2016

Data Persistence in Eiffel

Jimmy J. Johnson
University of Kentucky, boxer41a@yahoo.com
Digital Object Identifier: https://doi.org/10.13023/ETD.2016.444

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Jimmy J. Johnson, Student

Dr. Raphael A. Finkel, Major Professor

Dr. Miroslaw Truszczyński, Director of Graduate Studies

Data Persistence in Eiffel

DISSERTATION

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the College of Engineering at the University of Kentucky

By

Jimmy J. Johnson

Lexington, Kentucky

Director: Dr. Raphael A. Finkel

Professor of Computer Science

Lexington, Kentucky

2016

Abstract of Dissertation

Data Persistence in Eiffel

This dissertation describes an extension to the Eiffel programming language that provides automatic object persistence (the ability of programs to store objects and later recreate those objects in a subsequent execution of a program). The mechanism is orthogonal to other aspects of the Eiffel language. The mechanism serves four main purposes: 1) it gives Eiffel programmers a needed service, filling a gap between serialization, which provides limited persistence functions and database-mapping, which is cumbersome to use; 2) it greatly reduces the coding burden incurred by the programmer when objects must persist, allowing the programmer to focus instead on the business model; 3) it provides a platform for testing the benefits of orthogonal persistence in Eiffel, and 4) it furnishes a model for orthogonal persistence in other object-oriented languages.

During my research, I created a prototype implementation of the persistence mechanism using it effectively in several programs. Performance measurements showed acceptable performance with some increase in program memory usage. The prototype gives the programmer the ability to add automatic persistence to existing code with the addition of only a few lines of code. The size of this additional code remains constant regardless of the total number of lines of code in the project. Eiffel syntax remains unchanged and nonpersistent Eiffel code runs as is while incurring only a very small speed penalty.

KEYWORDS: data persistence, orthogonal persistence, persistent programming language, object-oriented programming, Eiffel

Jimmy J. Johnson

November 9, 2016
Date

Data Persistence in Eiffel

By

Jimmy J. Johnson

Dr. Raphael A. Finkel
Director of Dissertation

Dr. Miroslaw Truszczyński
Director of Graduate Studies

November 9, 2016

Acknowledgements

I would like to express my special appreciation to my advisor Dr. Raphael Finkel. His advice and encouragement have been invaluable. He put up with my zealousness for the Eiffel programming language and supported my use of it as a platform for this research. I greatly appreciate his dedication to my work and for the opportunity to study under him.

I also want to thank Dr. Jerzy Jaromczyk, Dr. Miroslaw Truszczyński , and Dr. Victor Marek for their guidance, instruction, and encouragement during my studies and research.

I want to thank Dr. Bertrand Meyer for his support as I began my graduate studies. Without him, I would not have had such a wonderful language in which to work, and this project may not have materialized. Mrs. Annie Meyer at Eiffel Software also provided much encouragement as she liaised between Eiffel Software's technical support team and me.

Others at Eiffel Software and at ETH Zurich have been a great help as well and deserve my thanks. Emmanuel Stapf, Jocylyn Fiat, and Alexander Kogtenkov provided help with the Eiffel compiler and many other technical issues for which I am very grateful. Roman Schmocker's help with the Eiffel runtime was indispensable.

Finally, I wish to thank my wife for tolerating my late-night research and programming and for mostly ignoring the ensuing grumpiness the next morning. She held the fort down, allowing me to focus on this research.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

This dissertation presents P-Eiffel[1], which extends the semantics of an Eiffel program to facilitate the automatic storage and retrieval of objects to and from long-term storage, eliminating almost all storage-related code from an Eiffel program and freeing the programmer to focus on the objective of the program. P-Eiffel provides the added functionality through a small change to the Eiffel runtime and a set of library classes, requiring no change to the Eiffel language or compiler. P-Eiffel provides a useful tool for Eiffel programmers, proves that storage and retrieval of objects need not require complex, database-related code or direct file manipulations, and provides a framework for future development and testing. P-Eiffel's semantics and library structure should be feasible in other object-oriented languages. This section introduces and defines P-Eiffel terms and describes the motivation behind this research. Subsequent sections show how a programmer uses P-Eiffel, detail some of its implementation, present some performance data, and describe possible improvements and future research possibilities.

Persistence is the ability of data to live beyond the lifetime of the program that creates the data. More specifically, persistence refers to the ability to capture an object's run-time state as defined by the stored values of the object's fields or attributes (defined below), and later, during a subsequent execution of that program or a different program, recreate an object that has the same state as the original one. The reconstructed objects are distinguishable only by their separation in time. Figure 1.1 illustrates this concept[2], showing executions of a program at two different times. The first execution creates objects with references to other objects and stores the objects and references to a persistent store (defined below). The second execution retrieves the objects and references, placing them into volatile memory. Though the retrieved objects likely occupy different location in volatile memory, they are logically equivalent to the objects in the first execution.

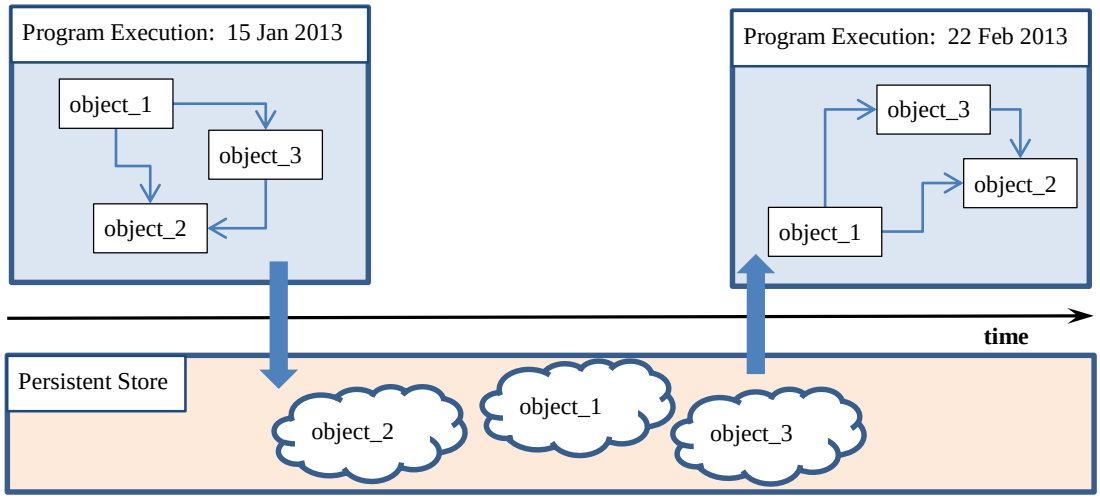

Figure 1.1 – Program executions and a persistent store

Many types of programs require data persistence. For example, word processors and spreadsheet programs preserve the state of a document by writing the document to a file. Online-shopping sites store the records of products, customers, and orders in a non-volatile (i.e. persistent) medium. Airline scheduling and reservation programs store flight, maintenance, and cus-

[1] The name, P-Eiffel, pays homage to the pioneering efforts of PS-Algol.
[2] Figure 1.1 was inspired by a similar diagram in the user manual for an Eiffel persistence framework [26].

tomer information. Company databases store records about employees, customers, and products. Even a long-running program for a single scientific calculation may occasionally save the computation state for resumption after a program failure. The need for easy access to persistence in my own Eiffel programs was the initial catalyst for P-Eiffel.

Meyer recognized the need for data persistence [55] and included basic persistence mechanisms in Eiffel from its beginning. A cursory internet search reveals some type of persistence mechanism for many programming languages, such as Java [88], C++ [17], Python [87], Swift [84], and Smalltalk [28], to name a few. This research, then, does not address whether or not programmers need persistence; it addresses the form persistence should take. This research does not accept that the current persistence methods are good enough; it demonstrates a better approach to persistence. P-Eiffel provides the programmer easy access to storing and retrieving long-lived objects and makes that access automatically available and native within the programming language.

As Atkinson and Morrison [9] say, programming languages have evolved within a paradigm in which persistence was not a major concern. Past engineering trade-offs, necessitated by the disparity between the speeds of system memory and long-term storage media, have led to two disparate technologies: programming languages that concentrate on expressiveness and calculations with short-term data and database systems that focus on the management and reuse of long-term data. The result is programming languages with little or no persistence support and database languages with inadequate computational power.

To facilitate the data movement depicted in Figure 1.1, programmers typically use limited file I/O mechanisms of the programming language or employ complicated interfacing facilities of a database API. In both cases, the programmer must develop code to translate the data to a form acceptable to the file or database system. Fu and Dasgupta [32] estimated that 15% of application code deals with this memory and storage mismatch. Other authors [9, 10, 85] quote a 1978 IBM report [49] that purportedly[1] states that 30% of application code concerns movement of data from the application to non-volatile memory.[2] Furthermore, interfacing with a database necessitates the development and maintenance of two programs, the application and the database; a change in one program forces a change in the other.

To avoid this extra coding effort, Persistent Programming Languages (PPLs), such as those described in Appendix A , attempt to bridge the gap between system programming languages and database languages by adding easier-to-use persistence mechanisms to an existing programming language[3]. These PPLs attempt to make the movement of data from short-term memory to long-term storage transparent to the programmer and independent to other elements of the language, hoping to reap the usually stated benefits of reduced code, higher quality, easier maintenance, and overall improvement in software systems[4]. Despite many years of study in this area, these PPLs have experienced limited success and have seen very little practical use.

This research revisits persistence, specifically persistence in Eiffel. I chose Eiffel as the target language for this research because of its support of most object-oriented concepts, clean

---

[1] I cannot find the actual report.

[2] The movement to object-oriented languages has not removed the requirement for this translation code. In fact, the popular use of a relational database backend for the long-term store has so highlighted the differences in system versus database data representation that the term "impedance mismatch" [95] is now in common use.

[3] Others try to bridge this gap from the other side in the form of Database Programming Languages, which attempt to add computational expressiveness to the data definition and access capabilities of an existing database language.

[4] One of the long-term goals of this study is to test Atkinson's Orthogonal Persistence Hypothesis, which says that a built-in persistence mechanism will produce better code at less cost [8].

syntax, and well-defined semantics, and because of my familiarity and pleasant experiences with its use. An open-source, modifiable compiler is available, which facilitates this research. Eiffel's existing persistence mechanisms illustrate and closely parallel the current state of persistence in other object-oriented languages. Existing persistence mechanisms force the programmer to choose between a severely limited but simple solution and a very capable but overly complex one. P-Eiffel provides a simple and capable alternative. Before summarizing at the end of this chapter the benefits of P-Eiffel and its contribution to computer language development, this dissertation explains Eiffel's existing persistence mechanisms. In order to do so, it first defines terms and explains some Eiffel constructs important to the discussion.

1.1 Terminology

Programming languages use different nomenclature for similar constructs. For example, the terms "member function" (C++), "method" (Java), and "routine" (Eiffel) describe the same general concept. This section defines some general terms used by this paper. The next section gives an overview of Eiffel and Eiffel-specific terms.

- **runtime** – functions included in every execution of a program that implement the core behavior of the language from which the program was built
- **object** – an instance of a type[1]; data that exists in volatile memory during execution of a program, but typically ceases to exist when the program terminates; any piece of program information, such as an integer, array, structure, or class instance
- **strongly typed object** – an object for which enough information exists during program execution to determine its type
- **invariant** – a consistency constraint applicable to an object that must be true at certain times during a program's execution
- **consistent object** – an object that satisfies its invariant (express or implied)
- **attribute** – a value or reference (i.e. a pointer) in an object, taking up space, such as an array element, structure field, or instance variable
- **routine** – a computation performed on the object, usually requiring more time than an attribute access; analogous to a class method
- **persistent store** – the logical construct where objects are maintained between program executions; ideally viewed as infinite, non-volatile memory containing strongly typed objects
- **store an object** – to place into the persistent store enough information about the object so an equivalent object can be reconstructed
- **retrieve an object** – to reconstruct an object based on information obtained from a persistent store, placing the result into volatile memory
- **persistent object** – an object that has been stored in a persistent store, allowing it to live beyond execution of the program that creates it
- **persistable object** – an object that can become persistent but is not required to be so
- **transient object** – an object that cannot be stored

Programs that automate persistence must distinguish persistent and transient objects. Mechanisms to identify persistable objects rely on allocation-based or store-time-based techniques. An allocation-based mechanism, usually operating at compile time, requires the pro-

[1] An object can also be an instance of more than one type. In an object-oriented program, an object that is created as an instance of type PIGEON may also be an instance of types BIRD and ANIMAL.

grammer to mark an attribute or class as persistent or potentially persistent. Run-time mechanisms, operating during program execution, identify persistent objects while traversing the object graph. Run-time identification usually begins at a designated object and follows references until all recursively-referenced objects have been stored.

- **allocation-based persistence** – a method that identifies persistent objects when they are defined or created
- **persistence-by-reachability** – a method that discovers persistent objects at store time by starting at a designated object, recursively storing it and all objects referenced from that designated object
- **persistence root** – a designated object in a persistence-by-reachability approach where a persistence operation begins; the top object in a tree of persistent objects
- **reachable object** – an object obtainable by following references from a persistence root

Type checking ensures compatibility between retrieved objects and program entities. The type checking of persistent objects can use structural equivalence, name equivalence, or a combination of the two.

- **structural equivalence** – type compatibility and equivalence of objects determined by the actual structure or definition [98]
- **name equivalence** – type compatibility and equivalence of objects determined by explicit declarations and/or the name of the types [94]

1.2 "Eiffel: The Language"[1]

Eiffel is a strongly and statically typed object-oriented language. It has a simple, Pascal-like syntax, yet it supports the development of large-scale systems with run-time performance similar to C and Fortran. AXA Rosenberg Investment Management, EMC Corporation, Hewlett Packard, Northrop Grumman, the Chicago Board of Trade, and others use Eiffel. [27] It has generic classes, dynamic binding, and automatic garbage collection. Through feature renaming and redefinitions, it encourages the use of multiple inheritance, leading to the reusability benefits promised by the object-oriented method. This research must anticipate the relationship between these various features of the Eiffel language and the desired functionality of a persistence mechanism. A somewhat contrived example shown below serves as a springboard from which to launch descriptions of this relationship. Before describing the example, though, this paper defines Eiffel-specific terms.

1.2.1 Eiffel terminology

Central to object-oriented programming are the terms class, object, and system. Because writers often interchange or use these terms in confusing ways[2], their definitions as used in this thesis follow.

- **class** – an abstract data type describing a set of possible run-time objects to which the same features are applicable [57]; text written by the programmer
- **object** – an instance of a type; the data that exist during execution of a program

---

[1] "Eiffel: The Language" [54] is the name the first book to cover Eiffel in full.

[2] The author of this paper recently encountered a college textbook that even used the two terms together, referring to a variable declared within a class as a class-object.

- **system** – an assembly of one or more classes, one of which has been designated as the root class, from which a compiler can produce an executable program
- **root class –** the designated class from which execution begins

> **System**, in the context of Eiffel, is a technical term, referring to one or more classes or groups of classes from which a compiler produces an executable program. Compiling an Eiffel system results in an executable program.

Programmers usually group related classes together into clusters using the operating system's directory mechanism. A configuration file directs the compiler to the location of the clusters of classes, from which the compiler automatically determines class dependencies and selects those classes required for compilation.

- **cluster** – a set of related classes grouped together, corresponding to the directory structure of the operating system

> In Eiffel, **class** is the only abstraction for the definition of a type. Eiffel does not define constructs such as `struct` or `union` as seen in other languages.

- **feature** – an operation for accessing or modifying instances of the class [54]
- **attribute** – a value or reference stored in the object

> The values of all the **attributes** of an Eiffel object taken as a whole define that object's state.

- **routine** – a computation, taking zero or more arguments, performed on the object
- **procedure** – a routine that performs some action on an object and does not return a value
- **function** – a routine that returns a **Result**, possibly modifying[1] the object
- **creation feature** – a feature that can be used to initialize an object of that type in a creation statement; similar to a C++ or Java constructor
- **feature call** – a fundamental program construct of the form `obj.some_feature`, applying the feature of name `some_feature`, from the corresponding class, to the object that `obj` denotes (i.e. the target of the call) at that moment in execution [57]
- **qualified call** – a feature call that explicitly lists the target object [57]
- **unqualified call** – a feature call that does not list its target object [57]

> Eiffel can ensure, through assertions (defined below), that an object is in a valid state immediately after returning from a **creation feature** and after any **qualified call**.

[1] Eiffel style guidelines recommend that functions return values without changing an object's state.

1.2.2 A sample system

Figure 1.2 shows the class diagram for a sample system. It models a community of heroes, superheroes, and their sidekicks. The red arrows indicate inheritance (i.e. IS-A) relationships, and the green arrows indicate client (i.e. HAS-A) relationships.

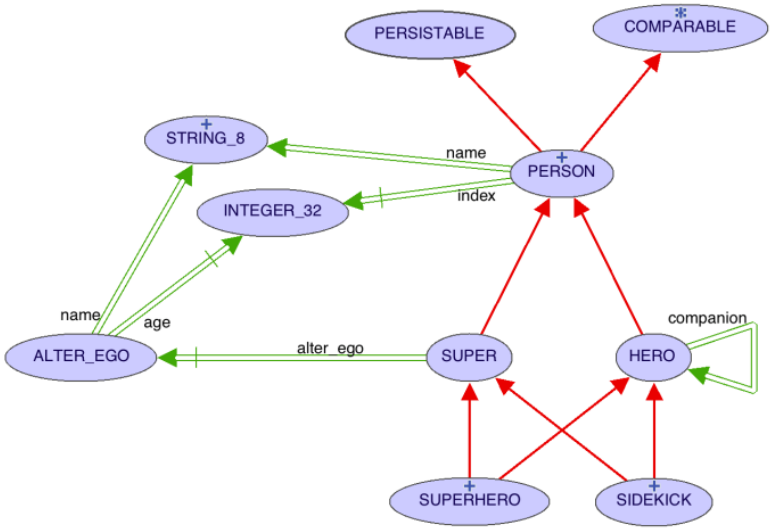

Figure 1.2 – A sample system

The diagram shows that SUPERHERO and SIDEKICK inherit from SUPER and HERO, both of which inherit from PERSON. (The diamond inheritance pattern, shunned in other languages, either by convention or by lack of multiple inheritance capability, presents no problem in Eiffel.) Class PERSON inherits from COMPARABLE, giving PERSON objects a total-order relationship with each other. Class PERSON introduces two stored attributes, name, which determines the ordering, and index, which is a simple basic field; class SUPER adds an alter_ego attribute, because supers cannot walk around in costume all the time; and class HERO gives the person a companion, since every hero needs a sidekick. All these attributes come together in SUPERHERO and SIDEKICK objects. Therefore, every HERO has a companion but not an alter_ego, and every SUPER has an alter_ego but no companion; a SUPERHERO or SIDEKICK has both. This system provides a framework with which to illustrate more Eiffel terminology, the Eiffel methodology, and, later, P-Eiffel.

1.2.3 Classes

The class PERSON shows the simple syntax and basic structure of an Eiffel class[1] and provides examples for more definitions.

```
class PERSON
inherit
    COMPARABLE
create
    make
feature {NONE} -- Initialization
    make (a_name: STRING)
            -- Create an instance, initializing `name'
        do
            name := a_name
        end
```

[1] We depict Eiffel code in colors and fonts similar to the default syntax highlighting used by Eiffel Software's integrated development environment. For brevity, this example lacks adequate whitespace and, with one exception, is devoid of comments.

```
feature -- Access
    name: STRING
    index: INTEGER
feature -- Comparison
    is_less alias "<" (other: like Current): BOOLEAN
        do
            Result := name < other.name
        end
end
```

In this class, make, name, index, and is_less are **features** of the class. (An object of this type actually has many more features, which it inherits from class COMPARABLE and other ancestors.) Features name and index are attributes, and features make and is_less are **routines**. Feature is_less is a **function** taking one argument that must be the same type as the object on which this feature is called. This feature determines the total ordering of PERSON objects based on comparison of the name of two objects. The **create** keyword designates make as a **creation feature**. Feature is_less contains two feature calls to name. The first targets the current object and the second targets the object to which other is attached.

```
feature -- Comparison
    is_less alias "<" (other: like Current): BOOLEAN
        do
            Result := name < other.name
        end
```

This feature compares the name of the current object to the name of another object that is of the same type. The assignment statement **Result** := **Current**.name < other.name is equivalent to the one above, but this form is uncommon.

- **current object** – the object to which the latest non-completed routine call applies [54]
- **Current** – the Eiffel keyword denoting the current object [57]; similar to **self** or **this** in other languages

The PERSON class segregates the features into three groups using **feature** clauses commented with Initialization, Access, and Comparison. These clauses allow the programming tools to present the features in ways that are helpful to the programmer. The clauses also aid information hiding. For example, the {NONE} in the clause before feature make, **feature** {**NONE**} -- Initialization, says the features following this clause down to the next clause are exported to no classes, making the feature uncallable from other classes (except from a creation statement.) A clause such as **feature** {SUPER, HERO} selectively exports the features that follow the clause (down to the next clause) only to the listed classes. Feature clauses without an export qualifier make the features that follow it available to any class.

- **NONE** – a fictional class that is logically the descendant of all classes, having no useful instance [54]
- ANY – the ancestor of all classes, containing general-purpose features; analogous to the Object class in Java or Smalltalk
- **exported feature** – a feature that may be used in a qualified call
- **non-exported feature** – a feature that may not be used in a qualified call
- **selectively exported feature** – a feature that may be used in a qualified call only from within those classes listed in its enclosing clause

As stated above, an object can be an instance of one or more classes; however, its type never changes. An object created by executing a creation feature from class SUPERHERO is an **instance** of class SUPERHERO. It is also an instance of its ancestral classes HERO, PERSON, COMPARA-

BLE, and ANY, giving it access to all the features in those ancestral classes. The object is a **direct instance** of only one class, SUPERHERO, its **generating type**.

- **generating type** – the class from which an object was created

1.2.4 Eiffel initialization rules

The following code segment illustrates the creation and use of the PERSON class by a client in feature get_john.

```
get_john: PERSON
    local
        y: INTEGER          -- silly, unused local
        p: PERSON
    do
        create john.make ("John Galt")
        print (p.name + " at index " + p.index)
        Result := p
    end
```

This feature, though silly, shows the use of the creation feature make from class PERSON. It also serves as a backdrop for illustrating the Eiffel initialization rules and introducing the difference between reference and value types.

The local variables y and p show the two kinds of Eiffel objects: values and references. The local variable y has one of the **basic types** (INTEGER, BOOLEAN, CHARACTER, REAL, and DOUBLE) all of which represent a value. Local variables of features and attributes of classes declared of these types hold the actual value, not a reference. The other local variable p contains a reference to a value, not the value itself. This situation is similar to a pointer or address in other languages but without the problems associated with pointers. Local variables, such as p in this example, and the attributes name from the PERSON class are reference types. When a local variable comes into scope or the program creates an object, the runtime initializes the involved variables to specific values.

Type	Default value
CHARACTER	null character, ''
BOOLEAN	False
INTEGER	0
REAL	0.0
DOUBLE	0.0
any reference type	Void

- Void – a predefined value representing the void value (a value not attached to an object) [57]; analogous to nil or null

When the program flow enters the above function, it sets the local variable y to zero and sets p and the Result of the function (a reference) to Void. After creating a PERSON object and printing a message, the function points Result to the newly created PERSON object and then passes the reference out of the function to the caller.

1.2.5 Reference and value semantics

Value types allow the construction of simple objects similar to Pascal records or C structures. Reference types allow the modeling of complex objects containing links to other objects, as shown in Figure 1.4, where there are multiple references to an object or circular references among two or more objects. Sometimes, however, an object must contain, not a reference to a complex object, but the actual object itself. Attributes declared to have a type that is an expanded class

store values, not references. In this case, that object appears as a sub-object of the enclosing object.

In the sample system, class ALTER_EGO supplies a SUPER with an identity behind which he can hide in order to function in everyday society.

```
class SUPER
inherit
    PERSON
create
    make
feature -- Access
    alter_ego: ALTER_EGO
        -- Every {SUPER} must protect his identity
end
```

In addition, a SUPER must always have an alter_ego (i.e. it cannot be **Void**) and no other SUPER can possess (i.e. reference) that same object. An **expanded** class provides this functionality. The only difference in form between a normal and an expanded class is the addition of the **expanded** keyword at the beginning of the class. (The void-safe compiler also forces the assignment of a non-void reference to the name attribute, which SUPER inherits from PERSON. Void-safety also requires the declaration of default_create, from ANY, as a creation feature. The runtime calls default_create to construct objects in settings where it cannot call a constructor with parameters.)

```
expanded class ALTER_EGO
inherit
    ANY
        redefine
            default_create
        end
create
    default_create,
    make
feature {NONE} -- Initialization
    default_create
        do
            name := "no name yet"
            age := 0
        end
    make (a_name: STRING; a_age: INTEGER)
        do
            name := a_name
            age := a_age
        end
feature -- Access
    name: STRING
    index: INTEGER
end
```

The following code and object diagram illustrate the difference between reference and value semantics. The code creates a SUPER named "Mr Incredible", setting his alter-ego name to "Bob" and his alter-ego age to 40. It then assigns values to local variables b and other_ego. Variable b is a SUPER, a reference type; and variable other_ego is an ALTER_EGO, an expanded type.

```
create incredible.make ("Mr Incredible", "Bob", 40)
b := incredible
other_ego := incredible.alter_ego
```

The diagram below shows the attachment status set up by the routine.[1]

[1] Passing reference or value objects as routine parameters has the same semantics as assignment.

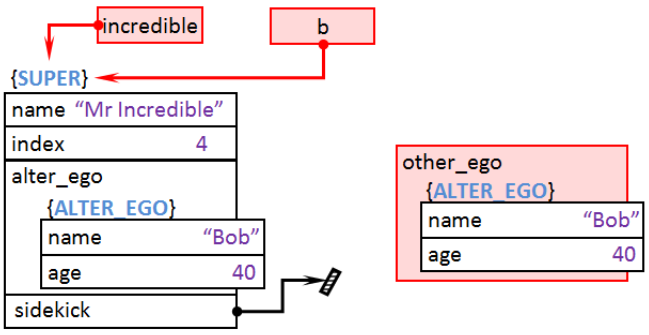

Figure 1.3 – Copy and reference semantics

Because the alter_ego attribute is an **expanded** type, the runtime makes a field-by-field shallow copy[1] into other_ego so that no aliasing occurs.

- **expanded class** – a class describing objects that have copy semantics
- **expanded object** – an object with an expanded generating type, which therefore has copy semantics
- **reference semantics** – attachments through assignment or argument passing results in aliasing
- **copy semantics** – assignment or argument passing produces a shallow copy

1.2.6 Tuples

A tuple is a language mechanism useful for describing an object consisting only of attributes and their accompanying setter features, where a more complicated class is overkill. The programmer creates an instance of a TUPLE with one or more values enclosed in square brackets, treating the resulting object like any other reference. Normal type-checking rules apply.

```
some_feature
    local
        tup: TUPLE [day: INTEGER, month: STRING, year: INTEGER]
    do
        tup := [31, "Jan", 2016]
            ...
```

The programmer accesses components of the TUPLE through the tags (e.g. day, month, and year) provided in the type declaration as if they were attributes of a class. TUPLE types are very useful, avoiding declaration of a class when a Pascal-like record suffices.

1.2.7 Once features

A **once** feature is a routine whose body is executed only when it is called the first time. Subsequent calls have no effect even if the arguments are different. For a **once** function, subsequent calls return the **Result** computed on the first execution.

- **once feature** – a feature whose body is called at most one time

---

[1] For simplicity, the diagrams depict STRING attributes as values residing in objects. Strings are actually reference types in Eiffel; therefore, the structure is more complicated than depicted. The name attribute of the ALTER_EGO object contained in bob and the name attribute in the copied ALTER_EGO object referenced by other_ego refer to the same STRING object, which, itself, has attributes, such as internal_hash_code and count, and a reference to a CHARACTER array called area.

