Profiling Initialisation Behaviour in Java

by

Stephen Frank Nelson

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2012

Abstract

Freshly created objects are a blank slate: their mutable state and their constant properties must be initialised before they can be used. Programming languages like Java typically support object initialisation by providing constructor methods. This thesis examines the actual initialisation of objects in real-world programs to determine whether constructor methods support the initialisation that programmers actually perform. Determining which object initialisation techniques are most popular and how they can be identified will allow language designers to better understand the needs of programmers, and give insights that VM designers could use to optimise the performance of language implementations, reduce memory consumption, and improve garbage collection behaviour.

Traditional profiling typically either focuses on timing, or uses sampling or heap snapshots to approximate whole program analysis. Classifying the behaviour of objects throughout their lifetime requires analysis of all program behaviour without approximation. This thesis presents two novel whole-program object profilers: one using purely class modification (#*prof*), and a hybrid approach utilising class modification and JVM support (*rprof*). #*prof* modifies programs using aspect-oriented programming tools to generate and aggregate data and examines objects that enter different collections to determine whether correlation exists between initialisation behaviour and the use of equality operators and collections. *rprof* confirms the results of an existing static analysis study of field initialisation using runtime analysis, and provides a novel study of object initialisation behaviour patterns.

Acknowledgments

I have been fortunate over the last six years to live and work in a great little city with a great little university and some wonderful people who have supported me through this long process and drawn out process.

First and foremost, I need to thank my supervisors Dr David Pearce and Dr James Noble who have been invaluable and extremely patient. Meetings with Dave have kept me grounded. I have immensely appreciate his down-to-earth advice and his practical approach to addressing problems, not to mention his willingness to treat me as a peer and discuss his research with me during our regular meetings — providing a much-needed sense of collaboration and companionship on the lonely journey towards a PhD. His support has ranging from incredibly rapidly covering any paper I had him in red ink, to logging onto Skype while on holiday on the other side of the world to check up on how I'm going. James has provided a foil to Dave's practical approach, encouraging me to consider the larger picture and aspire to change the world, while at the same time keeping me focused on the end goal of submitting a thesis. His insightful comments and 2am emails of encouragement and wisdom have helped me stay positive, and I will remember his advice that difficult meetings are best followed by chocolate. James in my mind is associated with witty culture references so it was a great disappointment that my attempt to emulate this with a Philip K. Dick reference in my final thesis draft came back annotated "–huh?".

My wife Melanie has been unfaltering in her love and support — no small feat as this PhD has encompassed our engagement, wedding planning, wedding, and four anniversaries. Melanie, I look forward to several more years of thesis angst as I attempt to replicate your love and support while you complete your PhD.

Many people have followed my thesis progress with the odd question about my progress, but no one has requested updates as frequently as my parents. Mum and Dad, thanks for all the encouragement. I know it's been difficult to explain to to people exactly what it is I'm doing but it's been amusing to hear the attempts, and I look forward to more analogies in the future. Your pride in my accom-

plishments and your support has kept me positive and moving forwards. I have especially appreciated the early morning walks around Wellington in the last few months.

I began this PhD among a strong community of peers in Memphis, and although things have changed over the years and most of you left academia long ago I'm sure that I will continue to look back on those few years as some of the best of my life. Andrew, Constantine, Hugh, and Vipul: you get a special mention in Chapter 4 for your help with coaxing *#prof* to spit out something useful on a deadline. Chris, Clare, Dave, and Joel, thanks for the help with proofreading. Gillian, you get an mention here as an honorary 'Memphisite', thanks for trying to teach me to write. Carlton, thanks for keeping me fit, both physically and mentally over the years. Finally, Chris Male, you've been a solid rock providing advice and feedback on everything from writing to programming to dealing with advisors to life in general. Thanks.

To Richard Bourne, thanks for supporting me both during my undergraduate years and then again in the last few years of my PhD by providing employment and encouraging my academic goals. Thanks too for taking a risk on me when I was a lowly student with little experience and a big ego. I hope that on balance it was worth the risk.

Finally, thanks to my examiners for their time, effort, and feedback, to Victoria University for numerous scholarships and grants for research and travel to conferences, and to the New Zealand government and people for providing financial support to higher education and especially student allowances for thesis students.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Objects, at the heart of the object-oriented paradigm, are fundamentally about modelling the world, breaking down complex problems into simple building blocks then combining them again to form complex systems. Unlike the world they mimic, objects in programming languages are not persistent. The chair you are sitting on is probably constructed from plastics, which are produced from refined oils, which are extracted from crude, which is pumped out of the ground, where it collected from the decomposed flesh of prehistoric organisms, etc etc back through billions of years. That particular chair is uniquely identifiable from all other chairs no matter how similar: they have a different history and different constituent parts. A typical object in a program has a much smaller timescale. Programs usually begin by bootstrapping from a clean slate. They request a chunk of virtual memory and rapidly carve out the chairs and other objects they need to catch up to the parts of physical world they are modelling. When the program has constructed sufficient objects to mimic the relevant real world processes it can begin performing its primary functions. Program objects do not have the rich history of real objects and are harder to distinguish than real objects, but programming languages assign program objects an *identity* so that they can be distinguished from all other objects — even if they are identical in every other way.

After a new object has been allocated programmers must populate it with initial values for mutable state and establish any constant properties of the object. Most mainstream languages use *constructor* methods to initialise the internal state of new objects. Constructor methods fill in the pertinent details from a real object's history that a virtual object lacks: names, dates, addresses, and preferences for electric livestock.

Like real objects, program objects have some properties that cannot or typi-

cally do not change; names and types for example. Other properties are mutable and can change freely; owner, colour, current location. Even though some properties should never change, programmers need to be able to establish their values. Java, like many other languages, represents immutable properties of objects as fixed-value fields – final fields – and permits constructor methods to initialise them.

Constructors and immutable fields are commonly used ways to initialise objects and represent constant properties, but they are not the only way to address this problem as Java enterprise experience shows. The prevalence of abstractions such as *JavaBeans*, *Dependency Injection* frameworks, *Factories*, *FactoryBean*s, and *BeanFactory*s in enterprise Java shows programmers battling with a lack of support for creating and persisting objects, and especially with initialising and configuring large groups of interacting objects.

This thesis examines the runtime behaviour of existing Java programs in an attempt to discover how Java programmers use Java to create objects, and how they distinguish between objects. We use runtime analysis to extract patterns of behaviour from running programs to examine how objects are initialised, how they are compared, and whether constructors, final fields, and equality methods are serving the needs of every-day Java programmers. Based on our analysis, we suggest some alternative language features for future languages and some potential optimisations that virtual machine authors could make.

1.1 Contributions

This thesis makes the following contributions:

- Definitions of observable behaviour patterns that identify the initialisation of immutable properties.

- A study of field and object initialisation behaviour, including confirmation of the results of an existing static study on field initialisation using runtime profiling.

- A study of the equality initialisation behaviour of classes and objects, especially in the context of collections.

- Two runtime profilers that track and analyse object initialisation behaviour.

The study of equality initialisation was published in TOOLS 2010 [68], while the confirmation of field initialisation behaviour was published in Runtime Verification 2012 [70].

1.2 Organisation

This thesis is organised as follows:

- Chapter 2 explores our motivations with this work and surveys the existing work in this domain.
- Chapter 3 defines behaviour patterns for observing and characterising initialisation.
- Chapter 4 presents a runtime profiling tool and a study of 30 different Java programs to determine how programmers implement object equality and whether this is related to the data the objects contain.
- Chapter 5 presents our design for a more capable tool for profiling running Java programs to extract and aggregate information about the behaviour of individual objects.
- Chapter 6 presents results obtained using this tool to examine the initialisation of fields and of objects in all 15 applications from a standard Java benchmark suite. It discusses the significance of these results and their implications for object-oriented language designers.
- Chapter 7 concludes this thesis with an overview of our results and the conclusions we draw.

Chapter 2

Background

This chapter begins with an overview of the work that motivated this thesis before introducing relevant concepts and related work.

2.1 Object relationships and equality

I began my doctoral studies with a grant to research "First Class Relationships for Object Oriented Programming Languages". A rather daunting task set, I began reading work by Bierman and Wren [12], and Pearce and Noble [80], which were relatively fresh and exciting when I began my doctoral studies in 2006. I then worked my way back through several decades of clusters of related work to find myself reading about the fundamentals of object-oriented programming and its origins in simulation in the 1960s. To summarise the first few years of my PhD and its influence on this thesis, this section begins with a whirlwind tour of relationships and object-oriented programming, then briefly covers my conclusions as a result of this research to motivate this thesis on initialisation and object profiling.

2.1.1 A brief history of objects, relationships, and identity

Object-oriented programming traces its origins back to simulation languages developed early in the history of computer science and digital simulation [72]. SIMULA showed how complex systems comprised of many concurrent processes could be implemented as a single program [32], breaking away from the prevailing programming models by allowing methods to maintain mutable state between calls. This simple change moved programs away from strict tree structures to more complex graph structures where data and behaviour was grouped to cre-

ate objects, creating the illusion of multiple components. In the process SIMULA created the concept of *identity* that has prevailed to this day: objects are located, exchanged, compared, and ultimately *identified* by their *reference*, an abstraction of their location in memory. Two objects that are equivalent in every way except their reference can be distinguished using the == operator.

While Nygaard et al. modelled interacting objects, Chen identified that some systems could be modelled more elegantly using entities and relationships [26]. Entities are composite values that nominate a *key* value that can distinguish them, rather than relying on memory location for identity as objects do. Relationships capture the interactions between entities explicitly, corresponding to the implicit interactions encoded in the behaviour of object-oriented programs.

Chen's model was extremely popular, eventually evolving into the relational databases that dominate data storage, while SIMULA's success at modelling collaborative behaviour has lead to the object-oriented paradigm dominating commercial software development for several decades. The conflict between entities distinguished by keys and objects distinguished by memory location, and explicit relationships vs implicit interactions has in part lead to the *object-relational impedance mismatch* problem identified in late '80s [62] that still remains an important concern for academic research and commercial software development [50].

Entities, relationships, and objects

Relationships as described by Chen were well used in databases and semantic data models but were, initially at least, ignored by object-oriented programming language designers until Rumbaugh proposed that they should be added to object-oriented languages. Rumbaugh recognised that objects correspond to the entities as described by Chen (so long as they are distinguished only by memory location), but the object-oriented paradigm has no explicit analog for relationships:

> It is possible to program [relationships] using existing object-oriented constructs, but only by writing a particular implementation in which the programmer is forced to specify details irrelevant to the logic of an application. *(Rumbaugh, [90])*

Rumbaugh introduced a relationship construct for object-oriented languages that adds relationships to object-oriented systems. His relationships are tuples of object references that are similar to relations in Chen's Entity-Relation diagrams; they are distinguished by the objects they refer to. Rumbaugh's relationships are

declared at the meta-level (class) as a tuple of class references that are instantiated as tuples of object references.

Rumbaugh and others implemented a language (DSM) that provided relationship support [95]. DSM was based on C, with extensions providing OO and Relationship features. The language did not achieve widespread use, possibly due to competition from C++ and Smalltalk, but Rumbaugh's work laid the foundations for much of the relationship literature that has followed.

Object-oriented modelling

The early 1990s saw the introduction of several design methodologies for object-oriented systems that modelled relationships explicitly [11, 17, 91]. Among other advantages, these mitigated many of the problems associated with the lack of relationships support. Object-oriented practitioners could model relationships at the design level without requiring explicit support at the language level. Programmers still implemented relationships without explicit support from their programming language, but they could refer to a model that abstracted the boilerplate code details of relationship implementation.

Two of the most popular object-oriented development systems were OMT (Rumbaugh et al.) [91] and the Booch Method (Booch) [17]. Both systems included a diagrammatic description of the classes in a system and their interactions, expressed as a graph where nodes represented classes or objects and edges represented relationships. Edges in both systems were more similar to the relation tables in Chen's work than to the behaviour implementations of relationships in object-oriented code because they were expressed independent from the objects/classes they relate.

Concurrently with the development of OMT and the Booch Method, Cunningham and Beck developed a brainstorming tool called the Class Responsibility Collaborator (CRC) cards [11]. The CRC card system models each class with a piece of card detailing the name of the class, its responsibilities, and the classes with which it collaborates. The collaborations define implicit unidirectional relationships between classes that are better suited for implementation as references than the predominantly bi-directional relationships in Booch and Rumbaugh's systems.

During the mid 1990s Rumbaugh, Booch and Jacobson produced a new modelling system, designed to unify their various techniques and provide an industry standard. The resulting system was the Unified Modelling Language (UML), which was accepted as a standard for modelling object-oriented systems by the

Object Modelling Group (OMG) in 1997 [92]. UML's approach to relationships followed OMT and Booch's method; modelling relationships as explicit entities independent of the participating classes [17, 91]. UML has subsequently become the dominant modelling language for object-oriented systems and the standard has been updated several times. One of the primary tools UML provides is the *class diagram*, which describes the structure of an object-oriented system as a collection of classes and relationships, based on the diagrams in OMT and Booch's method.

Explicit relationships

A binary relationship can be represented either as a pairs of references embedded in the two related objects, or as a set of pairs (that may be reified as a class). In the first case, there is burden placed on the programmer to ensure that the pairs of related references remain consistent and it is difficult to associate properties with the pairs because there is no clear place to store them. If links are reified as a class the programmer must take care to preserve the uniqueness of the links between objects (assuming this is desirable) as the link objects will gain an identity from their object reference that is independent of the objects they link, unlike Chen's relationships that are distinguished only by their target entities (objects). Both implementations create *strong coupling* [64] between the related classes, limiting their reuse potential and resulting in fragile code that is hard to maintain. Both implementations also obscure the relationship with implementation details of fields and methods spread among other fields and methods.

Some relationships are reasonably straight-forward to implement while other relationship implementations can be quite complex. Inspired by Gamma et al.'s "design patterns" [37] (a method for documenting complex techniques for solving recurring problems), Noble gives formulaic implementation strategies for five common types of relationships [71]. Noble claims that the prevalence of relationships in object-oriented design implies that relationships should be *easy to write*, *simple to represent*, and *immediately understood by later programmers* [71].

Østerbye identifies that relationship representation and use are largely orthogonal issues [76]. Relationships can be represented as properties of the participants, or independent of the participants (embedded references or sets of pairs) while independently allowing access directly from the class (via attributes) or through an external interface. Østerbye also demonstrates that relationships can be abstracted using a library to decouple representation from use [77].

Pearce and Noble also demonstrate a technique for abstracting relationship

implementation using a library [80]. They use *Aspects* to define a standard relationship interface and provide various relationship implementations. Like Østerbye's work, their aspect interface allows either internal or external implementation of relationships, however they limit programmers to external access: associations are accessed via a separate interface, rather than using fields of the participants.

Bierman and Wren present relationship support as a language extension: *RelJ* extends a small, functional subset of Java to provide first-class support for relationships [12]. RelJ allows programmers to define relationships between objects, specify attributes and methods on the relationships, and create relationship hierarchies. The authors provide a complete formalism for RelJ with a type system and small-step operational semantics for the system, however they do not provide an implementation. Balzer et al. build on RelJ to describe a relationship model that supports complex relationship constraints, further motivating explicit relationship support by demonstrating how such a system could increase expressiveness [10].

Vaziri et al. define a new language construct called a relation that has a dependent identity similar to Chen's relationships [110]. Though not designed specifically to support relationships, relation types are declared as relations between existing objects and have an identity that may be computed from one or more other objects/relations.

2.1.2 What are relationships?

UML Class diagrams and most other object-oriented modelling tools are concerned with meta-level objects, or classes. Although not all object-oriented languages are class-based, object modelling techniques seldom consider individual objects but rather aggregations of objects as classes with the same fields and the same behaviour. The same modelling techniques consider relationships (associations) at the same level as classes so it seems apparent that UML relationships are meta-level constructs. This is not consistent with Chen's ER model where 'relations' are expressed at the same meta-level as entities: ER Entities are analogous to objects and entity relations to object-oriented classes, but object-oriented relationships are analogous to ER *relationship relations*, rather than to ER relationships. In the terminology used by Bierman and Wren, the construct at the same meta-level as an object is a relationship *link*, which is analogous to an ER relationship (or relationship tuple). To further cloud the issue, Pearce and Noble, Østerbye and Balzer et al. also discuss relationships as sets of links at the same meta-level as

objects, but without a clear analogue from the class/object hierarchy[80, 77, 10]!

Relationships are about identity

Confused by the various names assigned to concepts in relationship literature, we referred again to Chen's ER model. It is elegant in its simplicity. Entities are discrete units of information expressed as a fixed number of named, typed attributes (a tuple). One of those attributes is the designated *primary key* that distinguishes one entity from all other entities of the same type. Relationships are similar to entities, they are also represented by a fixed number of named, typed, attributes (a tuple). Unlike entities, relationships are identified by composite primary key comprised of multiple attributes that refer to the primary keys of other entities.

ER entities and relationships diverge from objects in one fundamental way: entities and relationships have an explicit identity that is part of the model. Objects have an implicit identity based on language references that are not generally modifiable by the programmer. ER Relationships, where identity is derived from a dependency on other objects, just cannot be expressed in languages like Java without resorting to programmer-defined equals methods to mimic the composite primary-key behaviour ER Relationships require.

During the first few years of my PhD I had the pleasure of meeting and discussing my work with all of the Gavin Bierman and Alasdair Wren, Kasper Østerbye, Stephanie Balzer, and many of the other researchers working on explicit relationship support (not to mention David Pearce and James Noble, my advisers). It was from talking with Frank Tip, coauthor of the Relation Types work mentioned previously, that I became fascinated with the implications of adding support for ER-style delegated identity to object-oriented languages to support relationships. His work with Vaziri et al. demonstrates that ER-style relationships could be expressed in object-oriented programs by automatically generating constructors and equality methods and based on a subset of an object's immutable fields (the primary keys) then using *hash-consing* to ensure that a particular combination of keys was unique in a program [36] — using a global hash map to ensure that only one object exists with a particular permutation of immutable field values (key fields).

