default Polyglot scheduler: `JLScheduler`. One would then have to create a pass by extending an appropriate Polyglot visitor class and schedule the pass before the `CodeGenerated` pass, which is responsible for translation. The created pass will be responsible for rewriting language extensions AST nodes into Java AST nodes using the `NodeFactory` methods. In addition, one can also use the Polyglot *quasiquoting* feature, which enables the generation of Java AST nodes from a String (*polyglot.qq.QQ*). This standard technique ensures that the output is well-formed Java code.

Secondly, another way to translate to Java code is to simply override the method `prettyPrint(CodeWriter, PrettyPrinter)` of each new Node. This method is responsible for printing the generated code to the output file and is called by the default implementation of the method `translate(CodeWriter, Translator)`, which is called during the Translation pass. Although this is a quick and easy way to perform code generation, it makes it harder to extend the modified language later. Furthermore, Polyglot won't check that the generated Java code is valid, so errors may show up when the code is compiled.

As an example, Listing 2.19 shows the code generation for the Assert AST node and Listing 2.20 shows the code generation for the Throw AST node.

Listing 2.19: Assert node translation

```
1   /** Write the statement to an output file. */
2  public void prettyPrint(CodeWriter w, PrettyPrinter tr) {
3    w.write("assert ");
4    print(cond, w, tr);
5
6    if (errorMessage != null) {
7      w.write(": ");
8      print(errorMessage, w, tr);
9    }
10   w.write(";");
11 }
12
13 public void translate(CodeWriter w, Translator tr) {
14   if (! Globals.Options().assertions) {
15     w.write(";");
16   }
17   else {
18     prettyPrint(w, tr);
19   }
```

```
20 }
```

The translate method from the Assert node by default calls the prettyPrint method which handles the code generation to the output file. Note that the print(Node child, CodeWriter w, PrettyPrinter pp) method will handle the code generation for the Node instance child. Essentially it calls its prettyPrinter method.

Finally, the language extension compiler is ready and can be used by running *Main.java*.

Listing 2.20: Throw node translation

```
1 /** Write the statement to an output file. */
2 public void prettyPrint(CodeWriter w, PrettyPrinter tr) {
3   w.write("throw ");
4   print(expr, w, tr);
5   w.write(";");
6 }
```

Similarly to the Assert node, the Throw node's prettyPrint method handles code generation and writes code to the output file.

2.3.3 SwapJ

In this section, we bring the concepts introduced about Polyglot together to show how to create a compiler for a language extension. We create *SwapJ*, a language that extends Java with a swap functionality. The modification made to Java is simple: we introduce a new swap(x,y) keyword that swaps the contents of the arguments x and y if they are of the same type. We don't support swapping array accesses and field accesses, for simplicity.

We start by formally defining the syntax changes to the Java base grammar and also provide the semantics and type systems for the swap operation. After, we work step by step and implement the *SwapJ* compiler.

Formal Definition

We describe the syntax of our new built-in swap operation:

Listing 2.21: SwapJ BNF

```
1 <statement> ::= "swap" "(" <identifier> "," <identifier> ")" ";"
2                 | <statement>
```

We define the operational semantics of our swap operation. Swapping the two variables x and y means to assign the content of y to x and assigning the content of x to y in the store ϕ:

Swap$_{OS}$

$$\frac{\mathsf{stmts}, \phi[\mathsf{x} \mapsto \phi(\mathsf{y}), \mathsf{y} \mapsto \phi(\mathsf{x})], \chi \rightsquigarrow \mathsf{v}', \chi'}{\mathsf{swap}(\mathsf{x}, \mathsf{y}); \mathsf{stmts}, \phi, \chi \rightsquigarrow \mathsf{v}', \chi'}$$

We also define the type rule of our swap operation:

Swap$_{TS}$

$$\frac{\mathsf{P}, \Gamma \vdash \mathsf{x} : \mathsf{t} \qquad \mathsf{P}, \Gamma \vdash \mathsf{y} : \mathsf{t}}{\mathsf{P}, \Gamma \vdash \mathsf{swap}(\mathsf{x}, \mathsf{y}) : \mathsf{void}}$$

Implementation

1. **Build skeleton extension**
   As explained in the previous section, the first step is to run *newext.sh* to build the skeleton files and directories for our language extension:

   Listing 2.22: Creation of SwapJ files structure

   ```
   ./newext.sh swapJ swapj SwapJ sj
   ```

   The directory containing the skeleton file structure is SwapJ. The package name is swapj, the language name is SwapJ and the extension file for our SwapJ source files is .sj

2. **PPG grammar specification**
   The next step is to specify the syntactic grammar differences to the Java base grammar. We translate our BNF specification into *PPG* grammar and specify the changes in swapJ.ppg (Listing 2.23) as explained in the previous section. Since we are adding a new token `swap` we also have to modify the lexer grammar file (Listing 2.24).

   Listing 2.23: PPG grammar for SwapJ

   ```
   1
   2 terminal Token SWAP;
   3 non terminal Stmt swap_stmt;
   4
   5 start with goal;
   6
   7 swap_stmt ::= SWAP:a LPAREN name:l COMMA name:r RPAREN
         SEMICOLON:b {:
   8   RESULT = parser.nf.Swap(parser.pos(a,b),l.toExpr(), r.toExpr
           ());
   9 :};
   10
   11 extend statement_without_trailing_substatement ::= swap_stmt:a
         {: RESULT = a; :};
   ```

   We extend the Java statement and add a new Swap statement. Note that we also added a token SWAP that is defined in the lexer grammar file.

   Listing 2.24: Lexer grammar for SWAP token

   ```
   1  keywords.put("swap",          new Integer(sym.SWAP));
   ```

3. NodeFactory and AST Nodes

The next step is to define the new SwapJNodeFactory and create the necessary AST node. Listing 2.23 shows that if parsing is successful, a node `Swap` will be created through the node factory. We now need to specify the interfaces required, implement the concrete classes and follow UML diagram 2.8 describing the AST node creations.

We create an interface `Swap` (Listing 2.25), a concrete class `Swap_c` (Listing 2.26) providing the implementation, a new factory interface `SwapJNodeFactory` (Listing 2.27) which provides the template for a swap node creation but also implements the base `NodeFactory` interface and the concrete node factory `SwapJNodeFactory_c` (Listing 2.28) that handles the instantiation of concrete Swap nodes. UML diagram 3 summarises the hierarchy and relations between the different classes.

Listing 2.25: Swap interface

```
1 public interface Swap extends Stmt{
2
3 }
```

A swap node is a statement and therefore implements the Stmt interface.

Listing 2.26: Swap_c concrete class constructor

```
1  public class Swap_c extends Stmt_c implements Swap{
2
3    private Expr left_e;
4    private Expr right_e;
5
6    public Swap_c(Position pos, Expr left_e, Expr right_e) {
7      super(pos);
8      this.left_e = left_e;
9      this.right_e = right_e;
10
11   }
```

The constructor of a Swap node takes the Position in the source file from the parser and also two expressions representing the left variable and right variable to swap. The Swap_c also extends the Stmt_c class, which encapsulates behaviour of any Java statement.

Listing 2.27: SwapJNodeFactory interface

```
1 /**
2  * NodeFactory for swapJ extension.
3  */
4 public interface SwapJNodeFactory extends NodeFactory {
5
6   Swap Swap(Position pos, Expr left_e, Expr right_e);
7
8 }
```

Listing 2.28: SwapJNodeFactory_c concrete class

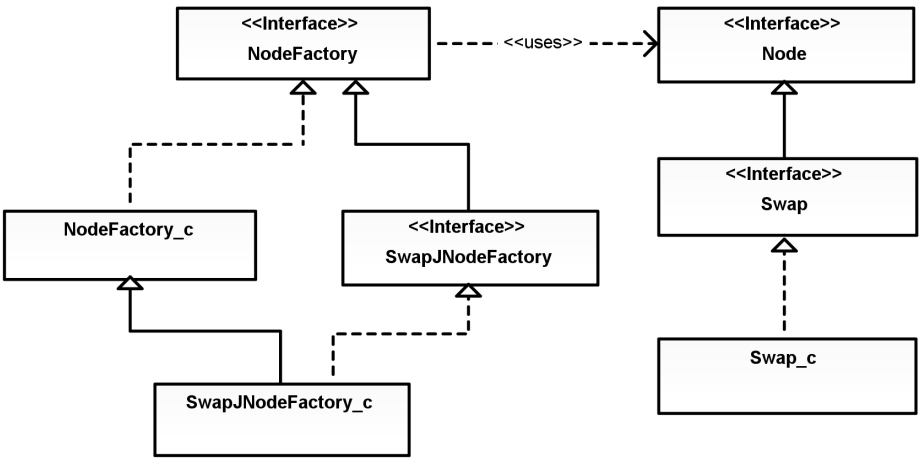
```
/**
 * NodeFactory for swapJ extension.
 */
public class SwapJNodeFactory_c extends NodeFactory_c
    implements SwapJNodeFactory {

  public Swap Swap(Position pos, Expr left_e, Expr right_e) {
    return new Swap_c(pos, left_e, right_e);
  }

}
```

The factory method Swap(Position, Expr, Expr) returns a concrete node Swap_c

Figure 2.9: SwapJ class diagram



4. **Semantic changes**
   The next step is to perform type checking on the swap operation: we need to ensure that the two arguments to swap are of the same type. As explained in the previous section, type checking is performed by the `typeCheck(ContextVisitor)` method of each node. We override this method so it checks if the two expressions of the Swap node are of the same type. Note that we also need to override the method `visitChildren(NodeVisitor)` to ensure the types of the Swap node arguments are disambiguated. Listing 2.29 shows how the `Swap_c` concrete class' methods `visitChildren` and `typeCheck` are implemented.

Listing 2.29: Swap node type checking

```
public Swap reconstruct(Expr expr_l, Expr expr_r) {
  if (this.left_e != expr_l || this.right_e != expr_r) {
    Swap_c n = (Swap_c) copy();
```

```
4       n.left_e = expr_l;
5       n.right_e = expr_r;
6       return n;
7     }
8     return this;
9   }
10
11  @Override
12  public Node visitChildren(NodeVisitor v) {
13    Expr expr_l = (Expr) visitChild(left_e, v);
14    Expr expr_r = (Expr) visitChild(right_e, v);
15
16    return reconstruct(expr_l, expr_r);
17  }
18
19  @Override
20  public Node typeCheck(ContextVisitor tc) throws
        SemanticException {
21
22    SwapJTypeSystem ts = (SwapJTypeSystem)tc.typeSystem();
23
24    Type left_t = left_e.type();
25    Type right_t = right_e.type();
26
27    if(!left_t.typeEquals(right_t))
28    {
29      throw new SemanticException("swap() arguments of different
            types!");
30    }
31
32    return this;
33
34  }
```

The overridden visitChildren disambiguate the Swap_c's fields and return a disambiguated Swap node. The overridden typeCheck method uses the type systems to verify the type equality between the two swap's arguments.

5. **Translation and code generation**
   The final step is translation. Since the swap translation to Java source code is straightforward, we can override the `prettyPrint(CodeWriter w, PrettyPrinter tr)` method directly and define code generation as explained in the previous section. We also need a fresh variable name to perform the swap, and Polyglot provides a helper static method for this: `Name.makeFresh()`. Listing 2.30 shows how to perform the code generation of a Swap node.

Listing 2.30: Swap_c node's code generation

```
1   @Override
2   public void prettyPrint(CodeWriter w, PrettyPrinter tr) {
3
4     // fresh variable name
5     String fresh = Name.makeFresh().toString();
6
7     // int temp = y;
8
9     left_e.type().print(w);
10    w.write(" " + fresh + " = ");
```

```
11    print(right_e,w,tr);
12    w.write(";\n");
13
14    // y = x;
15    print(right_e,w,tr);
16    w.write(" = ");
17    print(left_e,w,tr);
18    w.write(";\n");
19
20    // x = y;
21    print(left_e,w,tr);
22    w.write(" = " + fresh);
23    w.write(";\n");
24  }
```

6. **Testing**
   The *SwapJ* compiler is now operational, and we can test code generation. We create a swapTest class written in *SwapJ* and compile it with the *SwapJ* compiler. Listing 2.31 shows the class written in SwapJ and Listing 2.32 shows the generated Java output.

Listing 2.31: swapTest class written in SwapJ

```
1  public class swapTest {
2
3      public static void main(String[] args) {
4
5    String x = "Swapj!";
6    String y = "Java";
7
8      swap(x,y);
9      // Java Swapj!
10     System.out.println(x + " " +y);
11     swap(x,y);
12     // Swapj! Java
13     System.out.println(x + " " +y);
14     }
15 }
```

Listing 2.32: swapTest class after compilation

```
1  public class swapTest {
2
3    public static void main(String[] args) {
4      String x = "Swapj!";
5      String y = "Java";
6      java.lang.String id0 = y;
7      y = x;
8      x = id0;
9
10     System.out.println(x + " " + y);
11     java.lang.String id1 = y;
12     y = x;
13     x = id1;
14
15     System.out.println(x + " " + y);
16   }
17 }
```

Summary

We showed how to quickly implement a simple language extension by using Polyglot. This section was written as a tutorial for researchers and students interested to develop language extensions. In the next part of the tutorial, we will go in further depth and explore the scheduling of passes in Polyglot: we will add an AST Rewriting pass to directly transform SwapJ nodes into Java nodes. Table 2.3.3 summaries the lines of code added for each new class introduced to implement the *SwapJ* language extension.

Figure 2.10: SwapJ code modifications summary

Classes	Lines of code
SwapJNodeFactory	14
SwapJNodeFactory_c	14
Swap	7
Swap_c	117
Total	152

Chapter 3

Design Space of First-Class Relationships

This chapter presents the design choices available to supporting first-class relationships in an object-oriented language and compare them to existing work. We present examples using *ImperialRJ*, which extends Java with first-class relationships.

3.1 Language Design Requirements

There are several requirements for an effective first-class relationship language to keep in my mind before designing a language that supports first-class relationships. This section summaries the different requirements suggested recently in the literature. [9, 7]

- **Abstraction** : a good relationship abstraction that hides internal implementation complexity and provides a simple interface to work with.
- **Polymorphism** : different relationship implementations should be easily interchangeable without modifications to the participants.
- **Reusability** : relationship abstractions should be reusable just like classes can be reused by being instantiated several times.
- **Composition** : it should be possible to include the relationship implementation as the basis for another relationship. In other words, relationship implementations can be shared.
- **Separation of Concerns** : the participants should be decoupled from the relationship implementation so that they can be reused independently of the relationship if necessary
- **Syntax** : an intuitive syntax for expressing relationships

3.2 Terminology

We introduce the terminology used in the next sections:

- **Relationship:** A relationship abstracts the implementation of object collaborations. Relationships are the implementation abstraction of the UML association concept.
- **Participants:** The participants of a relationships are the objects' classes involved in the relationships.
- **Roles:** The participants of a relationship can be named to indicate the role they play within the relationship declaration. [41, 42, 13]
- **Tuple:** A tuple represents a possible association between the participants of a relationship.
- **Extent:** An extent represents the instance result of a relationship just like an object is the instance result of a class. In other words, it represents a set of tuples.

3.3 Exploring The Design Space

In this section we investigate the design choices available to support first-class relationships:

1. **First-Class Extents**
   Do we need one or many sets of tuples?
2. **First-Class Tuples**
   What does it mean to support first-class relationships?
   Are tuples first-class?
3. **State**
   Can tuples have processing ability?
   Can tuples have a state?
4. **Encapsulation**
   Do tuples have an existence outside an extent?
   How does an extent encapsulate a tuple?
   Do tuples store references or copies to the participants objects?
5. **Aliasing**
   How to prevent confusing situations due to tuple aliasing?
6. **Duplicates**
   Are duplicates of tuples possible within an extent?
7. **Arity**
   Can we support n-ary relationships?
8. **Relationship Constraints**
   How can we set constraints on the participants?
   Can we support multiplicity?
   How can we set constraints on the state of a tuple?

9. **Relationship querying**
   What are the possible ways of supporting querying of an extent?
   Can we support mathematical binary relations on extents?

10. **Persistence of Relationships**
    Is there a connection between first-class relationships and databases?

11. **Specialising Relationships**
    What are the issues involving relationship inheritance?
    How does it affect consistency constraints?
    Can we introduce relationship polymorphism?
    How to deal with covariant overriding of participants in a sub-relationship?

The table below in Figure 3.3 compares the existing languages supporting relationships with our design space [43, 6, 7].

	DSM	**RelJ**	**Rumer**	**ImperialRJ**
First-Class Extents	No	No	Yes	Yes
First-Class Tuples	No	Yes	**?**	No
Tuple State	?	Yes	Yes	Yes
Tuple Processing ability	**?**	**?**	Yes	In development
Encapsulation	by reference	by reference	by reference	by reference
Aliasing Tuples Protection	**?**	**?**	**?**	Aliasing of Tuples forbidden
Duplicate Tuples	No	No	**?**	In development
N-ary relationships	**?**	No	No	No
Multiplicities	Yes	Yes	Yes	In development
Constraints	**?**	No	Yes	In development
Querying	Yes	No	Yes	Yes
Aggregation functions	**?**	No	Yes	Yes
Relationship Persistence	**?**	No	No	In development
Relationship Inheritance	No	Yes	**?**	In development

Figure 3.1: Design Comparison of Existing Relationship Languages

3.3.1 First-Class Extents

Previous work have modelled relationships as a static concept that is available implicitly to the system [6, 7]. However, this approach reduces reusability since only one set of tuples is possible for a relationship declaration. Another alternative is to allow several extents from the same relationship declaration. We explore the two alternatives in terms of the following example: A *Department* has *Students* attending *Courses*. This is depicted in Figure 3.2.

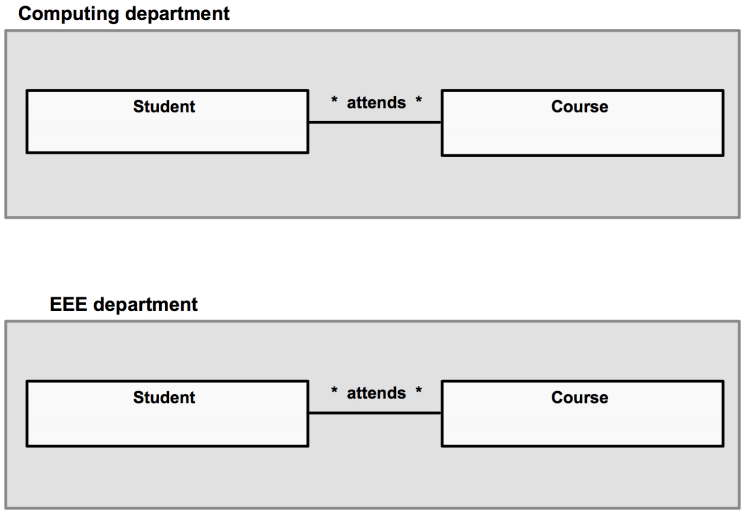


Figure 3.2: The Departments EEE and Computing have Students attending Courses

In the first alternative, if we only have one extent then modelling the relationship requires a third participant to take the *Department* into account: $Attends \subseteq Department \times Student \times Course$. This solution has several drawbacks. Firstly, it requires support for relationship of more than 2 participants, which are harder to manipulate, reason about and navigate. Secondly, the relationship now couples together 3 participants and cannot elegantly be used for the simple Attends relationship: $Attends \subseteq Student \times Course$. However, one advantage is that the relationship is available globally since it only has one extent, which makes it easier to access it.

In RelJ, one could define two relationships with the same signature but different names for each department to avoid using a relationship with three participants as shown in Listing 3.1. However, this solution is clearly not viable if there are lots of departments or worse if the system needs to be flexible for changes: new departments are created or renamed and old departments removed.