Our sample system imposes a total ordering for an object of type PERSON based on its index. The goal is to sort the objects based on their creation order. In the previous definition of make from PERSON, the initialization rules set the index to zero. Eiffel does not allow global variables, so there seems to be no way for a PERSON object to know what its index should be. A **once** feature, added to PERSON, comes to the rescue.[1]

```
feature {NONE} -- Implementation
    Index_imp: INTEGER_REF
        once
            create Result
        end
```

This non-exported feature returns on its first execution, not an INTEGER, but a reference to an INTEGER_REF, an object containing an INTEGER. Subsequent calls return a reference to that same INTEGER_REF object. This object serves as a global counter. A modified creation routine from the PERSON class increments the counter and records the count in the index attribute.

```
feature {NONE} -- Initialization
    make (a_name: STRING)
            -- Create and initialize using parameters
        do
            name := a_name
            Index_imp.set_item (Index_imp.item + 1)
            index := Index_imp.item
        end
end
```

1.2.8 Design by Contract®

Preconditions and other assertions are a very important part of recommended Eiffel practice. Assertions help programmers create correct software and serve as an aid to documentation.

- **assertion** – a Boolean statement expressing a formal property of runtime values
- **precondition** – an assertion expressing the constraints under which a routine functions properly [56]
- **post-condition** – an assertion guaranteed to hold at the end of a feature's execution if the precondition was satisfied

Preconditions and post-conditions express properties of features. The class invariant expresses properties of objects.

- **class invariant** – an assertion, expressing general consistency constraints applicable to every class instance as a whole [56]; checked after creation of an object and upon entry to and exit from exported features

The class invariant defines conditions under which the object's state, as defined by the values of its attributes, is valid. Recall that the PERSON class under section 1.2.3 has a state consisting of two attributes: name and index. So what constitutes a valid state for a PERSON object? Can a PERSON exist without a name? Can a PERSON have a negative index? Is the first PERSON the zero-th object or the first object? A class invariant answers these design questions and enforces the decisions. The following code segment illustrates these assertions, introduced with the keywords **require**, **ensure**, and **invariant**.

---

[1] Eiffel is not case sensitive, but the usual convention is to use a leading upper case letter for constants, **once** features, and the predefined value **Result**.

```
class PERSON
    ...
feature {NONE} -- Initialization
    make (a_name: STRING)
        require
            argument_exists: a_name /= Void
        do
            name := a_name
            Index_imp.set_item (Index_imp.item + 1)
            index := Index_imp.item
        ensure
            name_was_assigned: name = a_name
        end
    ...
invariant
    name_exists: name /= Void
    index_large_enough: index >= 1
end
```

These invariants also apply to objects of any descendant class such as HERO.

In the sample system, the HERO class models objects that usually have a companion. This **detachable** (i.e. possibly **Void**) companion attribute is also of the HERO type.[1] For this model, the companion of a companion must point back to the original HERO object. The invariant in class HERO expresses this referential integrity constraint.

```
class HERO
inherit
    PERSON
feature -- Access
    companion: detachable HERO      -- can be Void
invariant
    integrity: attached {HERO} companion as c and then c.companion = Current
end
```

The invariant of the HERO class is AND-ed to the invariant inherited from the PERSON class, which itself has invariants accumulated from its ancestors. The interface view of the HERO class shows this invariant accumulation. The interface view shows only the signatures of exported features, hiding implementation details. It has different colors to distinguish it from the normal text view of a class.

```
class interface
    HERO
create
    make (a_name: STRING)
            -- Create an instance, initializing `name'
        require
            argument_exists: a_name /= Void
feature -- Access
    name: STRING
            -- The person's name
    index: INTEGER
            -- Ordinal value tracking order of creation
```

[1] The companion attribute could have been modeled as a PERSON, for which the referential integrity would not be required, but modeling it as a HERO illustrates the point of invariant accumulation without adding uninformative complexity to the invariant.

```
feature -- Access
    companion: detachable HERO
            -- Possibly another hero that travels with Current
    generator: STRING
            -- Name of current object's generating class
            -- (base class of the type of which it is a direct instance)
            -- (from ANY)
        ensure -- from ANY
            generator_not_void: Result /= Void
            generator_not_empty: not Result.is_empty
    -- other exported features not shown
invariant
    integrity: attached {HERO} companion as c and then c.companion = Current
        -- from PERSON
    name_exists: name /= Void
    index_large_enough: index >= 1
        -- from COMPARABLE
    irreflexive_comparison: not (Current < Current)
        -- from ANY
    reflexive_equality: standard_is_equal (Current)
    reflexive_conformance: conforms_to (Current)
end
```

- **invariant accumulation** – the conjunction of [assertion] clauses appearing in the texts of [the current class and all its ancestor classes] [57]
- **interface view** – automatically generated documentation showing exported features and contracts of a class

1.2.9 Void-safe Eiffel

Invariants increase confidence that software is correct. Another, relatively new Eiffel mechanism, void-safety, also helps build quality software. Void-safe Eiffel ensures at compile time that if a program applies a feature to a reference, that reference is attached to some object. The compiler prevents a variable declared as an attached type from ever being set to **Void** or set to anything that can be set to **Void**. In the example code above, companion is the only attribute declared to be **detachable** and allowed to become **Void**; other attributes default to the **attached** type.

The attached rule applies to attributes in much the same way as the class invariant applies to features; an **attached** attribute must be non-void upon completion of the object's creation routine and must remain non-void throughout its lifetime.

1.2.10 Feature renaming

A class may rename features it inherits from other classes for convenience or to avoid name clashes. The SUPER_HERO class illustrates renaming.

```
class SUPER_HERO
inherit
    SUPER
    HERO
        rename
            companion as sidekick
        end
create
    make
end
```

- **feature renaming** – syntax to change the name of a feature in a descendant; used for convenience, as above, or to remove name clashes with inherited features

Eiffel is an object-oriented application development language that supports almost all object-oriented concepts[1]. Eiffel's characteristics, especially its assertion mechanism, support a development practice that helps developers create robust, reliable, and efficient software that scales up well to large systems. However, Eiffel lacks a lean mechanism for the persistence and concurrent sharing of objects.

1.3 The persistence problem

If a programming language does not have a built-in persistence mechanism, the programmer must resort to other, sometimes complicated, and hence more error-prone, mechanisms to move data to and from long-term storage. In older languages, the programmer relies on the input-output mechanisms to store and retrieve data via the file system. This technique may have sufficed for simple objects such as integers and characters, and even for arrays of these simple types; however, it is insufficient for object-oriented languages or any language that allows references through pointers. Consider a hypothetical organization that requires an Eiffel program to track its members. The following Eiffel code shows one possible class and its attributes.

```
class PERSON
feature -- Access
    name: STRING
    index: INTEGER
    companion: detachable PERSON
```

The entity companion is a possible reference to a PERSON object. The entities name and index are basic values stored in the object. (In Eiffel, STRING is a reference type, so name is really a reference to a STRING object, but that level of detail is not important for this discussion.) Figure 1.4 depicts a sample object graph that could exist during the execution of the program, showing circular, shared, and Void references.

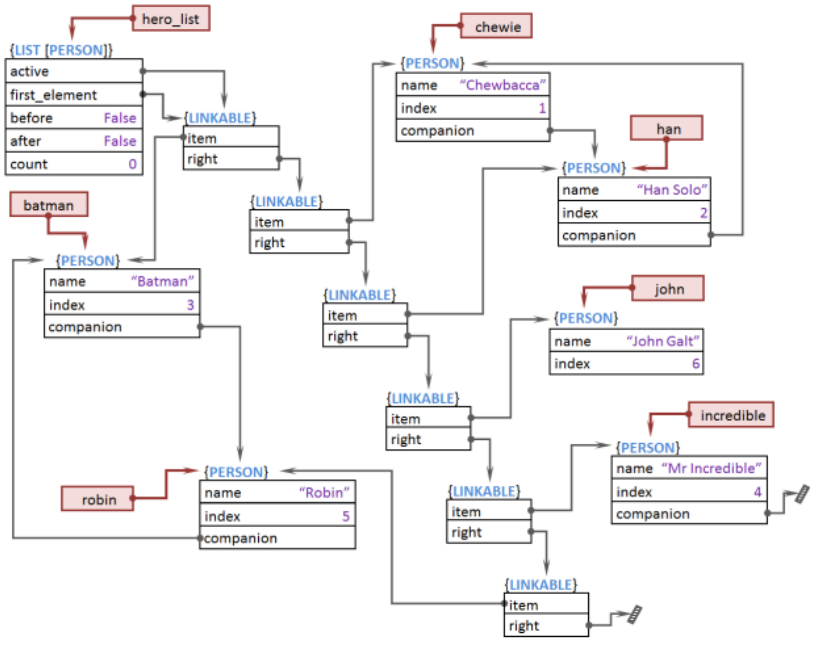

Figure 1.4 – Run-time object structure

[1] Eiffel does not have feature name overloading as found in Ada, C++, C# and Java. Meyer explains the reasoning behind this omission. [66]

The program could store the object referenced by john without storing other objects, but if it stores the chewie, han, or members object, it must also store other reachable objects as well. Bertrand Meyer calls this storage-by-reachability concept the **Persistence Closure Principle**:

> Whenever a storage mechanism stores an object, it must store with it the dependents of that object. Whenever a retrieval mechanism retrieves a previously stored object, it must also retrieve any dependent of that object that has not yet been retrieved. [56]

If a program stores members as the **root** object, it stores all the other objects as well, because they are recursively reachable through references. When the program retrieves members, it must retrieve the entire object structure.[1] In order to obtain this type of processing, programmers typically rely on serialization or database mapping.

1.3.1 Serialization

Serialization represents an object as a sequence of bytes. A serialization mechanism (e.g. the Serializable interface in Java[2] or the Eiffel mechanism shown below) traverses the complete object structure starting at a root object, visiting and converting all reachable objects in the process. Deserialization retrieves the byte sequence, building a complete copy of the original structure.

The Eiffel code below shows how to store and retrieve the objects depicted in Figure 1.4. The code assumes the entities employees, chewie, han, and john exist in the same class as the example routines, so these entities are available for use within the example routines.

```
store_example (a_file_name: STRING)
        -- Save `members' to a file named `a_file_name'
    local
        file: RAW_FILE
    do
        create file.make_open_write (a_file_name)
        file.basic_store (members)
        file.close
    end
```

Given a RAW_FILE object, the second line of the routine, file.basic_store (members) serializes and stores the entire object structure into a disk file. Serialization preserves enough information, including types, so a deserialization operation will be able to rebuild the complete object structure. Deserialization does not restore location in memory, but it does restore content and type. Eiffel programs can restore most objects using this method, but not objects that depend on the current execution context (e.g. threads, sockets, windows, and pointers to routines.) Objects of these types are meaningless outside the constructing process.

Retrieving the entire object structure is also straightforward, as shown below.

---

[1] After a program retrieves the stored structure into members, it loses the original list. Furthermore, the objects referenced from the other handles (e.g. chewie, han, and batman) are not the same objects referenced in the retrieved list. In fact, there may be two PERSON objects with the name "Chewbacca".

[2] Java serialization is similar to Eiffel, but it uses allocation-based persistence combined with persistence-by-reachability; the programmer must designate possibly persistent classes by implementing the Serializable interface.

```
retrieve_example (a_file_name: STRING)
        -- Read the object structure from a file named
        -- `a_file_name' and assign it to `members'
    local
        f: RAW_FILE
    do
        create f.make_open_read (a_file_name)
        if attached {LINKED_SET} file.retrieved as temp then
            members := temp
        else
            print ("Retrieval error")
        end
        file.close
    end
```

The routine retrieve_example calls retrieved on the RAW_FILE object, file, within an assignment attempt, introduced with the keyword **attached**. If the call to file.retrieved returns an object of the expected type, LINKED_SET, the temporary variable, temp, becomes attached to that object and is used within the if statement for assignment to the members attribute.

The serialization routines of the RAW_FILE class (or any other descendant of class IO_MEDIUM) provide an easy way to store and retrieve objects in simple applications, but these routines have a major weakness; there is no way to retrieve only a portion of the stored structure. The program cannot retrieve the object referenced by chewie in Figure 1.4 independently from the rest of the objects. Furthermore, after retrieve_example executes, chewie references an object that may differ from a programmer's expectations. The retrieved routine creates an entirely new object structure, but chewie still references the object that existed before the call to retrieved, not the second object in members. Furthermore, serialization is an all-or-nothing operation in which the identity of each object is lost. Serialization cannot selectively store and retrieve individual objects. In addition, there is no way for two or more running programs to access objects simultaneously. Overcoming these weaknesses and providing more complex store and retrieve operations requires the much heavier database-mapping approach provided by the EiffelStore library.

1.3.2 Database mapping

The EiffelStore library requires considerably more programmer effort, but it provides much greater control over the persistent store. This library allows an Eiffel program to interface with several different types of databases, such as ODBC, MySQL, Oracle, and Sybase. It provides benefits usually associated with a database, such as transactions, security, concurrency, and database triggers. This additional functionality comes with a price; the code is much more complicated. The following summarizes the steps required to use EiffelStore.[1]

The first step is to create an object of type DATABASE_APPL [G], where the generic parameter G is one of ODBC, MYSQL, ORACLE, or SYBASE. Using this object, the program logs into the database and initializes a database handle. Next, the program creates a session manager of type DB_CONTROL. This object allows the program to connect to the database, handle errors, and disconnect. To modify a table, the program creates an object of type DB_CHANGE and calls the modify feature on the object, passing an SQL statement as argument. Using the previously created DB_CONTROL object, the program commits the change. Class DB_SELECTION provides a feature to query the database, which places its result into an object of type DB_RESULT. After converting the DB_RESULT object to a DB_TUPLE, the program accesses individual tuple items using an index. Programs map Eiffel objects directly to relational tables using class DB_STORE and class DB_REPOSITORY.

[1] See [30] for a more complete tutorial.

This cursory description of the database-mapping approach of the EiffelStore library illustrates its complexity. The programmer must be aware of many classes and call the features in the correct order. The interrelations between the classes are complex, making the use of this library difficult to master. This approach also suffers from the two-system problem (i.e. the requirement to develop and maintain the application and a database in parallel.) This approach seems to fit Atkinson's and Morrison's [9] definition of glue-ware, hiding different technologies (a programming language and a database) behind an interface veneer in a hope of combining the two in an understandable and useful way.

This clash between the object-oriented data model and the relational data model has become known as the object-relational **impedance mismatch** [31]. An often-referenced blog post highlights the problems associated with object-relational mapping:

- **Object-to-table mapping problem** – As long as the structure of data in the system is simple, regular, and of fixed size, a class can map directly to a relational table. But object-oriented systems typically have complex relations. Indirect access to an object's property through another object, done simply in an object-oriented language with a call such as `chewie.companion.name`, requires the heavier relational algebra approach in the form of a join. Inheritance, which is hard to represent in relational tables, further exacerbates the problem.
- **Dual-schema problem** – Two programs, the application and the database, must maintain the form of the data, raising the question of schema ownership. Is the schema owned by the application developers or by the database developers? This ambiguity greatly complicates system evolution.
- **Entity identity issues** – An object in an object-oriented system has an implicit sense of identity separate from the state of the object, and this identity is not a concern for the programmer. An object-oriented program simply accesses the object through its references. In a relational database, the identity is part of the object's state—its key.
- **Data retrieval mechanism concern** – Relational database systems can arrange data in a way that optimizes typical retrievals. The relationships between objects in object-oriented programs can lead to very inefficient selections, projections, and joins. For example, a relational database requested to display just the name of a person may be able to select only the name field from a table; whereas an equivalent, object-oriented program may have to retrieve the entire person object, including irrelevant fields (i.e. address, job title, list of children, etc.) in order to construct a valid person object from which to obtain the person's name. [64]

The Database mapping approach, exemplified by the EiffelStore library, attempts to solve the lack of flexibility inherit in the serialization approach, but the problems caused by the object-relational impedance mismatch call for other solutions to the persistence problem.

1.4 Summary

The two Eiffel approaches illustrate the extremes of the persistence problem. The serialization approach is easy to use but suffers from an all-or-nothing dilemma: either the entire object graph is retrieved and useable, or nothing is useable. It suffers a big-exhale and big-inhale problem as the whole graph is stored and retrieved. Object identity is lost, and there can be no simultaneous sharing of objects between programs. Database mapping libraries remedy these problems but require code that is more complicated. Object-oriented database systems attempt to solve the impedance mismatch problem but seem to have had limited success. Persistent programming languages, which attempt to merge database-like functionality with a systems programming language, have also failed to see widespread use. I believe P-Eiffel fares much better.

P-Eiffel fills a long overdue void[1] in Eiffel as it bridges the ease-of-use-versus-functionality gap between the serialization and database-mapping approaches. It provides some of the database-mapping functionality while remaining at least as simple as the serialization approach. Though P-Eiffel requires further testing to determine its speed impact on the data-retrieval mechanism concern, it eliminates the other issues of the impedance mismatch problem. Testing with the current prototype, which is usable as is for programs requiring only local storage, indicates that the speed and memory overhead, though improvable, is acceptable. Testing also shows that P-Eiffel imposes a very small coding burden on the programmer. With P-Eiffel, the programmer who desires to add persistence to a program measures the amount of persistence-related code as a constant, extremely small number of lines, not as a percentage of code. P-Eiffel adds automatic persistence semantics to existing Eiffel constructs without a single change to the syntax of Eiffel. This simple-looking accomplishment coupled with the mentioned benefits proves that automatic persistence need not be complicated for the programmer. Empirical data on the benefits of automatic persistence can be gathered only after programmers have used P-Eiffel for some time. This paper details P-Eiffel's semantics and implementation to encourage its use and in hopes that developers of other object-oriented languages will add similar functionality to their languages.

1.5 Roadmap

The next chapter shows how to use P-Eiffel, illustrating its benefits. Subsequent chapters describe the implementation, list some of the difficulties encountered during development, show the result of some performance tests, suggest improvements, and introduce ideas for continuing research in persistence. Appendix A presents an overview of some of the past approaches to persistence from which this research greatly benefited. Appendix B offers a quick-start guide for setting up and using P-Eiffel.

[1] I remember that Bertrand Meyer, the developer of Eiffel, shortly after becoming Chair of Software Engineering at ETH Zurich, hired a full-time researcher and a PhD student for work on persistence. When I visited Dr. Meyer and this student in 2004, the research seemed promising; but it soon became upstaged by other concerns such as void-safety, concurrency, and automatic testing. Until now, persistence in Eiffel has progressed very little.

Chapter 2 Interface and semantics

P-Eiffel consists of a framework of classes coupled with a modified Eiffel runtime. The framework classes, residing in the jj_persistence cluster, provide the programmer interface to the persistence mechanism and implement most of the persistence functionality. A small modification to the Eiffel runtime, included in the P-Eiffel compiler, makes automatic persistence, which is selectable by the programmer, possible. The persistence classes, which are written in standard Eiffel and require no changes to the language, give a programmer access to persistence functionality through normal Eiffel techniques of inheritance and feature calls. Additionally, the persistence footprint is very small; only a few added lines of code give the programmer full control of the persistence mechanism. The interface classes provide this access. A programmer adds persistence to an Eiffel system by including the jj_persistence cluster in the system, inheriting from one or more of the framework classes, calling appropriate initialization features, and, if desired, turning automatic persistence on.

2.1 Interface classes

The persistence cluster divides the persistence classes into two sub-clusters, the interface cluster and the support cluster. The interface cluster contains about a dozen classes, some of which appear in Figure 2.1. The red arrows indicate inheritance (i.e. IS-A) relationships, and the green arrows indicate client (i.e. HAS-A) relationships.[1] Of the many classes and features available, the programmer need use only a few to take full advantage of the persistence mechanism.

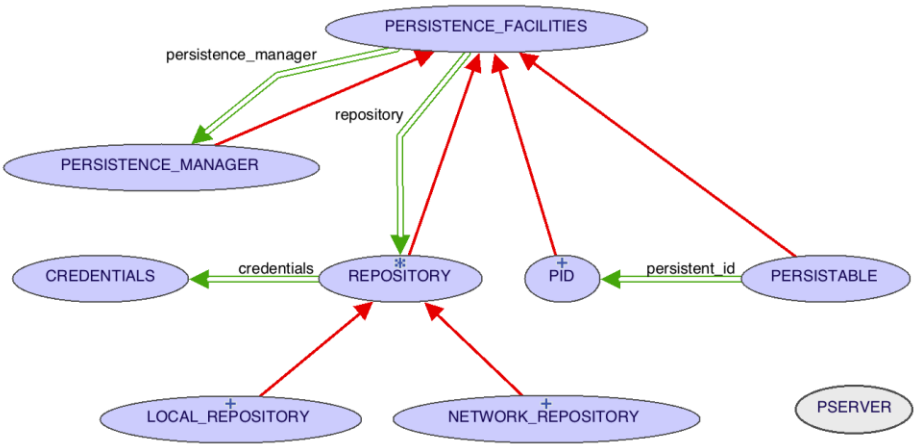

Figure 2.1 – Important interface classes

Class PERSISTENCE_FACILITIES contains queries, constants, and global attributes used by almost all of the persistence classes. A programmer who wishes to add persistence to a system should inherit from this class. The most convenient place for this inheritance relation is usually in the system's root class. Through inheritance, the programmer gains access to the program's single Persistence_manager.

Class PERSISTENCE_MANAGER adds commands to the queries, constants, and attributes of PERSISTENCE_FACILITIES to allow the programmer to initialize, activate, and perhaps fine-tune the persistence mechanism. A REPOSITORY is the abstraction for the data store, the location to which the persistent objects are stored. The programmer initializes the one REPOSITORY in a system as either a LOCAL_REPOSITORY or a NETWORK_REPOSITORY, creating it with the appropriate CREDENTIALS. A LOCAL_REPOSITORY connects to a local file, storing and retrieving objects to and from the hard drive. A NETWORK_REPOSITORY connects to a PSERVER, passing persistent objects across a network.

[1] Eiffel allows attributes to have the same name as a class.

To use these classes and their features effectively, the programmer must understand the semantics of the persistence mechanism.

2.2 Persistent and persistable

P-Eiffel distinguishes between persistent and persistable objects. P-Eiffel provides automatic persistence, but does not demand it. A **persistent object** is one that resides in a persistent store and continues to exist between program executions. A **persistable object** is one whose state P-Eiffel can monitor and whose attributes P-Eiffel can store without explicit programmer involvement. In P-Eiffel, (almost[1]) all objects can become persistable.[2]

P-Eiffel marks an object as persistable by associating the object with a PID (i.e. a persistent identifier). Once P-Eiffel pairs an object with a PID, that association remains for the lifetime of the object. An object gets a PID, becoming persistable, in one of three ways: by an explicit call to identify, through reachability from some other persistable object, or through creation as a PERSISTABLE type.

Though not the ideal method, a programmer can associate an object with a PID, manually making that object persistable, using an explicit call to feature identify. This feature obtains a new PID from the repository. The repository guarantees that the PID is unique. It is better to let P-Eiffel perform this object/PID pairing as it stores or creates objects.

When P-Eiffel stores an object, it uses persistence-by-reachability to achieve Meyer's persistence closure principle. As P-Eiffel explores an object's attributes during a persist operation, it calls identify on newly discovered objects, marking the newly discovered objects as persistable. After an object is so marked, P-Eiffel monitors that object for state changes. P-Eiffel, though, requires at least one object to be persistable already in order to initiate the persistence-by-reachability process. The easiest way to obtain this persistence starting point is to make at least one object begin its life as a persistable object.

P-Eiffel's persistence-through-allocation comes via inheritance from class PERSISTABLE. An object of this type is automatically marked as persistable by a call to identify in this class's creation feature. After an object is marked as persistable, explicitly or otherwise, P-Eiffel stores that object or updates its persistent representation automatically at the appropriate times during execution.

P-Eiffel stores or updates an object only when that object is in a consistent state, that is, when the object's invariant holds, which is after object creation and after execution of an exported feature. Eiffel allows qualified calls (e.g. my_object**.**do_something or Current**.**do_something) for exported features only, so the invariant is sure to hold after such a call. Therefore, if automatic persistence is enabled and the target object of a creation instruction or a qualified feature call is an automatically persistable object, then P-Eiffel stores that object.

When P-Eiffel stores an object, it sends all the attributes of that object to the repository, reducing references (i.e. pointers) to a persistable representation (i.e. a PID). It recursively follows

---

[1] The almost caveat is necessary, because there are some object types for which persistence is meaningless. For example, a network SOCKET does not maintain its connected state when a program using it terminates, therefore restoring its previous state in the context of a different program is meaningless. GUI objects, which rely heavily upon the operating system of the current platform, also lose their state and are not restorable once they go out of scope. Consider an EV_WINDOW. When a window is closed, the operating system removes that graphical element and those graphical elements contained in the window from the screen environment. Even though the underlying Eiffel objects still exist, restoring the corresponding graphical elements is impossible.

[2] Creating a persistable GUI should be possible if the GUI widgets only relied upon its corresponding object state. These self-contained widgets would have to draw themselves instead of relying on the operating system; however, they might lose the native feel of the platform.

references, storing all objects reachable from the original object. As it stores an object, it ensures the repository gets a terse and/or verbose representation of each object's type. Section 3.3 contains more details about the persistent type.

To reduce the time required to store objects, P-Eiffel only stores dirty objects during its recursive traversal of the object structure. P-Eiffel marks an object as dirty when an attribute of that object changes.

2.3 Example system

The sample system that was introduced in Chapter 1 along with its accompanying run-time object structure serves as a springboard to explore P-Eiffel's semantics in more detail and to show P-Eiffel's tiny persistence code footprint. The summary for this section collects all the following code snippets into one root class, highlighting the unobtrusiveness of the persistence code.

2.3.1 Access to persistence and initialization

The root class gains access to the basic persistence features through inheritance.

```
class                                   1
    ROOT                                2
inherit                                 3
    PERSISTENCE_FACILITIES              4
```

Inheriting from PERSISTENCE_FACILITIES grants the class access to the Persistence_manager, from which the programmer controls P-Eiffel's actions. Before making any calls to features that send data to the persistent store, the programmer must set up the repository, which also comes via the above-mentioned inheritance relation. The following feature does this.

```
initialize_repository                   39
        -- Set up the repository.       40
    local                               41
        c: CREDENTIALS                  42
        r: LOCAL_REPOSITORY             43
    do                                  44
        create c.make ("data_file.dat")  45
        create r.make (c)               46
        Persistence_manager.set_repository (r)  47
    end                                 48
```

Feature initialize_repository contains code required only for persistence that is unrelated to the business logic of the program. With P-Eiffel, the programmer is able to restrict this persistence-related code to this single feature instead of spreading intrusive code throughout a system. The root feature calls initialize_repository as its first action.

```
make                                    16
        -- Root feature for the system. 17
    do                                  18
        make_object_structure           19
        initialize_repository           20
        demo_manual_processing          21
        demo_automatic_processing       22
    end                                 23
```

With these preliminaries out of the way, the following sub-sections give details relating to the persistence mechanism. The example progresses from manual persistence, through an intermediate persistence level, to fully automatic persistence.

2.3.2 Create initial objects

The next feature called by the root feature sets up the initial object structure, initializing some of the following attributes that serve as handles to the objects used by the program. As the program progresses, it manually or automatically persists the objects referenced by these attributes. The attribute names themselves are never stored, because the root object, which contains these attrib-

utes, is never stored. Because all object types are persistable, it would have been possible to store the root and hence allow P-Eiffel to build an association in the REPOSITORY between the objects and these attribute names, but this example does not do that.

```
members: TWO_WAY_SORTED_SET [PERSON]                    8
john: detachable PERSON                                 9
chewie: HERO                                            10
han: HERO                                               11
incredible: SUPER                                       12
batman: SUPERHERO                                       13
robin: detachable SIDEKICK                              14
```

The type of each attribute from john down is an heir of PERSISTABLE, either directly in the case of john or indirectly through PERSON for the others. Therefore, these attributes reference objects that are automatically persistable by virtue of their types. Feature make_object_structure initializes the attributes and sets up the initial relationships among some of the resulting objects.

```
make_object_structure                                   24
        -- Set up the test objects.                     25
        -- Leave `john' and `robin' void.               26
    do                                                  27
        create members.make                             28
        create chewie.make ("Chewie")                   29
        create han.make ("Han Solo")                    30
        create batman.make ("Batman", "Adam West", 35)  31
        create incredible.make ("Incredible", "Bob", 40) 32
        chewie.set_companion (han)                      33
        members.extend (chewie)                         34
        members.extend (han)                            35
        members.extend (batman)                         36
        members.extend (incredible)                     37
    end                                                 38
```

Figure 2.2 shows the result of executing make_object_structure. The status reporting feature, is_persistable from PERSISTENCE_FACILITIES, now returns **true** for the objects referenced by batman, chewie, han, and incredible. Queries is_dirty, is_rootable, and is_persistent still return **false** at this point.