To implement hash-consing Vaziri et al. maintained *extent-sets* that behave remarkably similarly to the relationships as sets of links. As Vaziri et al. observed, these extent sets can cause garbage collection problems (when is it safe to collect an object, if state can always be retrieved by creating an instance with the same identity again?), or result in a new class of runtime errors that are hard to prevent.

For example, suppose that a popular library uses a relation type with a composed identity (and the runtime creates an extent-set to back it). Now suppose that the library is used by two disparate components within a program and both try to use a relation with the same composite identity: either they will both be accessing and potentially modifying the same tuple (breaking encapsulation) or one instance will receive an unexpected error and be unable to create the appropriate relation.

Unhappy with the implications of using global extent-sets we considered the possibilities resulting from adding an operator like Baker's *EGAL* [8] to object-oriented languages: an operator that compares entirely immutable objects using value comparisons, recursing until it encounters any non-immutable object then reverting to reference comparison (discussed in more depth in Section 2.2.2). This would allow programmers to easily create primary key objects for implementing relationships without requiring whole-program extent-sets to maintain their uniqueness (they are indistinguishable because they have no mutable state). We considered implementing relationships sets of primary key objects or maps from primary key objects to value objects containing mutable state for the relationship links, where the programmer could create relationship instances similarly to how they create collections, without requiring whole-program extent-sets. In fact, as we were able to demonstrate, programmers can already implement relationships in this style using collections and equals methods.

To investigate whether programmers already implement relationships using collections and equals methods we created *#prof*, the first of two profilers presented in this thesis. *#prof* was initially created to identify objects without any mutable state and with equality defined using all of their fields that were used as map keys. *#prof* is presented in Chapter 4. *#prof* did identify objects that behaved in a way we considered consistent with relationships, but it was hard to generalise this behaviour. We concluded that we needed a second profiler, *rprof*, that was more capable of tracking object behaviour. *rprof* is presented in Chapter 5, and its name is an abbreviation for *relationship* profiler, though in the end we did not use it to profile relationships.

During the development of *#prof* and later *rprof* my PhD focus gradually shifted from relationships and equality to initialisation. While creating these profilers we became aware that programmers initialised objects in several different ways and we began classifying them. This lead to this thesis on profiling object initialisation, with a secondary focus on profiling collection use and equality/identity inspired by our early work on relationships.

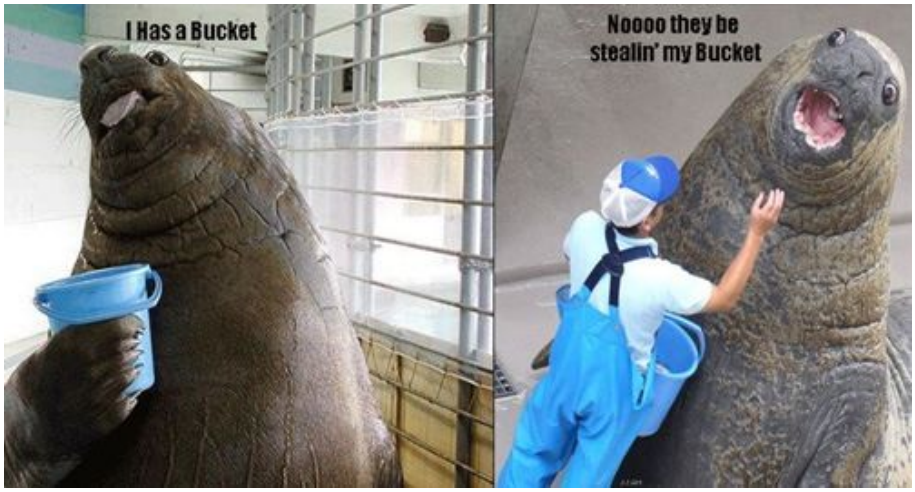


Image copyright `icanhascheezburger.com`

Figure 2.1: The difference between identity and equality can be very important in particular situations. For example, Java collections (buckets) define equals based on their contents rather than their identity (reference). If their contents change then their equals method will behave differently. This is fine if the user is using equals to comparing collections based on their equivalence to other collections, but can be undesirable they are particularly attached to a particular collection (bucket).

2.2 Object identity and equality

Object-oriented languages typically provide two types of equivalence for objects: *identity* and *equality*. Identity is the best operator for solving problems that require a specific object, *"Is this person Dave?"*, *"Is this my bucket?"*. Identity operators rely on object references, analogous to names or addresses that will locate a particular person among many. Equality operators are best for solving problems that require an object with specific properties: *"Does this person teach computer science?"*, *"Does this bucket contain fish?"*. Equality operators can rely on object references, but they can also rely on object properties to locate an object that is suitable for the task. It is important to use the correct operator for the correct task (see Figure 2.1).

Java's identity comparison uses object references to distinguish objects, references that also allow access to an object's state and behaviour. Java hides the implementation details of object references but provides two operators for performing reference comparisons: == and System.identityHashCode().

Equality can be determined using many approaches. Java facilitates this by providing a default equality based on references and allowing programmers to overwrite the default for particular classes. Java's equality is implemented by the equals and hashCode methods. Both methods are overridable methods defined on the root of the class hierarchy.

Java references are good for finding objects that reside within a program's memory but they are not useful for locating objects that are external to the program, such as "the object stored on my external disk drive", or "the student with database ID 4025". Java's identity operators cannot be modified by programmers so it is common for Java persistence and serialisation frameworks to recommend that programmers implement 'external' references by modifying equals and hashCode (e.g. [20, 79]). As a consequence, equals and hashCode could be implemented by programmers for two purposes: to determine whether objects are equal, or to determine whether objects represent the same object.

This section provides a survey of object identity and equality in languages like Java.

2.2.1 Identity

What is an object?

According to Booch:

> *"An object is an entity that has state, behaviour, and identity."* [17]

State and behaviour are familiar concepts to any programmer or mathematician, but what of identity?

Khoshafian and Copeland tell us that:

> *"Identity is that property of an object which distinguishes each object from all others."* [53]

In object-oriented programming languages such as Java that distinguishing feature is clear:

> "The term *identity* is used for reference equality: If two references are identical, then == between the two will be **true**." [5]

Java's conflation of identity and reference equality is elegant (and common in modern programming languages): object state and behaviour must be accessed by reference, ensuring that neither state nor behaviour can allow a wily programmer to distinguish between supposedly identical (==) objects. The discussion continues, describing equivalence in Java:

> "The term *equivalence* describes value equality—objects that may or may not be identical, but for which equals will return **true**." [5]

2.2.2 User-defined equivalence operators

SIMULA supported identity using reference comparisons (= in SIMULA, == in C-like languages) [14], but reference comparisons are not very useful for comparing entities that have an explicit identity from outside the program (a *key* value in Chen's terminology). Programmers were of course free to implement a method for comparing their objects using object-field values, but Smalltalk felt that this was sufficiently important to justify adding an additional operator. In Smalltalk, the == operator compares references while the = operator compares the operands by value; == asks whether the operands *are the same object* while = asks whether the operands *represent the same component* [40]. Smalltalk's == cannot be overridden (though other languages allow this) while = can be overridden to implement equality derived from object-field values.

Smalltalk's operators remain essentially unchanged in Java, where == compares objects by reference and equals allows users to define an arbitrary equivalence relation, but the notion of what they should represent has changed slightly. Java describes equals as an equivalence whereas Smalltalk's description indicates that these objects should represent the same entity. This means that Java objects that are *equal* could be distinguished by their behaviour while still meeting Java's definition: they don't "represent the same component". The actual behaviour of both Smalltalk's = and Java's equals relies on the programmers: both operators default to == and allow programmers to override the default, without enforcing particular behaviour. Programmers are free to violate the expectations of the language authors.

Many languages have experimented with different operators for identity and equality. Smalltalk has == for identity and = for equality, while Common Lisp has no less than five different equality operators: `eq`, `eql`, `equal`, `equalp`, `=`, along with a range of type specific functions such as `char-equal`, `string-equal`, and `tree-equal` [103]. Lisp's proclivity for equivalence operators may well be the reason Baker developed the EGAL operator, his 1993 paper presented a very comprehensive conceptual discussion of equality framed in terms of Lisp and its many operators [8]. Baker identified that object-oriented programs frequently contain a mix of immutable and mutable objects. He observed that *equality* comparisons are most useful for performing value-based comparisons of immutable objects that cannot change, while *identity* comparisons are necessary to compare mutable objects (equivalence comparisons should be stable, that is, two objects that are equivalent must be indistinguishable, so current state is not sufficient for comparing mutable objects). EGAL recursively performs an *equality* comparison for

immutable objects and an *identity* comparison for mutable objects.

Baker was not the first to identify the need for a distinction between objects and values in programming languages, MacLennan identified that objects have identity and mutable state, while values are immutable and any identity they possess is merely an implementation detail [61].

Object identity as an abstract concept for programming languages was first studied by Khoshafian and Copeland, who compared the identity mechanisms in different systems including databases and object-oriented languages, characterising not only object identity, but also shallow equality, deep equality, and the ability to merge objects [53]. Their work explicitly addresses tuples and sets as they attempt to cover both object-oriented languages and databases.

Grogono and Sakkinen discuss equality and object-copying and identify that the copy of an object should be equal to the object from which it was copied [43]. They propose four different equalities: identity, shallow equality; deep equality; and a structural equality that can distinguish between cycles and their unfoldings as trees.

Vaziri et al.'s Relation Types are notable not only for their relevance to relationships, but also for their equality based on multiple nominated key values [110]. Relation Types use hash-consing to ensure that each of their instances are unique as far as values for these key fields are concerned. Relation Types represent a different approach to Baker's EGAL: relations can have mutable state so long as their key values are immutable, allowing a mixture of mutable and immutable state in a single object while also allowing identity based on key values rather than memory address.

Hovemeyer and Pugh examine the behaviour of programmer-defined equality methods to demonstrate that a relatively simple static analysis can detect bugs, such as incorrect covariant signatures for `equals` or missing `hashCode` methods. They present results from static analysis of six Java applications [48]. Rupakheti and Hou also examine bugs in existing Java programs; they identify several recurring problems with the definition of equality using an observational study [93]. These results demonstrate that programmers make mistakes when implementing equality behaviour that are not immediately apparent when running the programs, and suggest that languages like Java could do a better job of communicating these contracts to programmers and ensuring correct implementations.

Costanza suggests that object identities should be explicit *comparands*, similar to entity-relational 'keys', or 'ids' [31]. Comparands allow an object's identity to be changed, allowing objects to share identities. For example, a decorator or

proxy could modify its identity so that comparisons between the proxy and the target object share an identity. Techniques similar to comparands are often used for identity comparisons between objects that are shared between networked applications, or for implementing identity hash codes in the presence of copying garbage collectors [30].

2.3 Immutability and initialisation

2.3.1 Immutability

Languages that allow mixing objects and values should ensure that the values cannot accidentally gain identity: a point value that can change can be distinguished from other points that nominally have the same value by observing those changes. As a consequence, many object-oriented languages support some form of immutability annotation.

C++ inherits the '**const**' modifier from C and applies it to field declarations. This modifier documents the programmer's intention that particular fields or references should not be modified. Java inherits this concept as the field-level **final** annotation, though it allows initialisation anywhere inside the constructor rather than using initialiser lists like C++.

Other languages support immutability at the class level: CLU [60] has immutable versions of primitive data structures, and Scala's standard libraries provide both mutable and immutable versions of most collections [73].

Bloch [16] advises Java programmers to "minimise mutability": to use immutable objects wherever possible, and to ensure objects are fully initialised by constructors. Bloch identifies that immutable classes are "easier to design, implement, and use" and "less prone to errors" than mutable classes. Certainly, it is easier to predict the behaviour of equality methods for immutable classes, as immutable objects that compare equal will remain equal for all future program states. More generally, programmers using immutable classes do not need to worry about calling methods that may unexpectedly modify state.

Implementing immutable classes in Java is not straightforward. In "Effective Java" [16], Bloch describes five steps required to implement immutable classes: don't provide methods that can modify state, ensure the class cannot be extended, make all fields final, make all fields private, and ensure *exclusive access* to any mutable components. This poses a significant burden on programmers implementing complex objects, for example, an object with cyclical references to other

objects of the same class cannot be implemented using final fields (step 3) or without mutator methods or non-private fields (steps 1 and 4).

Various researchers have proposed type-system approaches for maintaining class or object immutability. For example, Zibin et al.'s IGJ language provides parameterised mutability at both object and class level for a Java-like language [112]. IGJ enforces these restrictions statically using a generics-based type system, allowing a class to parameterise the mutability of an individual field, e.g. an IGJ map class can require its keys to be immutable while permitting its values to be either mutable or immutable.

Ensuring exclusive access to sub-components (step 5 of Bloch's requirements for immutable objects) requires access to the sub-components internal state (to ensure exclusive access via copying) or requires that the sub-component is created by the object. One proposed approach for controlling access to sub-components with programming language support is *ownership* [2, 29, 47, 85], which has also been applied to maintaining object immutability [78, 113].

2.3.2 Initialisation

Imperative languages that support immutable fields or objects must allow some way for the programmer to initialise those fields or objects. Most object-oriented languages accomplish this using constructor methods that run as soon as the object is created. C++ uses initialiser lists, a special attribute of constructor methods, to initialise **const** fields. Java and C♯ allow constructor methods to modify each final (readonly) field at most once per constructor method and enforce correct behaviour using static analysis.

It is sometimes necessary or desirable to modify inherently immutable fields after an object's constructor has run. Cyclic references in immutable fields cannot be established using constructor methods, and constructor methods cannot be virtual – the constructor must be called by name and cannot be parameterised or replaced by a subtype at runtime – limiting their usefulness in enterprise frameworks that handle dependency injection and persistence.

Programmers are free to implement *post-constructor initialisers* – methods that initialise fields after the constructor has returned – in Java, but Java cannot statically check that those fields are not accessed before they are initialised. This places the burden on the programmer to ensure correct implementation without assistance from the language, or face potential runtime errors.

Fähndrich and Xia's *Delayed Types* use an ownership-style type system of dynamically nested regions to allow programs to initialise fields even after construc-

tors have run while preventing programs from accessing uninitialised fields [35]. Haack and Poll have shown how this can be applied specifically to immutability [45], while Leino et al. demonstrate that ownership transfer can achieve a similar result [58]. Qi and Myers' Masked Types [87] use of type-states to address post-constructor initialisation by incorporating a list of uninitialised (or *masked*) fields into object types. Each of these approaches demonstrates that *post-constructor* initialisation can be statically verified to ensure correct access to potentially uninitialised state, but none of them determine whether or to what extent these approaches would be useful to Java programmers.

Unkel and Lam [109] describe an interesting approach to generalising Java's **final** field behaviour. A *stationary* field is a field that has no observable writes, that is, all writes precede all reads. A stationary field may be initialised multiple times during or after the constructor but is not modified once it has been read. They present a static analysis study of 26 Java applications, as well as dynamic analysis of 9 small benchmarks, and find that 40–60% of Java fields are stationary, and that at least 14–21% could not be initialised during the constructor. This demonstrates a need for static verification of post-constructor initialisation.

Porat et al. [83] conducted a similar analysis looking for "deeply immutable" fields (where neither the field itself nor any object reachable from that field is modified after the object's constructor completes) and found that around 60% of `static` fields were immutable.

Pechtchanski et al. [82] performed a dynamic (runtime) analysis of small programs to detect immutably fields and consequently eliminate unnecessary duplicate reads. They conclude that as many as 81% of field declarations are actually immutable, yielding a reduction in the number of field reads in their sample programs ranging from 33% to 99%. These results show the enormous potential benefits that could be obtained from better representation of programmer intentions regarding immutable fields.

2.4 Runtime analysis

This thesis deals extensively with the analysis of existing Java programs using runtime analysis, so in this section we consider literature techniques for extracting information from running programs. Static analysis, an equally valid approach to corpus analysis, was not used in this thesis because we are primarily concerned with aggregate properties of object behaviour.

Like static analysis studies, dynamic analyses require a corpus of programs

to analyse. Static studies typically select a group of open source programs, or analyse a selection of programs that have been analysed by other researchers. Poor corpus selection undermines the generalisability of results, as any trends observed by the researchers may be anomalies of the particular corpus. For this reason, and to aid reproducibility and allow similar research to be directly compared, it is best to use an established corpus of programs that have been subjected to peer review. Unfortunately, as Tempero et al. observe [106], there are few examples of standardised program corpora available. Their *Qualitas Corpus* is one of the largest and most actively maintained.

In addition to program corpus selection, dynamic studies must execute the programs to obtain results. This requires providing inputs, or *workloads*, to the programs that trigger their behaviour. Like corpus selection, it is important for generalisability that the program inputs are chosen without bias and that they are reproducible by other researchers. While language corpora are rare, benchmark suites, or collections of programs with workloads are relatively common. Benchmark suites for Java are primarily used for measuring JVM performance and for acceptance testing. Unfortunately, this can limit their generalisability to regular programs.

An additional concern for all dymanic analysis when compared to static analysis is *completeness*. While static analyses can demonstrate that a particular property cannot occur, in general dynamic analysis can only determine that the behaviour did or did not occur for the particular program invocation measured [9]. While dynamic analysis can never match static analysis in this regard, by its nature dynamic analysis is better at identifying the 'important' parts of the program: those parts that actually execute! Nevertheless, this property relies on the selection of good inputs: if the program's functionality is not well exercised by the workload it is given then it is not necessarily clear that the 'important' parts of the program even ran. One way to measure how 'good' a workload is for exercising a program's functionality is to identify which program statements were executed. This is known as measuring statement coverage. Brown et al. measured the statement coverage of the workloads provided by several benchmark suites and concluded that the workloads for most of the benchmarks resulted in good (greater than 80%) statement coverage [19].

2.4.1 Program corpuses for runtime analysis

This section considers the most commonly used Java benchmark suites.

SPECjvm98

SPECjvm98 is one the most frequently used benchmark suites for both industry and academia [100]. It was used by both Unkel and Lam [109] and Pechtchanski and Sarkar [82] for their runtime studies. SPECjvm98 consists of eight relatively small Java programs and will run on any Java Virtual Machine from JDK 1.1 onwards. SPECjvm98 is now deprecated in favour of SPECjvm2008, and both the size of the benchmarks and the age of the code limit its generalisability to modern Java programs.