Listing 3.1: Considering Departments in RelJ

```
1 relationship ComputingAttends(Student,Course){
2 }
3
4 relationship EEEAttends(Student, Course){
```

```
5 }
6
7 // department EEE
8 EEEAttends.add(sophia,signals);
9 EEEAttends.add(raoul,digital);
10
11 // department Computing
12 ComputingAttends.add(raoul,oop);
13 ComputingAttends.add(michael,oop);
```

In the second alternative, we simply allow several extents of the same relationship declaration. This way the relationship definition remains unchanged and we can reuse the same relationship declaration. We demonstrate the concept in Listing 3.2 using *ImperialRJ*. One disadvantage of this mechanism compared to a global extent is that the programmer needs to remember the different names for the extents of a relationship he is creating.

Listing 3.2: Multiple extents for the same relationship

```
1 relationship Attends(Student,Course){
2 }
3
4 // department EEE
5 Attends eee = newR Attends();
6 eee->add(sophia,signals);
7 eee->add(raoul,digital);
8
9 // department Computing
10 Attends computing = newR Attends();
11 computing->add(michael,oop);
12 computing->add(raoul,oop);
```

Extents in *ImperialRJ* are created using the `newR` keyword, intuitively similar to the `new` keyword for instantiating classes.

The decision to allow several extents leads to the decision to make extents first-class entities. Extents can be constructed at runtime and assigned to a variable and can also be passed as a parameter or returned from a method.

The example described can now be elegantly modelled in the UML diagram 3.3. The `Attends` relationship is a field of a `Department`.

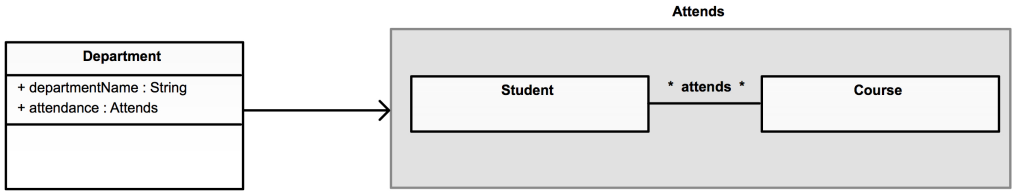


Figure 3.3: Departments have Students attending Courses

Note that the introduction of extents as first-class entities also enables relationship aggregation: an extent can be passed as a parameter to another relationship. Listing 3.3 shows an example of a `Recommends` relationship which takes a `Attends` extent as second participant.

Listing 3.3: Relationship Aggregation

```
relationship Attends(Student, Course)
{
  int mark;
}

public relationship Recommends(Tutor, Attends)
{
  String reason;
}

Attends attends = newR Attends();
Recommends recommends = newR Recommends();

a->add(raoul, oop);
a->add(sophia, oop);

// Tutor Michael recommends students Raoul and Sophia to attend OOP
recommends.add(michael, a).withReason("Do as I say! It's good for
    you");
```

3.3.2 First-Class Tuples

The support of tuples as first-class citizens means that we can store tuples in a variable and also pass them as arguments to methods. Such a design choice opens the question of whether a tuple exists outside an extent. As described in Section 2.2.2, first-class tuples raises issues regarding the fate of the tuple reference when the tuple is removed from an extent. Wren et al. suggested three solutions: taking no actions, deleting the tuple from the heap and nullifying references. The first option is not an accurate solution as it just doesn't deal with the issue and the last two introduce security risks when dealing with dangling references. In addition, Wren et al. suggestions do not cater for multiple extents. Consider the case when the same tuple is part of two different extents. What happens to the tuple reference if the tuple is removed from one extent or both?

Supporting first-class tuples also puts more overhead on the programmer because he has to keep track of the various tuple references, which makes it harder to reason about the program and leads to the typical aliasing problems like accidental conflicts [44]. However, it is easier to remember a tuple reference than refer to the participant objects to access the tuple.

An alternative design choice, is to take a more protective approach and only allow access to tuples through their extent and forbid reference to tuples. This mechanism ensures that no problems arises after removal of tuples as we don't need to deal with tuple references at all. However, first-class tuples could be useful in some situations as wrapper objects or as temporary storage for manipulation. In other words, in certain situations it could be more practical to process tuples one by one rather than accessing the whole extent holding them, in which case first-class tuples are desirable. In addition, accessing the tuples through the extent requires the programmer to write more code as he needs to keep track of both the extent and the participant objects of the desired tuple.

One possible solution to make life easier to the programmer is to introduce the generation of a macro, which groups the participant objects of a tuple.

Deal with tuple references

We give an overview of a mechanism to deal with tuple as first-class citizens. We differentiate between three states of the tuples:

1. The tuple hasn't been removed and it exists in an extent and has an external reference to it
2. The tuple has been removed and therefore doesn't exist in an extent but it still has an external reference to it
3. The tuple reference has been cleared and the tuple doesn't exist anymore

The three different scenarios are shown in Figure 3.4, 3.5 and 3.6. A possible choice is to let the programmer be responsible for clearing references to tuples that don't exist anymore if he decides to store tuples in variables. This principle is similar to allocating and deallocate a memory block in C: the programmer allocate an external reference to the tuple if he wish and then deallocates it. Listing 3.4 show cases a possible syntax in *ImperialRJ* for this mechanism.

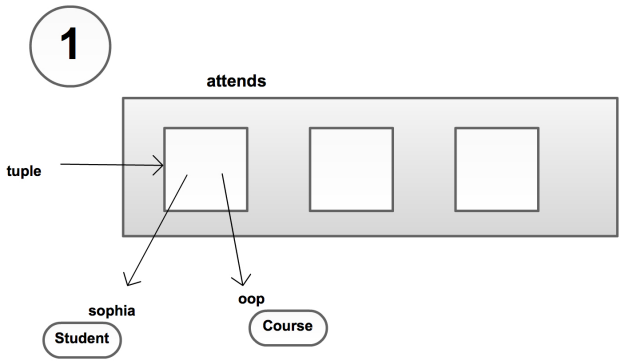


Figure 3.4: The tuple hasn't been removed

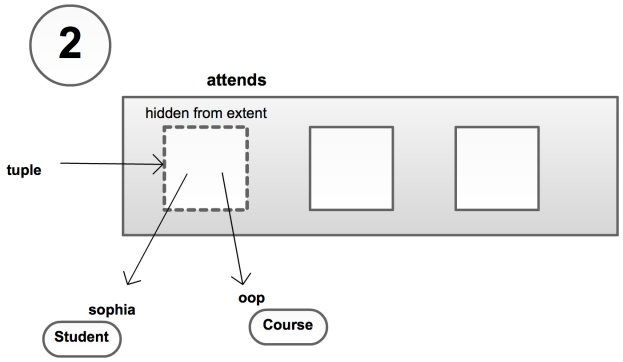


Figure 3.5: The tuple is hidden from the extent but an external reference exists to it

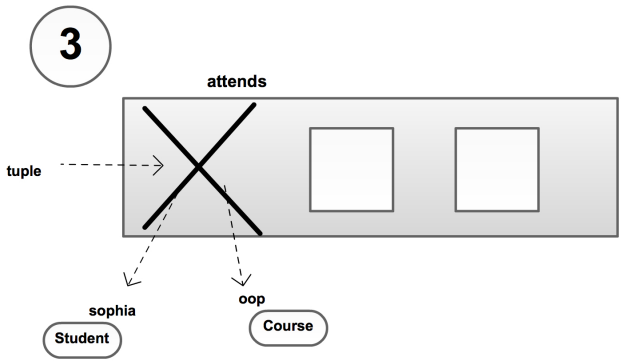


Figure 3.6: The tuple reference has been removed and the tuple structure cleared

Listing 3.4: The three states of a first-class tuple described in *ImperialRJ*

```
relationship Attends(Student,Course) : Attendance {
}

Attends attends = newR Attends();
Attendance tuple = attends->add(sophia,oop); // 1

attends->rem(sophia,oop); // 2
tuple.first(); // returns sophia
attends->get(sophia,oop); // invalid

tuple.free(); // 3
tuple.first(); // invalid
```

3.3.3 State and Processing Ability

Tuple State

Tuples can hold a state to conveniently include more information about a relationship. As an example, consider the relationship `Attends` where `Students` attend a `Course` but also get a `mark` for it. One way to include the mark in the relationship is to simply have a ternary relationship: $Attends \subseteq Student \times Course \times Mark$. However, as mentioned earlier, this requires support for n-ary relationships, which are harder to work with. Furthermore, the `mark` is not actually defining the `Attends` relationships but is rather an attribute of it. In addition, adding new attributes would require constantly modifying the `Attends` relationship definition.

An alternative is to allow tuples to hold the `mark` for a `Student` attending a `Course` and declaring the `mark` as an attribute of the relationship rather than a participant. Listing 3.5 shows how to manipulate tuple states in *ImperialRJ*. This mechanism makes it easy to navigate the relationship participants while at the same time query and modify the states of tuples if necessary.

Listing 3.5: A relationship attribute mark as tuple state

```
relationship Attends(Student,Course){
  int mark;
}

Attends attends = newR Attends();
attends->add(sophia,oop).withMark(8);
attends->add(michael,oop).withMark(10);
```

Tuple Processing Ability

It can also be convenient to define methods which manipulate a tuple within the relationship, especially if tuples are not first-class entities. These methods can access the state of the tuples through the relationship attributes and also the participant objects in the tuple by defining roles on the relationship declaration. Listing 3.6 shows an example where a tuple can generate an attendance report.

Listing 3.6: Tuple processing ability

```
relationship Attends(Student s, Course c){
  int mark;

  void printReport()
  {
    System.out.println(this.s + " is attending " + this.c " and has
        grade: " + this.mark);
  }
}

Attends attends = newR Attends();
attends->add(sophia,oop).withMark(8);
attends->add(michael,oop).withMark(10);

// sophia is attending oop and has grade 8
attends->get(sophia,oop).printReport();
```

Extent

Similarly to tuples, it can be convenient to encapsulate attributes and processing ability at the extent level. As an example consider Listing 3.7, which displays all the tuples within an extent. The keyword `extent` differentiate the declarations at the tuple level from the extent level within a relationship declaration. Notice that the keyword `this` within an extent declaration refers to the top extent level otherwise it refers to the current tuple.

Listing 3.7: Extent processing ability

```
relationship Attends(Student s, Course c){
  int mark;

  void printReport()
  {
    System.out.println(this.s + " is attending " + this.c " and has
        grade: " + this.mark);
  }

  extent void printAllReports()
  {
    for( (Student s, Course c) )
    {
      this.get(s,c).printReport();
    }
  }

}

Attends attends = newR Attends();
attends->add(sophia,oop).withMark(8);
attends->add(michael,oop).withMark(10);

// sophia is attending oop and has grade 8
// michael is attending oop and has grade 10
attends->printAllReports();
```

3.3.4 Encapsulation

What is actually stored within a tuple? One option is to store references to the objects composing a tuples. As a consequence, the extent does not encapsulate the participants of a tuple: the participant objects composing a tuple can be manipulated outside the extent. This is illustrated in Figure 3.7.

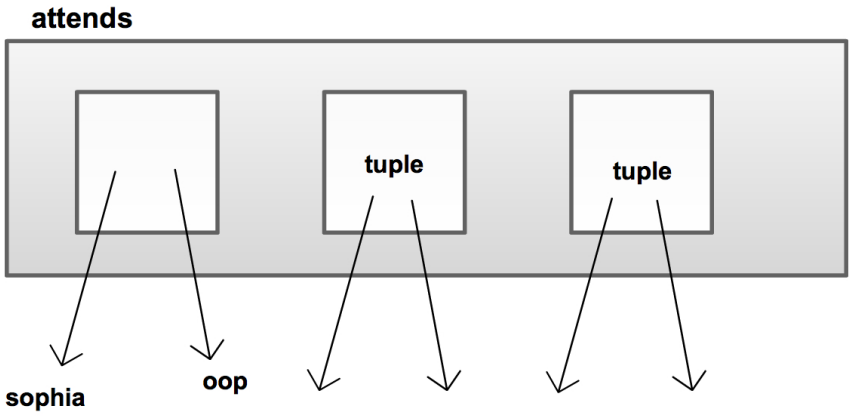


Figure 3.7: Storing participant objects references

A design alternative to storing the participant objects references is to store deep copies of the participant objects. This mechanism ensures that the extent completely encapsulates its tuples. However, this mechanism is expensive because a deep copy of the participant objects is required as well as equality comparisons for every methods of the extent storing the copies. This mechanism is illustrated in Figure 3.8.

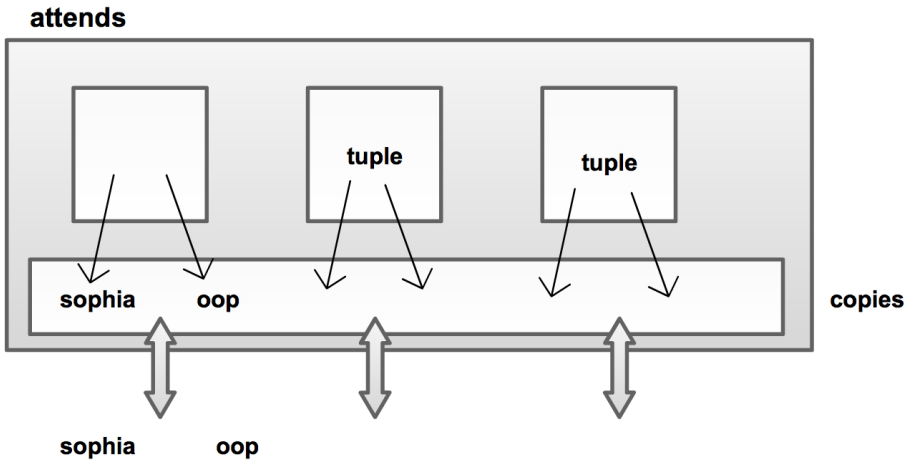


Figure 3.8: storing copies of participant objects

Similarly the introduction of attributes on tuples raises encapsulation issues if the state of a tuple is composed out of references to objects outside the extent. One solution is to only allow primitive values as attributes but this could be too restrictive.

3.3.5 Aliasing

The introduction of first-class tuples can lead to confusing situations. Take as example Listing 3.8 where two tuples references from two different extents of the same type are returned. Could a tuple reference be assigned to another one? In which case what happens after removal of the tuple within the extent and the deallocation of the tuple reference? Can the alias of the tuple on which `free()` wasn't called still access the tuple or not?

Listing 3.8: Confusion after tuple assignment

```
1  relationship Attends(Student,Course) : Attendance {
2  }
3
4  Attends a1 = newR Attends();
5  Attends a2 = newR Attends();
6
7  Attendance t1 = a1->add(sophia,oop).withMark(10);
8  Attendance t2 = a2->add(sophia,oop).withMark(9);
9
10 t1 = t2; // assignment
11 a2->rem(sophia,oop);
12 t1.free();
13
14 // 9?
15 t2.getMark();
```

Situations when a removed tuple has several aliases can lead to problems. In fact, it can accidentally modify the behaviour of the other aliases pointing to the same tuple unbeknownst to the programmer.

One possible strategy is to enforce control of the tuples alias through ownership types: a tuple reference from one extent cannot be assigned to another tuple reference from another extent even if the extents hold the same types because the tuples are considered to have different owners. Even though this mechanism can help clarify situations when two different extents are involved, it still doesn't solve the problem when two references from the same extents are assigned.

An alternative solution is to enforce immutability of the references pointing to a tuple. This way, two different tuples references cannot be reassigned and the problem doesn't occur.

Another less restrictive approach is to enforce read-only references to tuple, which cannot modify the state of the tuple they refer to after reassignment. This is technique is further described by Tschantz and Ernst in their work about extending Java with reference immutability [45]. In the example shown above, `t1.free()` wouldn't be allowed because `t1` cannot modify the state of the tuple it points to, only `t2` is entitled to do so.

3.3.6 Duplicates

Another question to raise is how to deal with duplicate tuples? The mathematical definition of a tuple only consider the participants of a tuple but not its state. We suggest three options that take into account the state of tuples:

1. Prevent duplicate tuples based on the participants and throw a runtime exception if a duplicate is added.
2. Override the existing tuple with the new one since the tuple state may be different.
3. Allow duplicate tuples with different states to be stored. When a specific tuple is required, a random or all are returned.

The desired behaviour can be annotated at compiled time. Listing 3.9 and 3.10 show an example in *ImperialRJ* using the annotations `@AllowOverrideTuples` and `@AllowDuplicateTuples`.

Listing 3.9: AllowOverrideTuples annotation

```
1  @AllowOverrideTuples
2  relationship Attends(Student s, Course c)
3  {
4    int mark;
5  }
6
7  Attends a = newR Attends();
8  a->add(sophia,oop).withMark(10);
9  a->add(sophia,oop).withMark(8); // override
10
11 // 8
12 a->get(sophia, oop).getMark();
```

Listing 3.10: AllowDuplicateTuples annotation

```
1  @AllowDuplicateTuples
2  public relationship Attends(Student s, Course c)
3  {
4    int mark;
5
6    void printReport()
7    {
8      System.out.println(s + " is attending " + c + " and has grade:
           " + this.mark);
9    }
10
11   extent public void printAllReports()
12   {
13     for( (Student s, Course c))
14     {
15       this.get(s,c).printReport();
16     }
17   }
18 }
19
20 Attends a = newR Attends();
21 a->add(sophia,oop).withMark(10);
22 a->add(sophia,oop).withMark(8);
```

```
23
24 // Sophia is attending OOP and has grade 10
25 // Sophia is attending OOP and has grade 8
26 a->printAllReports();
```

3.3.7 Arity

Could we implement relationships with more than two participants? This is possible by extending the relationship declaration to support a third participant object. However, dealing with more than two participants makes the relationship harder to navigate as a third dimension is introduced. An alternative solution, is to simply define two principal participants and consider the other participants as attributes which can be stored within the tuples of the relationship.

Consider the *Usage* relationship illustrated in Figure 3.9. A particular `Room` is booked at a certain `Time` for an `Activity`. Clearly the relationship is defined as $Usage \subseteq Room \times Activity \times Time$.

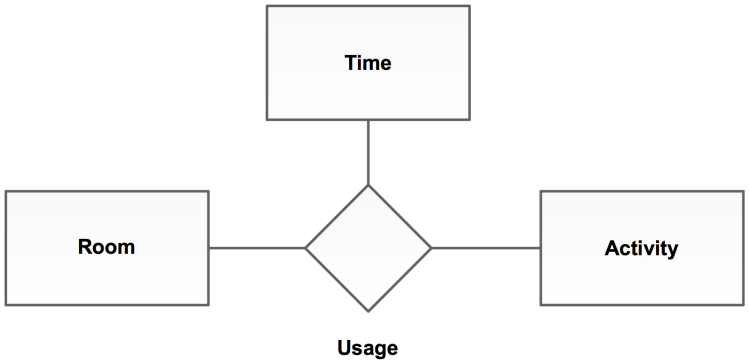


Figure 3.9: Storing copies of participant objects

This can simply be defined as a relationship $Usage \subseteq Room \times Activity$, which has an attribute `Time`. Listing 3.11 illustrates this example in *ImperialRJ*.

Listing 3.11: Time as an attribute of the Usage relationship

```
1 relationship Usage(Room, Activity)
2 {
3   Time time;
4 }
5
6 Usage usage = newR Usage();
7 usage->add(311,tutorialOOP).withTime(10am);
```

3.3.8 Relationship Constraints

Multiplicity

As described in the Background section, associations can be annotated with multiplicities to restrict the number of instances of participants within a rela-

tionship. There are two ways to check multiplicities within a language implementation: runtime and static checking.