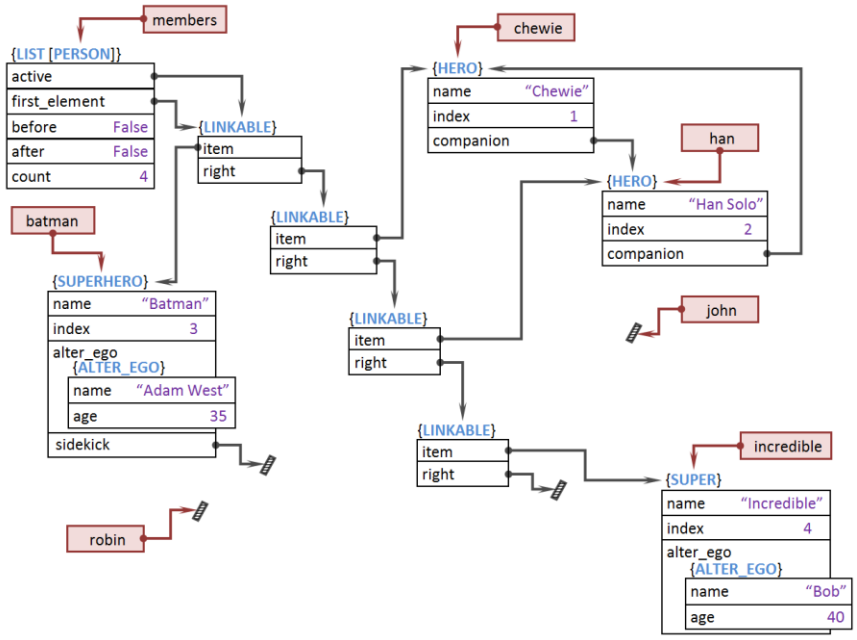

Figure 2.2 – Example structure after initialization

2.3.3 Manual persistence

After the repository is set up, the programmer can manually control persistence by marking and storing objects. Feature demo_manual_processing demonstrates how to explicitly call some of the persistence features. Its first few lines follow.

```
Persistence_manager.persist (chewie)                52
Persistence_manager.persist (incredible)            53
Persistence_manager.mark (batman)                   54
```

The first call to persist[1] stores chewie, and through persistence-by-reachability, stores the han object that is attached as the companion of chewie. The second call to persist demonstrates P-Eiffel's ability to handle expanded types as it stores incredible. Figure 2.3 highlights the persistent state of the three affected objects. (Persistent objects are orange, and dirty objects are grey. Basic (and STRING) attribute values, as opposed to references, are shown in purple.)

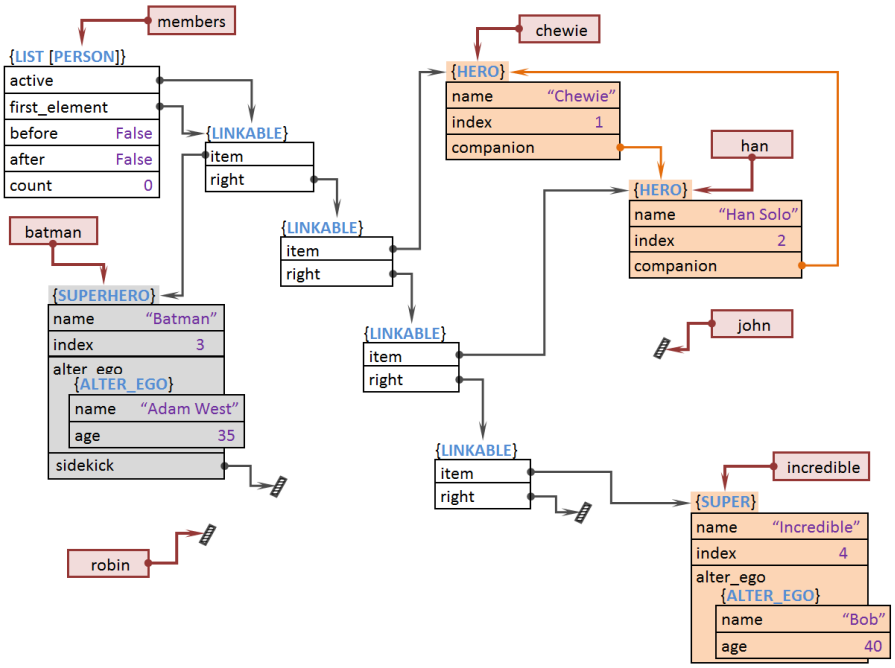

Figure 2.3 – Manual persistence of chewie and incredible

Query is_persistent now returns true for chewie, han, and incredible, but is_dirty returns false. Feature is_persistent_root is also true for the objects because they inherit from PERSISTABLE. The call to feature mark forces batman.is_dirty to return true.

The information sent to the repository includes the persistent type of each object and, because this is the first place where the persistence mechanism encounters the HERO and SUPER types, verbose type descriptions of those types. Subsequent persistence operations involving these two types require sending less information.

2.3.4 Automatic dirty marking

The PERSISTENCE_MANAGER class has features that allow the programmer to choose among three levels of persistence automation, represented by the constants No_automation, Marking_dirty, or Per-

[1] A cleaner alternative in this context is to replace the line "Persistence_manager.persist (chewie)" with "chewie.persist". Feature persist from PERSISTABLE wraps the call from Persistence_manager and allows an argumentless feature call. The example does not use this alternative, because it is only available for objects of type PERSISTABLE. The version used in the example is more general; it is available on all non-basic, non-expanded types.

sisting_automatic. The programmer sets the level of automation through one of the status-setting features or by passing one of the above constants to feature set_persistence_level. The programmer queries the current automation level with feature persistence_level or checks the status of the persistence level with feature is_marking_dirty or is_persisting_automatic.

The next section of code shows the Marking_dirty persistence level. This level of automation is a step up from the manual processing described above, providing some persistence automation while leaving persistence timing to the programmer. The first line of the following code segment calls set_mark_dirty. This feature causes P-Eiffel to automatically mark a persistable object dirty when one or more of its attributes changes. For example, the next line of the feature changes the name of incredible from "Incredible" to the more formal "Mr Incredible".[1]

```
Persistence_manager.set_mark_dirty                    55
incredible.set  name ("Mr Incredible")                56
```

After the assignment statement inside set_name changes the name of incredible, P-Eiffel ensures that incredible is now dirty. Figure 2.4 shows the resulting state.

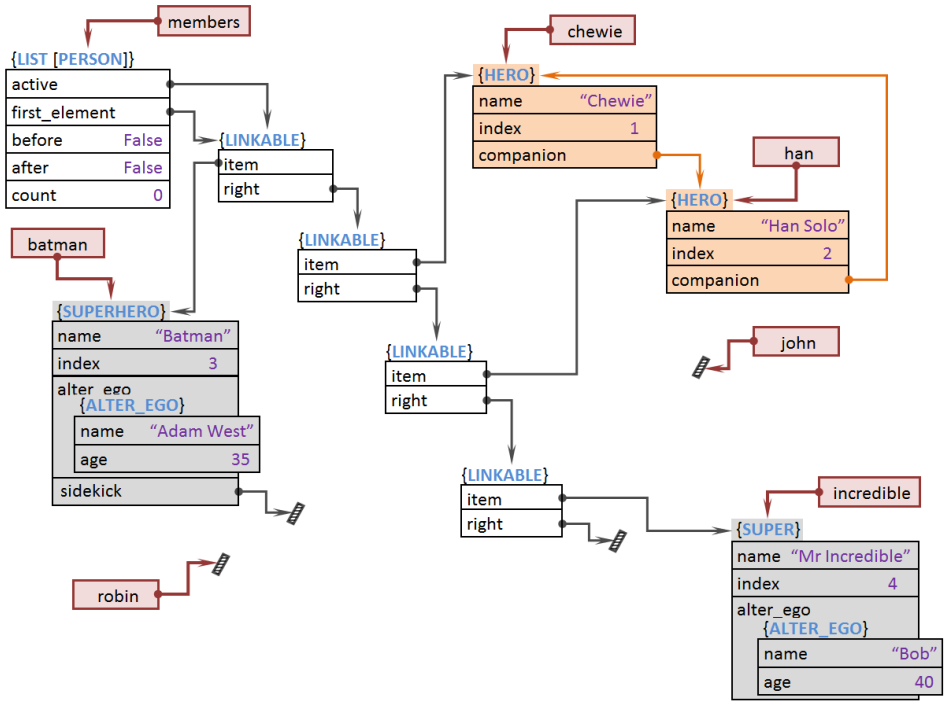

Figure 2.4 – Object states after becoming dirty

2.3.5 Checkpointing

At this point, two of the four PERSON objects are marked dirty. P-Eiffel allows the programmer to persist all dirty objects with a call to checkpoint.

```
Persistence_manager.checkpoint                        57
```

During the checkpoint, P-Eiffel sends all the attributes of batman, including the **void** value for sidekick, along with the verbose description of the SUPERHERO type, to the repository, demonstrating P-Eiffel's ability to handle void references. Because the repository already knows about the SUPER type from a previous persist operation on Incredible, P-Eiffel only sends reduced type information

[1] Eiffel does not allow direct changes to attributes from outside the enclosing class; it requires setter features to perform attribute changes.

for that type along with the new value of the name attribute, which was changed above. The previously dirty objects are no longer dirty and are persistent, as shown in Figure 2.5.

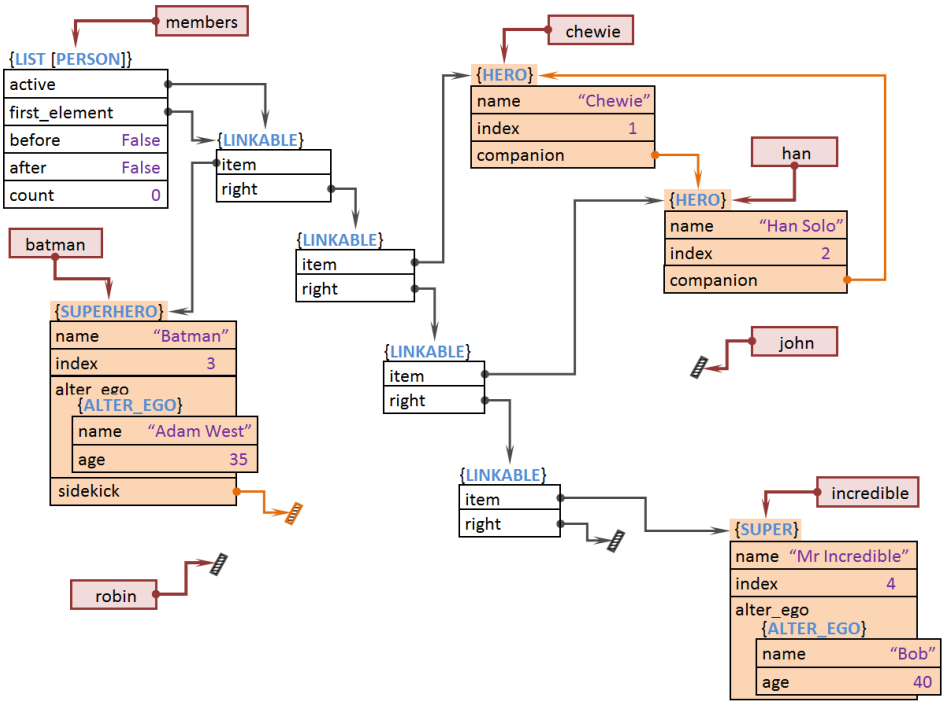

Figure 2.5 – Persistence_manager.checkpoint

As seen above, now that the persistence_level is set to Marking_dirty, an object that is persistable becomes dirty when one of its attributes changes or upon creation. The next two lines of code change chewie and create robin. As a **PERSISTABLE** object where the persistence_level is set to Marking_dirty, the Robin object begins life in a dirty state. It is not persistent, because the code creates it after the above call to checkpoint.

```
chewie.set_name ("Chewbacca")                          58
create robin.make ("Robin", "Dick Grayson", 16)        59
check attached robin as r then                         60
    members.extend (r)                                 61
end                                                    62
```

Setting the persistence level to Marking_dirty followed by intermittent checkpoint calls allows the programmer to control the timing of persist operations. However, a misplaced or forgotten call to checkpoint could lead to data loss. As shown in Figure 2.6, chewie and robin are dirty. If those objects are not manually persisted, the new information could be lost.

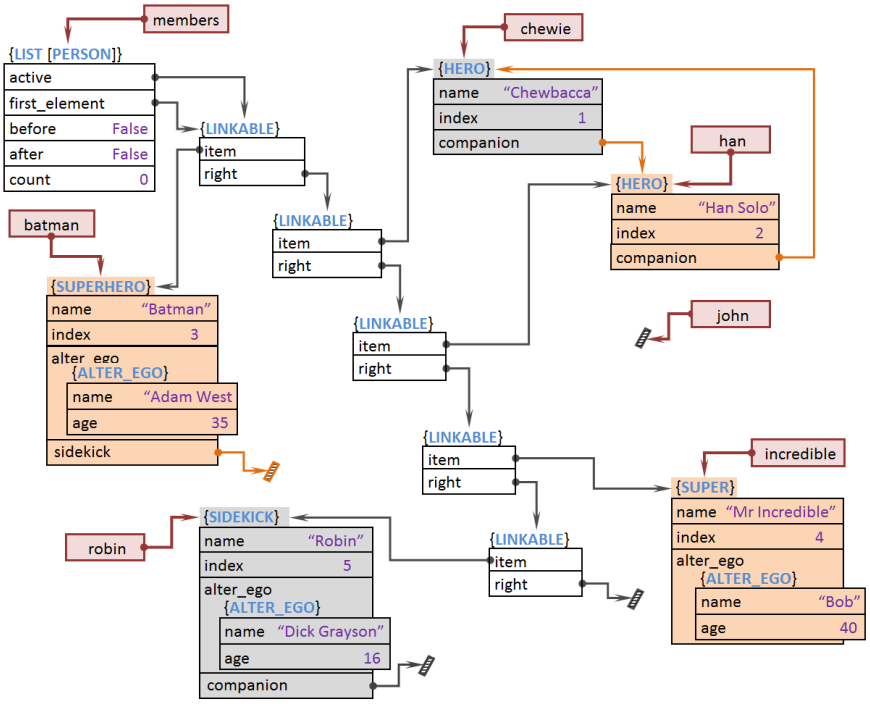

Figure 2.6 – Change chewie's name and create robin

2.3.6 Root-based persistence

One way to ensure the objects are persisted is to make the entire members list persistable, so that the two dirty objects, by reachability, also become persistent.

```
Persistence_manager.persist_as_root (members)                    63
```

Feature persist_as_root is the same as the call to persist except that it promotes its argument to a persistent root, protecting it from collection by the repository's automatic garbage collector.[1] Like persist, it is a manual call that ensures the persistence of its argument and all objects directly reachable from its argument. It strengthens the object's persistence status in relation to the garbage collector, but in the marking_dirty or no_automation level, it does not guarantee the persistence of all reachable objects. P-Eiffel might not visit a dirty object deep in the structure if that object's parent is not dirty at the time that persist_as_root is called. Furthermore, objects below this new persistent root may again be dirtied. Both cases require a call to checkpoint to ensure changes are not lost.

After the call, the persistence mechanism sends the header of members and all its linkable objects to the repository along with verbose type information about the LIST and LINKABLE types. Because this is the first time that robin is stored, P-Eiffel also sends verbose information about the SIDEKICK type. Because the chewie object is dirty from a previous change, P-Eiffel also sends its name attribute to the repository. Figure 2.7 depicts the resulting state.

[1] P-Eiffel does not yet implement the persistence garbage collector. When implemented, it will collect any object that is not a persistent root or that has become unreachable from a persistent root. The programmer must explicitly delete a persistent root object.

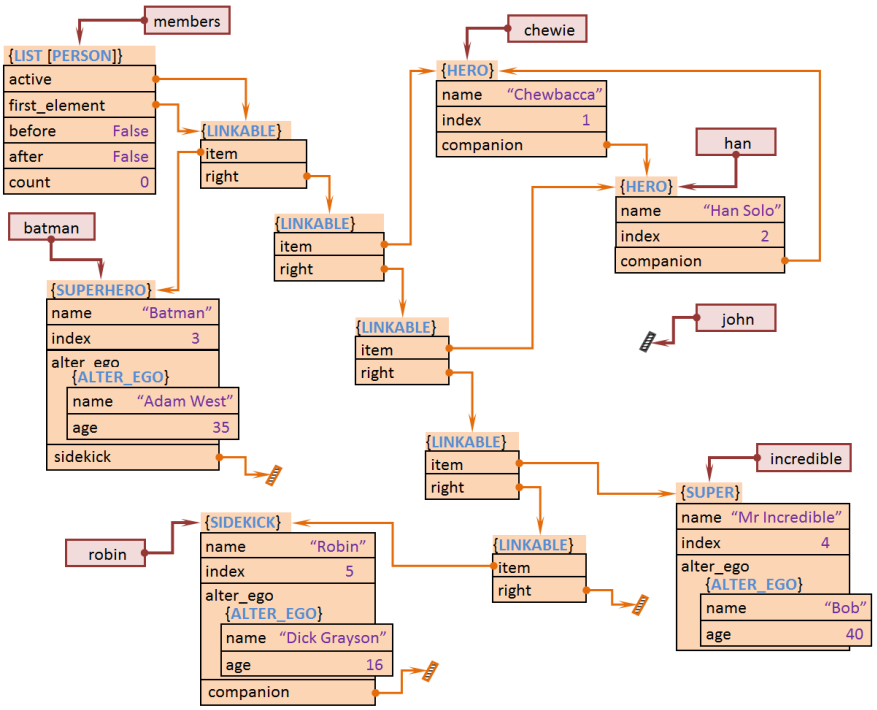

Figure 2.7 – After Persistence_manager.persist_as_root (members)

The manual persistence features mark, persist, persist_as_root, and checkpoint allow the programmer to control timing aspects of persistence, but an oversight or improper ordering of feature calls could lead to data loss. P-Eiffel removes the need for direct programmer involvement and eliminates this possible data loss with its fully automatic persistence mechanism.

2.3.7 Automatic persistence

Feature demo_automatic_processing shows how to initialize P-Eiffel's automatic persistence mechanism and details the resulting semantics. The first line in the feature enables automatic persistence.

```
Persistence_manager.set_automatic                    68
```

After execution of this line, the persistence mechanism begins to mark any persistable object as dirty upon modification of any of its attributes and automatically stores modified attributes of the dirty object upon completion of any qualified feature call on that dirty object. After the call, the programmer proceeds with normal Eiffel code, possibly never calling another persistence feature.[1] For example, the next line of the feature links batman to robin. Because the state, is_persisting_automatic, is now true, the qualified feature call triggers the persistence mechanism, storing the two new references, the sidekick reference from batman to robin. and the companion reference from robin to batman.

```
batman.set_companion (robin)                         69
batman.set_alter_ego_name ("Bruce Wayne")            70
```

The next line tests automatic persistence for an attribute change of an expanded object embedded inside another object by correcting the name of the expanded alter_ego object within batman from

[1] The programmer might occasionally call a query feature to get the persistence status of an object or to ask for the persistence identifier of an object.

"Adam West" to "Bruce Wayne"[1]. Again, after this qualified feature call, batman and alter_ego are dirty but P-Eiffel only sends the dirty attribute, the name of the alter_ego, to the repository.[2]

As described before, the attribute change within set_alter_ego_name causes the persistence mechanism to make batman dirty. When the qualified feature call ends, the persistence mechanism sends only the new name and the persistent type of name to the repository. The companion link is unchanged and robin is not dirty, so the persistence mechanism does not follow the reference or visit robin. In addition, the old name is now unreachable and subject to Eiffel garbage collection. Because the old name was not stored as a persistent root, the repository's garbage collector[3], if enabled, eventually removes the corresponding persistent version of the old name from the repository.

2.3.8 Creating persistable objects

The object structure is almost complete. The final lines of code in demo_automatic_processing create john and adds him to members. The code below creates a **PERSON** and attaches john to it.

```
create john.make ("John Galt")                                    71
```

Besides rounding out the organization's members, it shows the one remaining aspect of persistence semantics, the effect of a creation statement when persistence is automatic. After the creation statement, P-Eiffel checks the persistable status of john, the target of the creation statement, and, because john is persistable by virtue of its **PERSISTABLE** type, P-Eiffel sends john and the type information for **PERSON** to the repository. All six **PERSON** objects now exist, and a version of all of them exists in the repository.

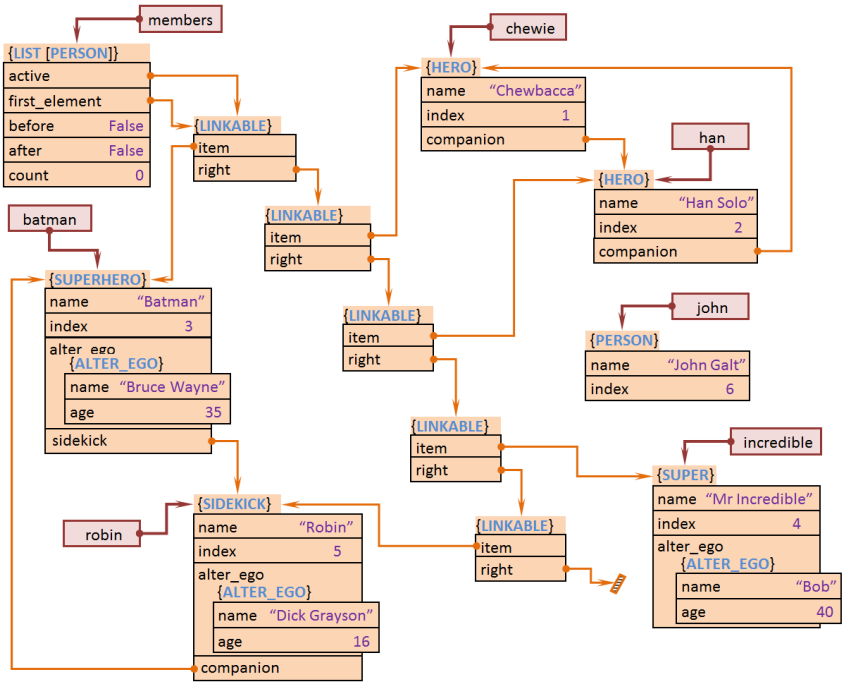

Figure 2.8 – After create john

---

[1] Adam West is the name of the actor that played Batman in the campy 1960's Batman TV series. Bruce Wayne is the character name of Batman's alter ego.

[2] There is a little sleight of hand here. Really, P-Eiffel sends all the basic attributes of both robin and alter_ego to the repository. Also, P-Eiffel does not store the sidekick attribute or follow the reference, because batman is not dirty.

[3] As previously noted, the repository's garbage collector is not yet implemented.

Now john is not yet reachable from members, but adding john to members is a normal Eiffel call.

```
check attached john as j then                    72
    members.extend (j)                           73
end                                              74
```

Because the insertion of john changes the count of members, P-Eiffel sends the count attribute of the list's header to the repository. P-Eiffel also adds the new persistable LINKABLE and the modified persistable references, circled in Figure 2.9, to the repository.

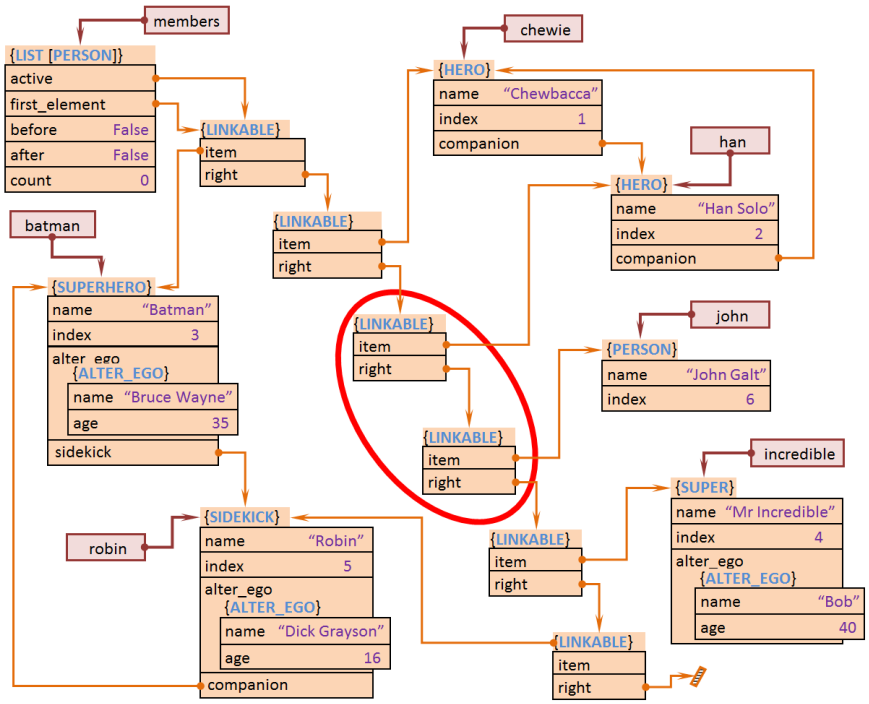

Figure 2.9 – After members.extend (john)

Even though the creation calls and assignment statements that set up these LINKABLE objects and trigger the persistence operations are far removed from the extend call itself, P-Eiffel automatically handles persistence, allowing the programmer to focus on the real objective of the program.

2.3.9 Loading persistent objects

The final lines of the example program show how to restore a persistent object from long-term storage. A persistence identifier for chewie provides a handle to the persistent representation of that object as stored in the repository. Using this identifier, the current session or a later one can load the corresponding, persistently-referenced object and any objects reachable from it.

```
pid := persistence_id (chewie)                                  81
if repository.is_stored (pid) then                              82
    check attached {PERSON} Persistence_manager.loaded (pid) as p then  83
        my_person := p                                          84
    end                                                         85
end                                                             86
```

This code segment simply obtains the pid from the chewie object that is still active, so the entity my_person becomes attached as an alias to that same chewie object. Normally, though, a program loads an object from a PID recorded during a previous session. When such code runs in a different session, entity my_person becomes attached to a newly created object, initialized with the values that are stored in the repository. This session, or a later session that knows the persistence identifier of the desired object, uses the pid value, whether stored in a variable or written on a piece of

paper, as the actual parameter to feature loaded[1] to refresh chewie and, through reachability, han. The attachment check verifies that the returned object is of type PERSON. In this context, it is easy to see by inspecting the preceding lines that the check statement succeeds. The conditional statement that checks repository.is_stored guarantees the precondition of loaded.