SPECjbb2005

SPECjbb2005 is a single simulation application designed to simulate a typical enterprise web-based ordering application [101]. As an example of a realistic workload it is extremely useful for JVM implementors and hardware designers, but as it consists of only one artificial application and it is not freely available, it is not particularly useful for academic research.

Java Grande

The *Java Grande Benchmarks* consist of several suites of benchmarks with 'large' demands: single-threaded, multi-threaded, distributed, and benchmarks for performance comparisons with C [22, 51]. Most of these benchmarks focus on solving academic problems in isolation, such as arithmetic problems, Fourier coefficient analysis, encryption, and raytracing. These benchmarks are designed for measuring the performance of JVMs for solving particular problems and as such are not good candidates for generating generalisable results.

DaCapo

The DaCapo benchmark suite consists of 14 open source, 'real' applications [15]. The suite was designed for programming language, memory management and computer architecture research and the authors included generalisability as a stated goal for their selection. The DaCapo benchmarks have gained significant adoption since their first release in 2006, and their latest release (2009) includes some relatively large programs.

SPECjvm2008

SPECjvm2008 is a significantly updated version of SPECjvm98, consisting of 10 benchmarks: a mix of 'real' programs and artificial benchmarks [102]. It is freely available and includes source code, making it a reasonable choice for academic research.

2.4.2 General profiling

Profiling – the dynamic (runtime) analysis of program behaviour – has been studied extensively as a tool for improving program performance, detecting bugs, and identifying behaviour patterns. Examining program execution can produce massive amounts of data if every program state must be recorded, so every profiling technique must include some approach for condensing or limiting this data.

The most common approach to data aggregation for profiling is sampling, where programs are interrupted at regular intervals and relevant properties are extracted. There are many examples of profiling techniques that focus on monitoring standard metrics, such as execution time (e.g. [42, 3, 4, 111, 23, 18, 99]) and heap usage (e.g. [89, 114, 57, 81]). A well-known example is `gprof` [42], that uses a combination of CPU sampling and instrumentation to approximate a call-path profile; that is, it reports time spent by each method along with a distribution of that incurred by its callees.

The increasing popularity of virtual machines and just-in-time compilation, thanks to the popularity of languages like Java, C♯, and JavaScript, has motivated significant investment in profiling research (e.g. [111, 7, 6, 55]). For example, Whaley described a system for profiling method execution time in a JIT using a compact representation of calling context [111] while Arnold et al. use profiling information to guide JIT optimisations, such as method inlining [6].

Binder and Hulaas present an interesting technique for precise flow profiling [13]. They used byte code modification to augment every method with an additional parameter for maintaining precise stack trace information and a thread local cache for storing results. They produce an exact count of executed byte codes for each possible stack trace. Their major concerns are producing an exact analysis of execution cost (measured in byte codes) without interference from the profiler (measurement perturbation), as well as profiler portability: they reject the use of JVMTI because it is not portable across platforms and JVMs. Using a tree structure for tracking method call information is an interesting idea, but it's not particularly relevant to our analysis that doesn't really deal with method track-

ing. They do whole program analysis by modifying byte codes prior to execution, which does not work well with external class-loaders. From their discussion it seems they have an additional approach for using JVMTI or an agent to modify byte codes but it is not described in the paper.

Sastry et al. present a hybrid approach to profiling using a combination of hardware and software to compress the data stream emitted by a profiler then analyse the compressed stream [94].

2.4.3 Object equality profiling

Marinov and O'Callahan present an approach to reducing program memory use by identifying and merging equivalent objects [63]. Their object equality is extremely relevant to the Baker's EGAL and a generalisation of Unkel and Lam's stationary fields to objects: an object may be a candidate for merging if, for all future program states, the object is never written to, never used in an == comparison, never used in a System.idenityHashCode() call, and never used as a monitor for synchronisation. Assuming the object is a merge candidate, it may be unified with other merge candidates if all fields of the object are merge-able. Their tool analyses several programs from the SPECjvm98 benchmark suite and samples program heap activity then applies a post-mortem analysis once execution is complete. Their analysis examines the object graph of snapshots, searching for sub-graphs that are structurally equivalent (isomorphic). They concluded that several programs from their suite exhibited large numbers of equivalent objects.

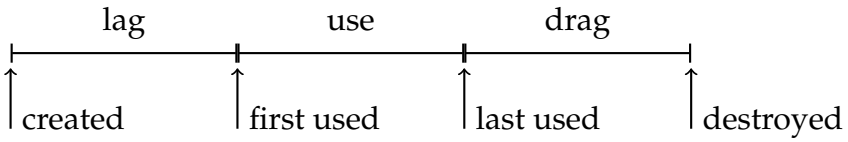
2.4.4 Object lifecycle profiling

Numerous works have focused on profiling object lifetimes for pretenuring in virtual machines (e.g. [1, 27, 52, 98]). Hirzel et al. studied a suite of benchmarks and concluded that object connectivity correlates strongly with object lifetime [46]. Contrasting with this, others have shown how stack state at the point of object allocation correlates with object lifetime [49]. Singer et al. studied a small benchmark suite in an effort to identify good predictions of long-lived objects [98]. Chen et al. consider the lifetime of object fields, rather than whole objects, since a field may not be active for the duration of its enclosing object's life; thus, fields with disjoint lifetimes can occupy the same memory, thereby reducing object footprint [25]. Similar work studied field lifetimes for the SpecJVM98 benchmark suite, and found on average a 14% reduction in heap space was possible [44].

Shankar et al. profiled Java programs in an effort to identify short-live objects

suitable for stack allocation [97]. Dieckmann and Hözle performed a detailed study of the allocation behaviour of the SPECjvm98 benchmarks [34]. Pearce et al. evaluated AspectJ as a profiling platform by considering different case studies [81]. They considered profiling execution time, heap usage, object lifetime and more.

Röjemo and Runciman introduced the notions of *lag*, *drag* and *use* to describe the lifecycle of *memory cells* in functional language implementations [89]:



Lag is the period after a memory cell is allocated and before it is first used in a computation. Drag is the period after a memory cell is last used but before it is garbage collected. Anything else is use. These concepts are interesting for measuring the causes of unused memory in functional programs, but this approach is are equally applicable for describing the 'life-cycle' of objects in an imperative language. They focused on improving memory consumption in Haskell programs and relied upon compiler support to enable profiling.

Shaham et al. measure the *drag time* for Java Objects: the time between last use and release by the system for garbage collection [96]. They maintain a record for each object containing a 'last used' timestamp that is updated each time an object is accessed. When the object is finally collected (collection is triggered every 100ms), they emit the drag time for that object. By multiplying drag with object size and sorting the result they create a list of the worst offenders that they used to manually optimise programs to reduce 'dragged' objects and so general memory use. This is an interesting application of similar profiling techniques to the techniques we used in Chapter 3. It would be interesting to use our Chapter 4 general profiler to verify their results.

2.4.5 JVM profiling support

Liang and Viswanathan introduced JVMPI, a general purpose profiling tool capable of measuring CPU time in a thread-aware manner, tracking object allocation, garbage collection, monitor contention and class loading [59]. JVMPI was designed to be the basis for JVM-independent profilers that could be distributed by vendors for use by programmers. JVMPI has been succeeded by JVMTI in more recent versions of Java [74, 75].

2.4.6 Heap analysis

Mitchell presented a novel approach to compacting the typically huge amounts of data generated during profiling [65]. His approach exploits the dominates relation for objects in the heaps.

Potanin et al. used the JVMPI interface [59] to profile object graphs in Java programs, concluding that these exhibit the property of being scale-free [86]. In particular, they observed a power-law distribution for edge degrees in the object graph of large programs: some objects were very highly connected, whilst most had low connectivity.

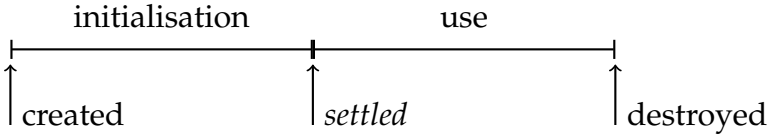
Chapter 3

Behaviour Patterns for the Initialisation of State and Equality

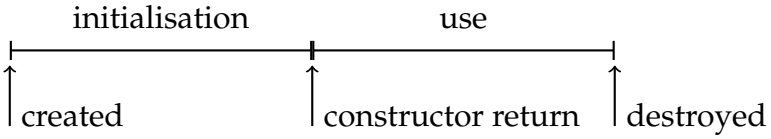
3.1 Settling

Imperative object-oriented languages such as Java use special *constructor* methods to allow freshly created objects to initialise their properties, including *immutable* or *constant* properties (e.g. **final** fields in Java, **const** in C++). Some objects have properties that cannot be initialised immediately after the object is created but are nonetheless constant (e.g. data structures with cyclical references).

Röjemo and Runciman introduced the notions of *lag*, *use*, and *drag* to describe the lifecycle of *memory cells* in functional language implementations [89] (see Section 2.4.4). We can apply similar concepts to describe the life-cycle and in particular the initialisation of objects in Java programs. Rather than considering lag, use, and drag, we consider *initialisation* and *use*, separated by *settling* events:



The intuition for settling events is that constant properties of an object must be initialised after the object is created but before they are used. A constant object property might fluctuate after the object is created, but at some point it will settle at a particular value. After that point it will not change again. For example, an object property that is settled by the end of the object's constructor could be illustrated as follows:



Different object properties may have different settling events. For example, objects that are stored in databases must be *persisted*, at which point the database will assign them a unique ID (e.g. [79]). Their ID is a constant property, but it cannot be allocated until after the object's constructor has returned.

This chapter defines five different *settling* events in an object's life-cycle: observable events that could demarcate the end of an object property's initialisation period, then describes how these events can be used to characterise program behaviour.

3.2 Settling events

An *object-field* – the value of a specific field of a specific object – may change throughout the existence of its object or it may settle on a particular value. The particular point in a program that an object-field settles is interesting because it represents the point at which the object-field was fully initialised. By generalising object-field behaviour to fields, objects, and classes we can use set theory concepts to divide program entities into sets based on their behaviour, and determine the relative importance of particular events by examining the relative sizes of the sets.

This section provides behavioural definitions of particular settling events that may occur in Java programs. Each event is introduced by describing how it can be observed for a particular object-field, then extended to objects, classes, and fields. Figure 3.1 shows a Java class definition that is used for examples in this section.

3.2.1 Constructor-settled (C)

All writes to a *constructor-settled* (C) object-field occur before its object's constructor method returns. Constructor-settled captures the intuitive notion of initialisation by constructor:

```
class Point {
   int x;
   int y;
   Point() {}
   Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int hashCode() {
      return x ^ y;
   }
   public boolean equals(Point other) {
      return x == other.x && y == other.y;
   }
}
```

Figure 3.1: A Point class. Java equals methods should take an Object parameter, this method takes a Point and assumes that it is not **null** for simplicity.

```
Point p = new Point(1,2);
p = null;
```

p.x is constructor-settled

```
Point p = new Point();
p.x = 1;
p = null;
```

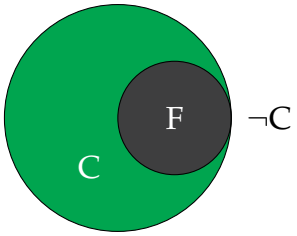
p.x is not constructor-settled

On the left, we see a point created and initialised using its constructor. The second line of code discards the only reference to the point so we can conclude that the point's object-fields are only written from the constructor. On the right we see another instance of the same class, but this time the instance's x object-field is modified after the constructor so it is not constructor-settled.

Final (F)

If a Java *field declaration* – the static description of a field in a Java class file – is annotated with final then it cannot be written to outside the constructor, so object-fields corresponding to final fields must be constructor-settled. final is a special case of settled behaviour because it is declared in code, unlike all other settled behaviours that are dependent on context: a class may be constructor-settled in one program but not constructor-settled in another depending on usage, but properties that are declared final must always be final.

We can plot the set of object-fields from a hypothetical program as a *Venn diagram* showing the subset relationship between final and constructor-settled object-fields:



Constructor-settled fields

A field declaration is constructor-settled if all the object-fields resulting from that declaration are constructor-settled. All writes to a constructor-settled field must originate in a constructor method in the field's class's hierarchy or from a method called from a constructor method. final fields, however, can only be written from a constructor method of the class that declares the field (in addition, Java final object-fields can only be written once whereas constructor-settled object-fields can be written multiple times).

Constructor-settled objects

An object is constructor-settled if all object-fields of the object are constructor-settled. All field writes to a constructor-settled object must occur before the object's constructor returns.

We do not consider static fields for object aggregations.

Constructor-settled classes

A class is constructor-settled if all objects of exactly that class are constructor-settled. We summarise objects by their declared class, so the subclasses of a constructor-settled class is not necessarily constructor-settled and vice versa. This definition represents a decision to produce class behaviour categorisations by aggregating objects rather than by aggregating field declarations. A subclass that is not constructor-settled may change a field declared in its super-class, causing that field declaration to be not-constructor-settled. This will not affect whether the superclass is constructor-settled. Consequently, a class that is constructor-settled may have field declarations that are not constructor-settled. We feel that this definition is consistent with the general focus of this thesis on dynamic rather than static properties, objects rather than field declarations.

We do not consider static fields for class aggregations.

3.2.2 Read-settled (R)

All writes to a *read-settled* (R) object-field occur before the first field read of that object-field. Read-settled captures the intuitive notation of *observably immutable*: properties that are never observed to change — sequential reads never return different values.

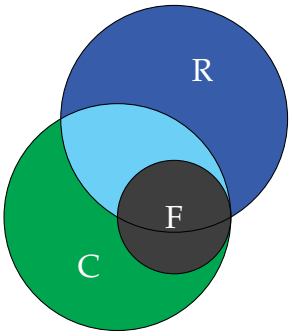
```
Point p = new Point(1,2);
p.x = 1;
p.y = p.x;
p = null;
```

p.x is read-settled

```
Point p = new Point(1,2);
p.y = p.x;
p.x = 1;
p = null;
```

p.x is not read-settled

Read-settled object-fields are not necessarily either a super-set or a sub-set of constructor-settled object-fields, or even of final object-fields. Java final fields can be read by methods called from a constructor before they have been assigned (this is also true of **const** fields in C++).



Read-settled fields

A field declaration is read-settled if all the object-fields resulting from that declaration are read-settled. In other words, all writes to a read-settled field must occur before reads from the same field of the same object. A field declaration is not read-settled if there exists an object of any class that writes to the field after it has been read.

Read-settled field declarations correspond to the *stationary* fields observed by Unkel and Lam, who characterise stationary fields as fields that are never written after they are read [109].

Read-settled objects

An object is read-settled if all object-fields of the object are read-settled.

Read-settled classes

A class is read-settled if all objects of that class are read-settled.

3.2.3 Whole-object read-settled (oR)

Whole-object read-settled (oR) is a special case of read-settled objects that uses an object-centric interpretation of read-settled: an object is object-read-settled if all field writes to its object-fields occur before the first field read from *any* object-field of the object. This class of objects is a subset of read-settled objects.

3.2.4 Write-settled (W)

A *write-settled* (W) object-field may only be written to once. This is similar to the restriction Java 8 uses to characterise *effectively-final* local variables for lambda-expressions [39].

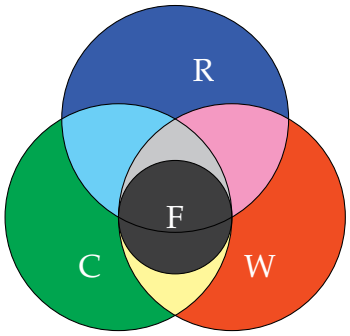
```
Point p = new Point();
p.y = p.x;
p.x = 1;
p = null;
```

p.x is write-settled

```
Point p = new Point();
p.x = p.y;
p.x = 1;
p = null;
```

p.x is not write-settled

While Write-settled object-fields are not necessarily either constructor-settled or read-settled, final object-fields are always write-settled.



Write-settled fields

A field declaration is write-settled if all the object-fields resulting from that declaration are write-settled. A field declaration is not write-settled if there exists an object of any class that has more than one write to that field.

Field declarations that are constructor-settled and write-settled (CW) correspond to the *undeclared-final* fields observed by Unkel and Lam: fields that are written at most once during the constructor and consequently could be labelled final without changing the program [109].[1]

Write-settled objects

An object is write-settled if all object-fields of the object are write-settled.

Write-settled classes

A class is write-settled if all objects of that class are write-settled.

3.2.5 Equals-settled (E)

All writes to an *equals-settled* (E) object-field occur before the first call to one of the object's equality methods (equals and hashCode). Equals-settled is only applicable to objects whose equals or hashCode method is called. Equals-settled captures the behaviour of object-fields used to compute equals and hashCode that settle before equals or hashCode is called.

```
Point p = new Point(1,2);
p.x = 1;
p.hashCode();
p = null;
```

p.x is equals-settled

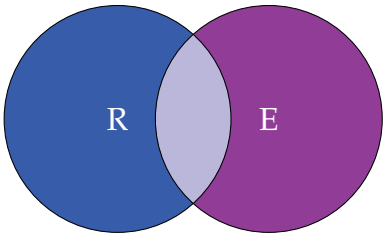
```
Point p = new Point(1,2);
p.hashCode();
p.x = 1;
p = null;
```

p.x is not equals-settled

Equals-settled object-fields do not necessarily settle before any of the other events we have defined: equals and hashCode methods can be called at any time

[1] In fact there is a distinction between CW field declarations and undeclared-final fields: undeclared-final fields, like Java's final fields, must be written directly from the constructor method. CW field declarations may be written from a method called from a constructor or super constructor. As these method calls could be inlined into the constructor we feel that this distinction is minor.

or not at all. equals and hashCode methods can even be called before final fields settle.[2]



Equals-settled fields

A field declaration is equals-settled if all the object-fields resulting from that declaration are equals-settled. All writes to an equals-settled field declaration in any object must occur before calls to an equality method for that object. Object-fields whose objects never have an equals or hashCode method called do not affect the classification of a field declaration and a field declaration is only equals-settled if at least one object-fields for that field declaration is equals-settled.

A field declaration is not equals-settled if there exists an object-field for that field declaration that is written after equals or hashCode has been called on its object, or if there are no object-fields for that field declaration whose objects have an equals or hashCode method called.