Runtime checking makes it easy to validate complex multiplicities including ranges, for instance `2..5`. The reason is because the check can be executed before adding or removing a new tuple to validate whether the multiplicity hold. However, this behaviour is not necessarily desirable because it means the programmer needs to cater for possible exceptions at runtime and write exceptions handler. Listing 3.12 shows an example on how to express such a multiplicity in *ImperialRJ*. Students are allowed to take between 6 and 8 courses.

Listing 3.12: Multiplicity On Participants

```
1 relationship Attends(many Student, 6..8 Course)
2 {
3   int mark;
4 }
```

Static checking prevents problems from arising at runtime because the validation was done at compile time. However, validating complex multiplicities would require intricate data-flow analysis. In fact, one would need to track every conditional branches and scenarios in order to find a violation of the multiplicity and this may not be possible in practice.

Note that for languages that only implements binary relations, multiplicities can also be specified on relationship attributes. For example, Listing 3.13 conveys that a `Student` can get up to three `marks` per `Course` that he is attending.

Listing 3.13: Multiplicity On Relationship Attribute

```
1 relationship Attends(Student, Course)
2 {
3  0..3 int mark;
4 }
```

Constraints On Participants and Relationship Attributes

Further constraints can be set on the participants and relationship attributes in order to ensure only valid information goes inside an extent and its tuples. One possible way to express these constraints is to specify declarative validations, which are checked at runtime. Listing 3.14 shows an example in *ImperialRJ* using a `constraints` block inspired from *Grails* [46]. Participants of the relationship are annotated with roles so they can be referred to inside the relationship declaration. The `constraints` block declares that `mark` must be in the range 1 to 10 and that the number of teaching hours of the `course` must be greater than 16.

Listing 3.14: Constraints On Relationship Attributes and Participants

```
1 relationship Attends(Student student, Course course)
2 {
3   int mark;
4
5   constraints
6   {
7      mark(range(1..10);
8      course(hours > 16);
```

```
  }
}
```

Relationship Invariants

In addition, it is possible to verify properties on the structure of a relationship by specifying structural invariants. This concept has been described by Balzer et al and can be expressed in a language implementation using for instance an `invariants` block. Listing 3.15 shows an example where the relationship `Substitutes` is defined as *irreflexive* and *asymmetric*.

Listing 3.15: Constraints On Relationship Attributes and Participants

```
relationship Substitutes(Faculty substitute, Faculty substituted)
{
  invariants
  {
    irreflexive;
    asymmetric;
  }
}
```

3.3.9 Relationship Operations and Querying

Relationship Operations

Similar to the Java Collection library which incorporate various relationship operations like union and intersection, a first-class relationship language should implement these operations built-in on extents as they are useful for different use cases. For instance, take the example of `Departments` holding an `Attends` extent. The `union` of all of the `Attends` extents represents the attendance within an `University`. In addition, various mathematical binary relations like inverse and transitive closure can be implemented for when relationships are used in the context of algorithms. As an example to find reachability in a graph structure represented by a `Connect` relationship. It is also useful to define navigation built-in methods to quickly scan the instances available in the domain and image of a relationship.

We list below the most common relationship operations in the *ImperialRJ* syntax:

- `unionof a with b`
  returns the union of the extent `a` and `b`
- `intersectionof a with b`
  returns the intersection of the extent `a` and `b`
- `compositionof a with b`
  returns the composition of the extent `a` and `b`
- `a(-)`
  returns the inverse of the extent `a`
- `a=`
  returns the reflexive closure of the extent `a`

- `a:`
  returns the symmetric closure of the extent `a`
- `a*`
  returns the transitive closure of the extent `a`

As an example, given a graph structure represented in Figure 3.10, a visitor could get all the visitable cities on his way starting from Brussels by combining the transitive closure and the extent built-in `from()` method to navigate the instances in the image of Brussels as shown in Listing 3.16.

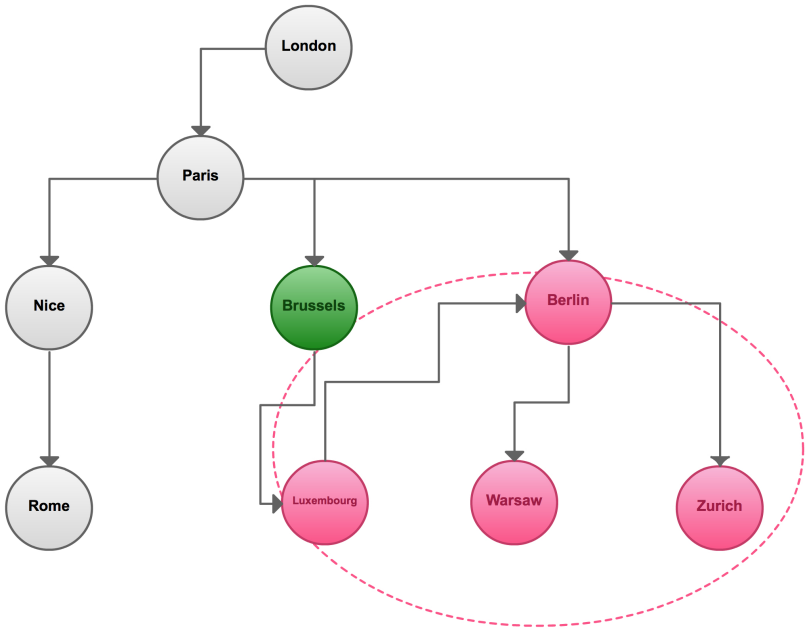


Figure 3.10: Cities linked together in a graph structure

Listing 3.16: Cities visitable starting from Brussels

```
1  public relationship Connect(City c1, City c2){
2  }
3
4  Connect cityConnections = newR Connect();
5  cityConnections->add(london,paris);
6  cityConnections->add(paris,nice);
7  cityConnections->add(nice,rome);
8  cityConnections->add(paris,brussels);
9  cityConnections->add(paris,berlin);
10 cityConnections->add(brussels,luxembourg);
11 cityConnections->add(luxembourg, berlin);
12 cityConnections->add(berlin, zurich);
13 cityConnections->add(berlin, warsaw);
14
15
16 // Set{luxembourg,berlin,zurich,warsaw}
17 (cityConnections*)->from(brussels);
```

Relationship Querying

Modern languages have little support for querying structured collections and objects [26]. As a consequence programmers are forced to handcode the queries, which can be inefficient. *LINQ* is an example of a language developed by Microsoft, which incorporate integrated queries on collections [47]. As relationships provide a structured interface for accessing data through extents and tuples, it is desirable to provide the programmer with built-in efficient querying constructs in order to prevent then from writing inefficient and error prone code. A possible way to express queries on relationship is through a `select()` construct acting on an extent. Common operations like *average*, *maximum*, *sum* and *minimum* performing calculations on relationship attributes could be defined separately like aggregate functions in SQL. Listing 3.17 shows how a filtering query is expressed and Listing 3.18 shows on how aggregate functions are expressed on an extent in *ImperialRJ*.

Listing 3.17: Filtering used macros

```
relationship Macro(Name, Definition)
{
  boolean used;
}

Macro macro = newR Macro();

macro->add(name1,d1).withUsed(true);
macro->add(name2, d2).withUsed(false);
macro->add(name3, d3).withUsed(true);

// filter all macros that are used
macro->filter(used == true);
```

Listing 3.18: Aggregate functions

```
relationship Follow(User, User)
{
  int interest;
}
// Twitter follow relation
Follow f = newR Follow();

// AVG
double avg = average interest from f;
// SUM
int sumInterest = sum interest from f;
// MAX
int maximum = max interest from f;
// MIN
int minimum = min interest from f;
```

3.3.10 Relationship Persistence

There exists a direct connection from associations in object oriented systems to relations in database systems. In fact, a relation in the database world is defined as a set of tuples, which conform to a common set of attributes. This

can be mathematically modelled as a relation over several sets where the sets are the possible attributes values of the database relation. A database relation is usually described as a table, which is organised into rows and columns. This means that relationships between entities can be modelled as a table. Figure 3.11 shows how the `Attends` relationship between `Student` and `Course` can be modelled using a relation over the attributes `StudentId` and `CourseId`.

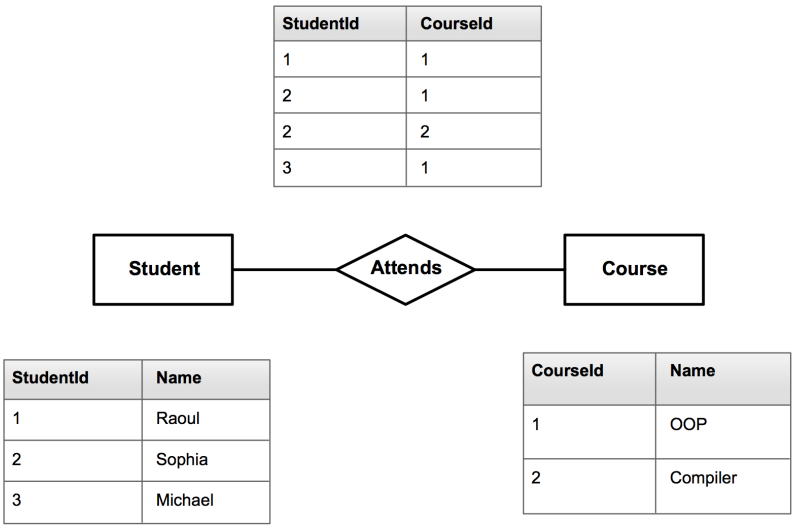


Figure 3.11: Attends relationships in Database

The same example is represented as a relationship in Object-Oriented systems.. How do we map the object model to a relational database? The two most common approaches are writing SQL conversion methods by hands or using sophisticated ORM frameworks. However, they are not intuitive and optimal for dealing with associations intra objects because object-oriented languages lack construct to explicitly define relationships. First-class Relationships can help facilitate the mapping of associations from an object-oriented system to a relational database.

The first approach available to the programmer is to set up SQL tables as well as write SQL conversion methods by hand to remove and add new tuples in the table persisting the relationship. This mechanism is inefficient for several reasons:

- it requires the programmer to keep the running system consistent with the database.
- it requires the programmer to write tedious and error prone code.
- the sql conversion methods are vulnerable to changes in the object-oriented system if new requirements comes in.

Another approach is to use a mapping system that acts transparently to the object model to store and retrieve objects directly to and from the database.

These systems are referred to as Object-Relational Mapping System (ORM). One of the most popular within the Java community is Hibernate. However, these sophisticated systems require a lot of knowledge to use. In addition they may not be optimal for dealing with associations as they map relationships that are implicitly defined in code since object-oriented languages are lacking explicit constructs. Moreover, such systems come with specialised querying languages, which introduce a mismatch between filtering logic in code and filtering logic on the relational database. As an example, Listing 3.19 show a mapping configuration for the `Attends` example, which assumes the `Student` class has a Set of `Courses` and the `Course` class a Set of `Students` to represent the relationship.

Listing 3.19: Filtering used macros

```
<class name="Student">
    <id name="id" column="studentId">
        <generator class="native"/>
    </id>
    <set name="courses" table="Attends">
        <key column="studentId"/>
        <many-to-many column="courseId"
            class="Course"/>
    </set>
</class>

<class name="Course">
    <id name="id" column="courseId">
        <generator class="native"/>
    </id>
    <set name="students" inverse="true" table="Attends">
        <key column="courseId"/>
        <many-to-many column="studentId"
            class="Course"/>
    </set>
</class>
```

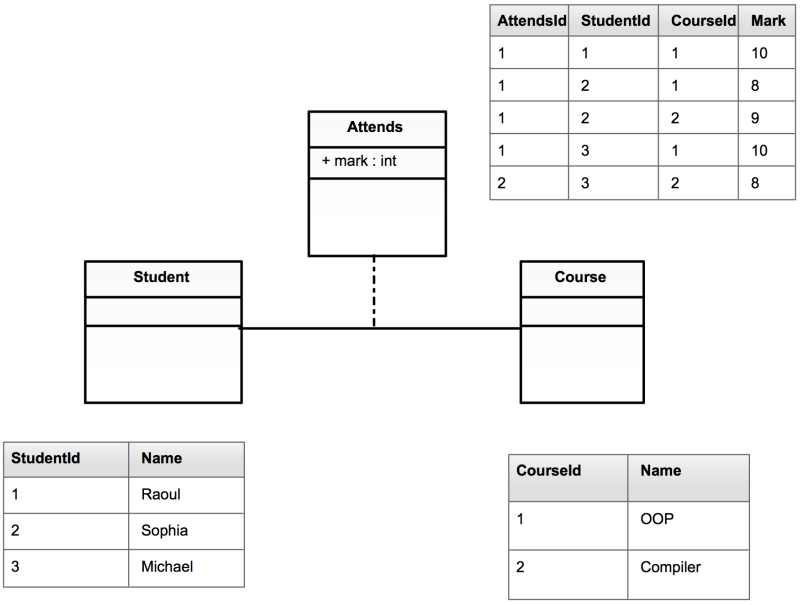
This mapping method of associations is undesirable for several reasons. First of all, it requires detailed knowledge of mapping associations to the database. Second, it is not optimal for changes. For example introducing a `mark` attribute for every attendance would require an overhaul of the configuration mapping as well as the object-oriented system code.

Since first-class relationships explicitly define relationship between objects, they could be used to automatically map the relationship to the relational database. There are several benefits:

- no detailed knowledge required of association mapping to relational database.
- optimal for new requirements about an association.
- transparent to the programmer.
- no mismatch between relational filtering within the object-oriented system and the relational database
- constraints from object-oriented model preserved to relational database

Figure 3.12 describe a graphical example of table generation from the `Attends` relationship declaration. Each `Attends` extent is assigned an `AttendsId`. The

table `Attends` associates tuple of `Students` and `Courses` with their `Mark` as well as their extent. The range constraint set on the mark attribute can be used to optimise the internal type on the relational database. For example to a byte value rather than an integer as only 10 values are used.



AttendsId	StudentId	CourseId	Mark
1	1	1	10
1	2	1	8
1	2	2	9
1	3	1	10
2	3	2	8

StudentId	Name
1	Raoul
2	Sophia
3	Michael

CourseId	Name
1	OOP
2	Compiler

Figure 3.12: Attends table generation with First-Class Relationships

The example above is generated from the following *ImperialRj* code:

Listing 3.20: *ImperialRJ* persistence

```
relationship Attends(Student, Course)
{
  int mark;
  constraints
  {
    mark(range(1..10);
  }
}

Attends a1 = newR Attends();
Attends a2 = newR Attends();

a1->add(raoul,oop).withMark(10);
a1->add(sophia,oop).withMark(8);
a1->add(sophia,compilers).withMark(9);
a1->add(michael,oop).withMark(10);

a2->add(michael,compilers).withMark(8);
```

3.3.11 Relationships Inheritance

Overview

One of the key benefits of class inheritance is to minimise duplicate code and enable reuse of code defined in a parent class by a child class. The same concept can be applied to relationships. Wren et al introduced relationship inheritance specifically to *RelJ* in a restricted form of delegation [7]. However, it is not suited for a language supporting first-class extents since RelJ doesn't support first-class extent.

We investigate a relationship inheritance implementation for *ImperialRJ* that is suited for first-class extents. It is similar to classes inheritance in Java: implementation of the parent relationship is inherited by a child relationship, however, the participants of the child relationship are a specialisation of the participants of the parent relationship.

A relationship can extend a parent relationship using the `extends` keyword and specialising the participants types. Figure 3.13 depicts a scenario, which requires relationship inheritance: `Students` get a mark for attending a specific `Course` in an `University`. In some cases, `HappyStudents` will happily attend a `Course` with a certain level of happiness.

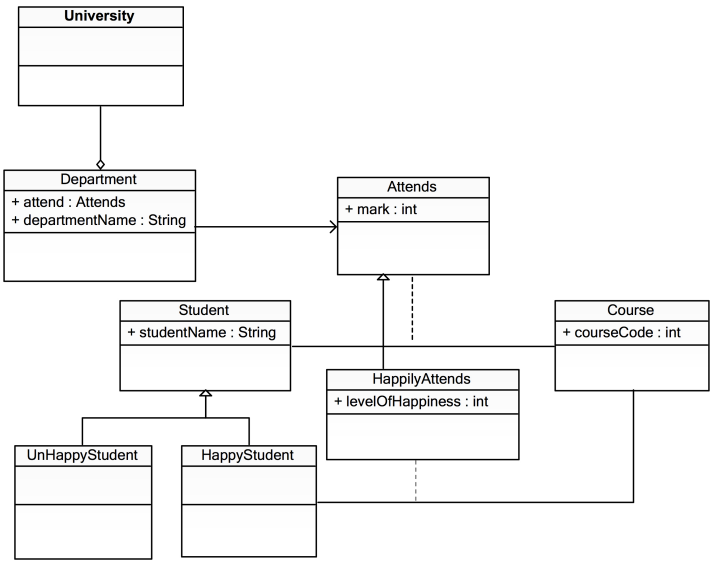


Figure 3.13: Typical University environment

As illustrated in the above UML diagram, the relationship `HappilyAttends` extends the `Attends` relationship. However, it provides more specialised participants: only `HappyStudent` can be part of the `HappyAttends` relationship. Similarly to class inheritance, the relationship `HappilyAttends` will inherit the implementation of its parent relationship; namely all the parent's fields and methods. In this case, a `HappilyAttends` relationship has a `levelOfHappiness` field but also a `mark` field for each of its tuples. Declaration and use of the `HappilyAttends` relationship is presented in Listing 3.21.

Listing 3.21: HappyAttends

```
1  public relationship Attends(Student s, Course c)
2  {
3    int mark;
4    String mood;
5  }
6
7  public relationship HappilyAttends(HappyStudent s, Course c) extends Attends
8  {
9    int levelOfHappiness;
10 }
11
12 HappyAttends a = newR HappilyAttends();
13 a->add(sophia,oop).withMark(8).withLevelOfHappiness(90);
14 a->add(raoul,oop).withMark(10).withLevelOfHapiness(100);
```

In addition, we introduce polymorphism of relationships by linking relationship inheritance with relationship sub-typing. In the above example, `HappyAttends` also becomes a subtype of `Attends`. The benefits of introducing relationship polymorphism are similar to class polymorphism: the developer can write code that deals with a family of relationship types. This is illustrated in Listing 3.22 where a `Department` holds attendance of `Students` regardless on whether they are happily attending the `Courses`.

Listing 3.22: Relationship Polymorphism

```
1  public relationship Attends(Student s, Course c)
2  {
3    int mark;
4    String mood;
5  }
6
7  public relationship HappilyAttends(HappyStudent s, Course c)
       extends Attends
8  {
9    int levelOfHappiness;
10 }
11
12 public class Department
13 {
14   Attends attends;
15   String departmentName;
16 }
17
18 Department eee = new Department();
19 eee.attends = newR Attends();
20
21 Department computing = new Department();
22 computing.attends = newR HappilyAttends();
```

Issues

The introduction of relationship inheritance and polymorphism raises several issues:

- Are relationship types covariant?
- How to deal with relationship's attributes specialisation?

- How to deal with multiplicity and constraints specialisation?

Relationship covariance

We illustrate the problem of relationship types covariance in Listing 3.23.