2.4 Summary

The following code consolidates all the code segments discussed above, highlighting the lines that involve programmer use of persistence features.

```
class                                                                    1
    ROOT                                                                 2
inherit                                                                  3
    PERSISTENCE_FACILITIES                                               4
create
    make
feature -- Access
    members: TWO_WAY_SORTED_SET [PERSON]                                 8
    john: detachable PERSON                                              9
    chewie: HERO                                                        10
    han: HERO                                                           11
    incredible: SUPER                                                   12
    batman: SUPERHERO                                                   13
    robin: detachable SIDEKICK                                          14
feature -- Initialization
    make                                                                16
            -- Root feature for the system.                            17
        do                                                              18
            make_object_structure                                       19
            initialize_repository                                       20
            demo_manual_processing                                      21
            demo_automatic_processing                                   22
        end                                                             23
    make_object_structure                                               24
            -- Set up the test objects.                                 25
            -- Leave `john' and `robin' void.                           26
        do                                                              27
            create members.make                                         28
            create chewie.make ("Chewie")                               29
            create han.make ("Han Solo")                                30
            create batman.make ("Batman", "Adam West", 35)              31
            create incredible.make ("Incredible", "Bob", 40)            32
            chewie.set_companion (han)                                  33
            members.extend (chewie)                                     34
            members.extend (han)                                        35
            members.extend (batman)                                     36
            members.extend (incredible)                                 37
        end                                                             38
```

[1] An alternative to is to replace the entire check statement with robin.load (pid). Feature load from PERSISTABLE wraps the contents of the check statement and has the same semantics, assuming its preconditions are met. Just like the parameterless version of persist, it is only available to PERSISTABLE objects.

```
    initialize_repository                                                    39
            -- Set up the repository.                                       40
        local                                                                41
            c: CREDENTIALS                                                   42
            r: LOCAL_REPOSITORY                                              43
        do                                                                   44
            create c.make ("data_file.dat")                                  45
            create r.make (c)                                                46
            Persistence_manager.set_repository (r)                           47
        end                                                                  48
feature -- Basic operations
    demo_manual_processing
        do
            Persistence_manager.persist (chewie)                             52
            Persistence_manager.persist (incredible)                         53
            Persistence_manager.mark (batman)                                54
            Persistence_manager.set_mark_dirty                               55
            incredible.set_name ("Mr Incredible")                            56
            Persistence_manager.checkpoint                                   57
            chewie.set_name ("Chewbacca")                                    58
            create robin.make ("Robin", "Dick Grayson", 16)                  59
            check attached robin as r then                                   60
                members.extend (r)                                           61
            end                                                              62
            Persistence_manager.persist_as_root (members)                    63
        end
    demo_automatic_processing
            -- Demonstrate automatic persistence features.
        do
            Persistence_manager.set_persist_automatic                        68
            batman.set_companion (robin)                                     69
            batman.set_alter_ego_name ("Bruce Wayne")                        70
            create john.make ("John Galt")                                   71
            check attached john as j then                                    72
                members.extend (j)                                           73
            end                                                              74
        end
    demo_loading
            -- Demonstrate loading features.
        local
            pid: PID
        do
            pid := persistence_id (chewie)                                   81
            if repository.is_stored (pid) then                               82
                check attached {PERSON} Persistence_manager.loaded (pid) as p then   83
                    my_person := p                                           84
                end                                                          85
            end                                                              86
        end
end  -- class
```

The listing shows that using P-Eiffel requires only a small amount of persistence-related code, and most of that code is confined to initialization, manual operations, and loading. Once initialized through a few features of the interface classes, mostly from PERSISTENCE_FACILITES and PERSISTENCE_MANAGER, persistence is automatic[1], as shown by feature demo_automatic_processing. Furthermore, other classes do not require awareness of persistence, nor do they need to make any

[1] The programmer may revert to one of the non-automatic persistence modes at any time. In Marking_dirty mode, subsequent changes to a persistable object still marks that object as dirty and the changes can be persisted with a call to checkpoint. If the programmer totally disables persistence, P-Eiffel no longer guarantees that changes to previously persisted objects are stored, even after a call to checkpoint.

calls to persistence features. P-Eiffel provides automatic persistence while requiring programmer familiarity with only a small subset of the persistence features of the interface. The use of P-Eiffel does not require programmer knowledge of its inner workings; nevertheless, the programmer may desire a cursory understanding of the implementation.

Chapter 3 Implementation details

P-Eiffel does a lot of work behind the scenes to make automatic persistence possible. The functionality P-Eiffel adds to Eiffel requires no change to the language. The persistence framework classes coupled with a few changes to the Eiffel runtime provide automatic persistence with acceptable memory and time overhead. Appendix B shows the changes to the runtime. This section delves into the inner workings of the persistence classes and the modified runtime. It describes the implementation of the persistence identifier and the interface to the modified runtime. It details the type tracking mechanism and the persistence algorithm. The section ends with a description of the open-ended nature of the actual data storage mechanism.

3.1 Persistence identifiers

In an object-oriented program, an object has implicit identity through a reference built into the language. In the P-Eiffel framework, class PID (for persistence identifier) represents a persistable object identity and persistable references. A PID wraps a 64-bit natural number in attribute item. The low-order 32 bits of a PID, the object_identifier, identifies a particular persistable object. The high-order 32 bits, the attribute_identifier, when combined with the object_identifier, provides a persistable representation of an Eiffel reference (i.e. a pointer). An object-attribute pair with a non-zero value for the attribute_identifier represents a reference. For example, the PID 4/2 represents the reference (i.e. a persistable representation of a pointer) to the second attribute of the fourth identified object. An object-attribute pair with a zero in the attribute_identifier identifies an object. The persistence framework classes use PID throughout to track and access persistable objects or to obtain the object referenced from an attribute of a persistable object.

Class PERSISTENCE_FACILITIES tracks persistable objects and monitors the persistence status of objects in once (i.e. global) hash tables, allowing O(1) lookup with a PID key. Feature Dirty_objects keeps track of objects that have undergone an attribute change since the last persist operation. Feature Rooted_objects keeps track of persistable objects that are persistable as persistent roots. The Expanded_links table is a necessary indirection required to link an expanded object to the attribute of its enclosing object. Finally, feature Identified_objects keeps track of each persistable object along with its persistable type. Associating an object with a PID in this table, which is automatic, allows the framework to find an object or the type of an object when it knows the PID of that object. The class also has feature persistence_id for O(1) lookup of a PID given an object. The feature obtains the persistence identifier stored in the header of each object by the modified runtime.

3.2 The P-Eiffel runtime

P-Eiffel preserves object identity through a PID, a 64-bit natural number[1] added, in the runtime, to the header of all non-basic objects. The framework classes use the low-order 32 bits of this value to track persistable objects during a session and to identify persistent objects saved to an external datastore. The persistence mechanism sets the persistence identifier of an object to a non-zero value in feature identify from class PERSISTENCE_MANAGER in order to mark that object as persistable. The creation features of PERSISTABLE call identify when initializing an object of that type. The persistence mechanism also calls the feature when it discovers an object that should be persistable because of reachability from some other object. To reiterate, the persistence identifier, accessible with query persistence_id from the PERSISTENCE_FACILITIES class and set by identify from PERSISTENCE_MANAGER, is not an attribute of an object; it is part of the object header provided by the P-Eiffel runtime.

[1] This modification of the object header requires an additional 64-bits as padding for memory alignment.

Class CALLBACK_HANDLER serves as the bridge between the framework classes and the modified runtime. This class contains the calls to the C routines of the runtime. Feature persistence_id_from_handler wraps the 64-bit header value and returns it as a PID. Feature set_persistence_id sets the header value of an object when given a PID. Besides access to the persistence identifier of an object, the CALLBACK_HANDLER also implements the callbacks to C routines that provide automatic persistence functions.

P-Eiffel's modified runtime executes a callback in three instances. It calls feature on_modify after all assignment statements to mark a dirty object, and it calls feature on_targeted after creation instructions and after qualified feature calls to persist the targeted object. When P-Eiffel persists an object, along with the attributes of the object, it also persists the type of the object.

3.3 Tracking object types

P-Eiffel tracks the type of each object to facilitate object loading and to ensure runtime checking of types between the repository and the session. During a normal Eiffel session, when the runtime encounters an object of a type previously unused, the runtime maps a dynamic type, represented by an integer, to the type. The dynamic type remains unchanged during the session. However, another execution of the same program, depending on execution order, might map that type to a different dynamic type. Because the dynamic type of objects may differ between each session, the persistence mechanism keeps its own type mapping for persistable objects, which is constant between executions.

When an object becomes persistable, P-Eiffel ensures that the persistence mechanism has a corresponding PERSISTENT_TYPE associated with the dynamic type of that object. The PERSISTENT_TYPE of an object is computed as the 160-bit SHA_1 message digest of the TYPE_DESCRIPTOR of the object's type. A TYPE_DESCRIPTOR is an internal representation, obtained through reflection on an object, of the object's generating type (i.e. the name of the class from which it was created) combined with the names and types of each attribute. Because P-Eiffel builds the TYPE_DESCRIPTOR, and hence the PERSISTENT_TYPE, from a generating type of an object, a descriptor and type remain the same during any execution of a system that was built using the underlying class, providing a combination of name and structural equivalence. P-Eiffel stores this descriptor and type in the repository. The stringified TYPE_DESCRIPTOR produced from the chewie object shows an example of the information stored in a descriptor.

```
<<{HERO} 3 fields [companion:HERO:2, index:INTEGER_32:3, name:attached STRING_8:1] >>
```

It contains the generating type HERO with its three fields, companion, index, and name. It also shows the field types and the position of the corresponding attribute within the HERO class. Here is the PERSISTENT_TYPE produced from that string.

```
3ab3f827c1a217d1c25408208c236162700b03ca
```

During a session, P-Eiffel maintains an association between a TYPE_MAPPING, a PERSISTENT_TYPE, and the current session's corresponding dynamic_type, storing these values in the once feature Type_mapping from class PERSISTENCE_FACILITIES. P-Eiffel builds the mapping, adding the three-value tuple when it first encounters a new type. After this first encounter, P-Eiffel only sends the persistent type, not the entire type description, to the repository as it runs the storage algorithm.

3.4 The storage algorithm

P-Eiffel launches its storage algorithm, manually or automatically, with a call to feature persist from class PERSISTENCE_MANAGER. This feature ensures the object passed as an argument is identified as persistable, creates a flattened representation of the object, and then asks the repository to store that representation.

Feature persist first ensures the object passed as argument is_persistable. If the object has not already been identified as persistable during a previous operation, the persistence mechanism

makes it persistable by a call to identify from PERSISTENCE_MANAGER, which obtains a unique, permanent, persistable identifier from the repository, assigns it to the object's header, and stores it in the session's Identified_objects mapping table. The repository guarantees identifier uniqueness.

Feature persist then creates a TABULATION for the object, calling tabulate to produce a flattened version of the object. The flattening process, encapsulated in the TABULATION class, recursively explores an object structure starting at a root object, following references to each reachable object. The resulting, tabulated form of the object structure consists of a set of hash tables that contain only basic, special values, and persistable PID references along with some type information.

After tabulating the object structure, feature persist asks the repository to store the TABULATION. Descendants of REPOSITORY implement the features that communicate with the underlying datastore. Class MEMORY_REPOSITORY, primarily for testing, simulates persistence by keeping the tabulated form of all persistent objects in memory, never writing the data to a permanent medium such as a file. The class FILE_REPOSITORY saves the tabulated data into a file on the local hard drive. Class NETWORK_REPOSITORY sends the tabulated data across a network to a PERSISTENCE_SERVER. The PESISTENCE_SERVER may then communicate with a FILE_REPOSITORY to save the data locally relative to the server. The next two subsections give more detail about the TABULATION and REPOSITORY classes.

3.4.1 The TABULATION class

The TABULATION class is the heart of the persistence mechanism. Given a PID, feature tabulate explores the object structure of the referenced object, flattening the structure into a format that is easily written to an external medium such as a file or network connection. Basically, a TABULATION is a collection of hash tables containing a representation of one or more object structures rooted at a particular persistent object, where each reference (i.e. pointer) is replaced with a PID (i.e. a persistable reference). As the example program described above persists chewie, and through reachability, han, it produces the TABULATION tables as shown next.

The descriptor_table contains a TYPE_DESCRIPTOR for each object type encountered during the traversal keyed by its corresponding type.

Table 3.1 – The descriptor_table

PERSISTENT_TYPE (key)	TYPE_DESCRIPTOR[1]
e161a18397628154b4114879dfcf87c24d8da95a	<<{HERO} 3 fields [companion:HERO:2, index:INTEGER_32:3, name:attached STRING_8:1] >>
92df2553bd413893615c1fcddb64089bdf944b07	<<{STRING_8} 5 fields [area:attached SPECIAL [CHARACTER_8]:1, count:INTEGER_32:5, internal_case_insensitive_hash_code:INTEGER_32:4, internal_hash_code:INTEGER_32:3, object_comparison:BOOLEAN:2] >>
da57f42995eab8cf94d252b7815a1b342d842c11	<<{ALTER_EGO} 2 fields [age:INTEGER_32:2, name:attached STRING_8:1] >>
0f8546a68e6f2e0336ed267e139ccfef70f77d7f	<<{SPECIAL [CHARACTER_8]} 1 fields [special:SPECIAL [CHARACTER_8]:1] >>

Both chewie and han are of type HERO as well as is the companion field of HERO. The name field is of type STRING_8, which itself contains an area field of type SPECIAL [CHARACTER_8]. The INTEGER_32, BOOLEAN, and CHARACTER_8 types are basic types that the persistence mechanism stores directly in

[1] This table shows the string representation of a TYPE_DESCRIPTOR. The actual value stored in the table is the Eiffel serialization of the descriptor.

the table, so there is no need to map those types. The persistence mechanism creates a table, shown later, for each of the mapped types.

The index_table serves as a dictionary for the type of an object and the time at which that object was last persisted. This example shows all the represented objects with the same time-of-storage, because this particular persist operation updates all the reachable objects.
The TABULATION class has features for storing and retrieving the values in the tables. Given a PID, the value of the corresponding PERSISTENT_TYPE leads the persistence mechanism to the table in which the attributes of all objects of that type are stored.

Table 3.2 – The index_table

PID (key)	PERSISTENT_TYPE	YMDHMS_TIME
2/0	3ab3f827c1a217d1c25408208c236162700b03ca	20160208T205703.247
11/0	92df2553bd413893615c1fcddb64089bdf944b07	20160208T205703.247
3/0	3ab3f827c1a217d1c25408208c236162700b03ca	20160208T205703.247
12/0	0f8546a68e6f2e0336ed267e139ccfef70f77d7f	20160208T205703.247
13/0	92df2553bd413893615c1fcddb64089bdf944b07	20160208T205703.247
14/0	0f8546a68e6f2e0336ed267e139ccfef70f77d7f	20160208T205703.247

The objects_table is a table of tables, where each sub-table holds the fields of all the persistent objects of a particular type keyed on a PID containing an object-attribute pair. Because this example encounters three types, there are three sub-tables.

Table 3.3 – The objects_table

PERSISTENT_TYPE	HASH_TABLE [ANY, PID]	
3ab3f827c1a217d1c25408208c236162700b03ca	PID (key)	ANY
	2/1	11/0
	2/2	3/0
	2/3	2000
	3/1	13/0
	3/2	2/0
	3/3	3000
92df2553bd413893615c1fcddb64089bdf944b07	PID (key)	ANY
	11/1	12/0
	11/2	False
	11/3	0
	11/4	0
	11/5	6
	13/1	14/0
	13/2	False
	13/3	0
	13/4	0
	13/5	8
0f8546a68e6f2e0336ed267e139ccfef70f77d7f	PID (key)	ANY
	12/1	C,h,e,w,i,e
	14/1	H,a,n, ,S,o,l,o

The first row of the objects_table contains a table that holds the attributes for the two HERO objects, object number two and object number three. Attribute number one of object number two, in the first row of the sub-table, shows a PID that references object 11. Looking up PID 11/0 in the index_table gives the PERSISTENT_TYPE of that object which in turn leads to the second sub-table of the objects_table. The five attributes of this STRING_8 object are stored here. For example, attribute five of object eleven shows the basic value 6, which corresponds to the count field of that STRING_8 object. Object chewie does indeed have six characters.

Flattening an object structure into a TABULATION replaces each Eiffel reference with a persistable representation and reduces the entire object structure to easily serializable tables, which a REPOSITORY then saves to a datastore or transmits over a network.

3.4.2 The REPOSITORY class

Deferred[1] class REPOSITORY in the interface cluster exposes the interface to the particular type of repository created during setup. It allows the programmer to interact with stored data. For example, given a PID, the programmer can ask the repository if an object with that PID is_stored or can query the repository for the stored_time and stored_type of an object. The store feature takes a TABULATION as argument and adds the tabulated object structure to the datastore. The loaded feature takes a PID and returns a TABULATION. Normally, though, the programmer does not call these features directly but delegates the calls to the automatic persistence mechanism.

One of the most important automatic features of the REPOSITORY class is next_pid, which supplies the persistence mechanism with a fresh identifier for eventual assignment to some object. The REPOSITORY manages a bucket of available identifiers, ensuring that a supplied identifier is unique. A REPOSITORY can identify up to 2^{31} - 1 unique identifiers. Descendant classes, such as MEMORY_REPOSITORY, LOCAL_REPOSITORY, or NETWORK_REPOSITORY, implement most of the features declared in the REPOSITORY class.

The MEMORY_REPOSITORY class, useful for testing, stores objects directly in memory. This type of repository does not provide true persistence, because the data is lost when the program ends. It stores incoming tabulated objects during the session in its data field, which is of type TABULATION. When the store feature receives an object structure as a TABULATION, it simply merges the incoming tables with its own data tables. This merging is easy to implement and test.

The LOCAL_REPOSITORY class works the same way as MEMORY_REPOSITORY, but it stores its data to a local file after a store operation[2]. Though slow, this class does provide real persistence of objects. A subsequent program execution or even another program can access the objects stored in the data file.

The NETWORK_REPOSITORY class provides the same interface, but instead of storing the persistent objects in memory or to a local file, it passes its information across a network through a socket, wrapping the data in a PMESSAGE. A PERSISTENCE_SERVER at the other end of the network connection interprets the message, answering with the appropriate PMESSAGE. For this research, the server stores its data in a file local to the server through a LOCAL_REPOSITORY implementation. Figure 3.1 shows this repository setup.

---

[1] An Eiffel deferred class is similar to a C++ "pure virtual" class and a Java "abstract" class.

[2] Class LOCAL_REPOSITORY, through the Eiffel kernel class IO_MEDIUM, relies on the underlying operating system to write data to stable storage. The current version of P-Eiffel does not confirm that the actual, physical write actually occurs. It does, however flush the local buffer, making the objects visible to other local processes as if the physical write has actually occurred. Future versions should address this shortcoming.

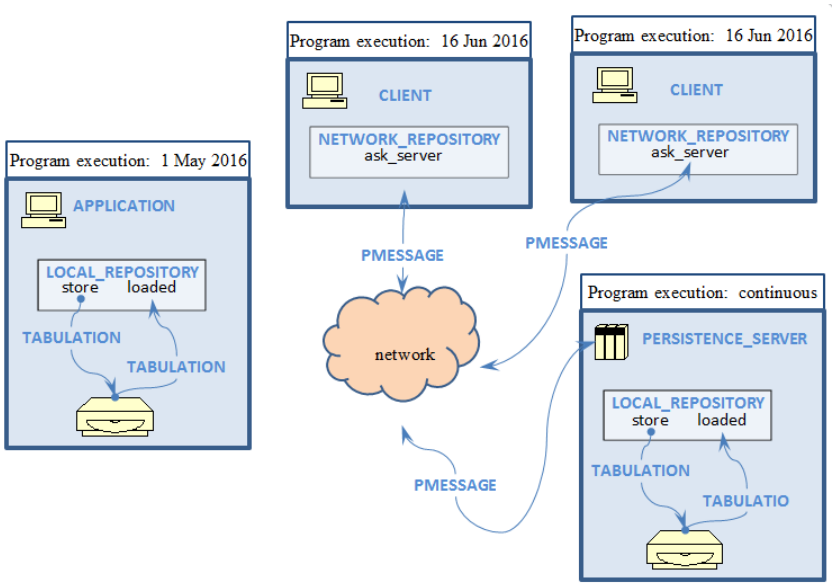

Figure 3.1 – Application and client/server dataflow

The stand-alone application on the left, which shows an execution date of 1 May 2016, stores the tabulated representation of its persistent objects on a local hard-drive through a LOCAL_REPOSITORY object. The two client-server applications on the right, which show an execution date of 16 June 2016, store their persistent objects through a NETWORK_REPOSITORY. Feature ask_server wraps the tabulated representation of its objects in a PMESSAGE, sending the message across a network to the PERSISTENCE_SERVER at the bottom of the figure, which runs continuously. The server unwraps the message and, through its LOCAL_REPOSITORY, stores the enclosed tabulated representation.

3.5 The underlying datastore

The three descendants of REPOSITORY show the flexibility of the framework's implementation by storing data to different types of medium. Though each type of REPOSITORY simply serializes its data to its medium using Eiffel's serialization library, a future descendant of REPOSITORY is free to implement its underlying datastore in other ways.

P-Eiffel's implementation does not dictate the storage method of tabulated objects. A new descendant of REPOSITORY could interface with a relational database or store its objects using a B-tree. Such classes could take advantage of well-known database techniques and greatly improve performance of the back end of the persistence mechanism.

3.6 Summary

The Table 3.4 summarizes the steps of the automatic persistence mechanism, giving the general ordering of the persistence features. The table shows the top-level features only. P-Eiffel associates each persistable object with a long-term persistence identifier, allowing the modified runtime to mark an object as dirty and to store that object and its type automatically. P-Eiffel minimizes programmer involvement in this process, making P-Eiffel very easy to use. Its non-commitment to a specific underlying datastore gives P-Eiffel implementation flexibility, allowing future speed and memory improvements.[1]

[1] See Chapter 5 for a description of planned improvements for P-Eiffel.

Table 3.4 – Automatic persistence steps

Feature	Class	Description
set_repository (r: REPOSITORY)	PERSISTENCE_MANAGER	The programmer calls this feature to tell P-Eiffel where to store persistent objects.
set_persistence_automatic	PERSISTENCE_MANAGER	The programmer uses this feature to enable the automatic persistence mechanism.
identify (object: ANY)	PERSISTENCE_MANAGER	This feature, called by the creation features of PERSISTABLE, queries the repository for a persistence identifier, assigning it to the object.
c_execute_callback (object: ANY)	CALLBACK_HANDLER	This feature is the entry point for P-Eiffel. The modified Eiffel runtime calls it after an object creation, after a qualified feature call, or after an assignment statement. This feature calls execute_callback.
execute_callback (object: ANY)	CALLBACK_HANDLER	This feature selects on_modified or on_targeted based on the context of the runtime's call.
on_modified (object: ANY)	CALLBACK_HANDLER	This feature, called after an assignment statement, marks the parent object as dirty, if that object is persistable.
on_targeted (object: ANY)	CALLBACK_HANDLER	This feature, called after a creation statement or a qualified feature call, persists the object, if that object is persistable.
persist (object: ANY)	PERSISTENCE_MANAGER	This feature produces the tabulation (the flattened representation of the object) by calling tabulate and passes that tabulation on to store.
tabulate (object: ANY)	TABULATION	This feature reduces the object and all dirty objects recursively reachable from that object to a tabulated form, replacing all pointers with persistence identifiers.
store (t: TABULATION)	REPOSITORY	This feature writes the tabulation, the persistable representation of an object structure, to a long-term storage medium.

Chapter 4 Performance

Execution speed and memory footprint of a production program are very important. This research, however, focuses on ease of use and proof of concept, which the previous chapters demonstrate. Nevertheless, it is prudent to look at the performance of a prototype implementation of P-Eiffel. The intent of the speed tests is to show relative performance across P-Eiffel's modes and in comparison to regular Eiffel. Specifically, this section compares P-Eiffel's automatic mode to its mark-and-then-checkpoint mode, showing that Persisting_automatic is more efficient than Marking_dirty followed by a manual call to checkpoint. It explores the computation burden imposed by P-Eiffel's modified runtime upon Eiffel's original runtime, demonstrating that P-Eiffel performs acceptably when there is no persistence despite some performance overhead. The section concludes with a discussion about the additional memory requirements of P-Eiffel as compared to Eiffel.

4.1 Testing method

P-Eiffel's three modes of operations or persistence levels, No_automation, Marking_dirty, and Persisting_automatic, place various demands on a program at different times during execution. A program in which the persistence level is set to No_automation requires very little processor time and scant memory until the programmer manually initiates persistence operations. Persistence levels Marking_dirty and Persisting_automatic, on the other hand, require more time and memory, but the requirements differ between these two modes as well. Executing a test program that persists thousands of objects gives some indication of the relative demands of the three modes.

The test program employs one of three persistence modes (e.g. No_automation, Marking_dirty, and Persisting_automatic) on various sets of objects, ranging from 10,000 objects to 70,000 objects. For each mode, the program performs ten runs, creating the proper number of objects and triggering persistence operations on those objects. The program records the time spent for creations, assignments, and qualified calls, because those three language constructs trigger the P-Eiffel runtime callbacks. It also makes a manual call to trigger checkpointing in order to time that operation. The graphs depict data that was produced by running the single-threaded test program with assertion checking turned off and as few other processes as possible in memory.

On each run of the test program, the main test feature calls three auxiliary features. Each starts a timer, performs the tested operation a bunch of times, and then records the time spent in the loop. Another auxiliary feature records the time required to checkpoint any dirty objects. Here are the calls made by the main test feature.

```
test_creations
test_assignments
test_assignment_calls
test_checkpointing
```

The test_creations feature records the time required to create many persistable objects and places it into an array. The other features then operate on those objects within a similar loop structure.

```
timer.start
from i := 1
until i > test_count
loop
    objects_array.extend (create {TEST_OBJECT})
    i := i + 1
end
timer.stop
statistics.record_creations_time (timer.duration)
```

The test_assignments feature gauges the time overhead incurred by P-Eiffel after every assignment statement. The timing loop occurs within a feature of a single TEST_OBJECT. Assigning a value within a feature of the enclosing object avoids a qualified feature call and thus does not

trigger a persist operation. The feature test_assignment_calls, on the other hand, calls a feature on each object, assigning a value to that object, so it might trigger a persist operation when the call returns. This feature tests the relative time required for P-Eiffel to mark an object as dirty and then possibly persist that dirty object after the qualified feature call returns.

The test program runs on two types of test objects, both of which descend from PERSISTABLE through TEST_OBJECT. The first type contains only basic attributes (e.g. INTEGER, CHARACTER, BOOLEAN, etc.), whereas the second type contains references. With the exception of the one attribute change made by test_assignment_calls, the references remain void. The use of two object types checks P-Eiffel's susceptibility to changes in attribute types. P-Eiffel always persists basic attributes of dirty objects, but it only follows references if the referenced object is dirty, leading to different timings for objects of different makeup.

4.2 Automatic versus manual checkpointing

The graph in Figure 4.1 shows the relative speeds between the persistence modes, Marking_dirty and Persisting_automatic. Under Persisting_automatic, P-Eiffel identifies, marks, and persists a persistable object when that object is created or after the object changes. Under Marking_dirty, it only identifies and marks a persistable object, persisting only on a manual call to checkpoint. Under no_automation, P-Eiffel creation and assignment times (shown in the next sections) are comparable to times in a normal Eiffel program.

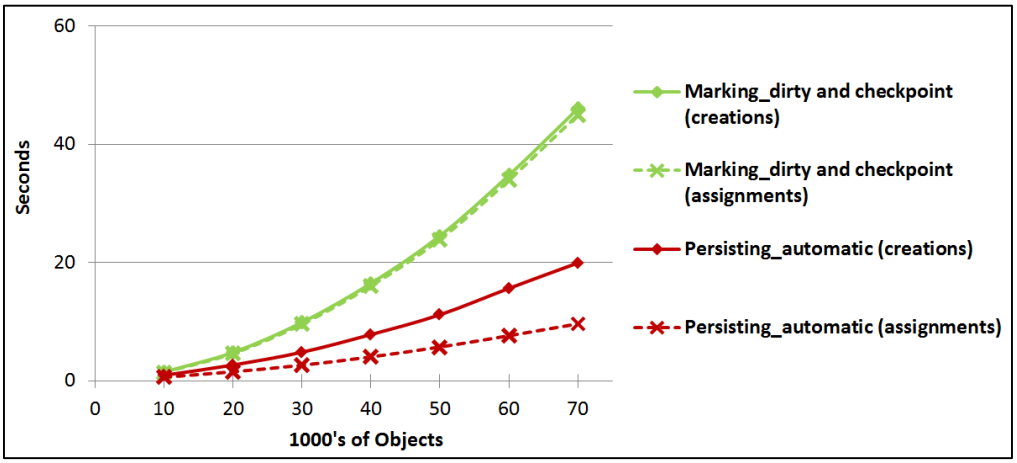

Figure 4.1 – Persisting_automatic versus checkpointing

From the graph, it is clear that the Persisting_automatic mode is faster than the Marking_dirty mode. P-Eiffel requires less time to persist objects as they are created or changed than it does to mark the objects as dirty and checkpoint them later. P-Eiffel marks an object as dirty with relative ease, but persisting a collection of dirty objects during a checkpoint requires a considerable amount of time. Checkpointing requires more time, because P-Eiffel must revisit dirty objects and manage the dirty-object list as each object is persisted. If persistence is fully automatic, P-Eiffel persists a dirty object immediately after it becomes dirty, and there is no dirty-object list to manage. The Persisting_automatic mode is normally the best mode to use.

4.3 P-Eiffel versus Eiffel

The next graphs compare P-Eiffel's time costs to Eiffel. For these tests, the test objects no longer inherit from PERSISTABLE and hence are not automatically persistable. Despite the fact that P-Eiffel persists no objects in these tests, it still checks if persistence is required after each object creation, assignment, and qualified feature call. Figure 4.2 and Figure 4.3 show the overhead of these ad-

ditional calls. Unlike the previous graph, which show time in seconds, these graphs depict time in hundredths of a second.