Equals-settled objects

An object is equals-settled if no object-fields of the object is written after equals or hashCode are called on the object. An object that never has equals or hashCode called is not equals-settled.

Equals-settled classes

A class is equals-settled if all objects of that class that have an equality method called are equals-settled and additionally at least one object of that class has an equality method called.

3.2.6 Collection-settled (K)

All writes to a *collection-settled* (K) object-field occur before the object is stored in a collection. Collection-settled is only applicable to objects that enter a collection.

[2]The Venn diagram for equals-settled and all previously defined behaviours is complex. For simplicity we only show equals-settled and read-settled because these behaviours are directly compared in Chapter 6.

Most Java collections use equals and hashCode to determine whether an object is a member of the collection, so collection-settled is a good candidate for the first use of an object-field.

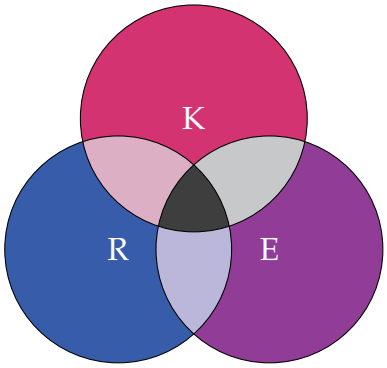
```
Point p = new Point(1,2);
p.x = 1;
new ArrayList().add(p);
p = null;
```

p.x is collection-settled

```
Point p = new Point(1,2);
new ArrayList().add(p);
p.x = 1;
p = null;
```

p.x is not collection-settled

Like equals-settled object-fields, Collection-settled object-fields are not necessarily settled before any other events occur: objects can enter a collection at any time including during the constructor.[3]



Collection-settled fields

A field declaration is collection-settled if all the object-fields resulting from that declaration are collection-settled. All writes to a collection-settled field declaration must occur before the target object enters a collection. Object-fields whose objects never enter a collection are ignored and a field declaration is collection-settled only if at least one object-field's object enters a collection.

A field declaration is not collection-settled if there exists a write to the field declaration for an object of any class that has entered a collection, or if there is no object-field for that field declaration whose object enters a collection.

[3]The Venn diagram for collection-settled including all defined behaviours is complex. For simplicity we only show collection-settled, equals-settled, and read-settled because these behaviours are directly compared in Chapter 6.

Collection-settled objects

An object is collection-settled if all object-fields of the object are collection-settled (this requires that the object enters a collection).

Collection-settled classes

A class is collection-settled if all objects of that class that enter a collection are collection-settled and at least one object of that class enters a collection.

3.2.7 Mutable (M)

Some object-fields never settle by a particular point in the program, a program will continue to write to them until the object is destroyed. If an object-field is not settled before any of the other points described then we say it is *mutable*.

3.3 Equality settling

In general it is not possible to conclude which references stored in an object's fields represent other objects that are part of the object (composite) and which represent relationships between that object and collaborating objects. Ownership and heap analysis can provide some insights (e.g. 2.4.6), but Java's equality methods (equals and hashCode) provide insight into the intentions of the programmer: if a programmer uses a particular field to compute object equality then we assume that the programmer considers that field part of the object (rather than a convenience reference to a collaborating object). The value stored in the field is sufficiently important that it can help distinguish this object from another similar object.

We define an object's *equality* as any program state (object-fields and array entries) used to compute equals or hashCode during the execution of the program. All of the settling patterns described in the previous section (Section 3.2) can be applied to *equality*, but some are particularly interesting and relevant.

This section discusses the different ways that a programmer may implement equals and hashCode and how they relate to our settling events. Equality is fundamentally a property of an object, so we consider objects and classes, but not object-fields or field declarations.

3.3.1 Identity-as-equality (Id)

Unless a programmer specifically implements equals and hashCode then the default implementations will use == and System.identityHashCode(), which rely on object references. Object identity cannot be modified in a Java program so unless a class (or its super-classes) implement equals and hashCode then its object's equality will be settled as soon as the object is created.

```
class Person {
  String name;
  public Person(String name) {
    this.name = name;
  }
}
```

This class does not define equals and hashCode so the default implementations that use reference comparisons will be used. All instances of this class will use identity-as-equality.

3.3.2 Constructor-settled equality (C)

Constructor-settled equality objects are objects whose equality settles before their constructor returns. These objects include identity-as-equality objects, but also objects whose class implements equals and hashCode that depend on fields that are constructor-settled.

Constructor-settled equality objects do not change their equality after their constructor returns, so two constructor-settled equality objects that are equals after their constructors return will be equals for the rest of the program.

```
class Point {
  public final int x;
  public final int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int hashCode() {
    return x ^ y;
  }
  public boolean equals(Point p) {
    return x == p.x && y == p.y;
  }
```

```
}
```

This class defines equals and hashCode using fields that can only be set from the constructor. Consequently, all instances of this class will have constructor-settled equality.

Constructor-settled equality is interesting because it indicates that an object's equality depends only on properties that are initialised during the constructor. This precludes equality based on cyclical references, for example.

3.3.3 Collection-settled equality (K)

Collection-settled equality objects are objects whose equality settles before they enter a Java collection. These objects include constructor-settled equality objects, but also objects whose equals and hashCode methods depend on properties that are collection-settled.

After entering an *equality collection* they never change their equality again.

```
class Student {
  private ID id;
  String name;
  public Student(String name) {
    this.name = name;
  }
  /** This method is provided so the ID
   *  can be set once the object has
   *  a database ID. */
  public void setID(ID id) {
    this.id = id;
  }
  public int hashCode() {
    return id.hashCode();
  }
  public boolean equals(Student p) {
    return id.equals(p.id);
  }
}
```

This class defines equals and hashCode using a field that can be set at any time during the program. A comment indicates that the field is intended to be set by a database framework when the object is persisted and the database allocates it a unique ID. This is a common pattern for objects that are persisted in

a database [20, 79]. Even though there are no static properties that ensure the equality will settle the comment makes it likely that all instances of this class will settle their equality.

Most Java collections (except IdentityHashMap) use equals to determine whether a collection contains a particular object:

```
public boolean contains(Object o) {
  Iterator<E> it = iterator();
  if (o==null) {
    while (it.hasNext())
      if (it.next()==null)
        return true;
    } else {
    while (it.hasNext())
      if (o.equals(it.next()))
        return true;
  }
  return false;
}
```

Implementation of the contains method from Java's AbstractCollection [105].

If an instance of the Student class was stored in a Java collection before its ID was set then any attempt to find an object in that collection would result in an error (a null dereference). Programmers that use an external property such as a database ID for implementing equals and hashCode take care to ensure that it is initialised before they store the object in a collection [20].

3.3.4 Mutable equality (M)

Sometimes it is appropriate to implement equals and hashCode using properties that do not settle. An object whose equality does not settle has mutable equality.

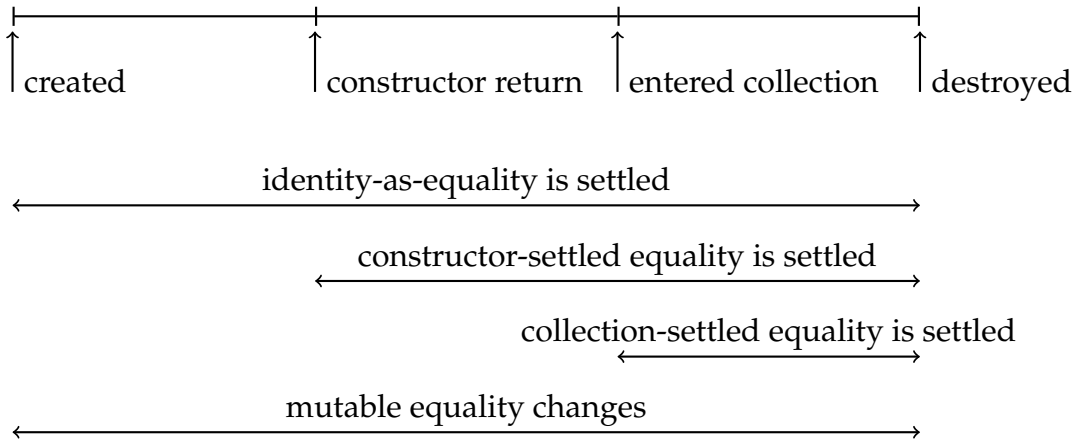
```
class Position {
  public int x;
  public int y;

  public int hashCode() {
    return x ^ y;
  }
  public boolean equals(Position p) {
    return x == p.x && y == p.y;
```

```
  }
}
```

This class is designed to represent the position of an object in a 2D space. Instances of this class will be updated as entities in the space move and the `equals` method can be used to determine whether two entities currently occupy the same position on that space. As long as the entities continue moving the equality will not settle.

3.3.5 Equality settling object life-cycles

The four types of equality settling behaviour presented in this section are summarised in this object-lifecycle chart:



Subsequent chapters use the settling behaviours defined in this chapter to observe the runtime behaviour of Java programs.

Chapter 4

#prof: Profiling Equality Initialisation

4.1 Introduction

This chapter introduces *#prof*, a runtime profiler designed and implemented for this thesis to study the behaviour of equals and hashCode in Java. *#prof* uses class file modification to add code to existing programs as they are loaded by the JVM. This allows *#prof* to observe the behaviour of individual objects as the program executes, summarise the observations by class, and output summary data when the program terminates. *#prof* uses a novel technique to detect changes to object equality by computing dependency graphs for hashCode method calls, then recalculating hashCode when any fields of objects in the dependency graphs change. *#prof* classifies objects by observing when their fields and their equality settle according to the definitions from Chapter 3.[1]

The work presented in this chapter, and *#prof* in particular, represents our first attempt at object profiling. *#prof* was developed using AspectJ [54] not only to facilitate code modification but also to aid rapid development. This was very successful and we were able to obtain interesting results and explore different approaches to object profiling with a relatively small time investment. This allowed us to identify some of the categories of settling events presented in the previous chapter. Nevertheless, *#prof*'s approach to profiling has several limitations (discussed in Sections 4.3.3 and 4.6) that have affected the scope of the results presented in this chapter and motivated the development of a new profiler, *rprof*, which is presented in Chapter 5.

[1]The work presented in this chapter was also presented at TOOLS Europe 2010 [68] and an extended version with tabular data is available as a technical report [69].

Chapter organisation

This chapter is arranged as follows:

- Section 4.2 describes the behaviour classifications used in this chapter, derived from the settling classifications presented in Chapter 3.
- Section 4.3 provides an overview of *#prof*, a runtime profiling tool developed for this thesis that monitors the behaviour of objects at runtime and classifies them using the behaviour characterisations from Section 4.2.
- Section 4.4 presents relevant details of *#prof*'s implementation.
- Section 4.5 presents results obtained using *#prof* to monitor the behaviour of 30 Java programs from the Qualitas Corpus [88].
- Section 4.6 summarises the contributions of this chapter and motivates the work presented in subsequent chapters.

4.2 Classifications for *#prof*

Programmers may implement equals and hashCode in many ways, but by observing their settling behaviour at runtime we can broadly classify different types of implementations and compare their relative popularity. This section uses the settling behaviours defined in Chapter 3 to characterise the particular types of implementations.

4.2.1 Equality vs. field settling behaviours

Chapter 3 defined several settling behaviours for object-fields and equality initialisation. This chapter classifies objects using a selected subset of these behaviours to characterise their object-fields and equality: constructor-settled or mutable object-fields (Section 3.2.1); and identity-as-equality, constructor-settled, collection-settled, or mutable equality (Section 3.3). Table 4.1 shows the eight categories for objects derived from these settling behaviours.

Assuming that a class actually defines equals and hashCode, then comparing the field and equality settling behaviour of its instances will result in one of four possibilities:

- Equality and fields both settled before the end of the constructor (C(C)). This could indicate that the objects represent constants such as numbers or points that have an inherent equivalence relation but no mutable state.

		Fields	
		Constructor-settled (C)	Mutable (M)
Equality	Identity-as-equality (Id)	*Id(C)*	*Id(M)*
	Constructor-settled (C)	*C(C)*	*C(M)*
	Collection-settled (K)	*K(C)*	*K(M)*
	Mutable (M)	*M(C)*	*M(M)*

Table 4.1: Object classifications based on equality and object-field settling behaviour. Frame colour indicates equality behaviour while background colour indicates field settling behaviour. These colours are also used in the results section of this chapter.

- Equality settles after fields (K(C), M(C)). This indicates that the class uses deep state – referenced arrays or objects – to calculate its equals and hashCode because equality changed independent of the object's fields. Unfortunately we cannot conclude anything else about these classes because *#prof* does not track deep state.
- Equality settles before fields (C(M), K(M)). This indicates that an object's equality is based on a subset of its properties and the object has fields that change. If this occurs, it could indicate that the class is using equals and hashCode to implement external references (e.g. [20, 79]). Alternately, the class could be caching external values in fields, or even caching the value of hashCode (e.g. java.lang.String).
- Neither equality nor fields settle (M(M)). This indicates that the class is most likely using equals and hashCode to determine whether its objects are currently equal (e.g. comparing coordinates to see if two objects are currently in the same position).

Classes that do not define equals and hashCode (Id(C), Id(M)) default to Java reference comparisons. They could be mapping an external ID onto a Java reference (Hibernate can do this, for example [79]) but we won't be able to tell using these designations.

4.2.2 Collections and equality settling

Java's standard libraries include the *Java Collections API*, a collection of general purpose implementations of many common data structures. Almost all of these

collections are capable of storing any type of object, yet they require stronger contracts than the equals method documentation provides. For example, documentation for java.util.Set states:

> *"Note: great care must be exercised if mutable objects are used as set elements. The behaviour of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element."* [24]

This contract is significant because it requires that while objects are in a Set their equality should not change. As no type constraint exists to prevent objects with mutable equality from entering a Set, programmers must take care to obey this contract or they may encounter subtle bugs. We will compare objects that enter collections like Sets to objects that enter other types of collections (e.g. Lists), as well as to objects that don't enter collections, to determine whether classes with objects in these categories are initialised differently (in some manner) to ensure that they meet the Set constraints. For example, it would be an error for mutable-equality/mutable-fields (M(M)) objects to change while they are in a Set.

The Java Collections API provides four major interfaces: Set, List, Queue, and Map, each of which imposes different constraint on the objects they contain. As we've discussed, the Set interface imposes a particularly strict contract on its contents. Lists and Queues are more permissive, but they have a related note of caution:

> *"Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a list."* [24]

This note is necessary because the provided implementations define equals and hashCode methods that depend on their contents. For example, ArrayList's equals method will recursively call equals on each List member. Programmers must be aware that if the List is stored in another collection – such as a Set – the other collection's contracts will transitively apply to the List's content.

The Map interface (also Hashtable) splits objects into keys and values: keys must conform to the same contract as Set objects, while value objects must conform to the same contracts as List's.

Collections naturally divide into two categories based on the contracts they impose on their contents:

- *Equality collections* — collections like Set whose implementations depend on equals and hashCode for correct functionality. These include Sets, and the parts of Maps and HashTables that contain keys. While an object is in one of these collections it must not change its equality.

- *Identity collections* – List, Queue, and the value parts of Map and HashTable – do not require equals and hashCode for correct functionality but they use equals to determine whether an object is in the collection, and their elements must obey any transitive constraints — if the collection enters an equality collection then all of the elements must behave as if they are in an equality collection.

There is a third type of collection that does not use equality: IdentityHashMap. As such, this collection does not exhibit any equality behaviour so we disregard it for the analysis in this chapter.

4.3 *#prof* overview

This section gives a high-level overview of the strategies used by *#prof* to categorise objects as by their equality-settling behaviour (Id, C, K, or M) and by their field-settling behaviour (C or M).

4.3.1 Detecting equality settling

Java's equals and hashCode methods describe how an object's equality can be calculated. *#prof* needs to detect changes to an object's equality at runtime to classify the object's equality settling behaviour. hashCode is very useful for detecting changes to equality if we assume that it does not have side-effects. If hashCode is side-effect free, then *#prof* can observe an object's equality at runtime without affecting the program's semantics by calling hashCode whenever an object's equality may have changed. If the value hashCode returns does not match the previous value, then *#prof* can conclude that the object's equality has changed.

#prof can determine which program events may cause an object's equality to change by observing each hashCode invocation and recording the properties it uses, then monitoring events that affect those properties. *#prof* does this using a two part strategy:

1. Immediately after an object's constructor returns, *#prof* calls the object's hashCode method. During the computation, *#prof* intercepts all method

```
public int hashCode() {
    int h = this.hash;
    if (h == 0 && count > 0) {
        int off = offset;
        char val[] = value;
        int len = count;

        for (int i = 0; i < len; i++) {
            h = 31*h + val[off++];
        }
        this.hash = h;
    }
    return h;
}
```

Figure 4.1: This hashCode method for java.lang.String demonstrates caching the result of hashCode to avoid recomputing it for subsequent invocations (adapted from [105]).

calls and adds the target objects of each call to a *dependency set* for the object. Once the hashCode method returns this set will contain all objects used to compute the object's hashCode. *#prof* records the dependency set and the result of calling hashCode for each object.

2. *#prof* tracks all field writes to all objects in dependency sets. When a field write occurs, *#prof* determines which object contains the field, then causes every object whose dependency set contains that object to re-compute hashCode. If the value returned by hashCode is different to the stored value, then the object has changed its equality so *#prof* records the change and updates the stored value. While *#prof* is recomputing hashCode it adds any additional objects encountered to the appropriate dependency set so that future changes can be detected.

If an object's hashCode method is not side-effect free then *#prof* could affect the program's behaviour. An idiomatic hashCode method will not have side-effects but a common optimisation for complex classes involves storing hashCode's value the first time it is calculated and returning the stored value for subsequent invocations (e.g. Figure 4.1). *#prof* could calculate hashCode too early and trigger an exception, or cause the object to cache a hashCode value that does not reflect its final state. *#prof* catches and discards exceptions while calculating hashCode. An object calculating and caching the wrong hashCode code could affect program behaviour, but this is an advanced strategy so we assume that pro-

```
public boolean equals(Object other) {
  if (this == other)
    return true;
  if (this == null)
    return false;
  if (!(other instanceof A))
    return false;
  A otherA = (A) other;
  return ...
}
```

Figure 4.2: An idiomatic Java equals method showing optimisation checks for **this**, **null**, and compatible types (adapted from [38]).

grammers creating caching implementations of hashCode also code defensively and ensure that any changes the object will invalidate the cached hashCode.