Listing 3.23: Assignment of extents with same family type problem

```
1  public relationship Attends(Student s, Course c)
2  {
3    int mark;
4    String mood;
5  }
6
7  public relationship HappilyAttends(HappyStudent s, Course c)
       extends Attends
8  {
9    int levelOfHappiness;
10 }
11
12 HappilyAttends happilyAttends = newR HappilyAttends();
13 // assignment of extent
14 Attends attends = happilyAttends;
15
16 UnHappyStudent gavin = new UnHappyStudent();
17 HappyStudent raoul = new HappyStudent();
18
19 happilyAttends->add(raoul,java)
20 attends->add(gavin,java);
21
22 // return gavin and raoul, but we expect only happy students
       happilyAttends->from(java);
```

The declaration of `HappilyAttends` is specialised to allow tuples of `HappyStudents` and `Courses`. However, by declaring an alias `attends` of type `Attends` and assigning it an alias of a `HappilyAttends` extent, the programmer can now access the `Attends` participants' interface and effectively corrupt the `happilyAttends` extent by adding tuples of type `Student` rather than `HappyStudent`.

We identify 4 ways to tackle this issue and suggest a simple solution based on an efficient internal structuring of tuples.

1. **Forbid assignment to extent aliases of different static type than right hand side**
   The simplest solution is to completely forbid assignment of extents to a different static typed receiver. However, this solution is not practical because we lose the benefits of relationship polymorphism. It basically means that HappilyAttends is not a subtype of Attends.

2. **Deep copy assignments of extents**
   Another solution is to deep copy assignment of extents. The receiver of the assignment would receive a deep copy of the assigned extent rather than a reference. This way, the receiver has a copy completely independent from the right hand side and therefore can not corrupt it. Although this solution is expensive due to the deep copy mechanism required, it is also not very practical because extents need to be copied when passed as parameters. For example, it prevents the storage of extents in collections. In fact, in Java, elements are added to collections by references.

3. **Filter input**
   A different solution is to use the dynamic type of the receiver in order to prevent tuples of the wrong type to be added in the extent. In the previous example, even though the alias `attends` has a static type `Attends`, it has a dynamic type `HappyAttends` and this information can be used to ensure that only tuples matching the `HappyAttends` declaration are added. However, this solution comes with two drawbacks. Firstly, it requires dynamic type checking on every addition of a tuple. Secondly, it is very restrictive as only `HappyAttends` tuples can be added even though the extent's alias static type accepts `Attends` tuples. This solution is similar to how Java deals with arrays.

4. **Filter output**
   An alternative solution is to perform filtering when fetching tuples from an extent rather than when adding them. One could use the static type of the receiver in order to fetch tuples matching the receiver's type. This solution is less restrictive because it allows the alias `attends` to access tuples matching the `Attends` participants' declaration and allows the alias `happyAttends` to access tuples of the `HappyAttends` participants' declaration. As a result, the result from the `from()`, `to()`, `foreach()` methods are filtered out to return results matching the static type of the receiver. However, this solution is relatively expensive as type checking is required every time tuples are fetched.

   4.1 **Structuring using static type of the extent's alias**
   We suggest a simple way of structuring the way tuples are added in an extent based on a russian nesting doll structure in order to provide an efficient filtering of tuples. The concept is illustrated in Figure 3.14. When an extent is created, internally a box is allocated for tuples matching the relationship participants' types. In addition, a surrounding box is created for each of the extent's parents relationship participants' types. This way, whenever tuples are added through an alias, they are added to the right box considering the static types of the participant objects passed as parameters. In the example described earlier, adding a tuple with participant types `HappyStudent` and `Course` through `happilyAttends` will add it inside the `HappilyAttends` box. Similarly, adding the same tuple through the `attends` alias will add it inside the `HappilyAttends` box. However, adding tuples through the `attends` alias with participant objects of static types `Student` and `Course` will add them inside the `Attends` box, which has also access to the `HappilyAttends` box. In case, the static types of the participant objects of a tuple don't match a specific box, the first matching parent type is used for placing the tuple. For example, in the previous example `gavin` has static type `UnHappyStudent` and will therefore be added inside the `Attends` box because `UnHappyStudent` is a subtype of `Student`.

   This structure allows to efficiently retrieve tuples based on the static type of the receiver. In fact, in the example described above, the `happilyAttends` alias has only access to tuples in the `HappilyAttends` box because its static type is `HappilyAttends`. However, `attends`

alias has access to the whole `Attends` box, which includes tuples in the `HappilyAttends` box. Consequently, no run time type checking is required to filter out fetched tuples.

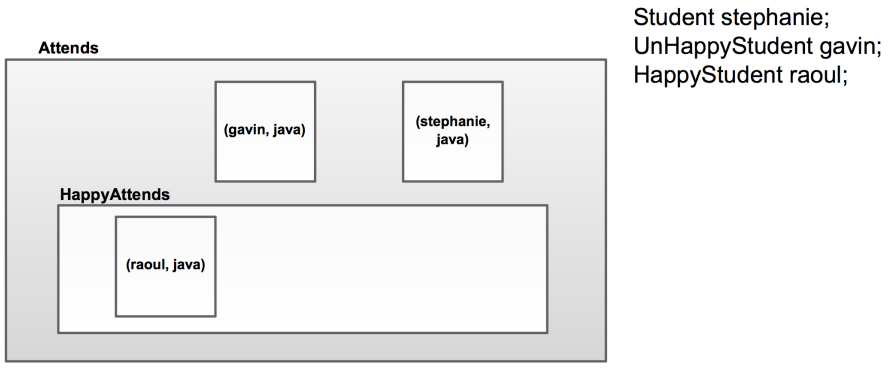


Figure 3.14: Placing tuples in a russian nesting doll structure

Relationship Attributes Specialisation

Take as example Listing 3.24. The Attends relationship defines an attribute tutor of type Tutor, whereas the relationship HappyAttends specialising Attends defines the attribute tutor as of type GreatTutor. How do we deal with relationships that define an inherited attribute with a different type?

Listing 3.24: Attribute Specialisation

```
1  public relationship Attends(Student, Course)
2  {
3    int mark;
4    Tutor tutor;
5  }
6
7  public relationship HappilyAttends(HappyStudent, Course) extends
       Attends
8  {
9    int levelOfHappiness;
10   GreatTutor tutor;
11 }
```

The first approach, is to take a purist view and simply forbid specialising an attribute name. The `HappilyAttends` relationship could simply define another attribute name `greatTutor` which takes a type `GreatTutor`. However, this may not be desired as the `HappilyAttends` relationship now has an unnecessary `tutor` attribute. In addition the code may lose consistency if a new attribute name is introduced representing the same information as another attribute. Another approach is to allow attribute specialisation and check based on the static type of the extent receiver whether the attribute `tutor` requires a general `Tutor` or a `GreatTutor`. However, this may not be accurate as for Attends extents only general tutors are allowed even though the participant object is a `HappyStudent`. A different approach is to allow attribute specialisation based on the static type of the participant objects. For instance, if a tuple

of `HappyStudent` and `Course` is added to an `Attends` extent, it will require a `GreatTutor` for the attribute `tutor`.

Multiplicity and Constraints Specialisation

Similarly, a specialised relationship will inherit constraints from its parent relationship. If required the specialised relationship should be able to set more restrictive multiplicities or constraints. For example if the `mark` attribute of the `Attends` relationship has a constraint range of 1..10, the `HappilyAttends` could refine the range to only good marks: 7..10. One possible way to ensure the constraint validation, is to apply the specialised constraints on the dynamic type of the extent receiver. This way the constraint always follow the most specialised form.

Chapter 4

ImperialRJ: The language

In this section we present *ImperialRJ* as a programming language *extending* Java. We introduce *ImperialRJ* by giving an example showcasing its features. We follow by providing the formal definition of *ImperialRJ*.

4.1 Example

To introduce *ImperialRJ* as a programming language, we program an example to show case in practice some of the available features in the language.

- Students attend Courses and get a mark for it.
- The University has two departments: computing and eee.
- What is the average mark at the University?
- Generate a Report with all Students who get a mark higher than the university average.
- There are 4 students: Raoul (computing), Sophia (computing), Michael (eee) and Stephanie (computing).
- There are 3 courses: oop, java and signals.
- Raoul attends oop with mark 10, Sophia attends java with mark 8 and oop with mark 6, Michael attends signals with mark 7, Stephanie attends oop with mark 8.

To program this problem, we declare a relationship `Attends` on `Student` and `Course` that has a relationship attribute `mark`. We create two extents for the two departments: `computing` and `eee`. We then add the tuples with their marks using the constructs available on relationship. To find the average mark on the university we use the union operator on the two extents as well as the average aggregate function. We use this result to filter all tuples with a relationship attribute greater than the university average. Finally, we define a method generateReport which takes an Attends extent to print an attendance report. The output of the program is shown below in Figure 4.1 and the code including comments is shown in Listing 4.2. The generated code by the *ImperialRJ* can be found in Appendix A.

Listing 4.1: University Example Output

```
Average is: 7.8
Raoul attends OOP with mark 10
Sophia attends Java with mark 8
Stephanie attends OOP with mark 8
```

Listing 4.2: University Example in *ImperialRJ*

```
import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;

public class UniversityExample
{
  public void launch() throws Exception
  {
    Student raoul = new Student();
    raoul.sName = "Raoul";

    Student michael = new Student();
    michael.sName = "Michael";

    Student sophia = new Student();
    sophia.sName = "Sophia";

    Student stephanie = new Student();
    stephanie.sName = "Stephanie";

    Course oop = new Course();
    oop.cName = "OOP";

    Course java = new Course();
    java.cName = "Java";

    Course signals = new Course();
    signals.cName = "Signals";


    // create two extents
    Attends computing = newR Attends();
    Attends eee = newR Attends();

    // add attendance with mark
    computing << (raoul,oop).withMark(10);
    computing << (sophia,java).withMark(8);
    computing << (sophia,oop).withMark(6);
    eee << (michael,signals).withMark(7);
    computing << (stephanie,oop).withMark(8);

    // university is union of computing and eee
    Attends university = unionof computing with eee;

    // calculate average mark at the university
    double averageMark = average mark from university;
    System.out.println("Average is: " + averageMark);
```

```
51
52      // filter the good students
53      university->filter(mark > averageMark);
54      // generate a report
55      generateReports(university);
56    }
57
58
59    private relationship Attends(Student, Course)
60    {
61     int mark;
62    }
63
64    public void generateReports(Attends a) throws Exception
65    {
66       foreach( (Student s, Course c) :  a)
67       {
68          int mark = a->get(s,c).getMark();
69          System.out.println(s +" attends "+ c +" with mark " + mark);
70       }
71    }
72
73
74    private class Student
75    {
76      String sName;
77
78      public String toString()
79      {
80        return sName;
81      }
82    }
83
84    private class Course
85    {
86      String cName;
87      public String toString()
88      {
89        return cName;
90      }
91    }
92  }
```

4.2 Formal Definition

In this section, we describe the language definition, type system and operational semantics of *ImperialRJ*.

4.2.1 Overview

Syntax

The main extension required to extend an Object-Oriented language with relationships is a new entity at the same level as a class. This is why we introduce a new syntactic category `EntityType`, which is composed of both classes and

relationships. A program is defined as a set of class declarations (`ClassDecl`) and relationship declarations (`RelDecl`). In addition, we added a new structure `RelMap` to map a relationship (`RelId`) to its definition which contains the two participant types and a map with the fields declared in the relationship.

Type System

Most of the rules introduced ensure that the arguments passed to a relationship method are in accordance with the participants types specified in the relationship declaration. Similarly, the rules setting relationship attributes ensure that the types of the arguments match the types of the relationship attributes specified in the relationship declaration.

Operational Semantics

We introduce a new `RelationshipObject` which represents an extent. It contains the `RelID` and a set of tuples contains participant objects as well as relationship attributes.

4.2.2 Syntax

```
CM ∈ ClassMap :  ClassId → ClassId × FieldMap × MethMap
RM ∈ RelMap :  RelId → EntityType × EntityType × FieldMap
FM ∈ FieldMap :  FieldId → Type
MM ∈ MethMap :  MethId → Type
VM ∈ VarMap :  VarId → Type

Program::= ClassDecl* | RelDecl*
RelDecl ::= class relationship R (p,p') {FieldDecl*}
ClassDecl ::= class C extends C' {FieldDecl* MethDecl*}

op ∈ Operators ::= < | > | == | ...

numeric ∈ NumericType ::= int | double | ...
p ∈ PrimitiveType ::= boolean | char | String | numeric
e ∈ EntityType ::= C | R
t ∈ Type ::= p | e | Set<t>

FieldDecl ::= t f
MethDecl ::= t m(t' x) { s }

v ∈ Value ::= PrimValue | {true, false, null}
pv ∈ PrimValue ::= charValue | intValue | StringValue | ...

e ∈ Expression ::= v | x | e.f | e→size() | ro | ae | se

ro ∈ RelationExpression ::= e^-1 | e^+ | e^= | e^° | e ∪ e' | e ∩ e'

ae ∈ AggregationExpression ::= average f from e |
                               sum f from e |
                               min f from e |
                               max f from e

se ∈ StatementExpression ::= new C() |
                             newR R() |
                             e→add(e',e'') |
                             e→add(e',e'')[FieldSet*] |
                             e→rem(e',e'') |
                             e→get(e',e'')[f] |
                             e→set(e',e'')[FieldSet*] |
                             e→from(e') |
                             e→to(e') |
                             e→filter(f op e') |

fieldset ::= f = e

s ∈ Statement ::= ϵ | se; se' | foreach((p x, p' x') :  e) { se }
se'
```

Figure 4.1: The syntax of *ImperialRJ*

4.2.3 Type System

Type System Rules

NEWR

$$\frac{P \vdash R}{\Gamma \vdash \mathsf{null} : R \qquad \Gamma \vdash \mathsf{newR}\ R() : R}$$

SIZE

$$\frac{\Gamma \vdash e : R}{\Gamma \vdash e \rightarrow \mathsf{size}() : \mathsf{int}}$$

ADD

$$\frac{\Gamma \vdash e : R \quad RM(R) = (t, t', \_) \quad \Gamma \vdash e' : t \quad \Gamma \vdash e'' : t'}{\Gamma \vdash e \rightarrow \mathsf{add}(e', e'') : \mathsf{void}}$$

ADDSET

$$\frac{\Gamma \vdash e : R \quad RM(R) = (t, t', FM) \quad \Gamma \vdash e' : t \quad \Gamma \vdash e'' : t' \quad \forall i \in \{1..n\}\ :\ FM(f_i) = t_i \wedge \Gamma \vdash e_i : t_i}{\Gamma \vdash e \rightarrow \mathsf{add}(e', e'')[f_1 = e_1, ..., f_n = e_n] : \mathsf{void}}$$

REM

$$\frac{\Gamma \vdash e : R \quad RM(R) = (t, t', \_) \quad \Gamma \vdash e' : t \quad \Gamma \vdash e'' : t'}{\Gamma \vdash e \rightarrow \mathsf{rem}(e', e'') : \mathsf{void}}$$

RELFIELDGET

$$\frac{\Gamma \vdash e : R \quad RM(R) = (t, t', FM) \quad \Gamma \vdash e' : t \quad \Gamma \vdash e'' : t' \quad FM(f) = t''}{\Gamma \vdash e \rightarrow get(e', e'')[f] : t''}$$

RELFIELDSET

$$\frac{\Gamma \vdash e : R \quad RM(R) = (t, t', F) \quad \Gamma \vdash e' : t \quad \Gamma \vdash e'' : t' \quad \forall i \in \{1..n\} \ : \ FM(f_i) = t_i \wedge \Gamma \vdash e_i : t_i}{\Gamma \vdash e \rightarrow set(e', e'')[f_1 = e_1, ..., f_n = e_n] : void}$$

FROM

$$\frac{\Gamma \vdash e : R \quad RM(R) = (t, t', \_) \quad \Gamma \vdash e' : t}{\Gamma \vdash e \rightarrow from(e') : Set < t' >}$$

TO

$$\frac{\Gamma \vdash e : R \quad RM(R) = (t, t', \_) \quad \Gamma \vdash e' : t'}{\Gamma \vdash e \rightarrow to(e') : set < t >}$$

FOREACH

$$\frac{x, x' \notin dom(\Gamma) \quad \Gamma \vdash e : R \quad RM(R) = (t, t', \_) \quad \Gamma[x \mapsto t, x' \mapsto t'] \vdash se \quad \Gamma \vdash se'}{\Gamma \vdash foreach((t\ x, t'\ x') : e)\{\ se\ \}\ se'}$$

INVERSE

$$\frac{\Gamma \vdash e : R \quad RM(R) = (t, t', \_) \quad RM(R^-) = (t', t, \_) \quad \Gamma \vdash R^-}{\Gamma \vdash e^{-1} : R^-}$$

FILTER

$$\frac{\Gamma \vdash e : R \quad \Gamma \vdash e' : t \quad FM(f) = t}{\Gamma \vdash e \rightarrow \text{filter}(f \text{ op } e')}$$

AGGREGATION

$$\frac{\Gamma \vdash e : R \quad \Gamma \vdash FM(f) : t \quad t \in \text{numeric}}{\Gamma \vdash \{\text{average}, \text{sum}, \text{max}, \text{min}\}\ f \text{ from } e}$$

Well-Formedness

WELL-FORMED RELATIONSHIP

$$\frac{\forall t, t', FM : (t, t', FM) \in RM(R) \Rightarrow P \vdash t\diamond \quad P \vdash t'\diamond \quad \forall f : \text{dom}(FM) \Rightarrow P \vdash FM(f)\diamond}{P \vdash R\diamond}$$

WELL-FORMED PROGRAMS

$$\frac{\forall C : P(C) \text{ is defined} \Rightarrow P \vdash C\diamond \quad \forall R : P(R) \text{ is defined} \Rightarrow P \vdash R\diamond}{\vdash P\diamond}$$

4.2.4 Operational Semantics

Meta Information

$\iota \in Addr \cup \{null\}$
$o \in EntityObject$
$r \in RelationshipObject$
$c \in ClassObject$

$Addr = \{\iota_i \mid i \in N\}$
$Val = Addr \cup \texttt{Set<Val>} \cup \{true, false, null\}$

$Heap : Addr \rightarrow EntityObject$
$Stack : Addr \times Val$

$RelationshipObject : \ \ RelID \times P(Addr \times Addr \times (FieldID \rightarrow Val))$
$ClassObject : ClassID \times (FieldID \rightarrow Val)$
$EntityObject : RelationshipObject \cup ClassObject$

Figure 4.2: Meta variables used for Operational Semantics

$Tuples(r) = r \downarrow_2$
$TupleFields(r, domain, image) = \{t \downarrow_3 \ \mid t \in Tuples(r), t \downarrow_1 = domain \ \wedge t \downarrow_2 = image \ \}$

$FD(r) = dom(RM(r) \downarrow_3)$
$FD(c) = dom(CM(c) \downarrow_2)$

Figure 4.3: Meta functions for Operational Semantics

Operational Semantics Rules

SIZE

$\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi'$
$\chi'(\iota) = \mathsf{ro}$
$\mathsf{v} = \#(\mathsf{ro} \downarrow 2)$

$\mathsf{e} \rightarrow \mathsf{size}(), \phi, \chi \rightsquigarrow \mathsf{v}, \chi'$

ADD

$$\frac{\begin{array}{l}\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi' \\ \chi'(\iota) = \mathsf{r} \\ \mathsf{e}', \phi, \chi' \rightsquigarrow \iota', \chi'' \\ \mathsf{e}'', \phi, \chi'' \rightsquigarrow \iota'', \chi''' \\ \mathsf{tuples} = \mathsf{Tuples}(\mathsf{r}) \setminus (\iota', \iota'', \_) \\ \mathsf{fields} = \{\mathsf{f} : \text{initial for } \mathsf{f} \mid \mathsf{f} \in \mathsf{FD}(\mathsf{r})\} \\ \chi'''' = \chi'''[\iota \mapsto (\mathsf{r}, \mathsf{tuples} \cup \{(\iota', \iota'', \mathsf{fields})\}]\end{array}}{\mathsf{e} \to \mathsf{add}(\mathsf{e}', \mathsf{e}''), \varnothing, \chi \rightsquigarrow \iota, \chi''''}$$