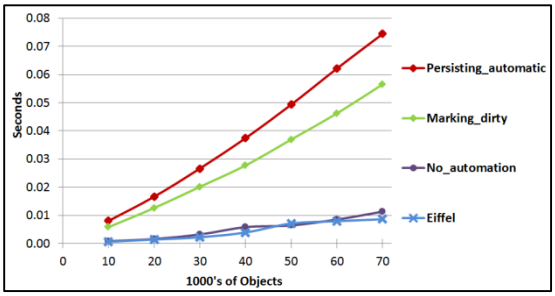

Figure 4.2 – P-Eiffel v Eiffel (creations)

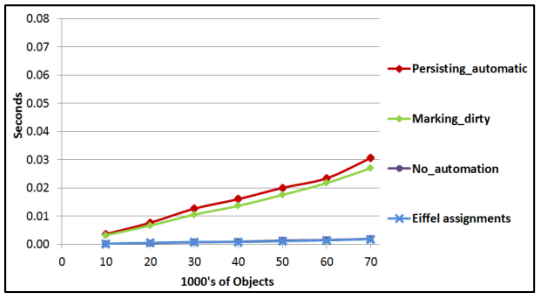

Figure 4.3 – P-Eiffel v Eiffel (assignments)

If the mode of the PERSISTENCE_MANAGER is set to No_automation, the difference in execution times between P-Eiffel and Eiffel is almost immeasurable. However, if the persistence_mode is anything other than No_automation, a program compiled with P-Eiffel requires more time per operation than the same program compiled with normal Eiffel, even if the program persists no objects. If the program requires great speed and does not need persistence, then use the normal Eiffel compiler, not P-Eiffel, or turn off automation.

On the other hand, if object persistence is important, then the programmer should consider using P-Eiffel. The test and timing programs exhibit no user-detectable delays during interactive executions.

4.4 Memory overhead

Program execution time is not the only concern. Execution of a P-Eiffel program requires more memory than a normal Eiffel program. The P-Eiffel framework classes as well as the modified P-Eiffel object header use additional memory.

The framework class PERSISTENCE_FACILITIES tracks persistable, rooted, and dirty objects in globally accessible tables. The impact of the tables themselves, which are **once** features, is small, but the content can become significant when there are many persistable objects. The tables store references to objects, indexing each object by a PID. Ideally, a persistence identifier should use only eight bytes, but because this version of P-Eiffel wraps the 8-byte identifier in a non-basic object, every PID instance incurs the overhead required by non-basic objects, which is driven by the size of the object header.

The largest impact on memory, then, comes from P-Eiffel's object-header format. First, P-Eiffel's runtime adds an 8-byte persistence identifier to each object. Furthermore, it requires an additional 8-byte pad for memory alignment. The resulting object header occupies 32 bytes instead of the 16 bytes required by the normal Eiffel runtime. Table 4.1 shows P-Eiffel's memory allocation scheme applied to a HERO object.

Table 4.1 – Memory allocation for HERO

	Fields	Size (in bytes)
Eiffel Header	type, flags, and processor identifier	16
P-Eiffel Header	persistence identifier plus padding	8 + 8
Reference Fields	name: STRING_8 companion: detachable HERO	4 4
Other Fields	index: INTEGER_32	4
Padding	to bring size to multiple of 32	20
	Total	64

Because P-Eiffel, like Eiffel, allocates space for objects in multiples of the header size, P-Eiffel objects always occupy at least 64 bytes, slightly more space than required by normal Eiffel objects.[1]

4.5 Meeting expectations

Atkinson and Morrison were among the first researchers to add persistence to a programming language. Their research includes PS-Algol [7], which adds persistence to S-Algol, and PJama [8], an early version of persistence for Java. When describing their research, they developed three main principles, collectively called **orthogonal persistence**, which they believe developers of programming languages should pursue. These principles, shown later, are often, like now, quoted in research papers, because they represent the minimum requirements of a persistent programming language. These principles have guided developers of persistent programming languages for the last 30 years. But programmers expect many more capabilities from a persistent programming language, particularly capabilities normally allocated to a database management system. This section describes how well P-Eiffel lives up to Atkinson's orthogonal persistence principles and then examines how well it meets programmers' expectations in database-like operations.

4.5.1 Orthogonal persistence

Atkinson and Morrison believe developers of programming languages should pursue **orthogonal persistence**, as defined by three principles:

1. **The Principle of Persistence Independence –** The form of a program is independent of the longevity of the data which it manipulates. Programs look the same whether they manipulate short-term [transient] or long-term [persistent] data.
2. **The Principle of Data-Type Orthogonality** – All data objects should be allowed the full range of persistence, irrespective of their type. There are no special cases where objects are not allowed to be long-lived or are not allowed to be transient.
3. **The Principle of Persistence Identification –** The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system. The mechanism for identifying persistent objects is not related to the type system. [9]

In older literature, the exact meaning of principle three is unclear or is assumed to have been subsumed by the other two principles. To clarify this ambiguity, Atkinson restates principle three:

3. **The Principle of Completeness or Transitivity –** If some data structure is preserved, then everything that is needed to use that data correctly must be preserved with it, for the same lifetime. [5]

This principle is the same as Meyer's Persistent Closure Principle.

P-Eiffel conforms to all these principles. It satisfies the first principle, persistence independence, because a P-Eiffel program looks just like a normal Eiffel program. Calling a persist feature or inheriting from PERSISTABLE are normal Eiffel constructs. P-Eiffel achieves persistence with no change to the language itself. Furthermore, the amount of application code actually required for persistence in P-Eiffel is minimal.

---

[1] Some of this memory overhead may be reduced in future implementations, as described in Chapter 5.

P-Eiffel places no restrictions or requirements on the type of an object in order for that object to persist. P-Eiffel's persistence type system is completely orthogonal to the Eiffel type system. A P-Eiffel program behaves the same as an Eiffel program, except for the additional persistence semantics. P-Eiffel satisfies principle number two.

P-Eiffel also satisfies Principle number three, completeness. Through persistence by reachability, P-Eiffel ensures all objects that should be persisted are persisted. Satisfying all three principles, P-Eiffel is truly a persistent programming language.

- **persistent programming language** – a programming language that provides some degree of orthogonal persistence

4.5.2 Database features

P-Eiffel achieves the minimum requirements for a persistent programming language, adding persistence to Eiffel without negatively affecting the language, but to be truly useful, it must also meet at least some of the expectations that programmers normally associate with a relational or object-oriented database system. The following paragraphs list these expectations and describe how well P-Eiffel lives up to them.

Database systems were developed to overcome the limitations of storing data in the 1960's era file-processing system, such as unnecessary data redundancy and inconsistency, difficulty in accessing data, data integrity violations, and security and concurrent-access problems [83]. Codd's landmark paper [23] defines the relational database model and sets the stage for its dominance among data models. The model's success has led to the expectation that a database system should achieve certain goals.

The goals required of any database system and of an object-oriented database system are listed in the forward to a collection of readings by pioneers on the subject [90] and summarized by Meyer [56]. Atkinson also stated similar goals in the often-quoted Object-Oriented Database System Manifesto [6]. The bulleted items below show the goals for databases listed by Zdonik et al. and Meyer. Bold font indicates items that Zdonik et al. and Meyer include in the Threshold Model [56, 90] for object-oriented databases.

- **Object identity** [47] – A database must be able to determine if two references point to the same object, to two equivalent objects, or to non-equivalent objects.
- **Encapsulation** – A database must hide the internal properties of objects and make them accessible through an official interface, hiding the data and implementation.
- **Complex state** – Objects must be able to refer to other objects through references.
- Inheritance, overriding, overloading, and dynamic binding – A database may provide these capabilities, which are common to object-oriented programming languages, but whose usefulness in a database language seems unclear and perhaps too complex to be practical.
- Computational completeness – A database may include the ability to express any computable function. This concept is really the whole point of persistent programming languages, but it may add too much complexity to a database language.
- **Integrity constraints** – The user should be able to describe and enforce the correctness and consistency of data.
- **Query mechanism** – There must be some provision to allow users (database users or application programmers) to access data based on properties of the data items.

P-Eiffel achieves the first of these goals, object identity, through class PID and the modified runtime's use of persistence identifiers. P-Eiffel satisfies the other goals (except for overloading, which Eiffel does not incorporate) simply by virtue of the language. P-Eiffel ensures integrity constraints within the application through Eiffel's built-in assertion mechanism, specifically the

class invariant. Enforcing integrity between two different systems would require invariant inclusion in the repository. In order to implement a fully capable query mechanism, P-Eiffel would need to store more type information, encompassing attribute renaming, which is not yet available in P-Eiffel's repository. Chapter 5 describes the plan for adding full integrity enforcement and query capability to P-Eiffel.

- **Programmable structure** – The database must support much more than streams of bytes. It must represent and store relationships and types in the database while keeping the logical and physical representation of the data separate. It must hide most of the data management tasks from the user.
- **Arbitrary size** – The processor or amount of memory must not limit the database's addressable space.

P-Eiffel does not require a separate database with a programmable structure, because the structure and programming occurs through the language within the application. The repository simply holds the data, hiding management of the data from the programmer. The amount of data in the repository is limited only by the underlying data representation as defined in a descendant of the REPOSITORY class. Eiffel, though, does impose some restrictions. For example, the implementation of feature count in the container classes limits the number of items in a container to the maximum value represented by an INTEGER_32 (over 2 billion items). Similar restrictions exist in Java and C++ class implementations. This restriction, though, affects the P-Eiffel (or other language) application only, not a properly designed repository.

- **Permanence** –The data must be accessible beyond execution of the process that created the data and be resilient to system failures. This capability may require some recovery mechanism and/or data redundancy.
- **Distribution** – A database system may distribute data over multiple computers in different geographic locations to improve performance or increase availability.

P-Eiffel's main function is data permanence. P-Eiffel achieves permanence gracefully with no impact on the Eiffel language and requires very little effort from the programmer. Though not yet resilient to system failures, P-Eiffel provides access to data from processes other than the one that created the data. Chapter 5 addresses failure recovery, data redundancy, and the closely related concept of data distribution and suggests possible paths for adding these functions to P-Eiffel.

- **Authorization or access control** – The database must allow users to own data and have a way to grant access to others.
- **Administration** – A database system requires tools to monitor, reorganize, and change users to the database.

The P-Eiffel prototype developed for this dissertation does not enforce access control, but it does contain the basic mechanism, class CREDENTIALS, which encapsulates the concept. Expanding the capabilities of that class should be straightforward. Adding a tool that manipulates user and connection information contained in a PSERVER should also be easy. These two goals are not necessary for demonstrating the feasibility of a persistent programming language like P-Eiffel, so they can be addressed after completion of other, more important functions.

- **Sharing** – A database system must allow multiple programs to simultaneously access data created by another program. A snapshot or checkpointing system, such as provided by class IO_MEDIUM is not enough.

P-Eiffel supports sharing through classes NETWORK_REPOSITORY and PSERVER. Multiple applications may connect through a NETWORK_REPOSITORY to communicate with a single PSERVER object.

- **Locking** – Users or programs should be able to obtain exclusive access to data items.
- **Transactions** – Users or programs should be able to achieve failure-atomic operations in isolation from other operations and allow rollback in the event of failure.

Object locking and transaction support are closely related and necessary for concurrent object access in a shared environment. The current version of P-Eiffel does not implement locking or transaction support, but future research should investigate the feasibility of meeting these goals.

- Object versioning – A database might retain earlier states of an object after the program changes the object.
- Class versioning and schema evolution – Systems always evolve and classes change. The stored objects must follow suit with any class changes if they are to remain usable.

These two goals fall outside the scope of this research. Object versioning provides a history or logging mechanism and may facilitate rollback operations, but this goal is not important for P-Eiffel at this time. Class versioning and schema evolution is important in the long run; some Eiffel classes (e.g. SED_RECOVERABLE_DESERIALIZER and MISMATCH_CORRECTOR) already provide a rudimentary support for schema evolution. Piccioni et al. describe an IDE-based solution to schema evolution integrated with Eiffel Software's Eiffel IDE and compiler [75, 76]. In the future, a new P-Eiffel compiler, developed for other reasons as described in Chapter 5, may be able to incorporate some of their work.

The following table shows each of the above goals along with the orthogonal principles, showing how P-Eiffel fares when evaluated against them. A "yes" in the status column means that P-Eiffel has that capability, and the provided-by column indicates if that capability comes from the Eiffel language (Lang), the P-Eiffel runtime (RT), or from the persistence framework (PF). A "no" in that column means that P-Eiffel does not yet provide that capability. The "yes and no" for inheritance and dynamic binding indicates that while Eiffel provides these mechanisms, P-Eiffel does not yet include the inheritance structure of objects within the repository. The requirement column indicates where to focus future work in order to provide a particular capability.

Table 4.2 – Persistence goals and P-Eiffel

Goal	Status	Provided by	Requirements
persistence independence	yes	PF	
data-type orthogonality	yes	PF	
persistence completeness	yes	PF	
object identity	yes	PF and RT	
encapsulation	yes	PF and Lang	
inheritance, dynamic binding	yes and no	Lang	Compiler extension and additions to PF
complex state	yes	Lang	
computational completeness	yes	Lang	
integrity constraints	partial	Lang	Compiler extension and additions to PF

Table 4.2 – Persistence goals and P-Eiffel

Goal	Status	Provided by	Requirements
query mechanism	partial	PF	Compiler extension and additions to PF
programmable structure	yes	Lang	
arbitrary size	yes	FW and Lang	
permanence	yes	PF	
distribution	no		Additions to PF
access control	partial	PF	Additions to PF
administration	no		Additions to PF
sharing	yes	PF	Additions to PF
locking	no		Additions to PF
transactions	no		Additions to PF
object versioning	no		NA
schema evolution	no		NA

The table shows that the prototype P-Eiffel compiler already facilitates the production of programs that meet over half of the listed capabilities. The main item deserving attention is a robust query mechanism, which requires an enhanced compiler. A new compiler that can gather the information required for queries will likely also eliminate the shortcomings in enforcing integrity constraints and including inheritance structure within the repository. Chapter 5 discusses the requirement of an enhanced compiler in more detail. A new compiler with some additions to the persistence framework could transform the existing prototype P-Eiffel compiler into a production-quality compiler of great benefit to programmers.

4.6 Measuring effectiveness

If P-Eiffel is useful, it should help a programmer produce persistence-related programs that are better than and cheaper than programs produced without P-Eiffel. Measuring the quality of software, though, is difficult. Often, writers describe quality software with terms such as reliability, efficiency, and usability [71]; simplicity and expressiveness [82]; or robustness, extendibility, and compatibility [55]. However, measurements of these software qualities are very subjective. To present a more objective assessment of P-Eiffel, this research relies on criteria similar to the measurements presented by Grimstad et al. [33] in their evaluation of the usability aspects of PJama.

First, Grimstad et al. link the subjective qualities: maintainability (the measure of effort required to change code), understandability (the ease of code comprehension), and reusability (the ability to use the code in other applications) to objective measurements: lines of code (LOC), persistent explicit lines of code (PLOC), and number of persistence affected classes (PNOC[1]). They argue that writing fewer lines of code improves maintainability and eases code understandability and that high cohesion and low coupling improve maintainability, understandability, and reusability. If we accept their premise, then we can at least anecdotally explore the usefulness of P-Eiffel.[2]

Grimstad et al. use a measurement tool to count the number of lines of code (LOC), defined as productions rather than line-shifts. Instead, the metric tool for this research counts all lines of code, including comments, which presents no problem, because the comments remain almost the same in all versions of the test programs. Besides, good comments are an integral part of a well-written Eiffel program. Because the programs contain so few persistence-related features, and a relatively small number of persistence-related lines of code, hand counting persistent affected lines of code (PLOC), where a PLOC is defined as a line containing a call, declaration,

[1] Grimstad et al. actually abbreviated this metric as NOPC.

[2] Future work will test the hypothesis that automatic persistence makes persistence programming easier and less error prone by comparing P-Eiffel programs with their Eiffel counterparts.

or variable that directly relates to the persistence framework classes or features, is sufficient. The number of persistence-affected classes (PNOC) is any class that contains a PLOC. To obtain the number of classes (NOC) and PNOC, this study counts only the application's classes; it does not include persistence framework classes or supporting kernel classes. It also counts the total number of features (NOF) and the number of features that contain persistence code (PNOF). Again, these metrics only count the number of immediate features, those features defined in the application's classes, excluding inherited features.

This research evaluates three programs: 1) Supers, a more involved version of the Demo program that served as the example in section 2.4 ; 2) Flipper, an Othello game with a graphical user interface; and 3) Victory in the Pacific (VITP), a computer version of that 1970's board game.

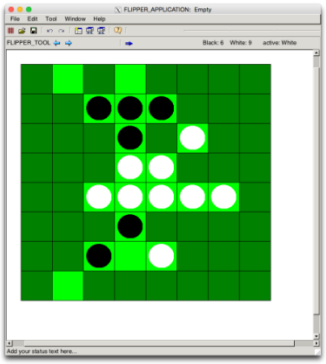

Figure 4.4 – Othello

Figure 4.5 – Victory in the Pacific

Program Supers performs the persistence actions described in Section 2.3 , displaying the results of each step in the terminal window. Parallel versions of the programs, one using serialization and one using database mapping, mimic the persistence version of their respective program as closely as possible. The parallel versions require additional features to emulate the dirty-marking and automatic-persistence features of the P-Eiffel versions. The serializable versions fail to maintain object identity. Flipper incorporates automatic persistence, saving the state of the current game after each player's move. VITP uses the intermediate level of persistence, marking objects as dirty, then checkpointing the changes when the player commits the changes. The VITP version using serialization saves the state of the entire game when the player commits his actions. I abandoned the database-mapping approach for VITP, because developing and testing a VITP database to accompany the application quickly became overwhelmingly difficult.

Figure 4.6 depicts the number of lines of code (LOC) for the Supers and Flipper programs, showing the lines of persistence-related code (PLOC) in red. The percentages at the top of the columns show the ratios of PLOC to LOC.

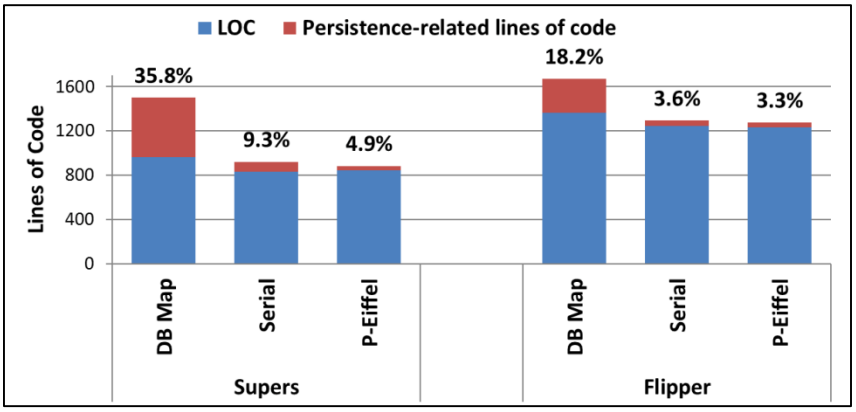

Figure 4.6 – Persistence-related lines of code

The serialized versions of the programs require slightly more persistence-related code than P-Eiffel versions. Much of the additional code in Supers emulates the marking-dirty and checkpointing features native to P-Eiffel.

The database-mapping versions require much more code than the serialized or P-Eiffel versions of the same program, mostly to build and access the database tables. The database in the Supers program must model the PERSON class and all its descendants, and account for multiple inheritance used by some of the program's classes. The database for Flipper, on the other hand, models only the GAME and DISK classes, so it does not require as much additional code. The amount of database-mapping code is dependent upon the number and complexity of the modeled classes. Increased complexity of the modeled classes also manifests as an increase in the number of persistence-related features (PNOF), as shown in Figure 4.7.

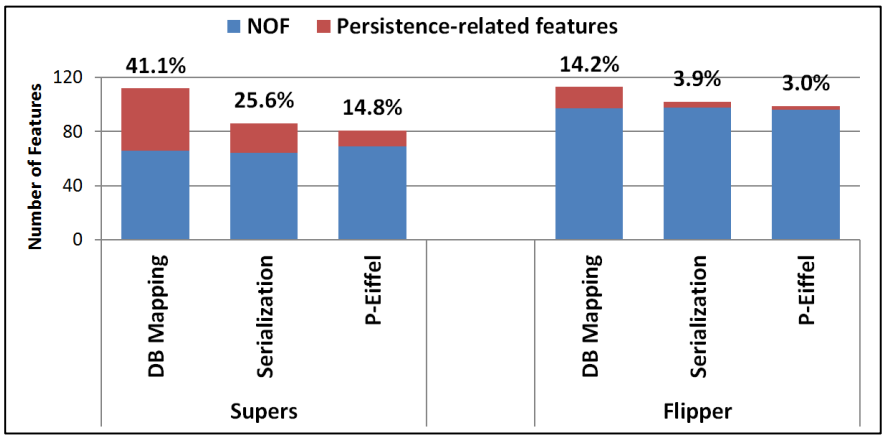

Figure 4.7 – Persistence-related features

The relationships between the classes in VITP were so complex that I abandoned the database-mapping version. Table 4.3 shows the metrics collected from the three programs, including class and feature counts as well as the number of lines of code.

Table 4.3 – Program metrics

		DB mapping	Serialization	P-Eiffel
Supers	PLOC / LOC	537 / 1499 = 35.82%	86 / 920 = 9.35%	43 / 884 = 4.86%
	PNOC / NOC	2 / 7 = 28.57%	1 / 7 = 14.30%	2 / 7 = 28.57%
	PNOF / NOF	46 / 112 = 41.07%	22 / 86 = 25.58%	12 / 81 = 14.81%
Flipper	PLOC / LOC	304 / 1670 = 18.20%	47 / 1292 = 3.64%	42 / 1275 = 3.29%
	PNOC / NOC	2 / 10 = 20.00%	1 / 10 = 10.00%	2 / 10 = 20.00%
	PNOF / NOF	16 / 113 = 14.16%	4 / 102 = 3.92%	3 / 99 = 3.03%
VITP	PLOC / LOC		56 / 34,115 = 0.16%	45 / 34,115 = 0.13%
	PNOC / NOC		1 / 155 = 0.65%	2 / 155 = 1.29%
	PNOF / NOF		5 / 1926 = 0.26%	4 / 1925 = 0.21%

The PLOC, PNOC, and PNOF measures do not increase as the size of the program increases. For P-Eiffel, the PLOC mainly represents initialization and loading. Storing operations require no additional code. These metrics show that adding P-Eiffel's persistence mechanism to an Eiffel system requires very little effort from the programmer.

4.7 Summary

Tests show that P-Eiffel's automatic persistence mode performs adequately, despite a small speed degradation and additional memory requirement when compared to a normal Eiffel program. The benefits of automatic persistence, though, offset these penalties. With P-Eiffel, a programmer adds persistence to a program using little additional code, which is restrictable to a small number of classes. The small overhead in lines of code and minimal impact on class coupling allow a programmer to easily add persistence to a system without affecting the system's maintainability, understandability, and reusability.

Chapter 5 Future research

The version of P-Eiffel described in this dissertation provides automatic persistence with very little programmer effort. Storing and retrieving objects with P-Eiffel is easier than with a serialization or database approach. P-Eiffel expands the usefulness of Eiffel with no change to the language. Though it has some memory and time overhead, it does show that automatic persistence in an object-oriented language is feasible and beneficial, and it lays the groundwork for future research into persistence. This continuing research should first fix some speed and memory inefficiencies of the framework classes. After improving performance, the research should branch into areas that have been too difficult or beyond the scope of this current research.

The early stages of P-Eiffel's development focused mainly on modifying the Eiffel compiler, but developing a new compiler required more manpower, expertise, and time than was available. As a compromise, P-Eiffel developed around modifying the runtime. This compromise prevents P-Eiffel from including class invariants in persistent type definitions and limits support of Eiffel's inheritance and type conformance mechanisms in the persistent data. A compiler that directly supports P-Eiffel's persistence mechanism would be a great improvement. P-Eiffel also needs improvement in its database functionality, such as security, transactions and locking, read-write error recovery, and data replication. This chapter looks back at some of the difficulties encountered during development. It then addresses some of P-Eiffel's inefficiencies, the inclusion of persistent invariants, a fully supported persistent type system, and database issues. It concludes by looking forward with hopeful optimism to P-Eiffel's future contributions.

5.1 Looking back

The beginning of the P-Eiffel project was characterized by enthusiasm, confidence, and ambition. Setbacks during development revealed that the project might also have had an abundance of overconfidence. At the start, I believed that I could easily modify Eiffel Software's compiler. After many painstaking hours, I realized that I lacked the expertise required to learn the code layout and modify it without breaking other aspects of the compiler. The code for the compiler and EiffelStudio, the Eiffel GUI integrated environment, is publicly available, but there is little documentation describing the code's logic. My lack of understanding of the compiler's implementation and its tight coupling with EiffelStudio and the runtime led to false starts.

The attempted compiler modification sought to modify the code it generates, so that it would inject at appropriate points in the abstract syntax tree new nodes that represent calls to persistence-related features. In order to give the compiler access to these new features, I modified class ANY, adding new attributes such as persistence_id and is_dirty. After many hours of debugging, I realized that the problem was not with the modified compiler, but with the new ANY class. The original text of class ANY states that the class "may be customized for individual projects or teams", but it failed to give details about restrictions placed on these modifications. Only after many hours wasted debugging code did I discover that the Eiffel compiler does not allow attribute additions.

After realizing that attributes could not be added to every class via class ANY, I took a different tack, attempting to preprocess a system of classes to inject the desired functionality by inserting new feature calls into the class texts. Though tedious, this approach could explore all the classes in a system and somewhat parse the code, but problems surfaced when a class text ventured beyond basic Eiffel structure. For example, the preprocessor successfully handles a feature call such as the following.

```
my_object.do_something
```

However, the preprocessor could not understand more sophisticated constructs:

```
my_object.do_something (a_object.get_object)
my_object.get_object.other_object
```

Feature calls nested inside a parameter list and chained feature calls proved too problematic to reliably intercept. As these constructs are not uncommon, a new approach became necessary. The new approach, a modified runtime, allowed the construction of the prototype version of P-Eiffel described in this thesis.

5.2 Correct inefficiencies

The current version of P-Eiffel abstracts the 64-bit persistence identifier with class PID. The PID class permeates the persistence framework, so there are many instances of this type in a P-Eiffel program, and each PID object incurs the memory overhead associated with complex objects. This overhead consists of the P-Eiffel object header, which itself contains an unused 8-byte persistence identifier, plus padding. As a result, every PID requires 64 bytes of memory just to wrap an 8-byte identifier. Removing the PID class and replacing that type with a NATURAL_64 type should save quite a bit of memory and may give a slight speed improvement. Unfortunately, feature signatures would then become less descriptive. For example, the signature of the feature loaded from PERSISTENCE_MANAGER informs the programmer that the argument is a persistence identifier.

```
loaded (a_pid: PID): ANY
```

The new form is a little less informative.

```
loaded (a_pid: NATURAL_64): ANY
```

Class PID has been very helpful during development, providing a very specific type for contexts that use a persistence identifier, making the implementation more readable and helping in debugging. Now that P-Eiffel is operational, this readability can be sacrificed for a substantial reduction in memory use.

P-Eiffel's memory usage also suffers in another way. Feature Identified_objects from class PERSISTENCE_FACILITIES keeps track of persistable objects, pairing each of them with its persistence identifier for quick lookup of an object given an identifier. Unfortunately, once P-Eiffel places a reference to a persistable object into the Identified_objects table, the garbage collector never reclaims the memory used by that object, even if the referenced object otherwise goes out of scope. Because a reference to the object still exists in the table, the garbage collector can never free that memory even if the program never uses that object again. The current arrangement defeats the Eiffel garbage collector, at least in spirit. A future version of P-Eiffel should address this issue, perhaps by using class WEAK_REFERENCE or by tying directly into the Eiffel garbage collector itself.