Detecting changes to equals is more difficult than detecting changes to hashCode because equals requires a parameter. We could compare the object to itself, to **null**, or to a new object, but many implementations begin with a short-circuit for common comparisons such as these (e.g. Figure 4.2). The only way a runtime analysis can determine whether a change to a program's state will cause equals to return a different value (without analysing the method's implementation) is to compare the object to all other objects to which it has been compared or will be compared in the future. A runtime analysis that attempts to monitor equals as a black box method must either compare all objects in the program every time a change occurs to reachable program state, or have foreknowledge of all possible execution paths (Marinov and O'Callahan achieve this using multiple program executions [63]).

Java's documentation includes a contract for hashCode() that requires that any two objects that are equal using equals must also have the same hashCode value:

> *"If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result."* [24]

Consequently, if the value computed using hashCode changes for a (correctly implemented) object, its equality as determined using equals must also have changed: monitoring hashCode will give at worst an under-approximation of equals changes. In practice, hashCode methods are implemented to closely mimic equals behaviour to avoid hash collisions between similar objects that would de-

teriorate the performance of equality collections, so hashCode is a sufficiently accurate approximation for equals for our purposes. Java does not enforce correct hashCode implementations, but there are automated tools available to developers that detect incorrect implementations at compile-time [48]. We assume the implementations are correct.

Even if we ignore the assumptions previously discussed, there are two additional problems that could threaten the validity of *#prof*'s equality settling detection. *#prof* should identify and monitor arrays that are used to compute hashCode but does not due to tool limitations, so changes to arrays will not trigger *#prof* to recalculate hashCode. Consequentially, some changes to equality may not be detected. In addition, a change to equality may not be detected if the previous and new hashCode values collide.

4.3.2 Detecting object-fields settling

Detecting changes that affect hashCode requires that *#prof* monitors all fields. When changes occur, in addition to calculating the effect on hashCode(), *#prof* marks the object that owns the field as mutable (M) if its constructor has completed. When the program terminates, *#prof* compares the number of mutable (M) objects of a particular class to the total number of objects of that class. If none of the objects were mutated then *#prof* concludes that the class was constructor-settled (C), and hence all objects of that class have constructor-settled fields. If a field of any object of that class was mutated after the constructor then *#prof* assumes that any object could have been mutated, so all objects of that class are classified as having mutable fields.

4.3.3 Aggregating results

#prof's design consists of independent modules that identify and publish particular events, and components that record events for each object. For example, suppose an object entered an identity collection then changed its equality:

- the *detect collection membership* component will identify that the object entered a particular identity collection and update the object's record to indicate that it (a) is in an identity collection and (b) has entered a collection.

- the *detect equality changes* component will detect the object's equality changing and generate an *equality changed* event.

- the *record changes in collection* component will receive the *equality changed* event, check the object record, observe that the object is currently in an identity collection, and update the object's record to indicate that the object changed its equality in an identity collection.

When the object is destroyed by the JVM's garbage collector or the program terminates, each component will update the record for the object's class:

- the *detect equality changes* component will record that an instance of the class changed its equality.
- the *detect collection membership* component will record that an instance of the class entered an identity collection.
- the *record changes in collection* component will record that an instance of the class changed its equality in an identity collection.

This approach ensures that the extra memory required to track objects is proportional to the number of live objects (so the JVM does not run out of memory) but means that *#prof* does not record information about the behaviour of particular objects, only the aggregate behaviour of classes. Using this information we can accurately classify classes using the classifications from Section 4.2, but we cannot always accurately classify individual objects. For example, suppose a class had two instances in total, one instance that entered an identity collection and one that entered an equality collection. We can conclude that instances of the class entered equality collections and identity collections, but we don't know whether each instance entered one collection or one instance entered both. On the other hand, if a class reports that one instance entered an identity collection and one changed in an identity collection we can infer that it was the same object.

#prof always produces class aggregations that match the observed results, but it is only possible to reconstruct object behaviour in cases where the majority of instances of a class behaved in the same way.

Instead of recording properties independently, *#prof* could use separate counters for each possible combination of properties. For example, *#prof* reports for each class:

- the number of objects in equality collections
- the number of objects in identity collections

Instead, *#prof* could record:

- the number of objects in equality and identity collections
- the number of objects in equality but not identity collections
- the number of objects in identity but not equality collections
- the number of objects in no collections

Unfortunately this would cause an exponential explosion in the number of properties to record: *#prof* records 18 independent properties, recording the permutations of these properties instead would require over 200K property counters per class. *#prof* represents an exploration of object initialisation behaviour, during development we did not know which properties would prove to be important. Even with only the six independent properties reported in this chapter plus membership of two types of collections we would require 256 property counters per class — feasible but it would have required significant architecture changes.

4.4 *#prof* implementation

#prof was implemented using AspectJ [54]. AspectJ is a language extension to Java that can systematically add new functionality to an existing program [41, 56]. AspectJ is a useful platform for building profilers [81], as it does not require recompilation of target programs or virtual machine modification, and allows the profiler to be developed using Java-like syntax.

AspectJ provides a Java compiler that can modify class files at compilation, and a class loader that replaces the JVM class loader and modifies classes as they are loaded by a running program. AspectJ allows profilers to identify important events using *pointcuts*, then specify Java code to handle event occurrences. *#prof* uses AspectJ to modify classes as the JVM loads them so that, as a program executes, *#prof* receives callbacks when particular method calls and field accesses occur. In addition, *#prof* uses AspectJ to replace Java collections with custom implementations that make it possible to track collection membership.

AspectJ load-time weaving cannot add code to the Java standard libraries, so *#prof* cannot directly record changes that occur within the standard library objects (except for collections, which we address specifically). *#prof* can partially observe some standard library objects that interact with user code but the results are not necessarily accurate — it cannot generate events for field modifications, for example. *#prof* is also unable to profile applications that use their own class loaders (like Eclipse) or applications that are close to the limit on method size —

AspectJ does not support breaking up methods to avoid overflowing the method size limit and as *#prof*'s use of AspectJ adds a lot of tracking code to methods this occasionally results in invalid class files, causing the program to terminate. This limits the programs that can be profiled by *#prof*.

The remainder of this section discusses relevant aspects of *#prof*'s implementation.

4.4.1 Generating events

In the terminology of aspect-oriented programming, an observable event is a *join point*. Code introduced at an event site is *advice*. AspectJ join points include method entry, method calls, and field access (read or write). AspectJ can insert advice at a single join point or to a set of join points (a *pointcut*).

#prof requires several different runtime events to be intercepted:

- object constructor return, to identify the beginning of the *initialisation phase*.
- object death, to aggregate recorded object data into class summaries.
- changes to an object's equality.
- changes to an object's fields.
- changes to an object's containment by collections.

Intercepting object creation with AspectJ

As a simple example, AspectJ can intercept object constructor return using advice similar to the following:

```
after () returning (Object o):
 call(*.new(..)) && !within(profiler.*){
 ...
}
```

In this example, `after` advice causes AspectJ to insert code after each constructor method call. The pointcut `call(*.new(..))` matches any constructor while the pointcut `!within(profiler.*)` prevents AspectJ from intercepting object creations caused by *#prof*.

Intercepting object death

#prof stores per-object information to record field modifications, object dependencies, collection membership, and hash code calculations. *#prof* must ensure that it does not cause memory leaks – which could cause the JVM to run out of memory – while maintaining a dependency graph and ensuring that *#prof* can aggregate the recorded information before program termination. For example, when an object enters a collection, *#prof* records that it is in that particular collection. If the collection becomes unreachable from the program then *#prof* should ensure that it can be collected by the garbage collector, as well as ensuring that the object's record is updated to indicate that it is no longer in the (garbage collected) collection.

#prof uses *weak references* to refer to Java objects so that the garbage collector is not affected by the presence of *#prof*'s tracking structures (this technique has been used for other profilers [1, 81]). *#prof* allocates weak references with a ReferenceQueue, a Java standard library class that interacts with the garbage collector. After collecting an object that is referenced by a weak reference, the garbage collector adds the weak reference to its associated ReferenceQueue, which allows *#prof* to identify that the object has been collected and take appropriate action. When a Java thread enters *#prof* tracking code (next time a *#prof* event occurs), *#prof* checks the queue and purges records for any objects that have been collected, aggregating their information into class statistics.

Profiling object equality

Section 4.3.1 outlined the algorithm *#prof* uses to detect changes to object equality by profiling hashCode and building dependency sets. When any object in a dependency set changes, *#prof* should detect the change, recompute hashCode (adding any new dependencies to the set), and conclude whether the object's equality has changed as a result of the field modification. This calculation assumes that hashCode methods are stable — as long as no reachable state changes, calling hashCode should always return the same value. The Java API documentation for hashCode asserts that implementations should have this property so this is not a major assumption [105]. *#prof* identifies changes that could affect object equality by intercepting all field writes (using AspectJ's set join point). This also allows *#prof* to determine whether object state is mutated after the constructor.

Some hashCode methods assume that state is available for computation based on programmer assumptions about when hashCode will first be called. *#prof* al-

ways calls hashCode immediately after construction ends, which can invalidate these assumptions and cause exceptions. #*prof* catches and discards any exceptions thrown during #*prof*-triggered hashCode calls as discussed in Section 4.3.3. Even if the hashCode method throws an exception #*prof* can build an accurate dependency set because one of the objects added to the set prior to the exception must change in order for a future call to complete without an exception.

Profiling collections

#*prof* tracks objects' collection membership to identify collection-settled equality and track objects that enter different types of collections. AspectJ cannot modify Java's standard libraries, which include collections, so #*prof* uses *proxies* to determine whether or not an object is in a collection. When a program creates a new collection #*prof* intercepts the call using AspectJ join points on collection constructor calls, returning instead a proxy collection that wraps the actual collection type that the program requested (see Figure 4.3). The proxy collections monitor collection interfaces and track objects as the program adds and removes them from the collections.

#*prof* replaces the following collections with proxy implementations: HashSet, HashMap, LinkedHashMap, Hashtable, LinkedHashSet, TreeSet, TreeMap, PriorityQueue, ArrayList, LinkedList, and Vector. Proxy collections *extend* the class they proxy, e.g. HashMapReplacement **extends** HashMap.

Some programs extend standard collections themselves. #*prof* exploits AspectJ's declare parents functionality to introduce its proxy collections into the hierarchy between the collection class and the application-defined subclass. In most cases this works well, however in a small number of cases it fails due to an outstanding bug in the AspectJ compiler. The benchmarks used in this chapter exclude programs that exhibit this behaviour.

It is possible for proxy collections to alter a program's behaviour. A program could use reflection or use the actual name of the class in computations. We did

```
...
new ArrayList<String>();
...
```

(a) Before AspectJ Modification

```
...
new ArrayListReplacement<String>(
        new ArrayList<String>());
...
```

(b) After AspectJ Modification

Figure 4.3: Replacing collections with proxy collections

not encounter any unexpected behaviour that could be attributed to this while profiling the benchmarks in this chapter but it could cause problems for other programs.

Finally, *#prof* cannot intercept collection object creations arising from code within the Java standard library itself because AspectJ cannot weave against it, and *#prof* can only profile collections from the standard libraries (application-specific collections are not supported).

Storing object-specific analysis state

#prof must associate state with objects during profiling. For example, for each object *#prof* records the last computed value of hashCode; the object's dependency set; and any collections that currently contain the object. AspectJ provides a mechanism for inserting fields into a class (*intertype declaration*), however, *#prof* must access object state after the JVM garbage collects an object, so object fields are not suitable. *#prof* instead uses a global store containing a record for each active object. When a program creates an object, AspectJ after advice causes *#prof* to create a record for the object and insert it into the global store. When advice intercepts an interesting event (for example, when the object enters a collection), *#prof* retrieves and updates the object's record. Since each record exists independently of the object it describes, it remains accessible after the JVM collects the object.

#prof's global store is a hash map, keyed on the objects themselves. Ideally *#prof* should use a unique identifier for each object that persists after the JVM collects the object. Unfortunately, this would cause a dependency problem: the profiler must store the identifier where it can retrieve it later from the object — using the identifier! In a language like C, *#prof* could use the memory address of an object as an identifier, however Java does not expose object memory addresses (and JVMs frequently relocate objects during garbage collection).

#prof uses a custom hash map similar to Java's WeakHashMap keyed on the actual objects, but wrapped with weak references to avoid affecting garbage collection. This allows *#prof* access to an object's record after the JVM collects the object. WeakHashMap itself is not suitable as it uses hashCode to store objects.

#prof stores objects using identity hash codes from System.identityHashCode(). This method returns a consistent value for an object regardless of any changes to object properties. Oracle's JVM implementation derives this value from the memory address of the object when the method is first called then preserves that value for future calls.

The use of System.identityHashCode() leads to an interesting complication: *#prof*

must insert each object into the global store as soon as the object is constructed so that it can be used by the dependency set algorithm. The JVM, however, stores all freshly-created objects in the nursery region of the heap — about 16MB of memory by default. This almost guarantees that there will extremely frequent hash collisions for live objects as the nursery is reused for creating the next generation of objects. These collisions significantly degrade performance as *#prof* must perform $O(n)$ search of hash buckets – where n is the size of the buckets – every time an event occurs that affects or could affect a particular object. This is a significant source of overhead for *#prof*, particularly for larger benchmarks. An implementation that, for example, included some time-based entropy in generated hash codes would significantly reduce *#prof*'s overhead.

Concurrency

Most Java programs are concurrent, at least to some degree, as most JVMs use threads internally. *#prof* relies on being able to compute `hashCode` after a field is modified without other field modifications occurring while it is doing so. To ensure this, *#prof* uses monitors to ensure that only one thread can be within profiler code at a time. This does not guarantee the program is thread-safe; one thread may still change a value while another thread is inspecting that object, but the first thread must then enter the profiler and wait for the other thread to finish before continuing. In practice we did not encounter any errors, but some programs that provided 'instant' feedback were sluggish (particularly AWT-based GUI applications).

4.5 *#prof* results

This section presents the results obtained from running *#prof* on a sample of applications from the Qualitas Corpus developed at Auckland University, NZ [88]. The Qualitas Corpus is comprised of source distributions from 100 open source Java projects to aid empirical research. The Qualitas research group designed the corpus primarily for static analysis, so the corpus includes libraries and platforms that cannot be run independently. In addition, some projects in the corpus use custom class loaders or include long methods that prevent *#prof* from modifying them (see Section 4.4). Of the 100 projects in the corpus, this section examines a sample population of 30 applications suitable for runtime profiling. These included compilers, command-line utilities, graphical tools, sample applications for libraries, and test suites. Figure 4.4 presents the complete list of applications profiled with a short description of each.

4.5.1 Experimental method

#prof profiled each benchmark as it ran on a standard Java HotSpot™ Server VM (build 1.5.0_15-b04, mixed mode), on one of several Dell Optiplex GX620 workstations (Pentium 4, 3.2GHz, 1GB memory) running NetBSD 5.0_RC2. Benchmarks were started using the following command line arguments:

```
java −Xmx1024M −javaagent:aspectjweaver.jar −classpath hashprof.jar [benchmark]
```

This starts Java with AspectJ's load-time weaving enabled. AspectJ will load any join points defined in AspectJ files listed by the META−INF declaration of hashprof.jar. Subsequently, when Java class files are loaded by the JVM, the AspectJ load-time weaver will augment them with the code necessary to instrument the specified join points.

Benchmarks

The Qualitas Corpus is not designed for runtime benchmarking and does not provide a framework for automatically executing programs, but most distributions include a binary package. For each benchmark listed in Figure 4.4 we investigated the binary packages available and determined how the benchmark could be run and what functionality could be exercised by running them in a standard Java environment. For each benchmark we identified a set of inputs designed to exercise program functionality, but we did not consult source code or measure profiling coverage. Figure 4.4 briefly outlines the test inputs used for each

Application	Synopsis
ant	A Java build system. Benchmarked building ant, uses javac.
antlr	A compiler-generator. Tested compiling Java grammar.
aoi	Art of Illusion, a 3D editor with raytracer. Built a simple model and rendered it.
columba	Java mail client. Connected to an imap server, browsed mail and sent a message.
derby	Java database. Ran tutorial on in-memory DB.
drawswf	SWF animation editor. Generated a small animation and exported to SWF.
fitjava	Testing framework. Ran tests distributed with framework.
freecs	Chat server. Ran server and connected several clients.
ganttproject	Graphical tool for task management.
hsqldb	Database tool. Created in-memory database and run various test scripts.
itext	Collection of tools for PDFs. Ran several tools.
jFin_DateMath	Date math library. Ran tests.
javacc	Java Compiler Compiler. Compiled JavaCC grammar.
jchempaint	Graphical molecule editor. Created and edited simple molecules.
jedit	Text editor. Created Java class, edited, searched, saved etc.
jfreechart	Graphical tool for creating charts. Tested UI.
jgraph	Library for drawing graphs. Ran several examples.
jgraphpad	Uses jgraph for drawing graphs. Created small graphs.
jgrapht	Views graphs, uses jgraph.
jhotdraw	Graphics framework. Tested sample application.
jmoney	Personal finance. Created sample accounts. Tested import/-export, saving, editing, and reporting.
nekohtml	HTML parser. Ran samples.
pmd	Source code analyser. Tested on various projects.
pooka	Java email client. Tested connecting to IMAP server, reading mail, sending mail.
trove	High-speed collection implementations. Ran the included benchmark suite.
velocity	Templating engine. Ran sample application.
weka	Data mining tool. Ran sample application.
xalan	XSLT processor. Ran some examples.
xerces	XML parser. Ran some examples.
xmojo	JMX implementation. Ran sample application.

Figure 4.4: 30 applications from Qualitas Corpus release 20080603 [88], using the most recent version available where applicable. Where relevant, the table lists the profiled application behaviour.

benchmark.