ADDSET

$$\frac{\begin{array}{l}\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi' \\ \chi'(\iota) = \mathsf{r} \\ \mathsf{e}', \phi, \chi' \rightsquigarrow \iota', \chi'' \\ \mathsf{e}'', \phi, \chi'' \rightsquigarrow \iota'', \chi''' \\ \mathsf{tuples} = \mathsf{Tuples}(\mathsf{r}) \setminus (\iota', \iota'', \_) \\ \mathsf{fieldsValues} = \{\mathsf{e_i}, \phi, \chi''' \rightsquigarrow \mathsf{v_i}, \chi'''' \mid \mathsf{e_i} \in \{\mathsf{e_1}, ..., \mathsf{e_n}\}\} \\ \mathsf{fieldsWithValue} = \{\mathsf{f_i} : \mathsf{v_i} \mid \mathsf{f_i} \in \{\mathsf{f_1}, ..., \mathsf{f_n}\}\} \\ \mathsf{fieldsWithDefaultValues} = \{\mathsf{f_j} : \text{initial for } \mathsf{f_j} \mid \mathsf{f_j} \in \mathsf{FD}(\mathsf{r}) \setminus \{\mathsf{f_1}, ..., \mathsf{f_n}\}\} \\ \chi'''' = \chi'''[\iota \mapsto (\mathsf{r}, \mathsf{tuples} \cup (\iota', \iota'', \mathsf{fieldsWithValue} \cup \mathsf{fieldsDefaultValues}))]\end{array}}{\mathsf{e} \to \mathsf{add}(\mathsf{e}', \mathsf{e}'')[\mathsf{f_1} = \mathsf{e_1}, ..., \mathsf{f_n} = \mathsf{e_n}], \phi, \chi \rightsquigarrow \iota, \chi''''}$$

REM

$$\frac{\begin{array}{l}\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi' \\ \chi'(\iota) = \mathsf{r} \\ \mathsf{e}', \phi, \chi' \rightsquigarrow \iota', \chi'' \\ \mathsf{e}'', \phi, \chi'' \rightsquigarrow \iota'', \chi''' \\ \mathsf{tuples} = \mathsf{Tuples}(\mathsf{r}) \setminus \{(\iota', \iota'', \_)\} \\ \chi'''' = \chi'''[\iota \mapsto (\mathsf{r}, \mathsf{tuples})]\end{array}}{\mathsf{e} \to \mathsf{rem}(\mathsf{e}', \mathsf{e}''), \phi, \chi \rightsquigarrow \iota, \chi''''}$$

FROM

$$\frac{\begin{array}{l}\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi' \\ \chi'(\iota) = \mathsf{r} \\ \mathsf{e}', \phi, \chi' \rightsquigarrow \iota', \chi'' \\ \mathsf{v} = \{\iota'' \mid (\iota', \iota'', \_) \in \mathsf{Tuples}(\mathsf{r})\}\end{array}}{\mathsf{e} \to \mathsf{from}(\mathsf{e}'), \phi, \chi \rightsquigarrow \mathsf{v}, \chi''}$$

TO

$$\frac{\begin{array}{l}\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi' \\ \chi'(\iota) = \mathsf{r} \\ \mathsf{e}', \phi, \chi' \rightsquigarrow \iota', \chi'' \\ \mathsf{v} = \{\iota'' \mid (\iota'', \iota', \_) \in \mathsf{Tuples(r)}\}\end{array}}{\mathsf{e} \rightarrow \mathsf{to}(\mathsf{e}'), \phi, \chi \rightsquigarrow \mathsf{v}, \chi''}$$

NEWR

$$\frac{\begin{array}{l}\iota \text{ is new in } \chi \\ \chi' = \chi[\iota \mapsto (\mathsf{R}, \emptyset)]\end{array}}{\mathsf{newR\ R}(), \phi, \chi \rightsquigarrow \iota, \chi'}$$

RELFIELDGET

$$\frac{\begin{array}{l}\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi' \\ \chi'(\iota) = \mathsf{r} \\ \mathsf{e}', \phi, \chi' \rightsquigarrow \iota', \chi'' \\ \mathsf{e}'', \phi, \chi'' \rightsquigarrow \iota'', \chi''' \\ (\iota', \iota'', \mathsf{fieldsForTuple}) \in \mathsf{Tuples(r)} \\ \mathsf{v} = \mathsf{fieldsForTuple(f)}\end{array}}{\mathsf{e} \rightarrow \mathsf{get}(\mathsf{e}', \mathsf{e}'')[\mathsf{f}], \phi, \chi \rightsquigarrow \mathsf{v}, \chi'''}$$

RELFIELDSET

$$\frac{\begin{array}{l}\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi' \\ \chi'(\iota) = \mathsf{r} \\ \mathsf{e}', \phi, \chi' \rightsquigarrow \iota', \chi'' \\ \mathsf{e}'', \phi, \chi'' \rightsquigarrow \iota'', \chi''' \\ \mathsf{tuples} = \mathsf{Tuples(r)} \\ (\iota', \iota'', \mathsf{fieldsMap}) \in \mathsf{tuples} \\ \mathsf{fieldsUnaffectedWithValue} = \{\mathsf{f} : \mathsf{fieldsMap(f)} \mid \mathsf{f} \in \mathsf{FD(r)} \setminus \{\mathsf{f}_1, ..., \mathsf{f}_n\}\} \\ \mathsf{e}_i, \phi, \chi_i \rightsquigarrow \mathsf{v}_i, \chi_{i+1} \text{ where } \chi_1 = \chi''' \text{ and } \mathsf{i} = 1..\mathsf{n} \\ \mathsf{fieldsAffectedWithNewValue} = \{\mathsf{f}_i : \mathsf{v}_i \mid \mathsf{f}_i \in \{\mathsf{f}_1, ..., \mathsf{f}_n\}\} \\ \chi'''' = \chi_{i+1}[\iota \mapsto (\mathsf{r}, \mathsf{tuples} \setminus (\iota', \iota'', \_) \cup (\iota', \iota'', \mathsf{fieldsUnaffectedWithValue} \cup \mathsf{fieldsAffectedWithNewValue}))]\end{array}}{\mathsf{e} \rightarrow \mathsf{set}(\mathsf{e}', \mathsf{e}'')[\mathsf{f}_1 = \mathsf{e}_1, ..., \mathsf{f}_n = \mathsf{e}_n], \phi, \chi \rightsquigarrow \mathsf{e}, \chi''''}$$

FOREACH

$$\frac{\begin{array}{l}\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi' \\ \chi'(\iota) = \mathsf{r} \\ \forall(\iota', \iota'', \_) \in \mathsf{Tuples(r)} : \mathsf{se}, \phi[\mathsf{x} \mapsto \iota', \mathsf{x}' \mapsto \iota''], \chi_{\mathsf{i}} \rightsquigarrow \mathsf{v_i}, \chi_{\mathsf{i+1}} \text{ where } \mathsf{i} \in \{1..\#\mathsf{Tuples(r)}\} \\ \mathsf{se}', \phi, \chi_{\mathsf{n+1}} \rightsquigarrow \mathsf{v}, \chi''\end{array}}{\mathsf{foreach}((\mathsf{t\ x}, \mathsf{t}'\ \mathsf{x}') : \mathsf{e})\{\ \mathsf{se}\ \}\ \mathsf{se}', \phi, \chi \rightsquigarrow \mathsf{v}, \chi''}$$

INVERSE

$$\frac{\begin{array}{l}\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi' \\ \chi'(\iota) = \mathsf{r} \\ \chi'(\iota') = \mathsf{r}^{-1}\end{array}}{\mathsf{e}^{-1}, \phi, \chi \rightsquigarrow \iota', \chi'}$$

FILTER

$$\frac{\begin{array}{l}\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi' \\ \chi'(\iota) = \mathsf{r} \\ \mathsf{e}', \phi, \chi' \rightsquigarrow \mathsf{v}, \chi'' \\ \mathsf{tuples} = \mathsf{Tuples(r)} \\ \mathsf{validTuples} = \{(\iota', \iota'', \mathsf{fieldsMap}) \mid (\iota', \iota'', \mathsf{fieldsMap}) \in \mathsf{tuples} \wedge \mathsf{fieldsMap(f)\ op\ v}\} \\ \chi''' = \chi'''[\iota \mapsto (\mathsf{r}, \mathsf{validTuples})]\end{array}}{\Gamma \vdash \mathsf{e} \rightarrow \mathsf{filter(f\ op\ e}')}$$

AGGREGATION

$$\frac{\begin{array}{l}\mathsf{e}, \phi, \chi \rightsquigarrow \iota, \chi' \\ \chi'(\iota) = \mathsf{r} \\ \mathsf{tuples} = \mathsf{Tuples(r)} \\ \mathsf{f} \in \mathsf{FD(r)} \\ \mathsf{values} = \{\mathsf{fieldsMap(f)} \mid (\iota', \iota'', \mathsf{fieldsMap}) \in \mathsf{tuples}\} \\ \mathsf{v} = \{\mathsf{average}, \mathsf{sum}, \mathsf{max}, \mathsf{min}\}(\mathsf{values})\end{array}}{\{\mathsf{average}, \mathsf{sum}, \mathsf{max}, \mathsf{min}\}\ \mathsf{f\ from\ e}, \phi, \chi \rightsquigarrow \mathsf{v}, \chi'}$$

Chapter 5

ImperialRJ Implementation

In this section we describe the implementation of *ImperialRJ* following the standard structure of a compiler. First, we describe the syntax choices and the grammar of *ImperialRJ* in *Polyglot*. Second, we explain the processing architecture by going over the AST generated from the grammar, then we briefly explain the various analysis supported; namely type checking and constraint checking. After that, we give an overview of the resulting Java code generation and of the *Java Relationship Library* that we implemented to work with relationships. Finally, we briefly give an overview of the testing platform used to validate the language implementation.

With respect to our design space presented in Chapter 3, the current features available in the *ImperialRJ* implementation are:

1. First-class extents
2. Tuples not as first-class citizens
3. Tuple states
4. Encapsulation by references
5. Querying
6. Aggregation functions

In addition, *ImperialRJ* provides new constructs to easily navigate relationships:

1. pattern matching `foreach` loop
2. `from()`/`to()` constructs to navigate the domain and image of the relationship

The table below summarises the lines of code and number of classes added in order to implement the *ImperialRJ* compiler.

Figure 5.1: *ImperialRJ* code summary

	Lines of code	Classes
Grammar extension	371	
ImperialRJ core	2422	63
Java Relationship Library	2388	71
Total	**5181**	**134**

5.1 Syntax choices

We made several decisions with regard to the syntax of *ImperialRJ* in order to make it intuitive and easy to work with. Firstly, extents are accessed with a different accessor operator: the arrow `->` instead of the dot accessor in order not to confuse them with class objects.

Secondly, the method names available on extents follow the Java naming convention. For example adding a tuple is simply done using the `add` method on an extent, getting and setting the value of relationship attribute is done using the standard getter/setter Java convention.

Thirdly, operations available on relationship like union and intersection as well as aggregation functions follow a syntactical form very close to English in order to be very readable. As an example the union of two extents is declared as: `unionof extent1 with extent2`;

Finally, we provide a new foreach construct which is similar to the extended `for` introduced in Java 5 but allows pattern matching on the participants of an extent for easy navigation.

5.2 Grammar

The grammar of *ImperialRJ* is written in the *ppg* format - Polyglot's parser. The grammar hooks in the standard Java grammar. For example, the new `foreach` construct extends a `statement` in order to be usable as a normal loop construct. This is shown in Listing 5.1

Listing 5.1: foreach grammar

```
1 extend statement ::= relationship_for_statement:a {: RESULT = a;
      :};
2
3 relationship_for_statement ::= FOREACH:n LPAREN LPAREN type:a
      identifier:b COMMA type:c identifier:d RPAREN COLON name:e
      RPAREN:f statement:g
4 {:
5   RESULT = parser.nf.RelFor(parser.pos(n,g),a,b,c,d,e.toExpr(),g);
6 :};
```

5.3 Abstract Syntax Tree Structure

We present a UML diagram in Figure 5.2 depicting the hierarchy of the relevant AST extension nodes created by the `NodeFactory`, which is called during the parsing phase based on the *ImperialRJ* grammar. We created all the classes and interfaces below except `NodeFactory`, `Node`, `Expr`, `Stmt` and `Term` which are part of *Polyglot*.

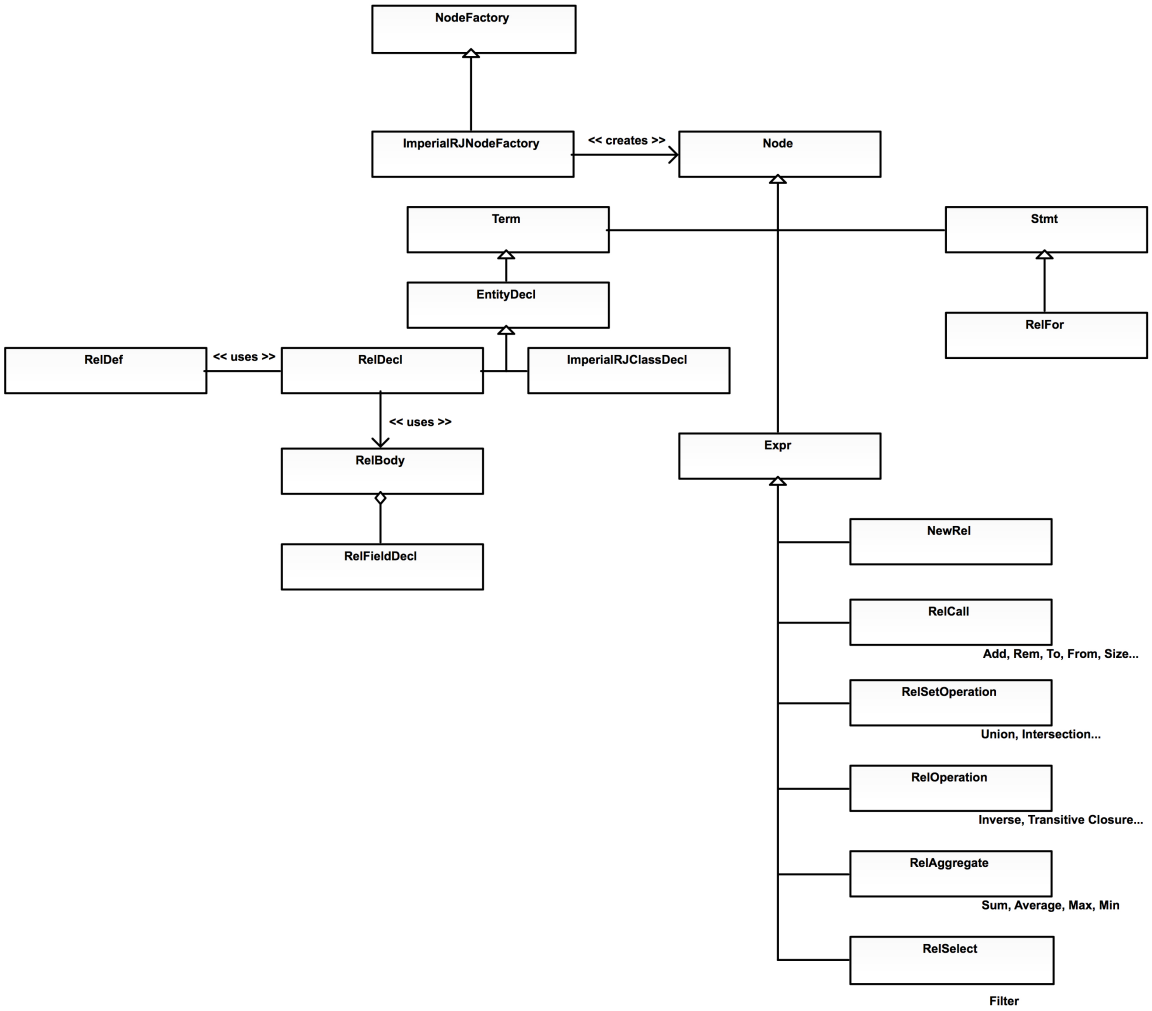


Figure 5.2: *ImperialRJ* AST Nodes structure

We introduce a new top level entity to classify object classes and relationship at the same level: `EntityDecl`. It is inherited by `RelDecl` which represents the declaration of a relationship and `ImperialRJClassDecl` which represents classes and interfaces declaration. In addition we introduce several interfaces representing specific actions on an extent:

- `NewRel`: represents the instantiation of an extent

- `RelCall`: represents a standard method call on an extent like `add`, `rem`, `to`, `from`...
- `RelSetOperation`: represents a set operation on an extent, e.g.. `union` and `intersection`
- `RelOperation`: represents the various mathematical operation on an extent including transitive closure, inverse, reflexive closure and symmetric closure
- `RelAggregate`: represents the various aggregate function calls on an extent including `sum`, `maximum`, `minimum` and `average`.
- `RelSelect`: represents a filtering querying with its three arguments: relationship attribute name, comparing operator and value to compare with.

All the interfaces mentioned above are implemented by concrete classes. For example, `RelCall` is implemented by `RellCallAdd_c` to represent an add method call on an extent, `RelAggregate` is implemented by `RelMinimum_c` to represent the minimum aggregation call.

5.4 Semantic Analysis

The *ImperialRJ* compiler supports semantic analysis: type checking and constraint checking. These are performed through the visitor hook methods provided by Polyglot for each phase of the compilation process. The constraints implemented are defined in the formal definition of *ImperialRJ* as typesystem and operational semantics rules.

5.4.1 Type Checking

All the type system rules presented in the formal definition of *ImperialRJ* are implemented. For example, the `ADD` rule specifies that the types of the tuple's participants must match the participants' types as specified in the relationship declaration. Listing 5.2 shows how this check is implemented within the `RellCallAdd_c` class. Basically, the domain and image type of the extent's definition are compared with the type of the arguments on the `add` call.

Listing 5.2: Add Type Checking: RellCallAdd_c.java

```
1  // Rule ADD
2
3  @Override
4    public Node typeCheck(ContextVisitor tc) throws SemanticException
         {
5
6      TypeNode domainType = reldef.domainType();
7      TypeNode imageType = reldef.imageType();
8
9      Type arg1Type = arg1.type();
10     Type arg2Type = arg2.type();
11
12     if(!arg1Type.typeEquals(domainType.type(), tc.context()))
13     {
```

```
14        throw new SemanticException("tuple first argument type ("+
              arg1Type.toString()+") doesn't match with Domain type ("+
              domainType.toString()+").");
15
16      }
17
18      if(!arg2Type.typeEquals(imageType.type(), tc.context()))
19      {
20        throw new SemanticException("tuple second argument type ("+
              arg2Type.toString()+") doesn't match with Image type ("+
              imageType.toString()+").");
21
22      }
23
24      return this;
25    }
```

As another example of type checking, the code in Listing 5.3 shows that the union of two extents is only possible if the extents are of the same type.

Listing 5.3: Union Type Checking: RelUnion_c.java

```
1   @Override
2   public Node typeCheck(ContextVisitor tc) throws SemanticException
        {
3
4     String leftRelStr = leftRelDef.asType().toString();
5     String rightRelStr = rightRelDef.asType().toString();
6     if(!rightRelStr.equals(leftRelStr)){
7       throw new SemanticException("Different relationship type
            arguments! left: "+leftRelStr+", right: "+rightRelStr);
8     }
9     return this;
10
11  }
```

5.4.2 Constraint Checking

The compiler also checks whether an attribute name in a filter query actually exists in the relationship declaration. Listing 5.4 shows how this check is implemented within the `RelSelect_c` class.