Future research must also investigate garbage collection of persistent objects in the REPOSITORY. At one point during execution of the example system of Section 2.3 , the program changes the name of one of the PERSON objects after that person and his name has been written to the repository. From the program's perspective, the old name is unreachable from any root object, and the persistent memory should be freed to prevent the accumulation of persistent garbage. A less myopic view reveals that persistent garbage collection is not so straightforward. Some other program, currently running or not, might need that object. In that case, deleting the object in question is wrong. The repository could track all the programs or users that refer to particular objects, but that solution just pushes the problem back. How long should a repository maintain a persistent object if that object is unreferenced for a long time? The version of P-Eiffel described in this dissertation does not yet implement the repository's garbage collector, and perhaps this solution is best. After all, long-term storage is cheap and, by definition, a persistent object never goes out of scope. Nevertheless, for completeness, future versions of P-Eiffel should explore persistent garbage collection, because automatic garbage collection is a very important feature of Eiffel.

5.3 Include persistent invariants

Another important feature of Eiffel is its built-in support for assertions, particularly class invariants, which express "consistency constraints applicable to all instances of a class." [16] Class

invariants, and the accumulation of invariants in descendants, serve to define the correctness of classes and help improve reliability of software. Invariants describe properties that must hold for every instance of that class. An assertion violation during execution does not represent an exception from which the system should recover, but an invalid state that should never exist; it indicates an error in the software. A correct program, then, should contain only consistent objects. P-Eiffel takes this view of assertions, storing an object only when it satisfies the object's assertions. Unfortunately, the current version of P-Eiffel cannot enforce consistency constraints between different programs retrieving the same object, because P-Eiffel lacks the ability to discover invariants during program execution and hence is unable to include the invariants in the repository.

Meyer considers the inclusion of attribute names and types along with objects to be the minimum amount of information necessary for a reasonable approach to persistence but concedes that the inclusion of the class invariant would be a better policy. While describing requirements for schema evolution, he suggests four possible levels of information that a persistence mechanism could store in a class descriptor to capture object types. His description of the four levels follows:

> C1 • At one extreme, the class descriptor could just be the class name. This is generally insufficient: if the generator of an object in the storing system has the same name as a class in the retrieving system, we will accept the object even though the two classes may be totally incompatible. Trouble will inevitably follow.
>
> C2 • At the other extreme, we might use as class descriptor the entire class text — perhaps not as a string but in an appropriate internal form (abstract syntax tree). This is clearly the worst solution for efficiency, both in space occupation and in descriptor comparison time. But it may not even be right for reliability, since some class changes are harmless. Assume for example the new class text has added a routine, but has not changed any attribute or invariant clause. Then nothing bad can happen if we consider a retrieved object up-to-date; but if we detect an object mismatch we may cause some unwarranted trouble (such as an exception) in the retrieving system.
>
> C3 • A more realistic approach is to make the class descriptor include the class name and the list of its attributes, each characterized by its name and its type. As compared to the nominal approach, there is still the risk that two completely different classes might have both the same name and the same attributes, but (unlike in case C1) such chance clashes are extremely unlikely to happen in practice.
>
> C4 • A variation on C3 would include not just the attribute list but also the whole class invariant. With the invariant, you should be assured that the addition or removal of a routine, which will not yield a detected object mismatch, is harmless, since if it changed the semantics of the class it would affect the invariant. [56]

P-Eiffel currently provides level C3. It builds a persistent type representation through reflection, discovering the class name, attribute names, and attribute types when it stores an object. P-Eiffel avoids the space and time overheads mentioned by Meyer, because it computes a digest to represent type information. Because Eiffel's reflection mechanism does not yet provide access to invariant definitions, P-Eiffel has no way to discover invariants at runtime. Runtime invariant discovery, which would improve P-Eiffel's ability to enforce object consistency, requires a new compiler. Not only could a new compiler discover invariants, it could also build the type information that P-Eiffel currently finds through reflection at runtime, improving P-Eiffel's efficiency.

5.4 Allow queries with an improved persistent type system

Besides improving persistent object consistency checking, a new compiler could allow P-Eiffel to fully support Eiffel's type system in the repository. The current implementation discovers type information through reflection on an object when it is stored. The resulting type information, though adequate for P-Eiffel to recreate any stored object, does not contain complete information about an object's lineage. P-Eiffel only knows the generating type of each stored object, and it knows the persisted types to which an object conforms. P-Eiffel does not account for feature, specifically attribute, renaming. P-Eiffel's rudimentary query mechanism allows queries based on object types but is incomplete for queries about specific attributes.

P-Eiffel has enough persistent type information to perform queries based on object conformance. In the context of the example program from previous chapters, the query, "Return all HERO objects" works as expected. A test program using the prototype P-Eiffel implementation retrieves all persisted HERO objects using the following code.

```
load_heros: LINKED_LIST [ANY]
        -- Load all the stored {HERO} objects.
    local
        pt: PERSISTENT_TYPE
    do
        pt := persistent_type_from_dynamic_type (({HERO}).type_id)
        Result := Persistence_manager.loaded_by_type (pt)
    end
```

After obtaining the PERSISTENT_TYPE corresponding to the run-time dynamic type of HERO, the code queries the Persistence_manager for a list of objects of that type. Calling this query at the end of the example system described in 2.3 returns chewie, han, batman, and robin, because the types of these objects conform to type HERO.

Though not yet implemented, it should be straightforward to implement in the repository the ability for P-Eiffel to answer queries such as, "Return all SUPERHERO objects that have a sidekick." The repository explores the type information and returns the representation of those objects that have a non-void entry in the representation of the sidekick attribute. But queries involving attributes are limited, because P-Eiffel is unable to account for feature renaming. A query such as, "Return all SUPERHERO objects that have a companion" would fail to find batman, because the SUPERHERO class renames the companion attribute inherited from HERO as sidekick. Because the persistent type system in the repository lacks information about the renamed attribute, the current implementation of P-Eiffel fails for such queries.

Accounting for attribute renaming and redefinition is problematic. While it may be possible to determine with reasonable probability that an attribute from an ancestor is renamed or redefined in a descendant type and to even infer the original name and type, there seems to be no way to positively identify all renamed or redefined attribute information at runtime. In the long run, a compiler built specifically to gather the persistent type information is necessary.

5.5 Add other database functionality

While work on a persistence-aware compiler proceeds, another branch of research should focus on extending the existing persistence framework to incorporate functions normally performed by a database. In order for P-Eiffel (or any other persistent programming language) to reach production status, it must robustly incorporate security, data replication and durability, and safe object sharing through transactions or locking.

P-Eiffel does not yet incorporate security, but the username and password features of class CREDENTIALS, unused in the example programs but required for connection to a repository, could provide authentication and access privilege enforcement. When a program initially creates a repository, P-Eiffel could record salted and hashed username and password values in the repository, setting the owner to this initial creator. Subsequent connections to that repository would verify

the username and password. Furthermore, envisioned NETWORK_REPOSITORY features could allow the repository owner to set the access mode to something other than the default read-and-write mode in which any user can modify objects in the repository. A private setting would allow only the owner to access the repository; a read-only setting would allow other users to access objects from the repository while allowing only the owner to make changes to those objects. P-Eiffel could implement the private setting in the framework code that connects a user to the repository; it could implement the read-only setting at the point of the persistence calls, allowing the program to run normally while avoiding calls that write objects to the repository. This multiple-tiered data access scheme implemented in the existing framework or runtime would enhance P-Eiffel's data security.

P-Eiffel's data security could be enhanced further by adding encryption. Simple data encryption of a repository would prevent programs other than P-Eiffel programs specifically designed for that repository's data types from viewing or modifying the contents. Encryption functions could be added at the point of data persistence and loading. The choice to encrypt or not could be controlled by the programmer with calls to features to be added to the REPOSITORY class. The speed and memory costs that these authentication and encryption features would add is unknown, but they would enhance P-Eiffel's usefulness.

Replication and durability of persisted data might also be useful. Google's Cloud Storage Service [22] and Amazon's Simple Storage Service [4] provide various levels of data access, security, disaster recovery, and backup. These functions could be added to P-Eiffel in a descendant of the REPOSITORY class built using the API of one of these services. On the other hand, a REPOSITORY for object replication and distribution built specifically for P-Eiffel might provide faster access to persistent objects. Because P-Eiffel's persistence mechanism, at least in full_automatic mode, persists objects after every qualified feature call, the amount of data transferred is usually quite small. This fine granularity provides a checkpoint of sorts for an object after every persist operation. The class invariant ensures object consistency at the time of the write. Combining this consistency checking with logging and repository mirroring could provide a stable and reliable persistence mechanism and make P-Eiffel much more useful.

For P-Eiffel to become unquestionably useful, one more, perhaps the most important, database-like capability must be added—the capability for many programs to concurrently and safely access the same repository. Using a common example, the following code segment illustrates how concurrent feature calls might violate object consistency even when preconditions are present. In this typical scenario, the transfer_off feature from class ACCOUNT attempts to move money from the current account into some other account. The precondition requires that the originating account starts with enough money to accommodate the transfer.

```
transfer_out (a_amount: INTEGER; a_other: ACCOUNT)
    require
        has_sufficient_funds: balance >= a_amount
    do
        withdraw (a_amount)
        a_other.deposit (a_amount)
    end
```

In a concurrent environment, between verification of the precondition and execution of the call to withdraw, a second program can deplete the account, invalidating the call. P-Eiffel must protect against this type of consistency violation while still allowing concurrent access to objects. It might be possible for P-Eiffel to provide object sharing through transactions with locking [29, 83] or with synchronization through semaphores or even with conditional critical regions [61], but both methods seem difficult, because they both require a sophisticated concurrency management system on top of the persistence mechanism. Instead, I hope that P-Eiffel can leverage Eiffel's built-in Simple Concurrent Object-Oriented Programming (SCOOP) mechanism.

SCOOP addresses synchronization issues such as race conditions, atomicity violations, and deadlocks. Using SCOOP, a programmer does not have to deal with semaphores, mutexes,

or critical regions. A SCOOP-enabled program has one additional reserved word and only slightly modified contract semantics. Though SCOOP is currently implemented using threads, it is progressing toward wider implementation strategies. The SCOOP developers envision an environment that handles simultaneous program executions not just with separate processors in the same CPU but by widely diverse computational engines, including separate networked computers [24].

Regardless of the implementation, simultaneous repository access by multiple programs requires some method to notify clients that a particular object in the repository was changed and a way for the client programs to react to these changes. This functionality could be achieved with the addition of a few features to the PSERVER and REPOSITORY class. The PSERVER could track each client that is currently accessing it and the objects in which each client is interested. When a client changes one or more objects, the server would send a list of persistent identifiers to interested clients. Upon receipt of a change, each client that requires an update to any of those objects would send a normal request for objects back to the server. A client would initially express its desire for such notifications by subscribing to them with the server. This subscription, notification, and callback functionality could be added to the framework implementation without changing the current programmer interface.

Safe and concurrent object sharing coupled with a subscription-notification-callback scheme would be a welcome upgrade to P-Eiffel's networking capabilities. Providing this upgrade while maintaining programming simplicity would make P-Eiffel production-ready and provide a platform for long-term testing of Atkinson's orthogonal persistence hypothesis.

5.6 Looking ahead

Malcolm Atkinson, likely the first advocate for persistence in programming languages, developed his Orthogonal Persistence Hypothesis (OPH) during his early studies of PS-Algol [7] and stated it more formally while describing the outcome of persistence studies using PJama [5], a language based on early versions of Java intended for long-term research into persistence. His hypothesis reads:

> If application developers are provided with a well-implemented and well-supported orthogonally persistent programming platform, then a significant increase in developer productivity will ensue and operational performance will be satisfactory. [5]

This hypothesis states what programmers intuitively suspect: less code is better. This intuition has inspired many attempts at orthogonal persistence. Some of the languages in which programmers have attempted to produce a viable orthogonal persistence mechanism include[1] Ada [25, 48, 65], C [46], C++ [18, 32, 41, 43, 50], Objective C , E [78, 79, 81], Eiffel [18, 21, 26], Java [11, 53, 63, 89], JavaScript [19], Lisp [46], Napier [9], Oberon [12], PM3 [36, 38], PS-Algol [7, 14], and Smalltalk [39]. None of these approaches seem to have lived beyond the initial research or achieved use in production systems.

In "Persistence and Java—A Balancing Act" [5], Atkinson describes the mixed success and limited influence that the PJama project has had on industry's adoption of a persistent languages. The PJama project was perhaps the most extensive study of orthogonal persistence, and it achieved most of the goals set for this research. However, despite access to a team of programmers, the use of a popular base language, and sufficient resources, PJama was never adopted by a sufficient number of users to prove the OPH. Atkinson attributes this failure, in hindsight, to several reasons, including improper technical decisions that forced the project to chase Java's

[1] This list is for illustration only and not all inclusive. Apologies to authors who have contributed to the study of persistence but are not listed here.

rapidly changing implementation and the availability to industry of other, more visibly attractive but short-range solutions to persistence, such as automated code generation and relational database technology.

In contrast, P-Eiffel builds on a very stable language. Though some may consider Eiffel obscure, it is very much alive and continues to influence development in other languages. For example, authors acknowledge the specification, design, documentation, and testing benefits of Eiffel's built-in assertion support [16, 42, 67, 86] or attempt to embed such support in other languages [3, 20, 34, 68, 80]. Furthermore, universities such as ETH Zurich continue to use Eiffel for research, some of which seems aimed at schema evolution [75]. Schema evolution and persistence are closely related, so this research and the research at ETH should benefit each other, making P-Eiffel more attractive to other users. Finally, programmers have expressed interest in persistence in Eiffel in newsgroup messages [92, 93, 97] and in Eiffel Software's web-based Eiffel persistence project [69]. Language stability, ongoing research, and interest in persistence has the potential to move P-Eiffel to a prominent position in the Eiffel community. If programmers then begin to use P-Eiffel, the usefulness of orthogonal persistence in programming languages should become evident, encouraging renewed interest in its applicability to other languages.

The approach to orthogonal persistence presented in this thesis should be workable in other object-oriented languages, but the ease of implementation in the other language will depend on the complexity of the language, the language's introspection capabilities, and upon the availability of a modifiable compiler or runtime. P-Eiffel's persistence semantics rely upon Eiffel's unambiguous qualified feature call and assignment statement constructs—an object becomes dirty when an attribute is changed through assignment and it is stored when a qualified feature call on that object exits. These semantics are easy to define, because 1) Eiffel confines attribute assignments to the body of a feature within the class, and 2) operations on objects occur only through feature calls. P-Eiffel also relies heavily upon Eiffel's introspection ability, which provides P-Eiffel with class names and gives it attribute types and names at runtime. Furthermore, P-Eiffel would not exist if not for the open-source availability of the Eiffel runtime and compiler.

Mimicking P-Eiffel's persistence mechanism in other languages may require more work. For example, Java, C++, and C# allow an assignment to an attribute from outside the context of the enclosing object.

```
my_object.some_attribute = a_value;
```

Following P-Eiffel's semantics, this code would be interpreted as an assignment to an attribute contained within my_object, which marks my_object as dirty. But it is unclear if my_object should be persisted immediately after this line of code, or if the persistence mechanism should wait, hoping for a normal, unambiguous feature call to occur later in the code. Another example of a problematic construct is the use of friend functions and procedural methods in C++. Because these calls generally operate outside the context of any enclosing object, it might be hard to determine which object to mark as dirty or to persist. These languages also give arrays special treatment, unlike Eiffel, in which an **ARRAY** is a class subject to normal feature call semantics. This special treatment would likely require special treatment in the persistence mechanism. Also, C++ and C# have very limited introspection capability, likely making P-Eiffel's approach to object persistence less applicable in these languages. On the other hand, Java's reflection API seems very capable, so implementation of object traversal and tabulation might be easier than in Eiffel. Regardless of the language, the addition of orthogonal persistence would probably require access and modification of the language's compiler or runtime.

Chapter 6 Conclusion

Even if P-Eiffel never moves the orthogonal persistence hypothesis to theorem status, it does provide a new service to Eiffel. It adds orthogonal persistence to the language, filling the gap between the all-or-nothing, serialization approach of STORABLE and the overly complex database-mapping approach of the EiffelStore library. The new mechanism fits into Eiffel with minimal impact on the language itself. Persistence in P-Eiffel, accessible through the interfaces of easy-to-use classes, is mostly automatic, requiring very little explicit persistence-related code. The persistence implementation automates persistent object identification, physical storage, persistent object access, and persistent-object type checking. P-Eiffel extends the strong type system of Eiffel to the objects in the persistent store and leverages Eiffel's built-in assertion mechanism to ensure persistent-object consistency. A basic persistence mechanism with an easy-to-use programmer interface paves the way for the development of other database-like functions such as concurrent access, security, distribution, and replication. Research using P-Eiffel should continue, hopefully leading to advancements in the study of persistence in other object-oriented languages, such as Java. Time will tell if P-Eiffel will persist.

Appendix A Review of persistent systems

The development of database systems and persistent programming languages has proceeded from differing viewpoints; programming languages have focused on computation and database systems have focused on data storage. Over the years, developers of database systems have added more and more computation capabilities while developers of persistent programming languages have attempted to add long-term data storage to system languages.

Built-in persistent mechanisms over the years have met with limited success. The successful languages were those that have aspired to achieve the goals mentioned in Section 4.5 while presenting a simple interface to the programmer. This simple interface usually hides a very complex storage or memory manager and/or a modified compiler that handled the database-like functionality. This separation of concerns into language issues and data management appears in the programmer interface of persistent languages reviewed below. A review of a few of these languages illustrates some desirable (and perhaps undesirable) characteristics and techniques for a persistent programming language.

A.1 PS-Algol (1982)

PS-Algol [7, 10] is one of the first languages to add persistence mechanisms directly to an existing language, extending S-Algol (1979) [58, 59] and introducing orthogonal persistence. For persistence, PS-Algol adds two sets of procedures, one set for managing persistence (i.e. accessing a database and performing transactions) and another set to manipulate tables for associative lookup (i.e. hash-table procedures.)

S-Algol creates complex objects on the heap and subjects them to garbage collection. PS-Algol extends this heap model to the persistence mechanism by adding new predeclared procedures and types. The new procedures provide methods for creating and opening a database and for accessing objects stored in the database. The compiler and language remains unchanged, but a modified runtime extends the dynamic type checking of S-Algol. PS-Algol adds a persistent object manager to the runtime to lazily[1] follow references using a persistent object identifier (PID) for each object. Following the run-time model of S-Algol, PS-Algol performs run-time, structural type checking on objects as they are loaded from the persistent store. To allow this checking, the persistent store holds fieldnames and field types for all the fields associated with each object.

As a PS-Algol program accesses or creates an object, the new runtime adds two mappings, one from the object's local address to a PID and another from a PID to the object's local address. These mappings prevent duplicate disk reads, because as PS-Algol dereferences a pointer, it consults the table to determine if that PID has already been associated with a local object. To avoid unnecessary writes, a commit routine writes back out changed or newly created objects only. The authors say these procedures give reasonable performance.

The PS-Algol code fragment in Figure 6.1 inserts a new person into a persistent store.[2] The PS-Algol persistence procedures used in the example are open.database, error.record, s.lookup, s.enter, and commit. (The dot notation used in procedure names and variables is a naming convention of PS-Algol and should not be confused with the dot notation used in object-oriented languages. The identifier **read.a.line**, for example, is the name of a routine that reads input from the terminal.)

---

[1] In other words, the runtime copies the objects to its heap only when a pointer is dereferenced. This is in contrast to a program that loads all persistent objects when it loads a root object.

[2] The code examples in this dissertation loosely follow a company-with-employees motif, which is similar to the superheroes example described in section 1.2 and depicted in the object structure of Figure 1.4.

```
structure person (string name, index ; pntr addr, other)
structure address (int no ; string street, town ; pntr next.addr)
let db = open.database ("Employees.list", "my-password", "write")
if db is error.record then write "Can't open database" else
begin
    write "Name: "          ; let this.name = read.a.line
    write "Index: "         ; let this.index = read.a.line
    write "House num: "     ; let this.house = readi
    write "Street: "        ; let this.street = read.a.line
    write "Town: "          ; let this.town = read.a.line
    let p = person (this.name, this.index, address (this.house, this.street, this.town, nil), nil)
    let addr.list = s.lookup ("addr.list.by.name", db)
    s.enter (this.name, addr.list, p)
    commit
end
```

Figure 6.1 – Add a new person to a persistent store in PS-Algol

After defining two structures, the above code attempts to open a persistent store named Employees.list in write mode. If successful, it creates a new person object from input gathered from the user. The code inserts that object, indexed by the object's name, into a table called addr.list.by.name. Finally, commit stores the changed table, the person object, and the address referenced inside the person object to the persistent store.

The example illustrates Atkinson's principles of persistence. PS-Algol achieves persistence independence. PS-Algol uses the same syntax used by S-Algol, and the added persistence routines appear as normal routine calls. PS-Algol is data-type orthogonal, because any created object can become persistent. PS-Algol obtains transitivity through persistence-by-reachability.

A.2 Galileo (1995)

As with PS-Algol, which extends S-Algol to add persistence features, Galileo [1, 2] extends Edinburgh ML to add persistence features. Galileo is a strongly and statically typed functional language. A program can only change attributes declared as modifiable. Galileo supports abstract types, type hierarchies, information hiding, and exception handling. It also includes a built-in assertion mechanism to restrict the domains of attributes. In contrast to PS-Algol, Galileo seems to have been more influenced by the needs of database programming than that of systems programming. Galileo code is reminiscent of SQL. The code in Figure 6.2 models a company that has departments and employees, and Figure 6.3 illustrates the model's use.

```
use
Company :=
(rec Departments class
    Department ↔
        (Name: string
        and Manager: var Person
        and Budget: num
        key: (Name)
and Employees class
    Person ↔
        (Name: string
        and Index: num
        and Salary: num
        and Dept: var Department
        key (Index));
```

Figure 6.2 – Galileo structures

For persistence, the language relies on the concept of environments. In the code above, **use** adds a new environment called Company to the global environment. The Company environment

contains two classes[1], Departments and Employees, into which it places objects of type Department and Person. A Department object has attributes Name, Manager, and Budget of the declared types. The program similarly defines a Person. Each class contains a **key** constraint that requires each Index attribute of the contained objects to differ. Attributes marked with **var** can change; if **var** is absent the attribute cannot change once set. A top-level, global environment is available and automatically managed and stored by the runtime. Because Company is part of this global environment, the runtime automatically stores it, its two classes, and the objects contained in those classes.

A programmer interacts with objects through a set of graphical primitives or associated operators during an interactive session. The graphical interface and the associated operators provide a query language for a Galileo persistent store. Figure 6.3 shows an interactive session using the previously defined classes.

```
enter Company:

mkPerson
    (Name := "John Galt"
    and Salary := 100
    and Index := 5
    and Dept := get Departments with Name = "Research");

for x in Employees
    with Salary of x
        < avg (for y in Employees
                with at Dept of x = at Dept of y
            do Salary of y)
do Name of x;
```

Figure 6.3 – Galileo object insertion and query

This example code adds an employee to the research department using an automatically generated operation, mkPerson. The second query, beginning with **for** x **in**…, asks for a list of names for all employees in the same department with a salary less than the average salary of the employees in that department. Galileo does not require the programmer who defines the structures or the user who interacts with the program to issue explicit commands to store or retrieve objects; persistence is transparent and automatic. The **enter** command, used to make Company the current environment, may be considered the closest Galileo comes to having an open-database command. Any type of object used by the Galileo system can become persistent. In this respect, Galileo provides orthogonal persistence. Unfortunately, concurrency and sharing of objects is not possible.

A.3 Napier88 (1988)

Napier88 [15, 60, 62] is as a proof-of-concept language with an integrated persistent programming environment. It allows parallel execution and provides facilities for schema evolution.

Like Galileo, Napier88 hides object persistence from the programmer using environments arranged as a tree. The predefined procedure PS provides access to the user's persistent root environment, into which the user places new objects (other environments or user-defined objects.) A retrieve operation attempts to **project** (type cast) a stored object onto an entity, failing if the retrieved object and the entity's definition are not structurally equivalent. The code in Figure 6.4, defines a **structure** called person containing the attributes name, index, and extra. This example assumes that the persistent store contains one previously persisted object that matches the person

[1] Use of the term **class** in Galileo differs from its use in object-oriented languages. A class in Galileo provides a container within the persistent environment and defines the structure of the objects that container can hold.

structure. The program loads the object and updates some attributes using the attributes' names as indexes.

```
type person is structure (name: string; index: int; extra: any)

let ps = PS()

project ps as X onto
person:
    begin
        X(name) := "John Galt"
        X(index) := "5"
        let this = X (extra)
        type extraInfo is record (sal: int; gender: bool)
        project this as Y onto
            extraInfo: write Y (sal)                    ! Output the person's salary
        default: …
    end
default:    {}
```

Figure 6.4 – Napier88

The point of this simple example is not to explain Napier88 in any detail but to illustrate orthogonal persistence in Napier88. Obtaining a reference to the persistent root is the only action required for storing objects. All objects reachable from that root are retrievable, and changes, such as the changes made to the name and index attributes, are stored when the program ends. The programmer does not write explicit code to store the objects.

Figure 6.4 also illustrates Napier88's dynamic type checking. The PS function returns a reference to an object of type **any**. When it is projected (type cast) onto X, the object is dynamically type checked to ensure the object matches X's type, that is, person. The runtime type checks attribute extra only when the program projects that object onto the extraInfo type at the second **project** operation. Programs that do not use the extra attribute need not declare the extraInfo type or perform the second projection.

Napier88 provides parallel execution through the **process** type, which can establish an Ada-like rendezvous [60].[1] A process object, just as all other objects, can be stored. Napier88 adheres to the three principles of persistence.

Finally, Nappier88 allows schema evolution. Because it type checks an object only when a program projects an object onto a type, modifications to unused types do not necessarily affect all programs that use objects based on the changed types. For example, if a programmer changes the extraInfo type, he must change only those programs that reference that type. Programs that use only the name and index attributes can retrieve the person object with no change to their code. This dynamic type checking allows incremental modification of an object's structure.

A.4 E (1989)

The E programming language [78, 79, 81], originally designed as a language for implementing database systems, has evolved into the first C++ extension to support persistence. Figure 6.5 shows the use of E for the same purpose as that given for Galileo above. The example is not intended to be complete (e.g. there are no constructors or member functions), but it should suffice to present the flavor of E as experienced by the programmer.

[1] Morrison [60] shows a Napier88 solution to Dijkstra's dining philosophers [35] that uses the **process** type.

```
dbclass Department {
    public:
        dbchar* name;
        Person* manager;
        dbint num;
};
dbclass Person {
    public:
        dbchar* name;
        dbint index;
        dbint salary;
        Department* department;
};
dbclass Departments: collection [Department];
dbclass Employees: collection [Person];

persistent dbstruct company {
    Departments departments;
    Employees employees;
};

main() {
    Employee* e1 = new (personnel.employees) Employee ("John Galt");
};
```

Figure 6.5 – E code

The programmer declares objects as potentially persistent using the **dbxxx** types, which mirror the standard C++ types. The **persistent** declaration of company causes that object to survive across all runs of the program. As descendants of the generic[1] **collection** class, Employees and Departments inherit an overloaded **new** operator, allowing the dynamic creation of Person and Department objects. The code creates a new Person object, with name "John Galt", within the employees collection. The entity, e1, references the newly created object. Because the employees collection is a member of a persistent structure, it and any object created within it are also persistent. The runtime automatically stores e1 when it creates the object. There are no calls to file routines or database-like functions such as open or read. With the exception of the new types, the code looks almost like standard C++. To complete the example, the query code seen in the Galileo example would be coded using normal C++ methods.[2]

The persistence mechanism used by E is different from the persistence-by-reachability seen in PS-Algol, Napier, and Galileo. E uses allocation-based persistence, which determines object persistence based on the creation method applied to the object, either statically with a **persistent** declaration or dynamically in a **collection**. The E runtime and programming environments use the EXODUS Storage Manager behind the scenes to take care of the actual physical placement of objects. Compiling an E source module that contains declarations of persistent objects produces both a C translation and a storage manager file, the persistent store, containing those persistent objects. The compiler binds the names of the persistent variables to the physical objects at compile time. This binding links the variable names in a module to a location in a physical file maintained by the Storage Manager and initially identified with an environment variable, EVOLUME. Because the compiler allocates memory for persistent objects based on the declarations in the code, subsequent deletions of the declarations leave unreferenced objects in the persistent store. To prevent this accumulation of persistent garbage, compilation and execution of E

[1] A generic class such as collection [T] is called a generator in E [79] and predated C++ templates.
[2] E also introduced CLU-like iterators to C++ to loop over the elements in a collection by calling a resumable iterator function that yields a result on each step through the loop.

programs must occur within a special environment that tracks dependencies between programs and persistent objects.