For compilers, build tools and libraries we preferred samples distributed with the benchmark. GUI tools required manual interaction; researchers attempted to trigger all of the application's major features, but did so without a deep knowledge of the applications behaviour. *#prof* introduces significant but extremely variable overhead to the applications, ranging from 2×–20× the original runtime. Large autonomous programs ran for several hours, but these can run unattended. The overhead introduced made manually driving interactive programs extremely tedious and we are grateful to several research students from our department for their help driving these benchmarks.

On termination, *#prof* generates tab-separated summary data for each class encountered that is stored in plain-text files (see Section 4.3.3). The results are presented in this section.

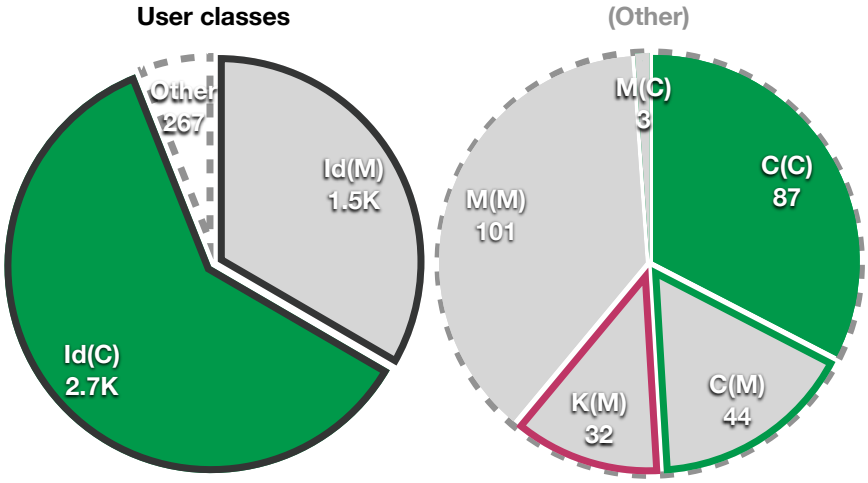
4.5.2 Overview

Table 4.2 shows summary data for each benchmark. The first four columns show the number of user-defined classes encountered by *#prof* at runtime (i.e. excluding java/javax). In order, they show: the total number of classes in each benchmark; the number of classes with instances in equality collections; the number of classes with instances in identity collections; and the number of classes with no instances that enter collections. The fifth column shows the total number of objects in user-defined classes, and the final two columns show the total number of classes and objects observed including system classes.

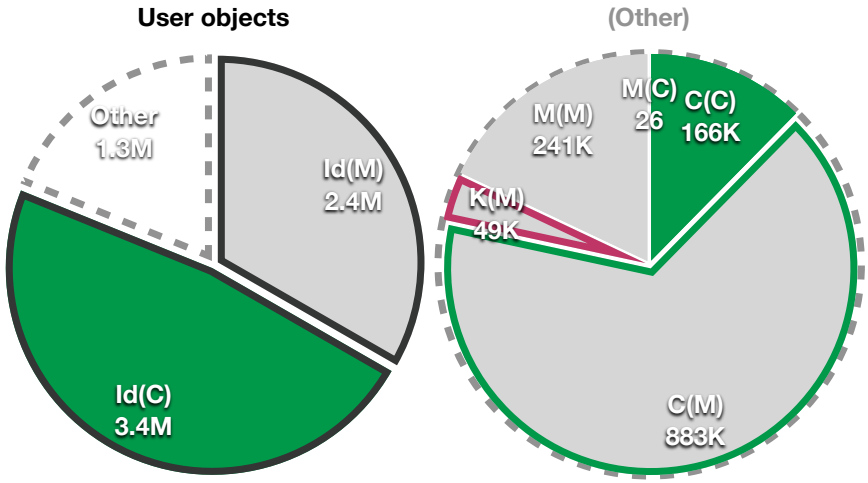
Figure 4.5 presents summary data for user-defined (i.e. excluding java/javax) classes and objects encountered (Figures 4.5a and 4.5b respectively) and demonstrates the result format used throughout the rest of this section. Each figure is split between the categories introduced in Section 4.2 (Table 4.1). The chart on the left of the figure shows the total population split between classes and objects that use identity-as-equality, Id(M) and Id(C), and *Other*, which includes all classes and objects that use non-default equals and hashCode methods. The chart on the right, *Other*, subdivides the classes and objects from the *Other* segment on the left: constructor-settled (C), collection-settled (K), and mutable (M) equality classes. Each category is further split between constructor-settled (C) and mutable (M) fields, giving six possible categories: C(C), C(M), K(C), K(M), M(C), M(M). No instances of K(C) were observed so it is omitted from all figures.

	User defined (i.e. excluding java/javax)					All	
	Classes				Objects	Classes	Objects
Benchmark	Total	Eq. Coll.	Id. Coll.	No Coll.	Total	Total	Total
ant	324	13	36	288	2,639,716	367	2,769,194
antlr	61	4	15	46	16,261	72	49,581
aoi	269	6	49	220	18,930	337	23,723
columba	509	8	35	474	13,059	607	34,402
derby	239	9	14	225	3,399	280	4,238
drawswf	90	1	6	84	1,873	146	5,887
fitjava	8	0	1	7	102	22	739
freecs	89	1	58	31	999	106	3,763
ganttproject	230	5	21	209	24,553	294	37,397
hsqldb	85	0	0	85	22,930	116	27,014
itext	49	2	13	35	6,617	77	15,928
jFin_DateMath	6	0	0	6	151	10	881
javacc	41	2	13	28	41,271	50	74,335
jchempaint	173	3	17	156	800,688	241	860,622
jedit	409	7	23	386	55,489	501	105,912
jfreechart	110	3	7	103	1,784	140	2,524
jgraph	83	4	9	74	4,728	117	74,372
jgraphpad	174	13	17	157	17,084	257	90,466
jgrapht	69	4	9	60	2,139	94	18,591
jhotdraw	125	1	2	123	7,262	164	48,073
jmoney	220	3	11	209	8,786	297	33,514
nekohtml	41	0	1	40	4,368	55	9,215
pmd	198	9	42	155	658,736	216	776,269
pooka	349	6	73	276	23,803	440	58,965
trove	24	0	0	24	2,100,189	35	2,400,310
velocity	49	1	3	46	623	70	1,302
weka	22	1	1	21	605,493	41	606,341
xalan	144	2	5	139	1,664	161	3,050
xerces	160	7	19	141	1,661	169	2,339
xmojo	52	1	5	47	323	95	1,293
Total	4,402	116	505	3,895	7,084,681	5,577	8,140,240

Table 4.2: Summary table showing the number of classes and objects in various categories, per benchmark.



(a) Classes excluding java/javax



(b) Objects excluding java/javax

Figure 4.5: Classes and objects excluding java/javax. The charts on the left show classes and objects in three categories: Id(M) — identity-as-equality and mutable fields, Id(C) — identity-as-equality and constructor-settled fields, and *Other* — which is expanded in the chart on the right. Within *Other* we see: C(C) — constructor-settled equality and constructor-settled fields, C(M) — constructor-settled equality and mutable fields, K(M) — collection-settled equality and mutable fields, M(M) — mutable equality and mutable fields, and M(C) — mutable equality and constructor-settled fields. There were no classes or objects with collection-settled equality and constructor-settled fields (K(C)).

Discussion

Table 4.2 and Figure 4.5 show that the majority of classes do not override the default equals and hashCode methods; in most cases, identity-as-equality is adequate to both identify and compare objects. About half of all classes and objects we observed have constructor-settled fields, suggesting that many classes and objects do not need to mutate their fields after their constructor returns.

Of classes that override equals and hashCode, about half have constructor-settled equality (131 of 267), but constructor-settled fields are less common among these classes. Objects that have constructor-settled equality account for about three quarters of objects that override equals and hashCode and collection-settled or mutable equality is less common among objects than among classes.

Classes and objects whose fields settle before their equality (deep state-based equality) are very uncommon: only three classes and 26 objects showed this behaviour, of a total 4402 classes and over 7M objects.

The following sections consider and compare different subsets of classes and objects from the whole population shown in Table 4.2 using the format established in this section.

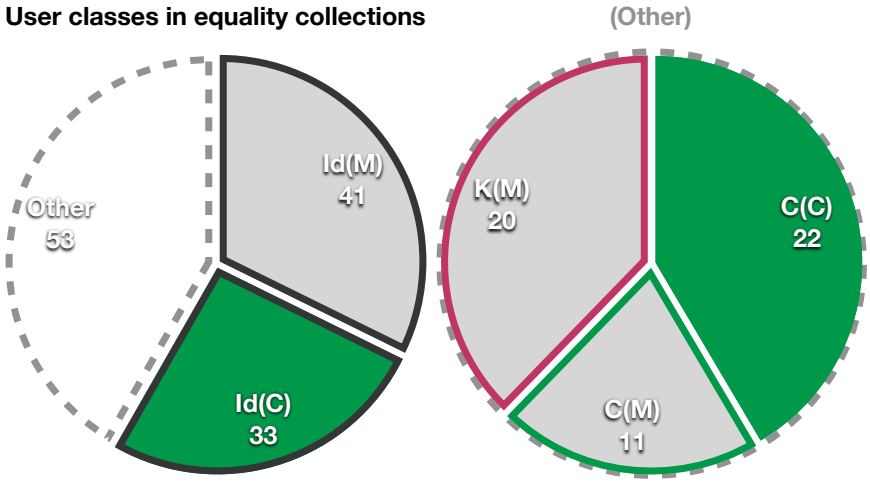
4.5.3 Results: classes and objects that enter equality collections

Figure 4.6 presents data for all user-defined (i.e. excluding java/javax) classes and objects encountered that entered equality collections — collections that require that equals and hashCode are not affected while an object is in that collection. These include Sets and the parts of Maps and HashTables that store keys.

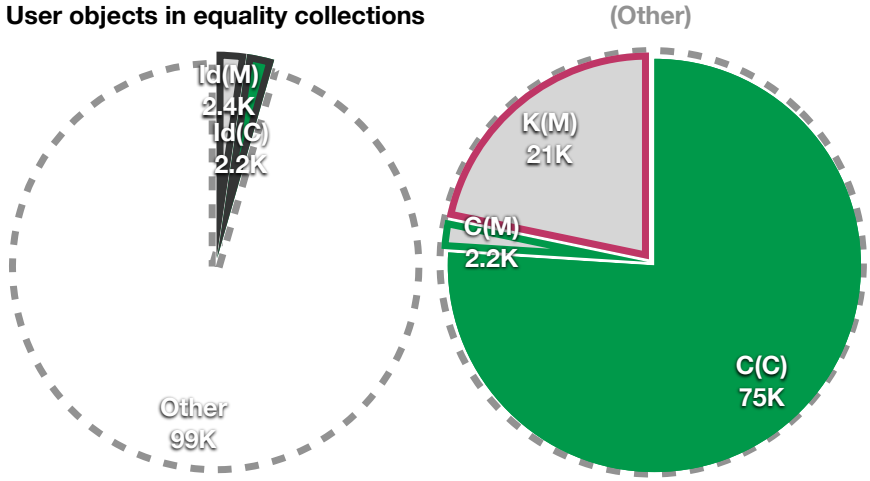
Discussion

Figure 4.6 shows that the vast majority of user-defined objects that enter equality collections do not use identity-as-equality, unlike in the general population where identity-as-equality objects are the norm. The relative proportion of classes that use identity-as-equality is also lower than in the general population, but not to the same extent as objects. This suggests that although identity-as-equality classes that enter equality collections are not uncommon, they tend to have fewer instances than classes that implement equals and hashCode.

One possible interpretation for the extremely high proportion of objects in equality collections that implement equals and hashCode is that collections are



(a) User Classes In Equality Collections



(b) User Objects In Equality Collections

Figure 4.6: User classes and objects that enter equality collections (sets, keys of maps). The charts on the left show classes and objects in three categories: Id(M) — identity-as-equality and mutable fields, Id(C) — identity-as-equality equality and constructor-settled fields, and *Other* — which is expanded in the chart on the right. Within *Other* we see: C(C) — constructor-settled equality and constructor-settled fields, C(M) — constructor-settled equality and mutable fields, and K(M) — collection-settled equality and mutable fields. There were no classes or objects with collection-settled equality and constructor-settled fields (K(C)) or with mutable equality (M(M) and M(C)).

commonly used to map external references onto Java references. For example, a database ID could be used as a map key, mapping that database ID to a unique Java object that represents the domain object associated with that ID (this technique is common among persistence frameworks). If equality collections were commonly used to maintain sets of unique Java objects then we expect to see more objects with identity-as-equality entering these collections.

More than half of classes stored in equality collections have mutable fields even though their equality always settled before entering a collection. It would be interesting to know whether these object's fields are also collection-settled, unfortunately *#prof* does not implement this analysis. Instances of these classes are uncommon: about three quarters of objects in this category have constructor-settled equality and constructor-settled fields (C(C)), a significantly higher proportion than among classes.

Reassuringly, none of the objects or classes we observed changed their equality in equality collections: this would be an error had it occurred. In fact, all of the observed objects *and their classes* had equality that settled before entering a collection. It would be entirely permissible for an object to enter an equality collection, leave, then change their equality, or for instances of a class that do not themselves enter a collection to change their equality after the constructor, but we did not observe this occurring.

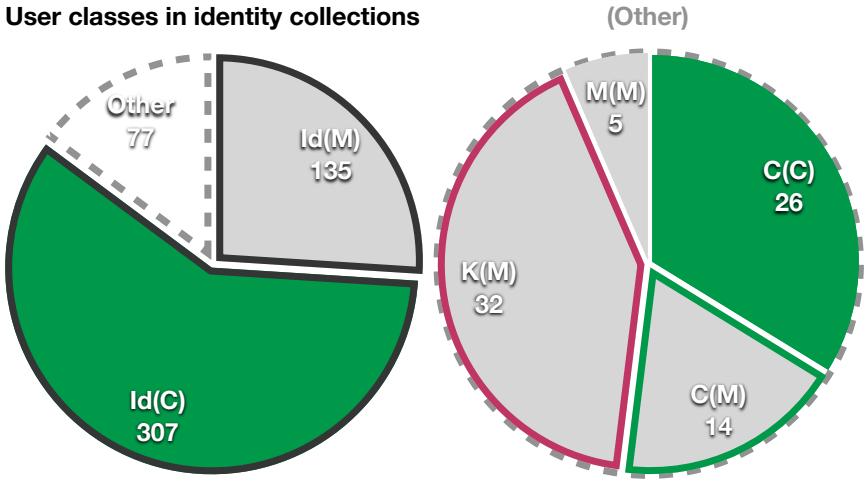
A significant proportion of classes and objects that enter equality collections use collection-settled equality; post-constructor settling is relatively common for these classes and objects.

4.5.4 Results: classes and objects that enter identity collections

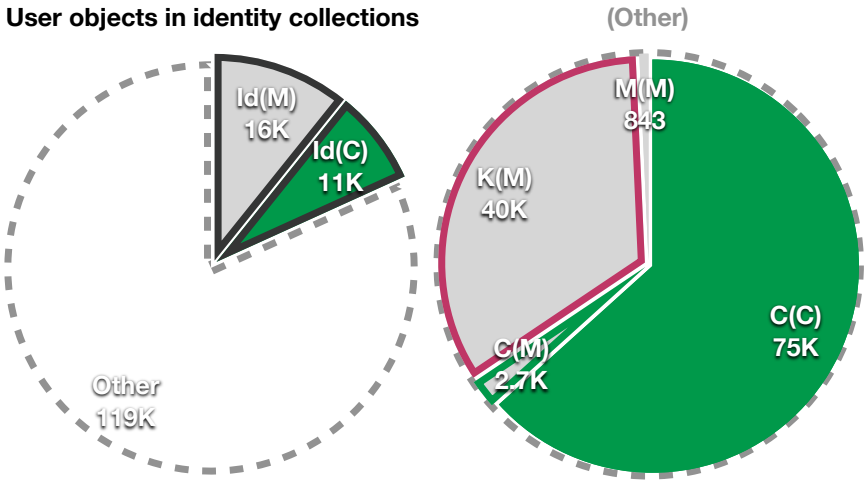
Figure 4.7 presents data for all user-defined (i.e. excluding java/javax) classes and objects encountered that entered identity collections — collections that do not require equals and hashCode for their implementation. These include lists, queues, and the parts of maps that store values.

Discussion

Almost all classes and objects that enter identity collections have settled equality (identity-as-equality (Id), constructor-settled (C), or collection-settled (K) equality). Classes and objects that enter identity collections also have very few instances of identity-as-equality when compared to the general population (particularly noticeable for objects). These trends are not as pronounced as classes and



(a) User Classes In Identity Collections



(b) User Objects In Identity Collections

Figure 4.7: User classes and objects that enter identity collections (lists, queues, values of maps). The charts on the left show classes and objects in three categories: Id(M) — identity-as-equality and mutable fields, Id(C) — identity-as-equality and constructor-settled fields, and *Other* — which is expanded in the chart on the right. Within *Other* we see: C(C) — constructor-settled equality and constructor-settled fields, C(M) — constructor-settled equality and mutable fields, and K(M) — collection-settled equality and mutable fields. There were no classes or objects with collection-settled equality and constructor-settled fields (K(C)) or with mutable equality (M(M) and M(C)).

objects in equality collections, but are less easy to explain: classes and objects that enter equality collections must ensure that their equality does not change while they are in those collections, and they can expect their equality to be used immediately by the collection as it stores the object. Classes and objects that enter identity collections do not have to meet this requirement: they may never have equals or hashCode called as identity collections only use equality to determine collection membership (contains()). There is significant overlap between classes that enter both equality collections and identity collections that could partially explain this trend, but as discussed in Section 4.3.3, *#prof*'s analysis cannot decide in general whether particular objects enter both types of collection.

Among objects that do not use identity-as-equality, objects in identity collections also include fewer constructor-settled equality objects with mutable fields than among the general population, where objects with mutable fields are common.