Listing 5.4: Constraint checking in filter query: RelSelect_c.java

```
1   @Override
2   public Node typeCheck(ContextVisitor tc) throws SemanticException
        {
3
4     if(!reldef.relDecl().relBody().existField(b.toString()))
5     {
6       String relName = reldef.asType().toString();
7       throw new SemanticException("The field " + b.toString() + "
            is not declared in " + relName);
8     }
9
10    return this;
11  }
```

A similar check is executed on aggregation queries: before execution, the existence of the relationship attribute which is queried is checked within the relationship declaration.

5.5 Code Generation

In this section, we present the code generation phase of the *ImperialRJ* compiler. An example of generated code can be found in Appendix A. In addition, we present the Java Relationship Library (*JRL*), a library we developed to single out all functionalities responsible for working with relationships in Java.

Figure 5.3 depicts the code generation phase of the *ImperialRJ* compiler. It shows that the *ImperialRJ* compiler is a source-to-source translator, in other words it outputs Java code, which is then compiled to Java bytecode using the standard *javac* compiler. The generated Java code uses the JRL to deal with relationships.

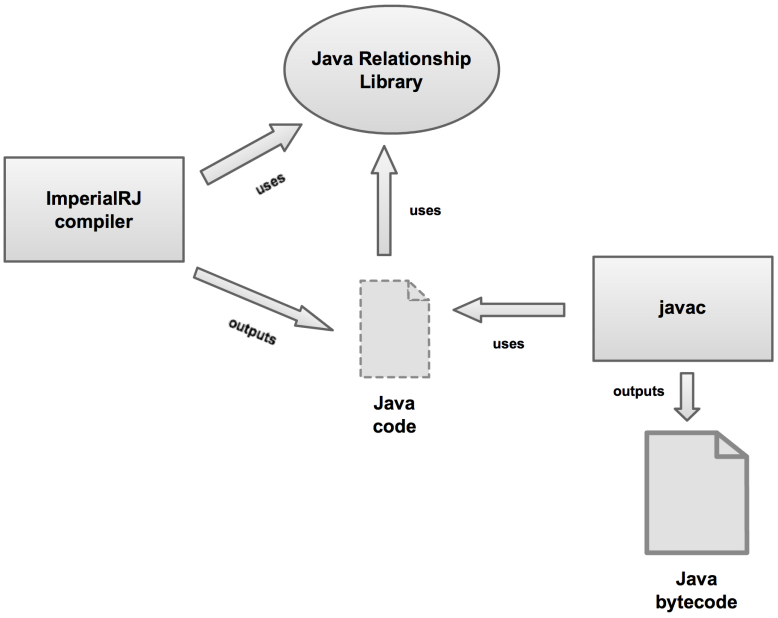


Figure 5.3: *ImperialRJ* Code Generation architecture

5.5.1 Java Relationship Library

We describe an UML diagram of the JRL in Figure 5.4. The JRL is composed of re-usable components that enable to:

1. Create and access tuples.
2. Create relationships.
3. Navigate a relationship as an iterable collection.
4. Navigate the domain and image of a relationship.
5. Add/Remove tuples in a relationship.
6. Apply visitors to a relationship to execute different operations.

The different visitors available include:

1. `MaxVisitor`: returns the maximum value of a relationship attribute within a relationship.
2. `MinVisistor`: returns the minimum value of a relationship attribute within a relationship.
3. `SumVisitor`: returns the sum of a specific relationship attribute for every tuple within a relationship.
4. `AverageVisitor`: returns the average of a specific relationship attribute for every tuple within a relationship.
5. `TransitiveClosureVisitor`: returns the transitive closure of the tuples within a relationship.
6. `SymmetricClosureVisitor`: returns the symmetric closure of the tuples within a relationship.
7. `ReflexiveClosureVisitor`: returns the reflexive closure of the tuples within a relationship.
8. `UnionVisitor`: returns the union of a relationship with one or more relationships.
9. `IntersectionVisitor`: returns the intersection of a relationship with one or more relationships.

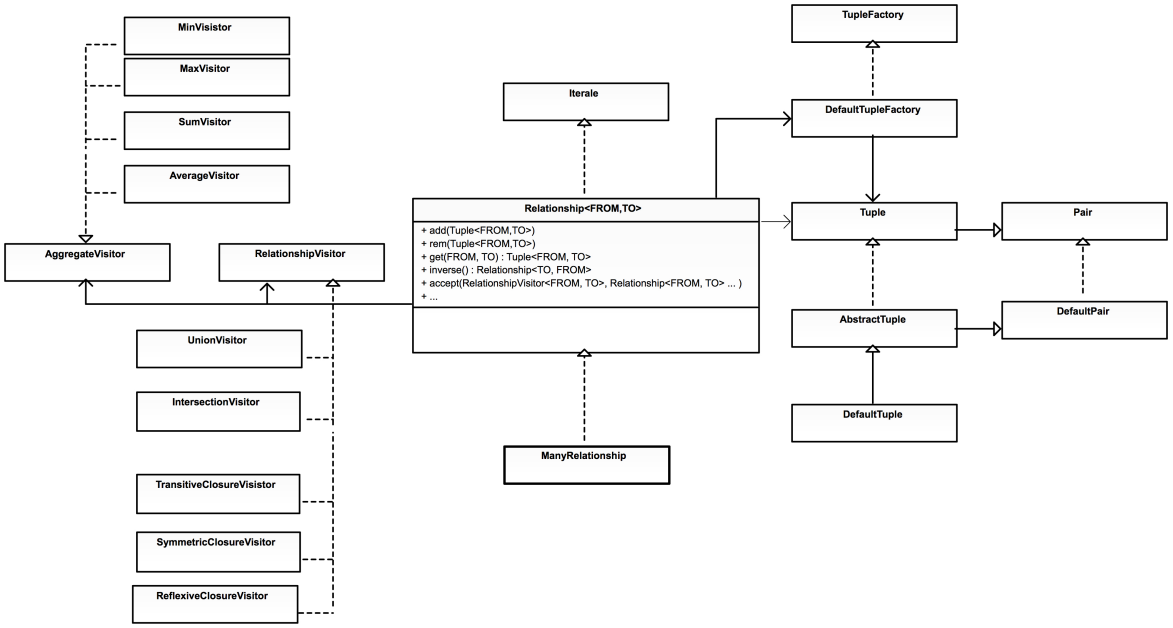


Figure 5.4: Java Relationship Library UML

As shown in the UML diagram, the `Relationship` interface provides a base for specific relationship implementations. At the moment only `ManyRelationship` is available. Further implementation can be added by implementing the `Relationship` interface to provide more specific multiplicities: `OneManyRelationship`, `ManyOneRelationship` etc.

5.5.2 Mapping ImperialRJ to Java

The *ImperialRJ* translation phase is executed within the `translate(CodeWriter w, Translator tr)` method of each AST node. The translate method by default calls the `prettyPrint(CodeWriter w, PrettyPrinter pp)` to output Java code to the generated Java class file. This is why, each AST node overrides the `prettyPrint` method to generate Java code which uses the JRL to interface with relationships. We describe three translation examples.

Firstly, when a relationship is declared in *ImperialRJ*, four classes are generated as shown in Figure 5.5:

1. the relationship class
2. the inverse class of the relationship
3. the relationship tuple class
4. the inverse relationship tuple class

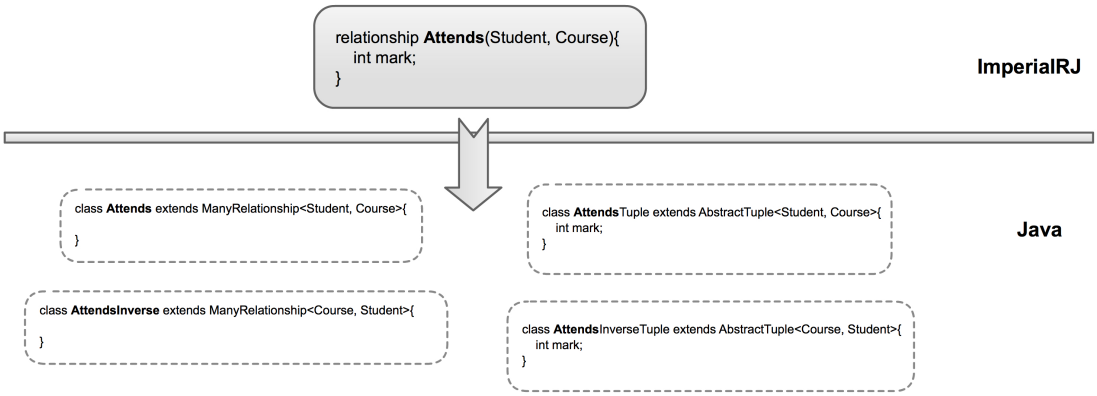


Figure 5.5: Translation of Relationship declaration to Java

The classes are accessed with the methods provided by the JRL. For example, adding a tuple to an extent uses the built-in `add` method of the relationship class as shown in Figure 5.6. Listing 5.5 shows the translation method within the `RelAdd` AST node.

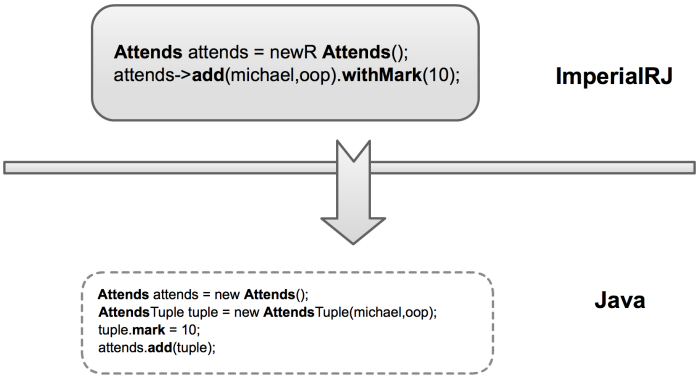


Figure 5.6: Adding a Tuple translation to Java

Listing 5.5: Adding a Tuple translation to Java

```
1  // in RelCallAdd_c.java
2
3  @Override
4  public void prettyPrint(CodeWriter w, PrettyPrinter pp) {
5
6    String fresh = Name.makeFresh().toString();
7
8    String domainType = this.reldef.domainType().nameString();
9    String imageType = this.reldef.imageType().nameString();
10
11   String tupleTypeName = reldef.name().toString() + "Tuple";
12
13   w.write(tupleTypeName + " " + fresh + " = new "+tupleTypeName
         +"("+this.arg1+","+this.arg2+");\n");
14   w.write(this.name + ".add("+fresh+")");
15 }
```

As another example, navigating an extent with the *ImperialRJ* foreach construct is translated to use the iterable property of the relationship class as specified by the JRL. This is described in Figure 5.7. Listing 5.6 shows the translation method within the `RelFor` AST node.

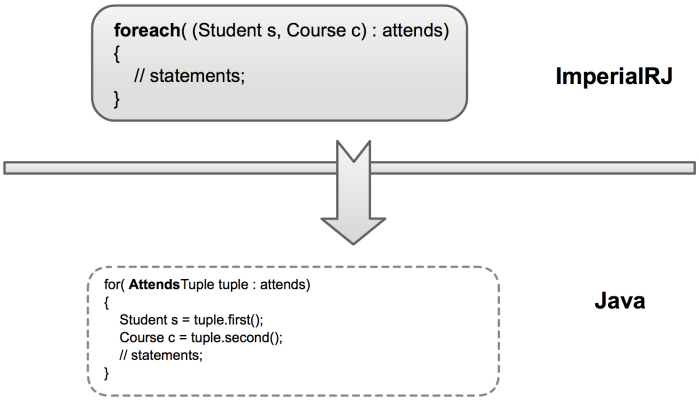


Figure 5.7: Foreach construct translation to Java

Listing 5.6: Foreach construct translation to Java

```
1  // in RelFor_c.java
2
3  @Override
4  public void prettyPrint(CodeWriter w, PrettyPrinter pp) {
5
6    String domainT = domainType.nameString();
7    String imageT = imageType.nameString();
8
9    String freshTupleName = Name.makeFresh().toString();
10
11   w.write("for(Tuple<"+domainT+", "+imageT+">" + freshTupleName +
         " : ");
12   relName.prettyPrint(w, pp);
13   w.write(")");
14   w.write("{");
15
16   w.write(domainT + " " + this.domainVar.name().toString() + " =
         " + freshTupleName+".first();\n");
17   w.write(imageT + " " + this.imageVar.name().toString() + " = "
         + freshTupleName.toString()+".second();\n");
18
19   body.prettyPrint(w, pp);
20
21   w.write("}");
22 }
```

5.6 Testing

In this section we briefly describe the testing infrastructure we developed in order to validate the compiler of *ImperialRJ*.

5.6.1 Java Relationship Library

The Java Relationship Library is used internally by the *ImperialRJ* compiler to represent relationships and provide all the methods and operations available

on them. Following standard software engineering practices, we unit tested the different functionalities provided by the library. Currently there are **60 unit tests**. The main functionalities tested include:

- Relationships and Tuples creation
- Accessing Relationships and Tuples states
- Navigation of Relationships
- Relationship Operations: inverse, transitive closure, reflexive closure, union, intersection...

Figure 5.8 depicts a sample of unit tests run in eclipse.

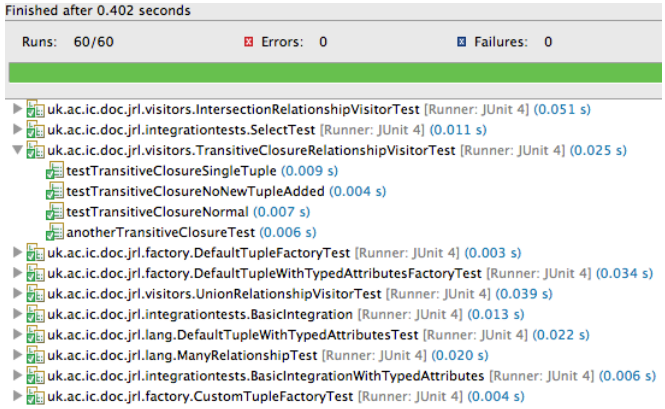


Figure 5.8: Unit Testing of the Java Relationship Library

5.6.2 Validation of ImperialRJ

We wrote **22 test cases** written in *ImperialRJ* that test the language features of *ImperialRJ*. The tested features include:

- extents as first-class citizen
- add/removal of tuples
- navigation constructs
- filtering of relationship
- relationship operations: inverse, transitive closure, union, intersection...
- aggregation functions

We built an automated testing tool which compiles each test case and run them individually. The tool captures the output of the compiled test case and compares it with the expected output for that test. Once all the tests have been run our tool produces a summary of passing and failing tests. All failing tests return the expected output and actual output to the console for debugging

purposes. Figure 5.9 shows an example output produced by our tool with one failing test: *Testing reflexive closure*.

```
17) Running RelationshipOperations2.irj : Testing intersection
OK!

18) Running RelationshipOperations3.irj : Testing transitive closure
OK!

19) Running RelationshipOperations4.irj : Testing reflexive closure
FAIL!
Expected output:25
Actual output:35

20) Running RelationshipOperations5.irj : Testing symmetric closure
OK!

21) Running RelationshipOperations6.irj : Testing Inverse same domain/image
OK!

22) Running RelationshipOperations7.irj : Testing Inverse different domain/image
OK!


Summary:
SUCCESS: 21/22
FAIL: 1/22
```

Figure 5.9: Output Example for *ImperialRJ* Automated Testing

The list of all test cases currently available with the *ImperialRJ* compiler can be found in Appendix B for reference.

Chapter 6

Evaluation of First-Class Relationships with ImperialRJ

In this section we evaluate the benefits of first-class relationships in *ImperialRJ* over implicit relationships built with language primitives. We perform our evaluation by showing implicit relationship implementation code from three popular Java applications: *JFlex* - a lexical analyser generator [39], *JFreeChart* - a Java chart library [48] and *PMD* - a scanner for problems in Java source code [49], and porting them using first-class relationships in *ImperialRJ*. We show the problems of implicit relationships implementation and demonstrate how first-class relationships in *ImperialRJ* can improve code readability and ease development in an agile context.

6.1 Issues with Implicit Relationships

From our evaluation we identified six issues with the implementation of implicit relationships:

1. **Boiler Plate Code**: Adding and removing data from a relationship requires the programmer to implement the same methods over again.
2. **Navigation**: Internal implementations of relationships with collections make it hard to navigate the relationship.
3. **Querying**: The programmer needs to implement himself the algorithm to query the internal structure representing a relationship, which may not be efficient.
4. **Encapsulation**: In some cases, the internal implementation of a relationship can poorly encapsulate its participants and lead to security issues.
5. **Consistency**: The implicit relationship implementation needs to ensure the relationship stays consistent, putting additional overhead on the programmer.

6. **Rigidity**: Implicit relationships are rigid to changes and make it difficult to be agile to new requirements without painful implementation changes.

We discuss each issue with code examples and refactor them to *ImperialRJ* to demonstrate the benefits of first-class relationships.

6.2 Boiler Plate Code

The following example shown in Listing 6.1 is taken from *JFreeChart*. It shows that `Series` are associated with `Labels`.

Listing 6.1: Adding Series Label: MultipleXYSeriesLabelGenerator.java

```
/**
     * Adds an extra label for the specified series.
     *
     * @param series  the series index.
     * @param label  the label.
     */
    public void addSeriesLabel(int series, String label) {
        Integer key = new Integer(series);
        List labelList = (List) this.seriesLabelLists.get(key);
        if (labelList == null) {
            labelList = new java.util.ArrayList();
            this.seriesLabelLists.put(key, labelList);
        }
        labelList.add(label);
    }
```

This relationship is internally represented as a `Map` of a series to a list of `Labels`. Adding new `Label` to a `Series` requires creating a special `addSeriesLabel` method, which deals with the internal representation of the relationship. This method implementation is also error-prone: if the list of `Labels` hasn't been instantiated it needs to be created first before adding `Labels` to it.

Similarly, more boiler plate code is required for removing tuples from that relationship. Listing 6.2 shows the method implementation to remove all `Labels` associated to a `Series`.

Listing 6.2: Removing Labels from a Series: MultipleXYSeriesLabelGenerator.java

```
    /**
     * Clears the extra labels for the specified series.
     *
     * @param series  the series index.
     */
    public void clearSeriesLabels(int series) {
        Integer key = new Integer(series);
        this.seriesLabelLists.put(key, null);
    }
```

In ImperialRJ

In *ImperialRJ*, the boiler plate code to manipulate a relationship is abstracted away from the programmer. The example above can be modelled as a relationship between a `Series` and `Labels` as shown in Listing 6.3. Adding and

removing tuples of series and `Label` can be done using the built-in constructs, which enables the programmer to focus on the logic of his application rather than the error-prone implementation details.

Listing 6.3: SeriesToLabels in *ImperialRJ*

```
relationship SeriesToLabels(Integer, String)
{

}

SeriesToLabels seriesLabels = newR SeriesToLabels();
```

6.3 Navigation

The following example shown in Listing 6.4 is taken from *PMD*. It shows an association between a `RuleSet` and a list of `Rules` participating in the `RuleChain`.