The E runtime calls routines that move objects to and from the persistent store.[1] The Storage Manager schedules the physical reads and writes and provides atomic, recoverable transactions with two-phase locking. An E compiler or interpreter injects the calls to store and retrieve objects and adds the code to mark objects as clean or dirty into the C code during E code preprocessing.

A.5 PM3 (1991)

PM3 [37, 38] adds orthogonal persistence, as defined previously by Atkinson and Morrison in 1985 to Modula-3 while incurring negligible performance costs. A modified runtime and two new library interfaces, Database and Transaction, adds persistence to the language. The example code in Figure 6.6 shows the use of the two interfaces. The code obtains a reference to a root object[2] from a previously created, named database and modifies one of the previously stored, reachable objects within a transaction. The call to Transaction.commit() ends the transaction and stores the object.

```
MODULE Company;
    IMPORT Text;
    IMPORT Database;
    IMPORT Transaction;
    CONST n = 30;
    TYPE
        Person = RECORD
            name: TEXT;
            salary: INTEGER;
        END Person;
        EmployeeList = OBJECT (* … *) END EmployeeList;
    VAR
        Employees: EmployeeList;
PROCEDURE SetSalary (aName: TEXT; aValue: INTEGER);
    VAR
        p: Person;              (* a reference type *)
    BEGIN
        Database.open ("aDatabaseName");
        Employees := Database.getRoot();
        Transaction.begin();
            (* assume a procedure to search Employees exists *)
        p := FindEmployeeByName (aName);
        p.salary := aValue;
        Transaction.commit();
    END SetSal;
END Personnel;
```

Figure 6.6 – PM3

A modified[3] compiler ties persistence to the Modula-3 garbage collector and allows the runtime to intercept calls to the operating system's virtual memory primitives. If a system call results in a memory fault, in the case of a read, the persistence mechanism allocates and maps

---

[1] Richardson et al call these data-movement operations pin and release.

[2] The statement assigning the root object to Employees triggers type checking. The runtime raises an exception if the obtained object is not structurally equivalent to the definition of an EmployeeList object.

[3] The authors say the Modula-3 compiler was not changed. They added the new Modula-3 compilation process as a front end to the GNU C compiler. This preprocessor produces C output, sending the output to the GNU C compiler.

new memory and reads in the missing object. During writes to memory as an object changes, the mechanism marks as dirty the page containing the changed object, so the page can be stored, if required, during a commit. The mechanism only stores pages that contain persistent objects. It discovers persistent objects through reachability analysis and copies newly persistent objects into persistent pages, mapping the pages to the virtual address space associated with the root object's persistent store. The commit operation, when called, stores any dirty, persistent pages.

Likewise, PM3 retrieves objects from a persistent store on demand. As PM3 discovers references to objects, it maps the references to pages in volatile memory. When the runtime references an object on a mapped but non-resident page, the runtime traps and reads in the required page from the persistent store, placing the page into volatile memory. The programmer needs only to request a root object from the persistent store; PM3 automatically retrieves the pages containing reachable objects as it accesses those objects.

A.6 PHP (1995)

Unlike the above languages, PHP [73, 96] is not a persistent programming language; it is a scripting language usually embedded within HTML and used primarily for web development. Nevertheless, PHP deserves a review along with persistent languages because of its popularity and the ease with which it interfaces with databases. It also provides an example of a persistence mechanism that is contrary to the orthogonal principles.

Internet documents seem to attribute PHP's popularity to its C-like syntax, extendibility, and availability on most operating systems. Netcraft, a company that provides web server and hosting analysis, claims that as of January 2013, 244 million sites, 39% of the sites surveyed, run PHP [72]. PHP's ability to access numerous types of databases also contributes to its popularity. The following code connects to a relational database, runs a query, and formats the result in HTML.

```
<?php
$dbhost = 'localhost:8889';
$dbuser = 'root';
$dbpass = 'root';
$dbname = 'Company';
        // Connect to the database
$connection = mysql_connect ($dbhost, $dbuser, $dbpass);
mysql_select_db ($dbname);
        // Define the query
$underpaid_employees_query =
    "SELECT e1.name, e1.salary, e1.department " .
    "FROM Employees AS e1, " .
    "((SELECT e2.department, AVG (salary) AS avg FROM Employees as e2 " .
    "GROUP BY e2.department) as e3) " .
    "WHERE e1.department = e3.department AND e1.salary < avg";
        // Execute the query
$result = mysql_query ($underpaid_employees_query);
        // Send query result as a formatted table in HTML
echo "<table border='1'>
<tr> <th>Name</th> <th>Salary</th> <th>Department</th> </tr>";
while ($row = mysql_fetch_array ($result)) {
    echo "<tr>";
    echo "<td>" . $row['name'] . "</td>";
    echo "<td>" . $row['salary'] . "</td>";
    echo "<td>" . $row['department'] . "</td>";
}
echo "</table>";
        // Terminate the database connection
mysql_close ($connection);
?>
```

Figure 6.7 – PHP with SQL query

The example code assumes a database defined similarly to the previously reviewed languages. The query asks for all employees in each department who have a salary less than the average salary of all the employees in that department. The code runs (interpreted or compiled) on the server, and sends the results back to the client.

The example code is straightforward, assuming the programmer is familiar with relational databases, SQL, and the hundreds of functions available in PHP. Because PHP requires programmer expertise in so many technologies, it is the antithesis of orthogonal persistence. Persistence related statements that have little to do with the actual processing of the objects litter the code and require extensive programmer knowledge of the database schema. Its reliance upon SQL subjects PHP scripts to security risks [74]. Type checking of persistent objects seems almost nonexistent. From the viewpoint of orthogonal persistence, the only redeeming quality of PHP is its model of execution, where the code that interfaces with the database runs entirely at the server.

A.7 Thor/Theta (1996)

Thor [51, 52] is a database system that allows sharing of persistent objects across a network by various types of applications. Thor allows safe sharing of persistent objects between applications written in different languages. Thor's client cache management scheme provides efficient store and retrieve operations.

Type-safe sharing means Thor uses and modifies stored objects only in a way consistent with the type of each of those objects. Thor enforces type safety across system boundaries by requiring interface routines to be included within the persistent store itself. These routines, written in an object-oriented language called Theta, form a well-defined interface through which applications access persistent objects, insulating the objects from potential invariant-breaking changes. The database author must implement the routines in a way that ensures each object in the persistent store remains in a valid state. Even programs written in other languages must go through the Theta interface by use of a Theta veneer (i.e. a wrapper) around the non-Theta code. Other database systems available at the time Thor/Theta was developed (GemStone, O2, SHORE, and ObjectStore) allow object sharing, but they do not enforce object consistency. Thor addresses this shortcoming.

The client side of Thor/Theta maintains objects in a cache, requesting objects as needed from Object Repositories (OR) duplicated on multiple servers. An OR contains persistent roots and accesses them through a string-to-object mapping. The OR prefetching policy dynamically determines the set of objects to be returned to the client. Instead of passing a page, which likely contains many objects not requested by the application, a Thor server passes only a subset of those objects that reside on the same page as the requested object. The prefetching mechanism culls or adds objects based on their frequency of use.[1] In addition, once the group of objects arrives at the client Front End (FE), the FE determines which of these objects to keep. When storing objects, a transaction sends only modified objects, not a whole page, back to the server. When compared to paging systems, this object-level granularity at the FE combined with the dynamic prefetching scheme usually reduces the number of objects that must pass over a network, greatly improving overall system performance.

[1] Besides the requested object, the Thor server returns other objects that are stored on disk next to the requested object. If these clustered objects are related, the client program is more likely to use them making the server continue to return a high number of objects. As clustering decreases, the number of objects returned by Thor decreases.

A.8 Java (1996)

Since version 1.1, Java has had facilities for storing objects. Besides the core classes for manipulating files, Java includes Java Object Serialization [40], which allows Java objects to be serialized and deserialized to and from sequences of bytes and the Java Database Connectivity [88] API, which allows Java programmers to work with objects stored in a DBMS. Predating these two persistence methods, and initially built on Java version 1.0, is PJava (later PJama[1]) [5, 11].

A.8.1 PJava

The first version of PJava [11] achieves persistence through a set of classes along with a modified Java VM. In PJava, one class, PJavaStore, encapsulates persistence. To store or retrieve objects the programmer creates a PJavaStore object and then associates a persistence root to a string key, likely within a try block.

```
try {
    PJavaStore companyDB = PJavaStore.getStore();
    // Obtain a persistent store
    companyDB.newPRoot ("Employees", employees);
    // Associate a root with a key
} catch (PJSException e) {
    // Handle the exception
}
```

Figure 6.8 – Create a new persistent store in PJava

The PJavaStore.getStore routine in the try block gives the program access to the persistent store; the newPRoot routine allows the runtime to store automatically the employees object during a commit, which occurs at program termination.

Accessing previously committed objects is almost as straightforward.

```
try {
    PJavaStore companyDB = PJavaStore.getStore();
    employees = (LinkedSet) companyDB.getPRoot ("Employees");
} catch (PJSException e) {
    // Handle the exception
}
```

Figure 6.9 – Using a persistent store in PJava

As before, the PJavaStore.getStore routine reveals the persistent store to the program. The program accesses the stored employees list via a string key with a call to getPRoot. The association of the stored list to the entity, employees, requires a cast to the correct type. In both examples, the employees entity is a persistence root. After obtaining a persistence root, the programmer manipulates the object and all reachable objects normally, with no further concern that the objects are persistent.

The first version of PJava stores changed objects only when the program exits. Later versions of PJava [8] provide checkpointing and resumption routines, allowing objects to be stored at other times. As a PJava program executes, it marks modified, persistent objects and promotes newly reachable, persistent objects to persistent status. These modified and promoted objects are stored during a checkpoint or commit. If an error occurs, PJava restores the objects to a previous, consistent state.

The class PJavaStore provides the interface; modification of the Java VM provides the functionality. The modified VM determines from a cache if an object in volatile memory differs from that in the persistent store and faults objects in from the persistent store when necessary. PJava's persistent store, called Sphere, operates on objects instead of disk pages, as used by other

[1] The PJava developers renamed the project PJama.

systems. The persistent store assigns a Persistent ID (PID) for each persistent object, allocates space for the object, and copies it to disk. For an object fault, the runtime locates the object via its PID and transfers the object to the program.

Persistence in PJava requires minimal effort from the programmer, and the automatic aspect of its persistence mechanism makes it mostly orthogonal, serving as an end-of-program checkpointing mechanism. PJava's persistent store also provides object migration, schema evolution, and disk garbage collection. The convenience provided by the named, persistence roots in PJava predated similar functionality provided by Java Object Serialization.

A.8.2 Java object serialization

Serialization in Java is similar to the serialization mechanism in Eiffel. It stores a persistent root and all reachable objects to a file or transmits them over a network as a stream of bytes. Most Java library classes are serializable, but some (e.g. Thread and Socket) are not. Serializable classes must implement the Serializable interface. Figure 6.10 shows the declaration for the Company class.

```
import java.io.Serializable;
public class Company implements Serializable {
    private static final long serialVersionUID = 1L;
    private LinkedSet <Department> departments = new LinkedSet <Department>();
    private LinkedSet <Person> employees = new LinkedSet <Person>();
// rest of class not shown
```

Figure 6.10 – Implementing the Serializable interface in Java

There are no routines to define; implementing the interface suffices to identify the class as serializable. The serialVersionUID[1] aids in version control. The Department class and the Person class must also implement Serializable. Other entities of the example classes are either serializable (e.g. String) or primitive data types (e.g. **int**), so the default serialization mechanism suffices. Figure 6.11 shows how to store an object in Java.

```
void store_example (String aFileName) throws FileNotFoundException, IOException {
    FileOutputStream fos = new FileOutputStream (aFileName);
    ObjectOutputStream oos = new ObjectOutputStream (fos);
    try {
        oos.writeObject (company);
    } finally {
        oos.close();
    }
```

Figure 6.11 – Store example in Java

The writeObject routine stores its argument, company, and all reachable objects via the ObjectOutputStream, oss. Figure 6.12 shows how to retrieve an object.

[1] The default value is generated from a SHA-1 hash [91] of the name and other components of the class, making the default serial version UID very sensitive to class changes.

```
void retrieve_example (String aFileName) throws FileNotFoundException, IOException {
    FileInputStream fis = new FileInputStream (aFileName);
    ObjectInputStream ois = new ObjectInputStream (fis);
    try {
        Company company = (Company) ois.readObject ();
    } catch (ClassNotFoundException e) {
        System.err.println ("Unknown class: " + e.getMessage());
    } finally {
        ois.close();
    }
}
```

Figure 6.12 – Retrieve example in Java

The readObject routine retrieves an object via the ObjectInputStream, ois, casting the result to the appropriate type. This simple code stores and retrieves serializable objects.

Non-serializable objects require more complex processing. Modification of the default serialization mechanism combined with the reflection API handles non-serializable superclasses and attributes and can improve efficiency. Redefinition of the writeObject and readObject routines furnishes special handling for non-serializable entities. Through reflection, a program examines an object's attributes and type information to make run-time decisions that can improve performance of the persistence mechanism. This customization, versioning control, and ease of use make Java serialization a good persistence mechanism for simple applications. This mechanism, though, is an all-or-nothing approach to persistence, where the entire structure must be stored and retrieved as a whole. It does not work well for large object graphs or shared objects. More sophisticated, database-like functionality requires another approach to persistence, such as the Java Database Connectivity API.

A.8.3 Java database connectivity

The Java Database Connectivity API, or simply JDBC, is a database-mapping approach to persistence. It provides access to objects stored as tabular data, such as a flat file or relational DBMS. JDBC helps programmers connect to a persistent store, query the store using SQL, process persistent objects, and transfer modified objects back to the persistent store. The code in Figure 6.13 illustrates a few of the capabilities of JDBC.

```
public void modify() {
    String url = "jdbc:mysql://localhost:3306/myDB";
    Connection con = DriverManager.getConnection (url, username, password);
    con.setAutoCommit (false);
    Savepoint savePt = con.setSavepoint();
    String query = "SELECT EMP_ID, NAME, SALARY FROM EMPLOYEES";
    Statement stmt = con.createStatement (
                ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
    ResultSet rs = stmt.executeQuery (query);
    boolean tooBig = false;
    while (rs.next() && (!tooBig)) {
        int s = rs.getInt ("SALARY");
        tooBig = s * 2 > maxSalary;
        rs.updateInt ("SALARY", s* 2);
    // double the salary
        rs.updateRow();
    }
    if (tooBig) {
        con.rollback (savePt);
    }
    con.commit();
    con.setAutoCommit (true);
}
```

Figure 6.13 – JDBC modify-with-rollback example

The example code accesses and modifies an existing database that represents a company with a set of departments and a set of employees. The modify routine attempts to double every employee's salary unless this doubling results in a salary that is above the maximum, in which case any changes made up to that point are undone. (For brevity, the code does not show **try**, **catch**, and **finally** blocks.) This code creates a Connection to a MySQL database. After saving the current state of the database in a Savepoint, an SQL query, called on a Statement object, retrieves information about each employee, placing the information in a ResultSet, a table containing the selected rows. Using the ResultSet's iterator, next, the **while** loop visits each row in the table, changing the salaries. If at any point a salary becomes too large, the program rolls back to the referenced Savepoint. The call to setAutoCommit at the beginning allows the code to treat the statements in the loop as a single, all-or-none transaction, but it requires an explicit call to commit at the end to store the changes and release locks obtained during the routine.

JDBC works well for interacting with a relational database, but the added convenience comes with a price; working with complex object-relational mapping can become quite tedious. The approach meets none of the three principles of orthogonal persistence.

A.9 C++ (1999)

Kasbekar et al. [43] describe a different and less ambitious approach for adding a sufficient level of user-transparency to the persistence mechanism for C++. Their preprocessing approach allows an application to checkpoint the state of C++ objects by adding routines Checkpoint() and Restore() to the classes for which the programmer desires persistence. The programmer then adds calls to the runtime's global object called RuntimeSystem.

```
RuntimeSystem.Checkpoint (thisPtr)
```

and

```
RuntimeSystem.Restore (thisPtr)
```

These calls result in the eventual execution of the Checkpoint() or Restore() routine of the object referenced from thisPtr. It also forces any objects reachable from the thisPtr object to be checkpointed in like manner. This deep checkpointing[1] is analogous to making a deep copy. A unique type-id attribute, added during code preprocessing, records the type of each object. This type information facilitates allocation of memory and object retrieval. The preprocessing also adds an ancestor class to each of the persistent classes.

Because source code must be available for the preprocessing, this approach cannot make all classes persistent. To accommodate missing source code and avoid preprocessing errors, users of this approach must specify, using **MA_Persistent class** T { … }, those classes that should be processed; or by specifying the names of the classes in a supplemental file. This falls short of Atkinson's three principles of persistence.

A.10 Persistent Oberon (2006)

Persistent Oberon [12, 13, 70] extends Oberon [77], a language in the Algol family with strict static typing of variables and functions. Oberon builds on the module concept of Modula-2, allowing the programmer to add attributes to a module, extending the abstract data types described by the original modules. Persistent Oberon runs in its own environment and loads modules as needed, after which they remain loaded, surviving system restarts. Each module is a persistent root, making all objects referenced in that module automatically persistent. Type declarations are the same in Persistent Oberon as in Oberon. The example code in Figure 6.14 creates the persistent object Employees the first time the module is loaded, and that object, along with any reachable

[1] Dr. Raphael Finkel, advisor and University of Kentucky professor, suggested the term "deep checkpointing."

objects, remains alive unless the module is explicitly unloaded. (Modules are unloaded for program modification.)

```
MODULE Company
    CONST n = 30;
    TYPE
        Person = RECORD
            name: ARRAY n of CHAR;
            salary: INTEGER;
        END Person;
        EmployeeList = OBJECT (* … *) END EmployeeList;
    VAR
        Employees: EmployeeList;
PROCEDURE SetSalary (aName: ARRAY OF CHAR; aValue: INTEGER);
    VAR
        p: POINTER TO Person;
    BEGIN {TRANSACTION}
            (* assume a search procedure exists *)
        p := FindEmployeeByName (aName);
        p.salary := aValue;
    END SetSalary;
END Company;
```

Figure 6.14 – Persistent Oberon

To capture the idea of consistent state, Persistent Oberon adds transactions, complete with atomicity, abort, and rollback capability. To define a transaction, the programmer annotates a normal **BEGIN-END** block with a **TRANSACTION** decoration, as shown in the example code. The entire block executes completely or, if an operation aborts, any persistent objects affected by the transaction role back to their previous state. Transactions execute serially (no overlap in time) within a system. Because Persistent Oberon runs within its own environment, only one instance of a particular module exists. It is unclear if a module can reference objects belonging to another module. All entities represent persistent objects unless specified by the programmer as **TRANSIENT** (does not survive restarts) or **WEAK** (reclaimable by the runtime.) When a top-level transaction completes, the runtime stores all modified objects. The Persistent Oberon environment incorporates a main-memory caching scheme to speed movement of objects between the persistent store and volatile memory. The system also facilitates schema evolution.

The addition of the **TRANSACTION** tag is the only change to Oberon. From the programmer's point of view, Persistent Oberon achieves orthogonal persistence.

A.11 Timor (2007)

Timor [44, 45] is an object oriented language that was designed to distribute persistent objects over the Internet. Ideally, a Timor program runs within the SPEEDOS operating system[1], which works directly with Timor's routines. Timor approaches orthogonal persistence, because it hides most of the persistent mechanism from the programmer and allows all [types of] objects to be [potentially] persistent.

Any type of Timor object can be made persistent by instantiating it as a persistent root, which the authors call a [Timor] **file**, using the **create** keyword. Objects reachable from this file automatically become persistent. Timor references the file via a **capability**, a special kind of reference that allows method calls on objects from the operating system as well as the programming language and which can protect objects by restricting access to a subset of the object's routines. The following code creates a file of type CompanyDatabase and makes it persistent.

[1] SPEEDOS is a new operating system under simultaneous development by the Timor developers to facilitate Timor programs. The website, www.speedos-security.org, is currently undergoing maintenance. It appears this project was never finished.

```
CompanyDatabase** companyDB = create CompanyDatabase.int();
```

A programmer accesses this file through the companyDB entity. The double star declares companyDB as a capability. This gives companyDB access to all the routines (or a subset of the routines) identified in the interface section of the CompanyDatabase type definition. The operating system (or an emulator) verifies whether a caller can use the called routine.

A globally unique identifier[1], assigned at creation time and available to the programmer, provides indexing of Timor files across a network. File-unique identifiers index sub-objects within each file. A combination of the file and sub-object identifiers can identify objects referenced across various files. The capability to distribute Timor files globally and the ability to reference objects across Timor file boundaries highlights the problem of garbage collection on a worldwide scale. Though Timor does not solve this problem, it does draw attention to the possibility and perhaps the need to identify and distribute persistent objects worldwide.

A.12 JavaScript (2010)

Cannon and Wohlstadter [19] present (without code examples) a framework-based approach to persistence in JavaScript for use with off-line, web-based applications. This framework detects changes made via assignment operations to the properties of an object. (JavaScript does not have classes; everything is an object. A **property** is analogous to an attribute or a routine.) It also detects changes made to an object's structure, such as the addition or removal of a property, using periodic iteration over specific groups of persistent objects to search for new properties. The framework relies on JavaScript Object Notation (JSON) to store individual objects. The framework serializes the references to objects, removing cycles, which JSON cannot handle, and generates a globally unique identifier (GUID) for mapping to the serialized objects. The framework then stores the objects locally for use by the application. Periodically, the application pushes modified objects to a persistent store over the internet, and returns objects changed at the server level to the application. The server notifies the application about any conflicts and gives the application opportunity to resolve them.

The framework abstracts object-level access and persistence timing issues. A key/value mapping provides access at the level of individual objects through a GUID instead of the traditional persistence-by-reachability. (The authors do not give details about this GUID, but it is clear that it allows the framework to match a particular stored object to the object model of the application.) This finer, object-level granularity allows the framework to balance the timing of store and retrieve operations between numerous, frequently called operations on small objects and fewer, less frequently called operations on a single large batch, reducing perceptible delays. This framework uses two methods for determining this set of changed objects, the use of JavaScript accessors and a developer-scheduled task that checks objects for changes in structure.

In JavaScript, a developer can bind a function to a property, turning a simple access or assignment into a function call. JavaScript interprets an assignment such as MyObject.attribute := aValue as a function call where aValue is the argument to a setter routine. These bound routines in the framework do additional work beyond the normal semantics of access or assignment. In other words, the framework intercepts the normal operations, adding its own semantics through a function call that marks an object as changed.

Besides the normal modification of object properties through assignment, JavaScript also allows the structure objects to change dynamically. (This is analogous to adding an attribute to a class, but JavaScript allows this addition during program execution.) The framework detects this type of change. The framework maintains a set of the referenced, persistent objects and periodically traverses the set, checking for new or deleted object properties. The programmer tunes the

[1] Neither of the referenced papers describes the format of this globally unique identifier.

time spent on this iteration and the frequency of this operation in order to balance the time spent on this maintenance task with the likelihood of data loss.

After the framework discovers a set of modified objects, the web browser locally stores the changed objects along with a log that records the reason for each modification (e.g. creation, update, or deletion.) The framework serializes the objects to disk using JSON, removing cyclic references and assigning a GUID to each object. The log, also stored locally, aids reliability and batch processing of changed objects by the remote server. Periodically, the application sends the modified objects and log to the server. The server simply stores any modified objects that do not conflict with the version on the server (i.e. they have not been changed elsewhere.) Conflicting modifications trigger a callback, forcing resolution at the application. In addition, the server responds by sending the GUID's [only] of objects that have changed since the last synchronization with the server. If the application requires the new objects, it requests them from the server [using the GUID's] and stores the resulting objects locally for possible off-line use by the application.

A.13 Summary

PS-Algol introduced the concept of orthogonal persistence, demonstrating persistence as a built-in part of the language. Like most of the languages, it uses a load-compute-save model combined with pointer faulting. The model used by Galileo and Napier is a continuous computation model. Each of these languages has a persistent environment where any object reachable from that environment is automatically stored. In contrast to persistence-by-reachability, the classes in Galileo and the collections in E demonstrate persistence-by-allocation where persistence of an object is determined at creation time. Persistent Oberon and PM3 add persistence mechanisms to Modula. Persistent Oberon wraps computations in transactions; PM3 focuses on the use of the garbage collector for determining object reachability. For C++, this paper presents a somewhat limited scheme that uses a precompiler to add persistence code to the language, allowing the state of selected objects to be checkpointed. PJava demonstrates many of the previous concepts (e.g. object faulting, persistent store with a cache, and persistent object identifiers.) PJava provides the possibility of referencing the persistent store by object instead of by memory page. The JavaScript approach also uses object-level granularity for persistence. It detects modification of an object's structure as well as modification of an object's values. It demonstrates a method of distributing persistent objects via client-server architecture, complete with callbacks from the server. Thor/Theta, a database programming language built for object sharing between different types of applications, also illustrates distribution of persistent objects. Timor combines a persistent programming language with an operating system built specifically for persistence. Timor also draws attention to the possibility of worldwide distribution of persistent objects. The other two approaches, serialization and database mapping, used in Eiffel and Java, illustrate two extremes of the persistence problem. The serialization approach is easy to use, but it induces an all-or-nothing dilemma; either the whole structure is retrieved and useable, or nothing is useable. The database-mapping approach remedies this problem, but its use involves code that is more complicated.

The reviewed languages use seemingly different approaches to persistence, but taken as a whole, illustrate some of the goals of a persistent programming language and show helpful techniques for developing an orthogonally persistent programming language. The main goal of all these languages is to abstract object persistence so programmers need not be concerned with the movement of objects between volatile memory and a persistent store. The developers of the above languages believe that orthogonal persistence will lead to improvements in software systems. However, proving that a language with orthogonal persistence provides improvement over one that does not provide that functionality can only be achieved, as pointed out by Atkinson et al., with the development of a platform in which this Orthogonal Persistence Hypothesis [8] can

be tested. The above languages provide anecdotal evidence that such a system is possible. These languages form a base on which a new orthogonally persistent platform should be built and point out likely characteristics required of a new system.

Appendix B P-Eiffel setup and use

To produce a P-Eiffel compiler, apply the patch file shown at the end of this appendix and also available online, as described below, to the Eiffel Software GPL source code, and compile the modified code with an Eiffel compiler. Include the persistence framework classes in a project and compile it with the new, modified compiler. (This patch and library works for version 16.05 of the Eiffel compiler.)

B.1 Building P-Eiffel

1. Download the patch file and library from:

   https://github.com/boxer41a/P-Eiffel

2. Download the Eiffel compiler from:

   www.eiffel.com

3. Download the Eiffel source code from:

   https://github.com/EiffelSoftware/EiffelStudio.git

4. Apply the runtime patch file:

   ```
   cd $EIFFEL_DEV/Src
   patch –p0 < /location_of_patch_file/auto_persistence_v4.patch
   ```

5. Compile the new runtime, runtime libraries, and compiler:

   See instructions at https://dev.eiffel.com/Compiling_EiffelStudio

6. Run the new EiffelStudio, including the jj_persistence cluster in a project.

B.2 Using P-Eiffel

The following code segments highlight the minimal persistence related code within some surrounding code context.