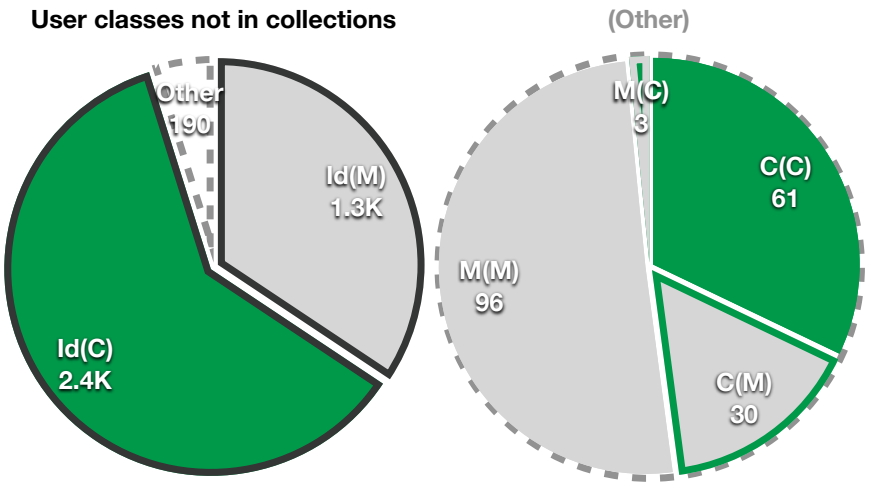
Unlike classes and objects that enter equality collections, identity collections contain some classes and objects with mutable equality and mutable fields, including some that change while in collections. Many of these were instances of a single class: org.columba.core.xml.XmlElement. XmlElement uses a Java Hashtable internally to maintain a collection of 'attributes' and a Vector to maintain a list of 'subelements'. Its hashCode implementation depends on both of these, so if either collection is modified its hashCode will be affected. This class is used to build a tree hierarchy where it is stored in 'subelements' lists, so it is not surprising that it changed its hashCode and its fields inside those collections.

4.5.5 Results: classes and objects that do not enter collections

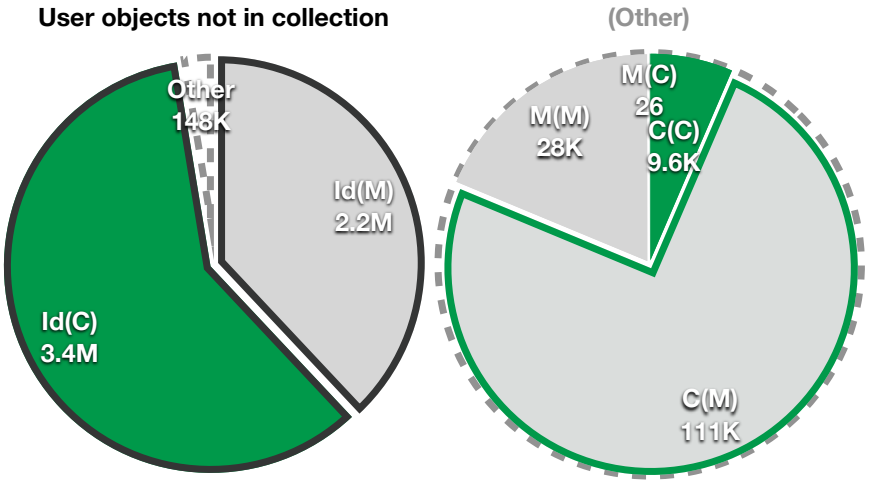
Figure 4.8 presents data for all user-defined (excluding java/javax) classes encountered whose instances did not enter any collections, and instances of those classes.

Discussion

Figure 4.8 shows a significant difference in the behaviour of objects that do not enter collections when compared with objects that do: identity-as-equality is the norm, especially for objects, whereas this is uncommon among objects that enter collections. Even among only those objects that do not use identity-as-equality, the majority have an equality that settles before the constructor ends. This is a surprising result: we expected that more classes would use mutable equality outside collections where they do not need to comply with collection requirements.



(a) User Classes Not In Collections



(b) User Objects Not In Collection

Figure 4.8: User classes and their objects that do not enter collections. The charts on the left show classes and objects in three categories: Id(M) — identity-as-equality and mutable fields, Id(C) — identity-as-equality and constructor-settled fields, and *Other* — which is expanded in the chart on the right. Within *Other* we see: C(C) — constructor-settled equality and constructor-settled fields, C(M) — constructor-settled equality and mutable fields, and K(M) — collection-settled equality and mutable fields. There were no classes or objects with collection-settled equality and constructor-settled fields (K(C)) or with mutable equality (M(M) and M(C)).

Most objects that do not use identity-as-equality also have mutable fields. This contrasts with objects in collections where mutable fields were relatively uncommon.

4.5.6 Threats to validity

While developing *#prof* and generating the results presented in this chapter we made several assumptions. If these assumptions do not hold then the results presented here may not be valid, so in this section we discuss those assumptions that could have the greatest impact if invalid.

Using `hashCode` to measure equality

We assume that programmers overriding `equals` also implement `hashCode`, and that the implementation depends on the same fields. Additionally, we assume that any change to `equals` will also cause a change to the `hashCode` value. This assumption is justified by appealing to the Java specification for the `equals` and `hashCode` methods, as discussed in Section 4.3.1. If programmers implement `equals` and `hashCode` methods to be inconsistent with the specifications then our observations of `hashCode` may not be representative of equality behaviour. We suspect that if this does occur, it is uncommon.

Even if the programmer has implemented `hashCode` to depend on the same fields as `equals` it is possible, due to hash collisions, that a change in `equals` will not cause a change in `hashCode`. This is unlikely to occur sufficiently often to affect our results.

Implementations of `hashCode` that cache the result

We assume that programmers using caching implementations of `hashCode` also code defensively to make sure that it is updated when the fields that it depends on change. It is possible that a programmer implements `hashCode` assuming it will not be called until after all the object's fields have been initialised (e.g. using post-constructor initialisation methods). If programmers make this assumption then, by calling `hashCode` as soon as the constructor ends, we break the assumption and cause the object to store a cached `hashCode` based on uninitialised state that will not be updated by subsequent calls to `hashCode`. This could cause the object to be incorrectly classified as C(M) when it could possibly be K(M) or M(M), as the `hashCode` should depend on post-constructor initialised fields but is instead determined at the end of the constructor by *#prof*'s explicit call to `hashCode`. It is

unlikely that this situation will result in incorrect program behaviour as at worst it could cause all objects to have the same hash code: a bad hash function but not an incorrect one.

Probably the most well-known class that caches its hash code is java.lang.String, which stores the computed value the first time hashCode is called and returns that value for subsequent calls. *#prof* cannot profile standard library classes so we do not observe this particular class, but there could be other classes with this behaviour. If we were able to observe the behaviour of String objects without *#prof* we would classify them based on their behaviour as C(M). Their fields change after the constructor because they write to their hashCode cache field, but their equality does not change because they always compute the same value for hashCode. In fact, *#prof* would also classify strings as C(M) for the same reasons.

Based on programming experience caching implementations of hashCode are not common, and when they do occur they tend to depend on constructor-settled fields (resulting in a correct C(M) classification). Nevertheless, it is possible that *#prof* classified too many objects/classes as C(M) instead of K(M) or M(M).

Representativeness of benchmarks

We assume that the selection of programs that we measured to obtain the results presented in this chapter are in fact representative of Java programs in general. This assumption is typical for researchers performing profiling, but is nevertheless a genuine threat to the validity of our results. It is possible that all the programs in our study exhibit behaviour that is not typical of other programs. We tried to select a sample of programs that did not share common features: some command line tools, some user-centric programs, etc. We also selected programs from different authors, and from a corpus specifically designed for research. However, all of the programs were open source, and all were relatively small compared to large enterprise programs. We were also unable to profile the whole corpus.

We made additional assumptions that the particular invocation of each program we measured was representative of the program's general behaviour, and we did not measure coverage to ensure that all possible program behaviour was executed. Whole-program dynamic analysis is not particularly common, but those examples we discussed in Chapter 2 (e.g. [13, 63, 96]) also made these assumptions. In fact, it is unlikely that most programs would achieve high code coverage from a normal execution as, from experience, most programs include code for handling exceptional conditions and future use that would not be exe-

cuted during normal execution. For example, a class that we classified as C(C) may have methods for modifying fields that were not called. We assume instead that, by generalising across multiple programs, we see overall trends in behaviour even if the classification of a particular class does not describe its behaviour in any possible program.

Finally, we did not normalise the contribution of objects and classes from individual benchmarks to the overall results presented in this chapter. This means that larger benchmarks like ant and trove had a larger contribution to the results than small benchmarks like fitjava and jFin_DateMath. We made this decision to avoid giving high weighting to classes from small benchmarks that have only one instance (singletons), as these classes could have a significantly different behaviour to regular classes.

4.6 Conclusion

This chapter presents our first approach to using runtime profiling to measure settling behaviour in Java programs. While it has several limitations, this work was very helpful for improving our understanding of this area and did yield some interesting observations:

- Most classes and objects use identity-as-equality, only around 6% of classes define equals and hashCode.
- Many objects and classes settle their fields during the constructor, more than half of classes and nearly half of objects.
- Non-default equals and hashCode are the norm among objects and classes that enter collections but uncommon among classes and objects that do not enter collections. This implies that equals and hashCode are primarily used for collections, either directly in the case of equality collections, or for "contains" operations in other collections.
- It is not uncommon for classes and objects that enter equality collections to have mutable fields: more than half of classes and a quarter of objects in these categories change their fields after their constructor returns. Identity-as-equality and collection-settled equality are common among these classes and objects.
- Equality implementations based on deep state are rare, or at least it is uncommon for equality to settle after an object's fields settle.

This chapter has discussed many limitations in our approach to profiling with #*prof*, but in light of the results we observed we consider these limitations to be most significant:

- #*prof* does not record information about individual objects. #*prof* performs all aggregation internally and only emits class summary information. This prevents after-the-fact aggregations from tracing the behaviour of a particular object and limited us to class-level observations in some cases.

- #*prof* cannot track the behaviour of standard library classes and objects. This is caused by AspectJ, which cannot modify classes which are included in rt.jar, the classes that the JVM loads on startup.

- The Qualitas Corpus, while the largest and most comprehensive benchmark suite available for Java, is not well-suited to runtime studies. Many of the benchmarks are not capable of running as applications and few of them included significant workloads that we could use for profiling. This severely limits the reproducibility of our results and their comparability to other studies.

- #*prof*'s is unable to profile applications that use a class loader. This severely limited the range of applications we could profile.

- #*prof* could not track arrays, creating a significant opportunity for error.

From an engineering perspective, the development of #*prof* was significantly impacted by not having an efficient means to identify objects. #*prof* distinguishes objects by their system hash code, falling back to == to resolve collisions. Its use of system hash codes interacted poorly with garbage collection resulting in frequent collisions. A better mechanism for uniquely identifying objects during and after the program execution would have aided development significantly.

Chapter 5

rprof: A General Object Profiler for Java

This chapter presents *rprof*, our second runtime object behaviour profiler for Java developed for this thesis. Like *#prof*, *rprof* characterises several object behaviour patterns, then records the frequency of objects exhibiting these behaviours in real-world Java programs. This chapter focuses on the contributions of *rprof*'s design and implementation, which led to the development of several novel techniques for whole-program dynamic analysis and especially state-independent aggregation. These techniques are broadly applicable to profiling object behaviour. This chapter also discusses numerous technical contributions. *rprof* development focused particularly on identifying and measuring the object-field settling behaviour patterns defined in Chapter 3. Chapter 6 presents results obtained by applying *rprof* to a selection of Java programs.

This chapter presents and discusses the following contributions:

- an object behaviour profiler capable of classifying the behaviour of classes, objects, and fields in real-world Java programs according to the classifications presented in Chapter 3.
- techniques for complete, exact tracking of Java objects including (almost) all standard library classes and objects, regardless of the class-load mechanism used by the program.
- a novel application of MapReduce [33] to event aggregation, allowing parallel, out of order aggregation.
- a modular design for object profiling – decoupling the generation and aggregation of events and using unique persistent identifiers for program enti-

ties – allows independent development of data acquisition and aggregation components.

Unlike #*prof*, *rprof* stores aggregate data for object-fields, allowing after-the-fact aggregations to be developed and refined without re-running the original benchmark. As a consequence, *rprof* requires significantly more resources than #*prof* — for both computation and data storage. Nevertheless, the extra data was invaluable for *rprof*'s development process and particularly for iteratively refining the aggregations that produce the results presented in this thesis.

This chapter is structured as follows:

- Section 5.1 introduces an example used throughout this chapter that gives an overview of the *rprof* approach to object profiling.
- Section 5.2 enumerates the major components of the *rprof* profiler and provides a high-level technical overview.
- Section 5.3 discusses *rprof*'s approach to tracking objects and generating object-related events.
- Section 5.4 presents details of *rprof*'s use of byte code modification for tracking method calls.
- Section 5.5 discusses tracking field access and modification.
- Section 5.6 details *rprof*'s approach to parallel data aggregation using in-memory MapReduce algorithms.

5.1 Example

This section presents a worked example that demonstrates the process *rprof* uses to generate data, and performs a simple hypothetical analysis. Subsequent sections utilise this analysis as a running example.

Figure 5.1 shows the small Java program used in this example. When executed, the code creates and initialises a new object of type `Example`. The code then calls the `hello()` method on the object, which returns a string. The program stores the string in a field belonging to the object. This program avoids I/O operations as they introduce complexity that is irrelevant to the analysis. Though simple, this program demonstrates all of the basic types of operations that *rprof* can track.

```
public class Example {
   String foo;
   public static void main(String args[]) {
      Example e = new Example();
      e.foo = e.hello();
   }
   public String hello() {
      return "Hello!";
   }
}
```

Figure 5.1: A running example for Chapter 4; this simple Java program creates an object, calls a method, and modifies a field.

5.1.1 Generating an event stream

rprof observes running programs and generates an event stream describing their behaviour. *rprof* can also analyse the event stream as it is generated.

Suppose the analysis tracks: object allocation, initialisation, and destruction; instance method calls; and field modifications. The example program – when executed on a JVM running *rprof* – generates the following event stream (omitting events related to JVM initialisation and termination):

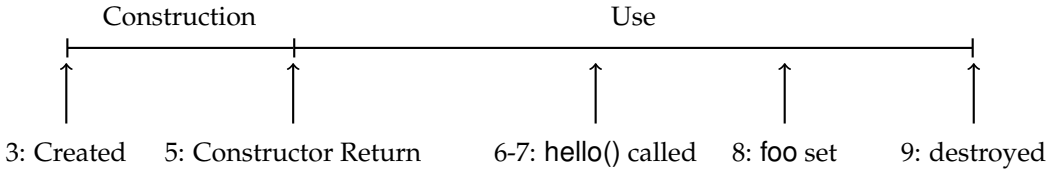
Event ID	Thread	Type	Class	Method/Field	Arguments (Object IDs)
1	1	*Method call*	`Example`	`<init>()`	`( )`
2	1	*Method call*	`Object`	`<init>()`	`( )`
3	1	*Object Init*	`Example`		`( 2 )`
4	1	*Method return*	`Object`	`<init>()`	`( 2 )`
5	1	*Method return*	`Example`	`<init>()`	`( 2 )`
6	1	*Method call*	`Example`	`hello()`	`( 2 )`
7	1	*Method return*	`Example`	`hello()`	`( 2, 3 )`
8	1	*Field store*	`Example`	`foo`	`( 2, 3 )`
9	`null`	*Object Free*			`( 2 )`

This event stream shows nine events that describe the behaviour of the program's objects. It shows two types of unique IDs: sequential event IDs 1-9 in the first column, and object IDs in the thread column (the thread objects) and in the arguments column. The stream refers to three object IDs: `1`, a thread object; `2`, an `Example` object; and `3`, the string "`Hello!`".

The event stream shows that the program initialised a new object using its default constructors (Events 1 and 2). Java's type system prevents programs from accessing freshly allocated objects until their constructors have run, so we do not

see a reference to the object's ID, 2, until the program reaches the default Object constructor (Event 3). This event also includes the type of the new object: Example. Events 4 and 5 show control flow returning through the object's constructors, then Event 6 shows the program calling Object 2's instance method, Example.hello(). Event 7 shows Object 2's method returning a previously unseen object: Object 3.[1] The program stores Object 3 in Object 2's field, foo, shown in Event 8. Finally, Event 9 records that the garbage collector has destroyed Object 3.

Using this information we can construct an object lifecycle diagram for Object 2 in the style of Röjemo and Runciman [89], using event ids to delimit phases (see Chapter 3):



Profiling a real program generates an enormous stream of events: some of the streams generated by running a real analysis were in excess of 500 GB. This excess of information makes it infeasible to extract useful information by manually inspecting the event stream and constructing life-cycle diagrams; we need an automatic approach.

5.1.2 Analysing the event stream

Extracting aggregate properties from a program requires analysis of the event steam. *rprof* uses a data-processing technique called MapReduce, a programming model developed at Google for writing algorithms that can scale from small test-cases to massively parallel computations spanning data-centers [33]. This style of programming simplified the process of reasoning about and prototyping whole-program dynamic analyses. *rprof* implements all data aggregations as context independent MapReduce algorithms.

MapReduce algorithms consist of two functions: map, which takes an input and produces a key and a partial record for that key; and reduce, which takes partial records for the same key and combines them. When all inputs have been processed and each key maps to one record the algorithm is complete.

Our running example continues in this section, analysing the event stream to determine the number of method calls and field writes performed per object.

[1] The full event stream would include JVM events prior to the main method running. These would show the Object 3's creation shortly after the JVM loaded the class file Example, and the creation of a thread object: Object 1.

Map

For our example, the map algorithm stage converts events from the stream into partial *instance records*: simple objects that record the number of method calls and field writes for a particular object contributed by a particular event. The records are identified by object ID *keys*: the first argument to the emit method.

```
void map(Event event) {
  if (event.type == OBJECT_INIT)
    emit(event.args[0], new Record(0, 0));
  if (event.type == METHOD_CALL && event.method.isInstanceMethod())
    emit(event.args[0], new Record(1, 0);
  if (event.type == FIELD_WRITE && event.field.isInstanceField())
    emit(event.args[0], new Record(0, 1);
}
```

This map function emits at most one record per event. More complex analyses might emit multiple records for a single event. The function emits partial records when the input is a method call or a field write. It also emits partial records for object allocations so that the aggregate data shows the total number of objects allocated — including objects that did not have any method calls or fields writes.