Listing 6.4: Navigating RuleSet and Rules: AbstractRuleChainVisitor.java

```
public abstract class AbstractRuleChainVisitor implements
    RuleChainVisitor {
    /**
     * These are all the rules participating in the RuleChain,
         grouped by RuleSet.
     */
    protected Map<RuleSet, List<Rule>> ruleSetRules = new
        LinkedHashMap<RuleSet, List<Rule>>();

 protected void initialise() {

 // Determine all node types that need visiting
        Set<String> visitedNodes = new HashSet<String>();
        for ( Iterator<Map.Entry<RuleSet, List<Rule>>> entryIterator =
            ruleSetRules.entrySet().iterator(); entryIterator.
            hasNext();){
            Map.Entry<RuleSet, List<Rule>> entry = entryIterator.next();
            for (Iterator<Rule> ruleIterator = entry.getValue().
                iterator(); ruleIterator.hasNext();) {
                Rule rule = ruleIterator.next();
                if (rule.usesRuleChain()) {
                    visitedNodes.addAll(rule.getRuleChainVisits());
                }
                else {
                    // Drop rules which do not participate in the
                        rule chain.
                    ruleIterator.remove();
                }
            }
}
```

This example shows that the programmer requires two nested loops in order to navigate the `Rules` associated to a `RuleSet`. The code is also not very readable due to the generics in the `Map`, which have to be redeclared to navigate the entries with a `Map.Entry<RuleSet, List<Rule>>`.

In ImperialRJ

ImperialRJ comes with a special pattern matching loop construct, which facilitates navigation on a relationship. The benefits of this construct is that the code becomes more readable. The example above can be refactored by declaring a relationship between `RuleSet` and `Rule` as shown in Listing 6.5.

Listing 6.5: Navigating RuleSetToRule in *ImperialRJ*

```
relationship RuleSetToRule(RuleSet ruleSet, Rule rule)
{

}

RuleSetToRule rulesExtent = newR RuleSetToRule();
// add stuff to it

// filtering algorithm
Set<String> visitedNodes = new HashSet<String>();
foreach( (RuleSet ruleSet, Rule rule) :  rulesExtent)
{
  if (rule.usesRuleChain())
  {
        visitedNodes.addAll(rule.getRuleChainVisits());
    }
    else
    {
        // Drop rules which do not participate in the rule chain.
        rulesExtent->remove(ruleSet,rule);
    }
}
```

6.4 Querying

JFlex

The following example shown in Listing 6.6 is taken from *JFlex*. It shows a filtering implementation to collect all unused macros.

Listing 6.6: JFlex Querying Unused Macros: Macros.java

```

  /** Maps names of macros to their definition */
  private Hashtable macros;

  /** Maps names of macros to their "used" flag */
  private Hashtable used;

  /**
   * Returns all unused macros.
   *
   * @return the enumeration of macro names that have not been used
       .
   */
  public Enumeration unused() {
    Vector unUsed = new Vector();
  Enumeration names = used.keys();
    while ( names.hasMoreElements() ) {
      String name = (String) names.nextElement();
```

```
18        Boolean isUsed = (Boolean) used.get( name );
19        if ( !isUsed.booleanValue() ) unUsed.addElement(name);
20      }
21      return unUsed.elements();
22    }
```

The introduction of a `used` attribute in a separate `HashMap` forces the programmer to implement manually a specific method to query the `Map` for unused macros.

PMD

The example in Listing 6.4 from *PMD* shows the difficulty to navigate the internal structure of the relationship. The navigation is required in order to filter all the rules which validate the method `usesRuleChain()`. If the programmer needs to filter rules matching a different property, most likely he will have to copy the navigation code and tweak the condition to filter the new property. This methodology can prove very problematic if changes to the internal structure of the relationship are needed. The programmer has now to modify several parts of his code and can introduce new bugs.

In ImperialRJ

The *JFlex* example querying over a relationship attribute is a perfect example showcasing the benefits of querying constructs in *ImperialRJ*. A macro can be defined as a relationship between its `Name` and its `Definition` with a relationship attribute used. The only thing required to the programmer is to filter all macros with `used==false` as shown in Listing 6.7.

Listing 6.7: JFlex Macros in *ImperialRJ*

```
1  relationship Macros(String, RegExp)
2  {
3    boolean used;
4  }
5
6  Macros macros = newR Macros();
7  macros->add("macro1",definition1).withUsed(true);
8  macros->add("macro2", definition2).withUsed(false);
9
10 // only contains ("macro2", definition2)
11 macros->filter(used == false);
```

6.5 Encapsulation

The example in Listing 6.2 from *JFreeChart* shows that the relationship is implemented as a `Map` of `Series` to a list of `Labels`. This internal representation can be problematic when the programmer accesses a reference to a List within the `Map`. This reference lives outside of the relationship and can accidentally be changed with another list or set to null, which can potentially delete several tuples at the same time.

In ImperialRJ

In *ImperialRJ* each tuple are encapsulated individually. The internal implementation of the relationship is encapsulated through safe constructs so the programmer can't accidentally abuse the implementation of the relationship by accident.

6.6 Consistency

The following example shown in Listing 6.8 is taken from *JFlex*. It describes the definition of a macro. A macro has a `Name`, a `Definition` and also a state whether it has been `used` or not.

Listing 6.8: JFlex Macro Consistency: Macros.java

```
1  /**
2     * Stores a new macro and its definition.
3     *
4     * @param name          the name of the new macro
5     * @param definition    the definition of the new macro
6     *
7     * @return <code>true</code>, iff the macro name has not been
8     *          stored before.
9     */
10   public boolean insert(String name, RegExp definition) {
11
12     if (Options.DEBUG)
13       Out.debug("inserting macro "+name+" with definition :"+Out.NL
             +definition);
14
15     used.put(name, Boolean.FALSE);
16     return macros.put(name,definition) == null;
17   }
18 }
```

This example shows a relationship between the `Name` of the macro and its `Definition`. Moreover, this relationship has an attribute, which tells whether the macro has been `used`. The internal implementation of this relationship is made out of two `Hashtable`, which respectively maps the `Name` of the macro to its `Definition` and the `Name` of the macro to its attribute `used`. The `insert` method shows that in order to add a new macro, the two `Maps` are forced to be kept consistent with respect to the `Name` of the macro. This is problematic for two reasons:

1. The programmer is responsible for the internal representation and consistency of the relationships.

2. The relationship becomes rigid to changes as we explain in the next section.

In ImperialRJ

As explained previously, this example can be implemented by defining a macro as a relationship between its `Name` and its `Definition` as shown in Listing 6.7.

6.7 Rigidity

The example in Listing 6.8 shows that the relationship attribute `used` is implemented by keeping an extra `HashMap` mapping the `Names` of macros to a `Boolean` indicating whether the macro has been `used` or not. This internal representation of the relationship makes it very rigid to changes.

Firstly, take the case when a second attribute indicating the importance of the macro is required. The programmer would need to add an extra `HashMap` to hold this new property and keep the additional `HashMap` consistent with the other `Maps`. In addition, he would need to change the `insert` method for adding new macros, as well as other methods to encapsulate the access to the relationship attributes within the `HashMap`.

Secondly, what if a macro can have several definitions? This new requirement would require a complete overhaul of the implementation. The programmer would need to change the internal data structure to store several definitions as well as modify all the methods dealing with the previous data structure, and potentially introducing bugs along.

In ImperialRJ

ImperialRJ provides a relationship abstraction designed to have any number of attributes and dynamic methods are generated to access them. This flexibility makes relationships agile for new requirements. The problem described above can be tackled by simply adding a new relationship attribute in the `Macros` relationship declaration as shown in Listing 6.9. In addition, the Listing below shows that the programmer doesn't need to worry about allowing several definitions for each macro as the relationship is by default many to many.

Listing 6.9: JFlex Macros in *ImperialRJ*

```
1  @AllowDuplicates
2  relationship Macros(String, RegExp)
3  {
4     boolean used;
5     int importance;
6  }
7
8  Macros macros = newR Macros();
9  macros->add("macro1",definition1).withUsed(true).withImportance(10)
       ;
10 macros->add("macro1", definition2).withUsed(false).withImportance
       (9);
```

Chapter 7

Conclusion

In this section we conclude our work; summarising the major achievements, outlining further work and possible optimisations. Finally, we reflect on the project and the future of first-class relationships in object-oriented languages.

7.1 Achievements

In this project we took up the challenge to investigate *all* the major topics involving first-class relationships. In Chapter 2, we analysed earlier working linking object-oriented techniques with relationships. In Chapter 3, we explored the design space available for implementing object-oriented languages supporting first-class relationships. This investigation led us to come up with a few novel ideas: multiple sets of relationships, mechanisms to deal with aliasing of tuples, covariant overriding of the participants of a relationship and relationship persistence. In Chapter 4, we defined a formal definition of *ImperialRJ* to provide a solid reference for further work. In Chapter 5, we described the implementation of *ImperialRJ*, *the first* language to *extend* Java with relationship constructs. Finally, in Chapter 6, we explored the issues with implicit relationships and validated the benefits of first-class relationships using *ImperialRJ*. Additionally, we wrote a tutorial on Polyglot, which is now official, for researchers and students interested in building language extensions [17].

7.2 Further Work

This project opens door for interesting further work.

ImperialRJ Compiler

Firstly, even though first-class relationships bring benefits from a software engineering point of view, the current code generation is not optimised. An interesting route would be to statistically infer properties about relationships and optimise their internal data structures accordingly. This idea can actually be extended to the use of collections in Java. Secondly, interesting features to *ImperialRJ* still need to be implemented: *multiplicities*, *roles* and *annotation of duplicate tuples*.

Design Space

We briefly looked at the opportunities to link relationships in the database world with the software world. However, current ORM framework deal with implicit relationships in mind since first-class relationships are not yet part of mainstream object-oriented languages. It would be interesting to research whether first-class relationships can provide a unified model for relationship persistence as well. In addition, we gave an overview of relationship inheritance but the need for it isn't yet clear. Further work is required to explore possible use cases and whether specialised mechanisms are needed to deal with relationship inheritance.

Empiral Study

We would also like to provide a comprehensive study on the use of implicit relationships in order to further validate the use of first-class relationships. In fact, our evaluation was limited to three popular Java applications. It would be interesting to compare implicit relationships in more applications and in different object-oriented languages in order to guide the development of better first-class relationship mechanisms.

Automated Refactoring

Finally, it would be interesting to develop a tool under the form of an *Eclipse* plugin that automatically refactors software to use first-class relationships in *ImperialRJ*. The benefits of such a tool will provide an automated way of improving the abstraction of software as well as reducing the systems coupling. In addition, with clever dataflow analysis it can also potentially speed up the application.

7.3 Reflection

We hope this report has laid strong foundations to fuel further interests in first-class relationships. Our preliminary results from the evaluation shows that relationships are omnipresent in object-oriented systems but are consistently implemented implicitly and dangerously by the developer. This is why, we believe support for relationship primitives is useful.

In our opinion, there's still work left to push first-class relationships into mainstream languages. Firstly, more research is required regarding the use of implicit relationships; our study only analysed three popular Java applications. Secondly, as applications are more and more enterprise related, it is important to fully understand the link between relationships at the software and database level. Thirdly, more work on possible compiler optimisations is necessary: first-class relationships are an opportunity to leverage new constructs and statically infer optimisation properties. Finally, we need more integrated tooling to help developers easily program using first-class relationships.

We hope to continue our work in the future and perhaps one day see first-class relationships as an integral part of the object-oriented paradigm.

Appendix A

University Example Java Output

Listing A.1: Java output of University Example

```
1  import uk.ac.ic.doc.jrl.lang.*;
2  import uk.ac.ic.doc.jrl.interfaces.*;
3  import uk.ac.ic.doc.jrl.factory.*;
4  import uk.ac.ic.doc.jrl.exceptions.*;
5  import uk.ac.ic.doc.jrl.visitors.*;
6  import java.util.*;
7
8  public class UniversityExample
9  {
10
11   public void launch() throws Exception {
12     Student raoul = this.new Student();
13     raoul.sName = "Raoul";
14     Student michael = this.new Student();
15     michael.sName = "Michael";
16     Student sophia = this.new Student();
17     sophia.sName = "Sophia";
18     Student stephanie = this.new Student();
19     stephanie.sName = "Stephanie";
20     Course oop = this.new Course();
21     oop.cName = "OOP";
22     Course java = this.new Course();
23     java.cName = "Java";
24     Course signals = this.new Course();
25     signals.cName = "Signals";
26
27     Attends computing = new Attends();
28     Attends eee = new Attends();
29
30     AttendsTuple id0 = new AttendsTuple(raoul,oop);
31     id0.mark = 10;
32     computing.add(id0);
33     AttendsTuple id1 = new AttendsTuple(sophia,java);
34     id1.mark = 8;
35     computing.add(id1);
36     AttendsTuple id2 = new AttendsTuple(sophia,oop);
37     id2.mark = 6;
38     computing.add(id2);
39     AttendsTuple id3 = new AttendsTuple(michael,signals);
```

```
40      id3.mark = 7;
41      eee.add(id3);
42      AttendsTuple id4 = new AttendsTuple(stephanie,oop);
43      id4.mark = 8;
44      computing.add(id4);
45
46      Attends university =
47        (Attends) (new Attends().copyFrom(computing.accept(new
              UnionRelationshipVisitor<UniversityExample.Student,
              UniversityExample.Course>(),eee)));
48
49      double averageMark =
50        (Double) university.acceptAggregate(new AverageVisitor<
              Student, Course>(), "mark",Integer.TYPE);
51
52      System.out.println("Average is: " + averageMark);
53
54      // startOf filter
55      Attends id5 = new Attends();
56      id5.copyFrom(university);
57      for(Tuple<Student,Course> t : id5){
58        int mark = ((AttendsTuple) t).mark;
59        if(!(mark > averageMark)){
60          university.rem(t);
61          }
62      }
63      // endOf filter
64
65
66      this.generateReports(university);
67    }
68
69    private class Attends extends ManyRelationship<Student,Course>{}
70
71    private class AttendsInverse extends ManyRelationship<Course,
          Student>{}
72
73    public class AttendsTuple extends AbstractTuple<Student,Course>{
74
75      public AttendsTuple(){super();}
76
77      public AttendsTuple(Student a,Course b){
78        super(a,b);
79      }
80
81      /*1*/
82      public int mark;
83    }
84
85    public class AttendsInverseTuple extends AbstractTuple<Course,
          Student>{
86
87      public AttendsInverseTuple(){super();}
88
89      public AttendsInverseTuple(Course a,Student b){
90        super(a,b);
91      }
92
93      /*1*/
94      public int mark;
95    }
96
```

```
97
98
99    public void generateReports(Attends a) throws Exception {
100     for(Tuple<Student, Course>id6 : a){
101     Student s = id6.first();
102     Course c = id6.second();
103     {
104       int mark = ((AttendsTuple) a.get(s,c)).mark;
105       System.out.println(
106           s +
107           " attends " +
108           c +
109           " with mark " +
110           mark);
111     }}
112   }
113
114   private class Student
115   {
116     String
117     sName;
118
119     public String
120     toString() {
121       return sName;
122     }
123
124     public Student() {
125       super();
126     }
127   }
128
129   private class Course
130   {
131     String
132     cName;
133
134     public String
135     toString() {
136       return cName;
137     }
138
139     public Course() {
140       super();
141     }
142   }
143
144   public UniversityExample() {
145     super();
146   }
147
148 }
```

Appendix B

Test Cases

The first commented line describe the purpose of the test. The second line states the expected output after running the test.

Adding One Tuple

Listing B.1: Adding One Tuple

```
1  // Adding one tuple
2  // 0\n1\n
3
4  import uk.ac.ic.doc.jrl.lang.*;
5  import uk.ac.ic.doc.jrl.interfaces.*;
6  import uk.ac.ic.doc.jrl.factory.*;
7  import uk.ac.ic.doc.jrl.exceptions.*;
8  import uk.ac.ic.doc.jrl.visitors.*;
9  import java.util.*;
10
11 public class BasicExtent1
12 {
13   public void launch() throws Exception
14   {
15     User raoul = new User();
16     raoul.userName = "Raoul";
17
18     User sophia = new User();
19     sophia.userName = "Sophia";
20
21     Follow f = newR Follow();
22     System.out.println(f->count());
23
24     f << (raoul,sophia);
25     System.out.println(f->count());
26   }
27
28   private relationship Follow(User, User)
29   {
30   }
31
32   private class User{
33     public String userName;
34     public String toString(){ return userName;}
35   }
36 }
```

Adding Two Tuples

Listing B.2: Adding Two Tuples

```
// Adding two tuples
// 0\n2

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;

public class BasicExtent2
{
  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    Follow f = newR Follow();
    System.out.println(f->count());

    f << (raoul,sophia);
    f << (sophia,sophia);
    System.out.println(f->count());
  }

  private relationship Follow(User, User)
  {

  }

  private class User
  {
    public String userName;
    public String toString()
    {
      return userName;
    }
  }
}
```

Adding Two Tuples Removing One

Listing B.3: Adding Two Tuples Removing One

```
// Adding two tuples removing one
// 3\n2\n

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;

public class BasicExtent3
{
  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    Follow f = newR Follow();

    f << (raoul,sophia);
    f << (sophia,sophia);
    f << (sophia,raoul);
    System.out.println(f->count());
    f->rem(raoul,sophia);
    System.out.println(f->count());
}

  private relationship Follow(User, User)
  {

  }

  private class User
  {
    public String userName;
    public String toString()
    {
      return userName;
    }
  }
}
```

Testing from()

Listing B.4: Testing from()

```
// Testing from()
// 2\ntrue\ntrue

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;

public class BasicExtent4
{
  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    Follow f = newR Follow();

    f << (raoul,sophia);
    f << (sophia,sophia);
    f << (sophia,raoul);

    Set results = f->from(sophia);

    System.out.println(results.size());
    System.out.println(results.contains(sophia));
    System.out.println(results.contains(raoul));
  }

  private relationship Follow(User, User)
  {

  }

  private class User
  {
    public String userName;
    public String toString()
    {
      return userName;
    }
  }
}
```

Testing to()

Listing B.5: Testing to()

```
// Testing to()
// 1\nfalse\ntrue

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;

public class BasicExtent5
{
  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    Follow f = newR Follow();

    f << (raoul,sophia);

    f << (sophia,raoul);

    Set results = f->to(sophia);

    System.out.println(results.size());
    System.out.println(results.contains(sophia));
    System.out.println(results.contains(raoul));
  }

  private relationship Follow(User, User)
  {

  }

  private class User
  {
    public String userName;
    public String toString()
    {
      return userName;
    }
  }
}
```

Set attribute value

Listing B.6: Set attribute value

```
1  // Set attribute value
2  // 10
3
4  import uk.ac.ic.doc.jrl.lang.*;
5  import uk.ac.ic.doc.jrl.interfaces.*;
6  import uk.ac.ic.doc.jrl.factory.*;
7  import uk.ac.ic.doc.jrl.exceptions.*;
8  import uk.ac.ic.doc.jrl.visitors.*;
9  import java.util.*;
10
11 public class AttributeExtent1
12 {
13   public void launch() throws Exception
14   {
15     User raoul = new User();
16     raoul.userName = "Raoul";
17
18     User sophia = new User();
19     sophia.userName = "Sophia";
20
21     Follow f = newR Follow();
22
23
24     f << (raoul,sophia).withInterest(10);
25     int interest = f->get(raoul,sophia).getInterest();
26     System.out.println(interest);
27   }
28
29   private relationship Follow(User, User)
30   {
31     int interest;
32   }
33
34   private class User
35   {
36     public String userName;
37     public String toString()
38     {
39       return userName;
40     }
41
42   }
43
44 }
```