Before any persistence operations, automatic or manual, the program must connect to a repository, preferably in the creation feature of the root class. The root class should inherit from PERSISTENCE_FACILITIES and call set_repository. The creation feature of the REPOSITORY class sets up the underlying physical datastore.

```
class
    ROOT
inherit
    PERSISTENCE_FACILITIES
create
    make
feature {NONE} -- Initialization
    make
            -- Set up Current.
        local
            c: CREDENTIALS
            r: LOCAL_REPOSITORY
        do
            create c.make (“data_file.dat”)
            create r.make (c)
            Persistence_manager.set_repository (r)
                -- The rest as normal.
        end
```

Additionally, classes describing objects that are automatically persistable should inherit from class PERSISTABLE, calling the precursor version of default_create.

```
class
    MY_CLASS
inherit
    PERSISTABLE
        redefine
            default_create
        end
feature -- Initialization
    default_create
            -- Set up Current.
        do
            Precursor {PERSISTABLE}
                -- The rest as normal.
        end
```

The following segment shows the minimal code required to retrieve an object given its PID.

```
            if repository.is_stored (pid) then
                check attached {PERSON} Persistence_manager.loaded (pid) as p then
                    -- Use the PERSON p normally.
                end
            end
```

B.3 The runtime patch file

```
Index: C/run-time/eif_auto_persistence.h
===================================================================
--- C/run-time/eif_auto_persistence.h                           (nonexistent)
+++ C/run-time/eif_auto_persistence.h                           (working copy)
@@ -0,0 +1,27 @@
+
+#ifndef _eif_auto_persistence_h_
+#define _eif_auto_persistence_h_
+#if defined(_MSC_VER) && (_MSC_VER >= 1020)
+#pragma once
+#endif
+
+#include "eif_portable.h"
+
+#ifdef __cplusplus
+extern "C" {
+#endif
+
+/* Constants for different tasks. */
+#define EIF_AP_DIRTY 1
+#define EIF_AP_QUALIFIED_CALL 2
+#define EIF_AP_CREATION 3
+
+/* Additional code for the automatic persistence framework. */
+RT_LNK void eif_auto_persistence_init (EIF_REFERENCE a_object, EIF_POINTER a_routine);
+RT_LNK void eif_auto_persistence_callback (EIF_REFERENCE a_object, EIF_INTEGER_32 a_task);
+
+#ifdef __cplusplus
+}
+#endif
+
+#endif

Property changes on: C/run-time/eif_auto_persistence.h
___________________________________________________________________
Added: svn:eol-style
## -0,0 +1 ##
```

```
+native
\ No newline at end of property
Added: svn:keywords
## -0,0 +1 ##
+Author Date Id Revision
\ No newline at end of property

Index: C/run-time/eif_eiffel.h
===================================================================
--- C/run-time/eif_eiffel.h                                 (revision 98126)
+++ C/run-time/eif_eiffel.h                                 (working copy)
@@ -61,6 +61,7 @@

 #include "eif_macros.h"

+#include "eif_auto_persistence.h"

 /* Platform definition */

Index: C/run-time/eif_types.h
===================================================================
--- C/run-time/eif_types.h                                  (revision 98126)
+++ C/run-time/eif_types.h                                  (working copy)
@@ -301,6 +301,8 @@
                        EIF_TYPE_INDEX dftype;
                        uint16 flags;
                        EIF_SCP_PID scp_pid;                /* SCOOP Processor ID to which object belongs */
+                       EIF_NATURAL_64 persistence_id;      /* Object persistence ID for automatic persistence. */
+                       EIF_NATURAL_64 persistence_id_2;    /* Padding */
                } ovs;
           } ovu;
           rt_uint_ptr ovs_size;
Index: C/run-time/garcol.c
===================================================================
--- C/run-time/garcol.c                                     (revision 98126)
+++ C/run-time/garcol.c                                     (working copy)
@@ -4180,6 +4180,7 @@
     uint16 age;                                             /* Object's age */
     uint16 flags;                                           /* Eiffel flags */
     uint16 pid;                                             /* SCOOP Processor ID */
+    EIF_NATURAL_64 l_persistence_id;
     EIF_TYPE_INDEX dftype, dtype;
     EIF_REFERENCE new;                                      /* Address of new object (tenured) */
     rt_uint_ptr size;                                       /* Size of scavenged object */
@@ -4190,6 +4191,7 @@
     dftype = zone->ov_dftype;
     dtype = zone->ov_dtype;
     pid = zone->ov_pid;
+    l_persistence_id = zone->ov_head.ovu.ovs.persistence_id;

     if (gen_scavenge & GS_STOP)                             /* Generation scavenging was stopped */
         if (!(flags & EO_NEW))                              /* Object inside scavenge zone */
@@ -4306,6 +4308,7 @@
                zone->ov_dftype = dftype;
                zone->ov_dtype = dtype;
                zone->ov_pid = pid;
+               zone->ov_head.ovu.ovs.persistence_id = l_persistence_id;
                zone->ov_size &= ~B_C;                       /* Object is an Eiffel one */

                CHECK("Valid size", size <= (zone->ov_size & B_SIZE));
Index: C/run-time/malloc.c
===================================================================
--- C/run-time/malloc.c                                     (revision 98126)
+++ C/run-time/malloc.c                                     (working copy)
@@ -1432,6 +1432,7 @@
```

```
          zone->ov_dftype = HEADER(ptr)->ov_dftype;
          zone->ov_dtype = HEADER(ptr)->ov_dtype;
          zone->ov_pid = HEADER(ptr)->ov_pid;
+         zone->ov_head.ovu.ovs.persistence_id = HEADER(ptr)->ov_head.ovu.ovs.persistence_id;

              /* Update flags of new object if it contains references and the object is not
               * in the scavenge zone anymore. */
@@ -2982,6 +2983,7 @@
          HEADER(zone)->ov_dftype = HEADER(ptr)->ov_dftype;
          HEADER(zone)->ov_dtype = HEADER(ptr)->ov_dtype;
          HEADER(zone)->ov_pid = HEADER(ptr)->ov_pid;
+         HEADER(zone)->ov_head.ovu.ovs.persistence_id = HEADER(ptr)->ov_head.ovu.ovs.persistence_id;
          if (!(gc_flag & GC_FREE)) {                    /* Will GC take care of free? */
              eif_rt_xfree(ptr);                         /* Free old location */
          } else {
@@ -3956,6 +3958,7 @@
     if (EIF_IS_EXPANDED_TYPE(System (dtype))) {
         zone->ov_flags |= EO_EXP | EO_REF;
     }
+    zone->ov_head.ovu.ovs.persistence_id = 0LL;

 #ifdef ISE_GC
     if (flags & EO_NEW) {                              /* New object outside scavenge zone */
@@ -4038,6 +4041,8 @@
     zone->ov_pid = (EIF_SCP_PID) 0;
 #endif
     zone->ov_size &= ~B_C;                             /* Object is an Eiffel one */
+
+    zone->ov_head.ovu.ovs.persistence_id = 0LL;

 #ifdef ISE_GC
     if (in_scavenge == EIF_FALSE) {
Index: C/run-time/misc.c
===================================================================
--- C/run-time/misc.c                                  (revision 98126)
+++ C/run-time/misc.c                                  (working copy)
@@ -68,6 +68,8 @@
 #include <ctype.h>                                     /* For toupper(), is_alpha(), ... */
 #include <stdio.h>

+#include "eif_auto_persistence.h"
+
 /*
 doc:    <routine name="eif_pointer_identity" export="public">
 doc:            <summary>Because of a crash of VC6++ when directly assigning a function pointer to an array of function
pointer in a loop, we create this identity function that cannot be inlined and thus prevents the bug to occur. As soon
as VC6++ is not supported we can get rid of it. Read comments on ROUT_TABLE.generate_loop_initialization for
details.</summary>
@@ -556,6 +558,51 @@
 }
 #endif

+/* Variables needed to store the handler object and routine. */
+rt_private EIF_OBJECT eif_auto_persistence_handler = NULL;
+rt_private EIF_PROCEDURE eif_auto_persistence_callback_routine = NULL;
+
+/* Initialize the auto_persistence callback module with the two given arguments. */
+rt_public void eif_auto_persistence_init (EIF_REFERENCE a_object, EIF_POINTER a_routine)
+{
+    EIF_OBJECT l_protected = NULL;
+
+         /* Convert 'a_object' to an EIF_OBJECT indirect reference that is protected by the garbage collector. */
+    if (a_object) {
+         l_protected = eif_protect (a_object);
+    }
```

```
+
+       /* Release the old object from the protection (if any). */
+   if (eif_auto_persistence_handler) {
+       eif_wean (eif_auto_persistence_handler);
+   }
+
+       /* Set the new handler object. */
+   eif_auto_persistence_handler = l_protected;
+
+       /* Set the callback function. */
+   eif_auto_persistence_callback_routine = (EIF_PROCEDURE) a_routine;
+}
+
+/* Perform a callback into Eiffel code. */
+rt_public void eif_auto_persistence_callback (EIF_REFERENCE a_object, EIF_INTEGER_32 a_task)
+{
+   if (eif_auto_persistence_handler && eif_auto_persistence_callback_routine) {
+
+           /* Temporarily set the callback routine to NULL.
+            * That way we can avoid infinite recursion when the callback triggers another callback. */
+       EIF_PROCEDURE l_routine = eif_auto_persistence_callback_routine;
+       eif_auto_persistence_callback_routine = NULL;
+
+           /* Execute the routine */
+       l_routine (eif_access (eif_auto_persistence_handler), a_object, a_task);
+           /* NOTE: After the call to 'eif_auto_persistence_callback_routine', 'a_object' may be invalid. Do not use it
any longer. */
+
+           /* Reset the callback routine. */
+       eif_auto_persistence_callback_routine = l_routine;
+   }
+}
+
 /*
 doc:</file>
 */
Index: Eiffel/eiffel/byte_code/access_b.e
===================================================================
--- Eiffel/eiffel/byte_code/access_b.e                  (revision 98126)
+++ Eiffel/eiffel/byte_code/access_b.e                  (working copy)
@@ -615,6 +615,19 @@
                    buf.put_new_line
                    buf.put_character ('}')
                end
+
+               if a_result = Void and a_target.c_type.is_reference and (call_kind = call_kind_creation or call_kind =
call_kind_qualified) then
+                   buf.put_new_line
+                   buf.put_string ("eif_auto_persistence_callback (")
+                   a_target.print_register
+                   buf.put_two_character (',', ' ')
+                   if call_kind = call_kind_qualified then
+                       buf.put_string ("EIF_AP_QUALIFIED_CALL")
+                   else
+                       buf.put_string ("EIF_AP_CREATION")
+                   end
+                   buf.put_two_character (')', ';')
+               end
           end

 feature -- Conveniences
Index: Eiffel/eiffel/byte_code/assign_bl.e
===================================================================
--- Eiffel/eiffel/byte_code/assign_bl.e                 (revision 98126)
+++ Eiffel/eiffel/byte_code/assign_bl.e                 (working copy)
```

```
@@ -347,6 +347,11 @@
                else
                    generate_assignment
                end
+
+               if not target.is_predefined then
+                   buffer.put_new_line
+                   buffer.put_string ("eif_auto_persistence_callback (Current, EIF_AP_DIRTY);")
+               end
            end

        Simple_assignment: INTEGER = 4
@@ -725,7 +730,7 @@
            end

    note
-       copyright:   "Copyright (c) 1984-2013, Eiffel Software"
+       copyright:   "Copyright (c) 1984-2015, Eiffel Software"
        license: "GPL version 2 (see http://www.eiffel.com/licensing/gpl.txt)"
        licensing_options: "http://www.eiffel.com/licensing"
        copying: "[
```

References

[1] Albano, A., Cardelli, L. and Orsini, R. 1985. GALILEO: a strongly-typed, interactive conceptual language. *ACM Trans. Database Syst.* 10, 2 (Jun. 1985), 230–260.

[2] Albano, A., Ghelli, G., Occhiuto, M.E. and Orsini, R. 1986. A strongly typed, interactive object-oriented database programming language. *Proceedings on the 1986 international workshop on Object-oriented database systems* (Los Alamitos, CA, USA, 1986), 94–103.

[3] Amalio, N. and Kelsen, P. 2010. Modular Design by Contract Visually and Formally Using VCL. *2010 IEEE Symposium on Visual Languages and Human-Centric Computing* (Sep. 2010), 227–234.

[4] Amazon Simple Storage Service (S3) - Cloud Storage: *//aws.amazon.com/s3/*. Accessed: 2016-07-12.

[5] Atkinson, M. 2001. Persistence and Java — A Balancing Act. *Objects and Databases*. K. Dittrich, G. Guerrini, I. Merlo, M. Oliva, and M.E. Rodriguez, eds. Springer Berlin / Heidelberg. 1–31.

[6] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. and Zdonik, S. 1989. *The Object-Oriented Database System Manifesto*.

[7] Atkinson, M., Chisholm, K. and Cockshott, P. 1982. PS-algol. *ACM SIGPLAN Notices*. 17, 7 (Jul. 1982), 24.

[8] Atkinson, M., Jordan, M., Atkinson, M. and Jordan, M. 2000. *A review of the rationale and architectures of PJama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform*. Sun Microsystems Laboratories Inc and Department of Computing Science, University of Glasgow.

[9] Atkinson, M. and Morrison, R. 1995. Orthogonally persistent object systems. *The VLDB Journal*. 4, 3 (Jul. 1995), 319–402.

[10] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, P.W. and Morrison, R. 1983. An Approach to Persistent Programming. *The Computer Journal*. 26, 4 (Nov. 1983), 360–365.

[11] Atkinson, M.P., Daynès, L., Jordan, M.J., Printezis, T. and Spence, S. 1996. An orthogonally persistent Java. *ACM SIGMOD Record*. 25, 4 (Dec. 1996), 68–75.

[12] Bläser, L. 2006. A programming language with natural persistence. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), 637–638.

[13] Bläser, L. 2007. Persistent Oberon: A Programming Language with Integrated Persistence. *Programming Languages and Systems*. Z. Shao, ed. Springer Berlin / Heidelberg. 71–85.

[14] Brown, A. 1988. *Persistent Object Stores*. University of St Andrews.

[15] Brown, A.L. 1996. *Napier88 Standard Library Reference Manual: Release 2.2.1*. Department of Computer Science, University of Adelaide.

[16] Buschmann, F. 2011. Unusable Software Is Useless, Part 2. *IEEE Software*. 28, 2 (Mar. 2011), 100–102.

[17] C++ Object Persistence with ODB: 2016. *http://www.codesynthesis.com/products/odb/doc/manual.xhtml#0.1*.

[18] Cahill, V., Horn, C., Kramer, A., Martin, M., Starovic, G. and College, T. 1990. *C** and Eiffel**: languages for distribution and persistence*.

[19] Cannon, B. and Wohlstadter, E. 2010. Automated object persistence for JavaScript. *Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), 191–200.

[20] Chalin, P., Kiniry, J.R., Leavens, G.T. and Poll, E. 2006. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. *Formal Methods for Components and Objects*. F.S. de Boer, M.M. Bonsangue, S. Graf, and W.-P. de Roever, eds. Springer Berlin Heidelberg. 342–363.

[21] Chignoli, R., Farre, J., Lahire, P. and Rousseau, R. 1994. *FLOO: A Strong Coupling Between Eiffel Language and O2 DBMS*.
[22] Cloud Storage - Online Data Storage: *https://cloud.google.com/storage/*. Accessed: 2016-07-12.
[23] Codd, E.F. 1970. A relational model of data for large shared data banks. *Commun. ACM*. 13, 6 (Jun. 1970), 377–387.
[24] Concurrent programming with SCOOP: 2016. *https://www.eiffel.org/doc/solutions/Concurrent programming with SCOOP*. Accessed: 2016-07-25.
[25] Crawley, S. and Oudshoorn, M. 1994. Orthogonal Persistence and Ada. *IN PROCEEDINGS TRI-ADA'94, BALTIMORE MD*. (1994), 29--8.
[26] Crismer, P.G. and Fafchamps, E. 2003. EPOM Eiffel Persistent Object Management.
[27] Customers: *https://www.eiffel.com/company/customers/*. Accessed: 2016-11-03.
[28] Databases and Persistence: 2016. *http://wiki.squeak.org/squeak/512*. Accessed: 2016-11-03.
[29] Daynes, L. and Czajkowski, G. 2001. High-performance, space-efficient, automated object locking. (2001), 163–172.
[30] EiffelStore Tutorial: *https://www.eiffel.org/doc/solutions/EiffelStore*. Accessed: 2016-04-20.
[31] Franco-Japanese Symposium on Programming of Future Generation Computers 1988. *Programming of future generation computers II: proceedings of the Second Franco-Japanese Symposium on Programming of Future Generation Computers, Cannes, France, 9-11, November, 1987*. North-Holland ; sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co.
[32] Fu, M.-M. and Dasgupta, P. 1993. A concurrent programming environment for memory-mapped persistent object systems. *Computer Software and Applications Conference, 1993. COMPSAC 93. Proceedings., Seventeenth Annual International* (Nov. 1993), 291–297.
[33] Grimstad, S., Sjøberg, D.I.K., Atkinson, M. and Welland, R. *Evaluating Usability Aspects of PJama based on Source Code Measurements*.
[34] Guerreiro, P. 2001. Simple support for design by contract in C++. *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39* (2001), 24–34.
[35] Hoare, C.A.R. 1978. Communicating Sequential Processes. *Communication of the ACM*. 21, 8 (Aug. 1978), 666–677.
[36] Hosking, A.L. and Chen, J. 1999. Mostly-copying reachability-based orthogonal persistence. *SIGPLAN Not.* 34, 10 (Oct. 1999), 382–398.
[37] Hosking, A.L. and Chen, J. 1999. PM3: An Orthogonal Persistent Systems Programming Language - Design, Implementation, Performance. *Proceedings of the 25th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1999), 587–598.
[38] Hosking, A.L. and Moss, J.E. 1991. *Compiler Support for Persistent Programming*. University of Massachusetts.
[39] Hosking, A.L., Moss, J.E.B. and Bliss, C. 1990. Design of an Object Faulting Persistent Smalltalk. (1990).
[40] Java Object Serialization Specification: *http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html*.
[41] Jayasekara, K.A.T.A. and Jayasena, S. 2010. Persistent data structure library for C++ applications. *2010 1st International Conference on Parallel Distributed and Grid Computing (PDGC)* (Oct. 2010), 356–361.
[42] Jazequel, J.M. and Meyer, B. 1997. Design by contract: the lessons of Ariane. *Computer*. 30, 1 (Jan. 1997), 129–130.

[43] Kasbekar, M., Yajnik, S., Klemm, R., Huang, Y. and Das, C.R. 1999. Issues in the design of a reflective library for checkpointing C++ objects. *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, 1999* (1999), 224–233.
[44] Keedy, J.L., Espenlaub, K., Heinlein, C. and Menger, G. 2007. Persistent Objects and Capabilities in Timor. *The Journal of Object Technology*. 6, 4 (2007), 103.
[45] Keedy, J.L., Espenlaub, K., Heinlein, C. and Menger, G. 2007. Persistent Processes and Distribution in Timor. *The Journal of Object Technology*. 6, 6 (2007), 91.
[46] Kempf, J., Paepcke, A., Beach, B., Mohan, J., Mahbod, B. and Snyder, A. 1988. Language level persistence for an object-oriented application programming platform. *Software Track, Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, 1988. Vol.II* (0 1988), 424–433.
[47] Khoshafian, S.N. and Copeland, G.P. 1986. Object identity. *ACM SIGPLAN Notices*. 21, 11 (Nov. 1986), 406–416.
[48] Kienzle, J. and Romanovsky, A. 2002. Framework based on design patterns for providing persistence in object-oriented programming languages. *Software, IEE Proceedings* -. 149, 3 (Jun. 2002), 77–85.
[49] King, E. 1978. *IBM Report on the Contents of a Sample of Programs Surveyed*. IBM Research Center San Jose.
[50] Lamb, C., Landis, G., Orenstein, J. and Weinreb, D. 1991. The ObjectStore database system. *Commun. ACM*. 34, 10 (Oct. 1991), 50–63.
[51] Liskov, B., Adya, A., Castro, M., Ghemawat, S., Gruber, R., Maheshwari, U., Myers, A.C., Day, M. and Shrira, L. 1996. Safe and efficient sharing of persistent objects in Thor. *Proceedings of the 1996 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1996), 318–329.
[52] Liskov, R.B., Liskov, B., Adya, A., Castro, M., Chung, E., Curtis, D., Ghemawat, S., Gruber, R., Johnson, P., Maheshwari, U., Myers, A.C., Ng, T., Shrira, L. and Zondervan, Q. *Thor/Theta Users Guide*.
[53] Lunney, T. and McCaughey, A. 2003. Object persistence in Java. *Proceedings of the 2nd international conference on Principles and practice of programming in Java* (New York, NY, USA, 2003), 115–120.
[54] Meyer, B. 1992. *Eiffel The Language*. Prentice Hall International (UK) Ltd.
[55] Meyer, B. 1988. *Object-oriented Software Construction*. Prentice Hall International (UK) Ltd.
[56] Meyer, B. 1997. *Object-oriented software construction*. Prentice Hall PTR.
[57] Meyer, B. 2009. *Touch of class learning to program well with objects and contracts*. Springer.
[58] Morrison, R. 1979. *On the Development of Algol*. University of St Andrews.
[59] Morrison, R. 1979. S-algol Reference Manual. University of St Andrews.
[60] Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. and Atkinson, M.P. 1989. The Napier Type System. *Persistent Object Systems*. Rosenberg, J. & Koch, D. (ed), Springer-Verlag. 3–18.
[61] Morrison, R., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. and Munro, D.S. A PROGRAMMING LANGUAGE AND DATABASE INTEGRATION TECHNOLOGY. University of St Andrews, Scotland.
[62] Morrison, R., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Munro, D.S. and Atkinson, M.P. The Napier88 Persistent Programming Language and Environment.
[63] Moss, J.E.B. and Hosking, A.L. 1996. *Approaches to Adding Persistence to Java*.
[64] Neward, T. 2006. The Vietnam of Computer Science. *The Blog Ride*.
[65] Oudshoorn, M. and Crawley, S. The Addition of Persistence to Ada95 and its Consequences.

[66] Overloading vs. Object Technology: 2001. *http://se.ethz.ch/~meyer/publications/joop/overloading.pdf*.
[67] Ozkaya, M. and Kloukinas, C. 2013. Towards Design-by-Contract based software architecture design. *2013 IEEE 12th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT)* (Sep. 2013), 157–164.
[68] Panchapakesan, A., Abielmona, R. and Petriu, E. 2013. A python-based design-by-contract evolutionary algorithm framework with augmented diagnostic capabilities. *2013 IEEE Congress on Evolutionary Computation* (Jun. 2013), 2517–2524.
[69] Persistence unified: *https://dev.eiffel.com/Persistence_unified*. Accessed: 2016-08-13.
[70] Persistent Oberon Language Specification: *http://www.composita.net/PersistentOberon.html*.
[71] Pfleeger, S.L. 2006. *Software engineering: theory and practice*. Pearson/Prentice Hall.
[72] PHP Just Grows & Grows: *http://news.netcraft.com/archives/2013/01/31/php-just-grows-grows.html*. Accessed: 2013-11-18.
[73] PHP Manual: 2013. *http://us1.php.net/manual/en/*.
[74] PHP Programming/SQL Injection: *http://en.wikibooks.org/wiki/PHP_Programming/SQL_Injection*.
[75] Piccioni, M., Oriol, M. and Meyer, B. 2013. Class Schema Evolution for Persistent Object-Oriented Software: Model, Empirical Study, and Automated Support. *IEEE Transactions on Software Engineering*. 39, 2 (Feb. 2013), 184–196.
[76] Piccioni, M., Oriol, M., Meyer, B. and Schneider, T. 2009. An IDE-based, integrated solution to Schema Evolution of Object-Oriented Software. (Auckland, New Zeland, Nov. 2009).
[77] Reiser, M. 1992. *Programming in Oberon: steps beyond Pascal and Modula*. ACM Press ; Addison-Wesley Pub. Co.
[78] Richardson, J.E. and Carey, M.J. 1989. Persistence in the E Language: Issues and implementation. *Softw. Pract. Exper.* 19, 12 (Nov. 1989), 1115–1150.
[79] Richardson, J.E., Carey, M.J. and Schuh, D.T. 1993. The Design of the E Programming Language. *ACM Transactions on Programming Languages and Systems*. 15, (1993), 494–534.
[80] Rubio-Medrano, C.E., Ahn, G.J. and Sohr, K. 2013. Verifying Access Control Properties with Design by Contract: Framework and Lessons Learned. *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual* (Jul. 2013), 21–26.
[81] Schuh, D., Carey, M. and Dewitt, D. 1990. Persistence in E Revisited — Implementation Experiences. *In Dearle et al. [DSZ90* (1990), 345–359.
[82] Sebesta, R.W. 2010. *Concepts of programming languages*. Addison-Wesley.
[83] Silberschatz, A. 2011. *Database system concepts*. McGraw-Hill.
[84] Swift Guides and Sample Code: 2016. *https://developer.apple.com/library/content/referencelibrary/GettingStarted/DevelopiOSAppsSwift/Lesson10.html*.
[85] Takasaka, S. 2005. *Survey of Persistence Approaches*. Stockholm University.
[86] Tantivongsathaporn, J. and Stearns, D. 2006. An Experience With Design by Contract. *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)* (Dec. 2006), 335–341.
[87] The Python Standard Library: *https://docs.python.org/3.4/library/persistence.html*. Accessed: 2016-11-03.
[88] Trail: JDBC(TM) Database Access: *http://docs.oracle.com/javase/tutorial/jdbc/index.html*.
[89] Vangari, S.S. 1998. *An object relationship framework and persistence in Java*. West Virginia University.
[90] Zdonik, S.B. and Maier, D. 1990. *Readings in object-oriented database systems*. Morgan Kaufmann.

[91] 2012. Federal Information Processing Standards Publication: Secure Hash Standard (SHS). National Institute of Standards and Technology.
[92] 2016. General Question: Object Persistence Mechanism. *Eiffel Users - Google Groups*.
[93] 2015. .NET guys, do you use DataSets or write your own persistence layers? *Eiffel Users - Google Groups*.
[94] 2013. Nominative type system. *Wikipedia, the free encyclopedia*.
[95] 2012. Object-relational impedance mismatch. *Wikipedia, the free encyclopedia*.
[96] 2013. PHP. *Wikipedia, the free encyclopedia*.
[97] 2015. Simple persistence mechanism for stand-alone app. *Eiffel Users - Google Groups*.
[98] 2013. Structural type system. *Wikipedia, the free encyclopedia*.

Vita

Jimmy Jack Johnson

Education

PhD – Computer Science, University of Kentucky (expected Dec 2016)
MS – Computer Science, Louisiana State University in Shreveport (1989)
BS – Basic Academics, United States Air Force Academy (1984)

Military Education

Air Command and Staff College Associate Correspondence Program (1999)
Squadron Officer School (1991)
Squadron Officer School Correspondence Program (1988)

Experience

PhD candidate – University of KY (Jan 10 – present)
- Instructor, Computer Languages
- Instructor, C++ Data Structures (Berea College)
- Instructor, Algorithm Analysis and Design (Berea College)

ATS Construction – Lexington, KY (Feb 07 – Sep 09)
- Part 91 corporate pilot / PIC – CE-750
- Part 91 corporate pilot / PID – Bell 407

Self-employed – Pueblo, CO (Jan 06 – Jan 07)
- Part 91 contract pilot / PIC, G-100

201st Airlift Squadron, DC Air National Guard – Andrews AFB, MD (Oct 96 – Aug 05)
- C-38A (G-100) chief evaluator and instructor pilot
- CONUS and OCUNUS operational support airlift pilot
- Flight commander
- C-38 lead training officer
- Squadron scheduling officer
- MEDEVAC training officer
- Advance Instrument School graduate
- C-21A pilot

1st Helicopter Squadron, USAF – Andrews AFB, MD (Jan 94 – Sep 96)
- Flight commander, classified plans and programs
- UH-1N transport and MEDEVAC pilot

52nd Flying Training Squadron, USAF – Lubbock, TX (Jul 91 – Jan 94)
- T-1A initial cadre
- T-1A operation test and evaluation pilot
- T-1A instructor pilot
- FAA liaison and airspace officer
- T-37 pilot

71st Flying Training Wing, USAF – Barksdale AFB, LA (Aug 87 – Jul 91)
- Deputy Detachment Commander
- T-37 evaluator pilot
- T-37 instructor pilot
- Life support officer
- Computer programmer and computer systems manager

71st Flying Training Squadron, USAF – Vance AFB, OK (Aug 84 – Aug 87)

T-37 Instructor pilot
Squadron Scheduling Officer
Student pilot, Undergraduate Pilot Training

Computer Science Department – USAF Academy, CO (May 84 – Jul 84)
Computer programmer
Academic assistant

Special Awards

Meritorious Service Medal (2)
Air Force Aerial Achievement Medal
Air Force Commendation Medal (4)
Air Force Achievement Medal (2)
Air Force Outstanding Unit Award (7)
National Defense Service Medal with Star
Armed Forces Expeditionary Medal
Global War on Terrorism Service Medal
Air Force Longevity Service Award (6)
Armed Forces Reserve Medal with Mobilization
Small Arms Expert Marksmanship Award
Air Force Training Ribbon Award
Washington DC National Guard Meritorious Service Medal
Washington DC National Guard Homeland Defense Medal
Washington DC National Guard Emergency Service Medal
Washington DC Faithful Service Medal (3)
Washington DC Special Recognition Award