Re-Set attribute value

Listing B.7: Re-Set attribute value

```
1  // Re-Set attribute value
2  // 11
3
4
5  import uk.ac.ic.doc.jrl.lang.*;
6  import uk.ac.ic.doc.jrl.interfaces.*;
7  import uk.ac.ic.doc.jrl.factory.*;
8  import uk.ac.ic.doc.jrl.exceptions.*;
9  import uk.ac.ic.doc.jrl.visitors.*;
10 import java.util.*;
11 public class AttributeExtent2
12 {
13   public void launch() throws Exception
14   {
15     User raoul = new User();
16     raoul.userName = "Raoul";
17
18     User sophia = new User();
19     sophia.userName = "Sophia";
20
21     Follow f = newR Follow();
22
23
24     f << (raoul,sophia).withInterest(10);
25     f->set(raoul,sophia).withInterest(11);
26     int interest = f->get(raoul,sophia).getInterest();
27     System.out.println(interest);
28   }
29
30   private relationship Follow(User, User)
31   {
32     int interest;
33   }
34
35   private class User
36   {
37     public String userName;
38     public String toString()
39     {
40       return userName;
41     }
42
43   }
44
45 }
```

Filter used macros

Listing B.8: Filter used macros

```
// Filter used macros
// 2

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;

public class FilterExtent1
{
  public void launch() throws Exception
  {
    Name n1 = new Name();
    Name n2 = new Name();
    Name n3 = new Name();

    Definition d1 = new Definition();
    Definition d2 = new Definition();
    Definition d3 = new Definition();

    n1.name = "macroName1";
    n2.name = "macroName2";
    n3.name = "macroName3";

    d1.definition = "definition1";
    d2.definition = "definition2";
    d3.definition = "definition3";

    Macro m1 = newR Macro();
    m1 << (n1,d1).withUsed(true);
    m1 << (n2,d2);
    m1 << (n3,d3).withUsed(true);

    m1->filter(used == true);

    System.out.println(m1->count());
  }

  private relationship Macro(Name, Definition)
  {
    boolean used;
    int importance;
  }

  private class Name
  {
    String name;
    public String toString() { return this.name; }
  }

  private class Definition
  {
    String definition;
    public String toString() { return this.definition; }
  }
}
```

Filter Interest greather than 9

Listing B.9: Filter Interest greather than 9

```
// Filter Interest > 9
// 1

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;
public class FilterExtent2
{
  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    User janek = new User();
    janek.userName = "Janek";

    Follow f = newR Follow();

    f << (raoul,sophia).withInterest(10);
    f << (raoul, janek);
    f << (sophia, raoul).withInterest(8);

    f->filter(interest > 9);
    System.out.println(f->count());
  }

  private relationship Follow(User, User)
  {
    int interest;
  }

  private class User
  {
    public String userName;
    public String toString() { return this.userName; }
  }
}
```

Pass Extent as parameter to method

Listing B.10: Pass Extent as parameter to method

```
// Pass Extent as parameter to method
// 1

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;

public class FirstClassExtent1
{
  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    Follow f = newR Follow();


    f << (raoul,sophia).withInterest(10);
    printFollowSize(f);
  }

  private void printFollowSize(Follow f)
  {
    System.out.println(f->count());
  }

  private relationship Follow(User, User)
  {
    int interest;
  }

  private class User
  {
    public String userName;
    public String toString()
    {
      return userName;
    }

  }

}
```

Return extent from method

Listing B.11: Return extent from method

```
1  // Return extent from method
2  // 2
3
4  import uk.ac.ic.doc.jrl.lang.*;
5  import uk.ac.ic.doc.jrl.interfaces.*;
6  import uk.ac.ic.doc.jrl.factory.*;
7  import uk.ac.ic.doc.jrl.exceptions.*;
8  import uk.ac.ic.doc.jrl.visitors.*;
9  import java.util.*;
10
11 public class FirstClassExtent2
12 {
13   public void launch() throws Exception
14   {
15     Follow f = getInitialisedExtent();
16
17     System.out.println(f->count());
18   }
19
20   private Follow getInitialisedExtent() throws Exception
21   {
22     User raoul = new User();
23     raoul.userName = "Raoul";
24
25     User sophia = new User();
26     sophia.userName = "Sophia";
27
28     Follow f = newR Follow();
29     f << (raoul,sophia).withInterest(10);
30     f << (sophia,raoul).withInterest(11);
31     return f;
32   }
33
34   private relationship Follow(User, User)
35   {
36     int interest;
37   }
38
39   private class User
40   {
41     public String userName;
42     public String toString()
43     {
44       return userName;
45     }
46
47   }
48
49 }
```

Testing Average Fct

Listing B.12: Testing Average Fct

```
1  // Testing Average Fct
2  // 7.0
3
4  import uk.ac.ic.doc.jrl.lang.*;
5  import uk.ac.ic.doc.jrl.interfaces.*;
6  import uk.ac.ic.doc.jrl.factory.*;
7  import uk.ac.ic.doc.jrl.exceptions.*;
8  import uk.ac.ic.doc.jrl.visitors.*;
9  import java.util.*;
10 public class RelationshipAggregations1
11 {
12   public void launch() throws Exception
13   {
14     User raoul = new User();
15     raoul.userName = "Raoul";
16
17     User sophia = new User();
18     sophia.userName = "Sophia";
19
20     User janek = new User();
21     janek.userName = "Janek";
22
23     Follow f = newR Follow();
24
25
26     f << (raoul,sophia).withInterest(10);
27     f << (raoul, janek);
28     f << (sophia, raoul).withInterest(11);
29
30     double avg = average interest from f;
31
32     System.out.println(avg);
33   }
34
35   private relationship Follow(User, User)
36   {
37     int interest;
38   }
39
40   private class User
41   {
42     public String userName;
43     public String toString() { return this.userName; }
44
45   }
46
47 }
```

Testing Max Fct

Listing B.13: Testing Average Fct

```
// Testing Max Fct
// 11

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;

public class RelationshipAggregations2
{


  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    User janek = new User();
    janek.userName = "Janek";

    Follow f = newR Follow();

    f << (raoul,sophia).withInterest(10);
    f << (raoul, janek);
    f << (sophia, raoul).withInterest(11);

    int maximum = max interest from f;

    System.out.println(maximum);
  }

  private relationship Follow(User, User)
  {
    int interest;
  }

  private class User
  {
    public String userName;
    public String toString() { return this.userName; }
  }
}
```

Testing Min Fct

Listing B.14: Testing Min Fct

```
// Testing Min Fct
// 0

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;
public class RelationshipAggregations3
{


  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    User janek = new User();
    janek.userName = "Janek";

    Follow f = newR Follow();


    f << (raoul,sophia).withInterest(10);
    f << (raoul, janek);
    f << (sophia, raoul).withInterest(11);

    int minimum = min interest from f;

    System.out.println(minimum);
  }

  private relationship Follow(User, User)
  {
    int interest;
  }

  private class User
  {
    public String userName;
    public String toString() { return this.userName; }

  }

}
```

Testing Sum Fct

Listing B.15: Testing Sum Fct

```
// Testing Sum Fct
// 21

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;
public class RelationshipAggregations4
{


  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    User janek = new User();
    janek.userName = "Janek";

    Follow f = newR Follow();


    f << (raoul,sophia).withInterest(10);
    f << (raoul, janek);
    f << (sophia, raoul).withInterest(11);

    int sumInterest = sum interest from f;

    System.out.println(sumInterest);
  }

  private relationship Follow(User, User)
  {
    int interest;
  }

  private class User
  {
    public String userName;
    public String toString() { return this.userName; }

  }

}
```

Testing union

Listing B.16: Testing union

```
1  // Testing union
2  // 3
3
4  import uk.ac.ic.doc.jrl.lang.*;
5  import uk.ac.ic.doc.jrl.interfaces.*;
6  import uk.ac.ic.doc.jrl.factory.*;
7  import uk.ac.ic.doc.jrl.exceptions.*;
8  import uk.ac.ic.doc.jrl.visitors.*;
9  import java.util.*;
10 public class RelationshipOperations1
11 {
12   public void launch() throws Exception
13   {
14     Name n1 = new Name();
15     Name n2 = new Name();
16     Name n3 = new Name();
17
18     Definition d1 = new Definition();
19     Definition d2 = new Definition();
20     Definition d3 = new Definition();
21
22     n1.name = "macroName1";
23     n2.name = "macroName2";
24     n3.name = "macroName3";
25
26     d1.definition = "definition1";
27     d2.definition = "definition2";
28     d3.definition = "definition3";
29
30     Macro m1 = newR Macro();
31     m1 << (n1,d1);
32     Macro m2 = newR Macro();
33     m2 << (n2,d2);
34
35     Macro m3 = unionof m1 with m2;
36     m3 << (n3,d3);
37     System.out.println(m3->count());
38   }
39
40   private relationship Macro(Name, Definition)
41   {
42     boolean used;
43     int importance;
44   }
45
46   private class Name
47   {
48     String name;
49     public String toString() { return this.name; }
50   }
51
52   private class Definition
53   {
54     String definition;
55     public String toString() { return this.definition; }
56   }
57 }
```

Testing intersection

Listing B.17: Testing intersection

```
1  // Testing intersection
2  // 1
3
4  import uk.ac.ic.doc.jrl.lang.*;
5  import uk.ac.ic.doc.jrl.interfaces.*;
6  import uk.ac.ic.doc.jrl.factory.*;
7  import uk.ac.ic.doc.jrl.exceptions.*;
8  import uk.ac.ic.doc.jrl.visitors.*;
9  import java.util.*;
10 public class RelationshipOperations2
11 {
12   public void launch() throws Exception
13   {
14     Name n1 = new Name();
15     Name n2 = new Name();
16     Name n3 = new Name();
17
18     Definition d1 = new Definition();
19     Definition d2 = new Definition();
20     Definition d3 = new Definition();
21
22     n1.name = "macroName1";
23     n2.name = "macroName2";
24     n3.name = "macroName3";
25
26     d1.definition = "definition1";
27     d2.definition = "definition2";
28     d3.definition = "definition3";
29
30     Macro m1 = newR Macro();
31     m1 << (n1,d1);
32     m1 << (n2,d2);
33     m1 << (n3,d3);
34     Macro m2 = newR Macro();
35     m2 << (n2,d2);
36
37     Macro m3 = intersectionof m2 with m1;
38     System.out.println(m3->count());
39   }
40
41   private relationship Macro(Name, Definition)
42   {
43     boolean used;
44     int importance;
45   }
46
47   private class Name
48   {
49     String name;
50     public String toString() { return this.name; }
51   }
52
53   private class Definition
54   {
55     String definition;
56     public String toString() { return this.definition; }
57   }
58 }
```

Testing transitive closure

Listing B.18: Testing transitive closure

```
// Testing transitive closure
// 2\n6

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;
public class RelationshipOperations3
{
  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    User janek = new User();
    janek.userName = "Janek";

    Follow f = newR Follow();

    f << (raoul,sophia);
    f << (raoul, janek);

    Follow transitive = f(*);

    System.out.println(transitive->count());

    f << (sophia,raoul);
    transitive = f(*);
    System.out.println(transitive->count());
  }



  private relationship Follow(User, User)
  {

  }

  private class User
  {
    public String userName;
    public String toString() { return this.userName; }
  }
}
```

Testing reflexive closure

Listing B.19: Testing reflexive closure

```
// Testing reflexive closure
// 3\n5

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;
public class RelationshipOperations4
{

  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    User janek = new User();
    janek.userName = "Janek";

    Follow f = newR Follow();


    f << (raoul,sophia);
    f << (raoul, janek);

    Follow reflexive = f(=);

    System.out.println(reflexive->count());

    f << (sophia,raoul);
    reflexive = f(=);

    System.out.println(reflexive->count());
  }



  private relationship Follow(User, User)
  {

  }

  private class User
  {
    public String userName;
    public String toString() { return this.userName; }

  }
}
```

Testing symmetric closure

Listing B.20: Testing symmetric closure

```
// Testing symmetric closure
// 4

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;
public class RelationshipOperations5
{

  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    User janek = new User();
    janek.userName = "Janek";

    Follow f = newR Follow();


    f << (raoul,sophia);
    f << (sophia, janek);

    Follow symmetric = f(:);

    System.out.println(symmetric->count());

  }

  public void printFollow(Follow f) throws Exception
  {
    foreach((User u1, User u2) : f)
    {
      //int i = f->get(u1,u2).getInterest();
      System.out.println( "("+u1+ "," +u2+ ") ");
    }
  }

  private relationship Follow(User, User)
  {

  }

  private class User
  {
    public String userName;
    public String toString() { return this.userName; }

  }
}
```

Testing Inverse same domain/image

Listing B.21: Testing Inverse same domain/image

```
// Testing Inverse same domain/image
// 2\ntrue\ntrue

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;
public class RelationshipOperations6
{
  public void launch() throws Exception
  {
    User raoul = new User();
    raoul.userName = "Raoul";

    User sophia = new User();
    sophia.userName = "Sophia";

    User janek = new User();
    janek.userName = "Janek";

    Follow f = newR Follow();

    f << (raoul,sophia);
    f << (sophia, janek);

    Follow inverse = f(-);
    System.out.println(inverse->count());

    Set fromSophia = inverse->from(sophia);
    System.out.println(fromSophia.contains(raoul));
    Set fromJanek = inverse->from(janek);
    System.out.println(fromJanek.contains(sophia));
  }

  private relationship Follow(User, User)
  {

  }

  private class User
  {
    public String userName;
    public String toString() { return this.userName; }
  }
}
```

Testing Inverse different domain/image

Listing B.22: Testing Inverse different domain/image

```
// Testing Inverse different domain/image
// 2\ntrue\ntrue

import uk.ac.ic.doc.jrl.lang.*;
import uk.ac.ic.doc.jrl.interfaces.*;
import uk.ac.ic.doc.jrl.factory.*;
import uk.ac.ic.doc.jrl.exceptions.*;
import uk.ac.ic.doc.jrl.visitors.*;
import java.util.*;
public class RelationshipOperations7
{

  public void launch() throws Exception
  {
    Student raoul = new Student();
    raoul.studentName = "Raoul";

    Student sophia = new Student();
    sophia.studentName = "Sophia";

    Course oop = new Course();
    oop.courseName = "OOP";


    Attends a = newR Attends();
    a << (raoul,oop);
    a << (sophia,oop);

    AttendsInverse inverse = a(-);
    System.out.println(inverse->count());

    Set fromOOP = inverse->from(oop);
    System.out.println(fromOOP.contains(raoul));
    System.out.println(fromOOP.contains(sophia));
  }

  private relationship Attends(Student, Course)
  {

  }

  private class Student
  {
    public String studentName;
    public String toString() { return this.studentName; }

  }

  private class Course
  {
    public String courseName;
    public String toString() { return this.courseName; }

  }
}
```

Bibliography

[1] Deborah J. Armstrong. The quarks of object-oriented development. In *COMMUNICATIONS OF THE ACM*. 2006.

[2] Alan Kay. On the meaning of object-oriented programming. `http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en`.

[3] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell*. O'Reilly Media, Inc., 2005.

[4] Russ Miles and Kim Hamilton. *Learning UML 2.0*. O'Reilly Media, Inc., 2006.

[5] Peter Pin-Shan Chen. The entity-relationship model - toward a unified view of data. Technical report, Massachusetts Institute of Technology, 1976.

[6] James Rumbaugh. Relations as semantic constructs in an object-oriented language. In *OOPSLA '87 Proceedings*, 1987.

[7] Gavin Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *ECOOP 2005*, 2005.

[8] Alisdair Wren. *Relationships for object-oriented programming languages*. PhD thesis, University of Cambridge Computer Laboratory, 2007.

[9] Stephen Nelson. First-class relationships in object-oriented programs. Technical report, University of Wellington, 2008.

[10] James Noble. Basic relationship patterns. Technical report, Macquarie University, Sydney, 1997.

[11] Alexandra Burns. The relationship detector. Master's thesis, ETH Zurich, 2006.

[12] Stephanie Balzer, Alexandra Burns, and Thomas R. Gross. Objects in context: An empirical study of object relationships. Technical report, ETH Zurich, 2008.

[13] Stephanie Balzer, Thomas R. Gross, and Patrick Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *ECOOP 2007*, 2007.

[14] Apache commons collection library. `http://commons.apache.org/collections/`.

[15] Guava: Google core libraries for Java 1.5+. http://code.google.com/p/guava-libraries/.

[16] Stephanie Balzer and Thomas R. Gross. Verifying multi-object invariants with relationships. In *ECOOP 2011*, 2011.

[17] Raoul-Gabriel Urma. Swapj: An introduction to polyglot. http://www.cs.cornell.edu/projects/polyglot/doc/swapJ-tutorial.pdf.

[18] First year discrete maths course. http://www.doc.ic.ac.uk/~yg/Discrete/notes.pdf.

[19] Robert Sedgewick. *Algorithms in Java, Third Edition, Part 5: Graph Algorithms*. Addison-Wesley Professional, 2003.

[20] *UML 2.0 OCL Specification*.

[21] David J. Pearce and James Noble. Relationship aspects. In *AOSD 2006*, 2006.

[22] David J. Pearce and James Noble. The relationship aspect library. http://homepages.mcs.vuw.ac.nz/~djp/RAL/index.html.

[23] Aspectj library. http://eclipse.org/aspectj/.

[24] Xml path language. http://www.w3.org/TR/xpath/.

[25] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in cw. In *ECOOP 2005*, 2005.

[26] Darren Willis, David Pearce, and James Noble. Efficient object querying for Java. In *ECOOP 2006*, 2006.

[27] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *In 12th International Conference on Compiler Construction*, 2003.

[28] Torbjorn Ekman and Gorel Hedin. The jastadd extensible Java compiler. In *OOPSLA 2007*, 2007.

[29] Gorel Hedin. An introductory tutorial on jastadd attribute grammars. Technical report, Lund University, 2011.

[30] Jaco. http://lamp.epfl.ch/~zenger/jaco/.

[31] Jsr 308: Annotations on Java types. http://jcp.org/en/jsr/detail?id=308.

[32] Torbjorn Ekman and Gorel Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 2007.

[33] Nate Nystrom, Lantian Zheng, Steve Zdancewic, Andrew Myers, Stephen Chong, and K. Vikram. Java information flow. http://www.cs.cornell.edu/jif/.

[34] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *ECOOP 2008*, 2008.

[35] Andrew Jonas, Daniel Lee, and Andrew Myers. J0: A Java extension for beginning (and advanced) programmers. http://www.cs.cornell.edu/Projects/j0/.

[36] Milan Stanojevic and Todd Millstein. Polyglot for Java 5. http://www.cs.ucla.edu/~todd/research/polyglot5.html.

[37] Michael Brukman and Andrew C. Myers. A parser generator for extensible grammars. http://www.cs.cornell.edu/projects/polyglot/ppg.html.

[38] Princeton University and Technical University of Munich. Lalr parser generator for Java. http://www2.cs.tum.edu/projects/cup/.

[39] Jflex - the fast scanner generator for Java. http://jflex.de/.

[40] Polyglot svn. http://polyglot-compiler.googlecode.com/svn/trunk/polyglot/.

[41] Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. Technical report, 2000.

[42] Matteo Baldoni, Guido Boella, and Leendert van der Torre. Relationships meet their roles in object oriented programming. Technical report, 2007.

[43] Stephanie Balzer and Thomas R. Gross. Rumer. http://www.mcs.vuw.ac.nz/raool/papers/rumer.pdf.

[44] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3:11–16, April 1992.

[45] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA 2005*, 2005.

[46] Grails constraints. http://grails.org/doc/1.0.x/guide/single.html#7.1DeclaringConstraints.

[47] Linq. http://msdn.microsoft.com/en-us/netframework/aa904594.

[48] The Java free chart library. http://www.jfree.org/jfreechart/.

[49] Pmd - project mess detector. http://pmd.sourceforge.net/.

Listings

List of Figures