The following table depicts the output of running the map function on the nine events shown in the previous section. An empty line indicates that the function did not generate a record for that event.

Input	Key	Record	
(Event ID)	*(Object ID)*	Calls	Writes
1			
2			
3	2	0	0
4			
5			
6	2	1	0
7			
8	2	0	1
9			

As the table shows, the map function emitted three partial records for the example event stream — all for the same object, each with a different value for the record.

Reduce

The reduce algorithm stage combines partial results to produce a complete record for each object:

```
Record reduce(ObjectID id, List<Record> partialRecords) {
    Record result = new Record();
    for (Record input: partialRecords) {
        result.calls += input.calls;
        result.writes += input.writes;
    }
    return result;
}
```

This function takes an arbitrary number of partial records in any order and combines them, returning a new record in the same format. This example only has records for one object; when aggregating data from a real program reduce will combine partial records for many objects.

	Key	Record	
	(Object ID)	Calls	Writes
	2	0	0
Input	2	1	0
	2	0	1
Output	2	1	1

The reduce function combines the three partial records for Object 2 into a single instance record. The completed record shows that Object 2 received a single method call and a single field write during the program, consistent with the object life-cycle diagram in the previous section.

The example presented in this section is simple, but illustrates the fundamental concepts and processes used to perform much more complicated analyses. The remainder of this chapter discusses the design and implementation of *rprof*, a profiler that can generate event streams and provides a framework for performing this type of analysis.

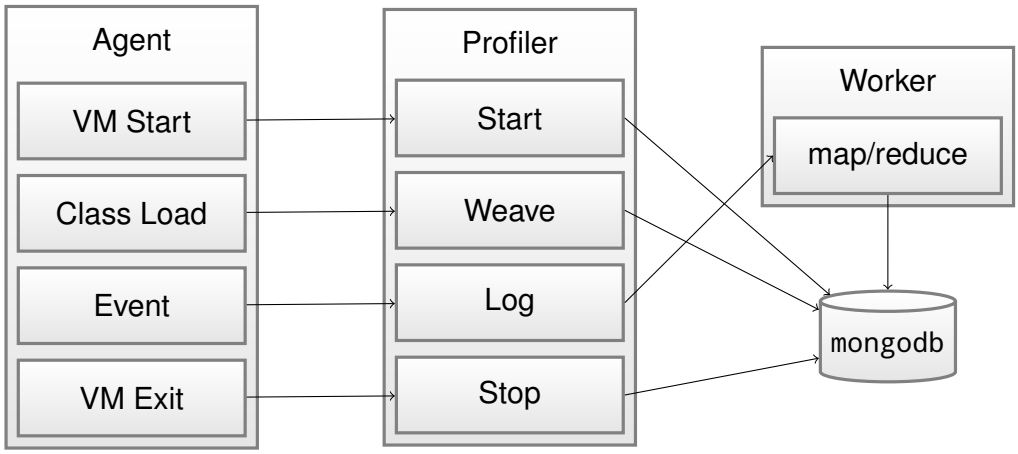


Figure 5.2: A depiction of *rprof*'s design, showing major components and the connections between them.

5.2 Design overview

The *rprof* profiler consists of four discrete components:

- The *Agent*, a `C` library loaded by the JVM running the profiled application.
- The *Profiler*, a Java application running in a separate JVM that provides utility functions for the *Agent* and routes its output.
- The *Worker*s, Java applications that perform aggregation of the event stream and handle storing the generated results.
- `mongodb`, a commercial *NoSQL* database used by the *Profiler* and *Worker* applications to store persistent data [28].

By running the *Agent* and the *Profiler* in different JVMs, *rprof* minimises the execution of additional Java byte code on the same JVM as the profiled application. This helps ensure that *rprof* does not affect the event stream generated by the *Agent*, while allowing the *Profiler* to use common Java libraries (e.g. for manipulating byte code and accessing `mongodb`). Figure 5.2 shows each of these components, their subcomponents, and the interactions between them.

5.2.1 Component interactions

The *Agent* and the *Profiler* each consist of four distinct sub-components that work in pairs: *VM Start* and *Start*, *Class Load* and *Weave*, *Event* and *Log*, and *VM Exit* and *Stop*.

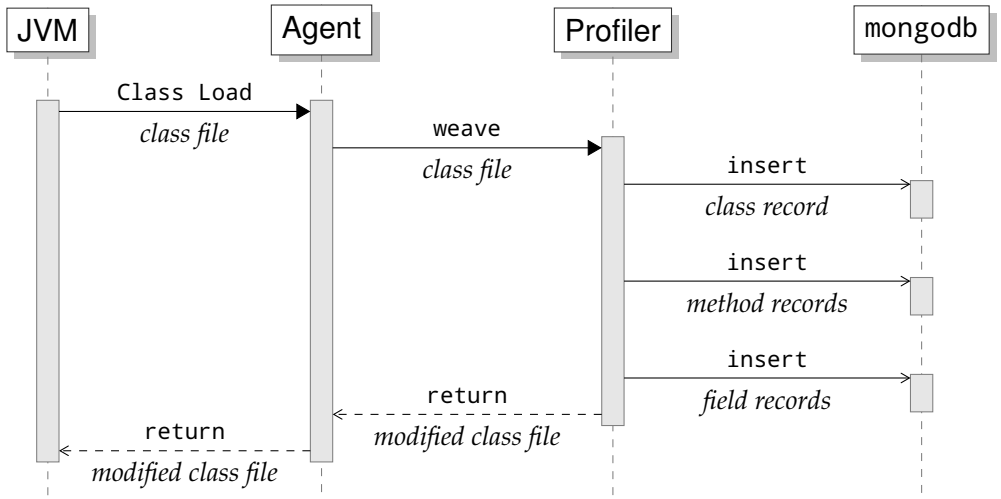


Figure 5.3: A UML sequence diagram showing the modification of a class file by *rprof*.

VM Start and *Start*

The *VM Start* sub-component handles *Agent* initialisation and notifies the *Start* sub-component with parameters describing the profiled application. *Start* creates a new database on the `mongodb` server, initialises it appropriately, and returns an ID to *VM Start*.

A single *Profiler* instance stores data and marshals workers for all profiler runs from the same batch. After initialisation, all additional communications between the *Agent* and the *Profiler* require an appropriate ID. The *Profiler* and *Worker* applications use the ID to locate the appropriate database.

Class Load and *Weave*

The *Class Load* sub-component handles class load events. When the JVM loads a class it triggers *Class Load*, which can modify the class before returning it to the JVM.

Class file modification is a common requirement for commercial and academic purposes. Several Java libraries exist for this task, but there are no comparable `C` libraries. Loading a Java library into the same JVM as the profiled application risks collisions between library classes and profiled classes, result contamination, and complicates JVM initialisation. *Class Load* avoids these problems by transmitting the new class file to *Weave* (that runs within a different JVM), as shown in Figure 5.3.

Before modifying the class file, *Weave* allocates unique IDs and creates records

describing the class, its methods, and its fields. *Weave* stores the IDs and records in mongodb so events can refer to IDs instead of using textual descriptions. *Weave* then uses the ASM byte code modification library [21] to modify the class file appropriately (see Sections 5.3.2, 5.3.4, 5.4.1, and 5.4.2).

Finally, *Weave* returns the modified class file to *Class Load*, which injects it back into the JVM.

Event and *Log*

Several mechanisms trigger the *Event* sub-component (discussed in Sections 5.3.2, 5.3.3, 5.4.1, and 5.5). When triggered, *Event* generates a new event record and appends it to the event stream.

Agent represents the event stream as an array of structs. *Event* stores these locally until a sufficient number accumulate, then transmits them to *Log* (this reduces communication overhead).

Log blocks *Event* until a *Worker* becomes available, preventing the *Agent* from generating data faster than the *Profiler* can process it. Having obtained a *Worker*, *Log* transfers control of the event batch and unblocks *Event*.

MapReduce and mongodb

The *Worker* uses MapReduce functions to map the event stream to appropriate aggregation records and store these records to mongodb [28]. *rprof* performs all data aggregations using MapReduce algorithms. These algorithms run concurrently on *Worker* processes, using mongodb as a backend (see Section 5.6).

NoSQL databases typically provide infrastructure for running MapReduce analysis and mongodb is no exception. mongodb's implementation of MapReduce, however, is not designed for high-performance aggregation of large amounts of data: it translates to and from JavaScript for each stage of the analysis causing unacceptable overhead and is single-threaded. Instead of using mongodb's MapReduce framework, *rprof* runs a custom MapReduce framework concurrently using *Worker* processes.

VM Exit and *Stop*

When the profiled application terminates and the JVM begins shutting down, it triggers *VM Exit*. *VM Exit* transmits any remaining batches of events to *Log*, then sends a notification of completion to *Stop*. *Stop* notifies the *Worker*s that the program has completed (see Section 5.6.2).

5.2.2 Technical overview

The *Agent* uses the JVMTI framework [75] to register `C` functions as callbacks for certain events that occur inside the JVM. Only the callbacks that interact with the *Profiler* appear in Figure 5.2 and merit discussion in subsequent sections. The *Agent* registers additional callbacks with JVMTI that provide essential functionality but are not relevant to this chapter (e.g. JVM startup, shutdown, GC).

rprof uses *TCP* for all interactions between major components, in most cases specifically *HTTP* requests. This serves two purposes:

- Simplified development: components implemented in different languages can interact using common libraries. This isolates faults to a single component and allows the use of commodity tools for analysing transmitted data, measuring throughput, and replaying interactions.

- Parallel computation: components can run in separate JVMs on different computers, distributing the workload between processors.

The *Profiler* uses Jetty's implementation of the HTTP Servlet 3.0 specification to handle requests [108, 66]. In particular, Jetty's *HTTP continuations* provided a simple mechanism for the *Profiler* to marshall *Worker*s waiting for event batches. The *Agent* uses the *cURL* library to implement requests [104].

rprof performs CPU-intensive or slow I/O tasks using the *Worker* applications. The *Worker*s can perform these tasks in parallel using multiple threads on multiple computers, without impacting the progress of the profiled application.

rprof utilises a high-performance NoSQL database, `mongodb` [28]. An early version of *rprof* attempted to use PostgreSQL, a relational database [84]. The overhead caused by maintaining referential consistency resulted in runtimes in the order of weeks for real-world Java programs. Switching to `mongodb`, designed for scalability and performance in a concurrent environment without requiring referential integrity, reduced this to hours.

5.3 Tracking objects

The example presented in Section 5.1 shows objects identified by unique IDs. These IDs allow *rprof* to identify events that reference the same object. As discussed at the end of Section 4.4.1, Java object memory addresses make poor IDs because the JVM's garbage collector moves objects during program execution;

an object's physical address may change at any point. *#prof* used Java's *identity hashcodes* to identify objects, obtained by calling a standard library method:

```
System.identityHashCode(Object x);
```

This method uses VM internal properties to consistently return the same integer for an object, even if the object moves. In practice we found that JVMs derive the integer from the physical address of the object when the method is first invoked, i.e. while the object was in the garbage collection nursery pool. As a result, all identity hash codes the JVM generated for *#prof* pointed to the small block of memory that contains the nursery pool: the most frequently reused heap space! This resulted in frequent hash collisions, which are not acceptable for a profiler like *rprof* that performs analysis after the program terminates.

If we exclude the identity hash code approach used by *#prof*, three options remain for uniquely identifying objects within the JVM:

- Use Java references (or weak references), as we did in Chapter 4.
- Extract actual memory addresses, track garbage collection events, and use these to maintain a mapping between addresses and objects.
- Generate a unique ID for each object and store the ID with the object.

Tracking objects using garbage collection knowledge is possible: JVMTI provides access to garbage collection events. Nevertheless, this approach presents additional challenges during analysis: the analysis would need to reconstruct a complete model of program execution to successfully identify objects after program termination.

Creating a unique ID and storing it with the object presents different challenges: generating unique IDs for objects created by different threads requires synchronisation and storing the ID requires either class modification or JVM support. In fact, JVMTI provides support for profilers and debuggers to associate state with Java objects. The remainder of this section discusses *rprof*'s use of JVMTI to associate unique IDs with Java objects:

- Section 5.3.1 describes the use of Java's profiling interface to associate persistent state with objects.
- Section 5.3.2 discusses object allocation detection.
- Section 5.3.3 explores tracking system objects allocated by native code.
- Section 5.3.4 considers tracking class objects.

5.3.1 Using JVMTI for object tracking

JVM developers recognised the difficulty for profilers to associate state with objects without program modification. Java 5 introduced JVMTI (Java Virtual Machine Tool Interface): a native interface to the JVM designed for profilers and debuggers [75]. The JVM uses JVMTI to load native libraries, which can register listeners for JVM events including: field reads and writes, class loads, and garbage collection events.

The JVMTI framework includes a method that allows JVMTI agents to associate a 64 bit (long) tag with any object (SetTag). The JVM maintains this tag itself, unlike a field inserted into an object through source or byte code modification. The JVM will inform the agent that it has collected an object even after its finalisers terminate and the JVM reclaims its memory. JVMTI documentation suggests using SetTag to store a pointer to a struct describing properties of the object stored in the agent's heap space. *rprof* uses it to store a unique object ID.

Figure 5.4 shows the actual source code *rprof* uses to tag objects using the JVMTI SetTag method. It also demonstrates retrieving an object ID (GetTag).

5.3.2 Detecting object allocations

JVMTI expects profilers to rely on byte code instrumentation to monitor object allocations; it does not generate events. Profilers can raise their own object allocation events at multiple points as object allocation on the JVM is a multi-step process. The first opportunity for tracking an object occurs when the call stack for an object's constructor reaches the default constructor for java.lang.Object. Java requires that all objects call a constructor before they can be used, and each constructor must call a super-constructor. *rprof* cannot access a fresh Java object reference until the chain of constructor calls reaches java.lang.Object (*top*). Fortunately, field accesses and instance method calls cannot occur until the constructor chain reaches java.lang.Object either.

rprof modifies the default constructor for java.lang.Object to generate events when any new object is created from Java code. Figure 5.5 shows the constructor after modification. This modification results in a call to the function shown in Figure 5.4 for every object created in Java. The JVM provides java.lang.Object as a bootstrapped class, rather than using a class-loader. Fortunately, JVMTI allows class modification of bootstrapped classes — this would not be possible using AspectJ or similar class modification tools.

```
void
RPROF_native_newobj(JNIEnv *env, jclass tracker, jthread thread,
                    jclass klass, jobject o, jlong id)
{
  jlong type, threadId;
  jvmtiEnv *jvmti = gdata->jvmti;
  EventRecord *event = CreateEvent(jvmti, 1);

  if (id == 0) { /* generate an object id */
    enterCriticalSection(jvmti); {
    id = generate_object_tag();
    }; exitCriticalSection(jvmti);
  }

  /* tag object with id */
  tag_object(jvmti, o, id);

  /* retrieve class id */
  type = get_tag(jvmti, klass);

  /* retrieve thread id */
  threadId = get_tag(jvmti, thread);

  event->type = RPROF_OBJECT_ALLOCATED;
  event->thread = thread;
  event->type = type;
  event->args_len = 1;
  event->args[0] = id;

  /* send the event to the profiler */
  comm_log(gdata->comm, event);

  deallocate(jvmti, event);
}
```

Figure 5.4: The C function used by *rprof* to tag an objects and generate allocation events. *rprof* injects code into java.lang.Object.<init>() (the root method in the constructor hierarchy) that calls this method *via* JNI, passing a reference to the current thread, the class of the new object (found using reflection), the new object, and a fresh ID for the new object (generated without blocking). If this function is called before the JVM has fully initialised the Java code cannot generate an ID, so this method *blocks*, generates an ID, then continues. This function is a simplified version of the actual function used by *rprof*.

```
package java.lang;
public class Object {
  public Object() {
    Class<?> type = this.getClass();
    nz.ac.vuw.ecs.rprof.Tracker.newobj(type, this);
  }
  ...
}
```

Figure 5.5: java.lang.Object after modification by *rprof*. All freshly created objects must call a constructor, which calls a super constructor, which eventually results in this method being called. *rprof* actually modifies byte code not Java code, this code approximates the modification.

5.3.3 Tracking system objects

The JVM loads native agents such as *rprof*'s *Agent* before executing any byte codes, loading any classes, or creating any objects. Nevertheless, many JVMTI functions are not available until the JVM has been initialised so some objects are created before the *Agent* can track object creations. Once the JVM has finished initialisation, the *Agent* uses JVMTI to iterate over all previously created Java objects, allocating them unique ID tags and generating events for each object.

Programs allocate some objects using processes that *rprof* cannot detect using byte code modification: the JVM generates some objects directly, and native code creates objects using JNI. JVMTI provides a callback for these object allocations that the *Agent* uses to tag the objects and generate events recording their creation.

5.3.4 Tracking class objects

Many of the objects created before the *Agent* can track object allocations are *class* objects: special meta-objects that represent static classes and offer access to static properties. The *Agent* uses these class objects to identify the types of regular Java objects using *reflection*, but it must first tag each class object with the persistent class ID that the *Profiler* generated for that class.

To facilitate class tagging, the *Profiler* sends the persistent class ID it has generated to the *Agent* with the modified class file. The agent cannot access the class object until the class has been loaded by the JVM, so it stores the class ID in a list of untagged classes, along with the *fully qualified class name*, which can uniquely distinguish the class within the JVM. The *Agent* uses three mechanisms to assign IDs to class objects:

- In most cases as soon as a class has been loaded, JVMTI will notify the *Agent* that the class is available. The *Agent* can tag it by looking up the class ID in its list of untagged classes.

- During JVM initialisation JVMTI will not notify the *Agent* that a class is available. When initialisation concludes the *Agent* iterates through the list of untagged classes, retrieves class objects by name, and tags them appropriately.

- JVMTI never notifies agents when object array classes are available. When these untagged classes are first encountered by the *Agent* it will use reflection and JNI functions to tag them correctly.

In addition to tagging class objects correctly for identifying object types the *Agent* uses class objects to generate field events (discussed in Section 5.5). The first approach to class object tagging is the most desirable because it ensures that *rprof* does not miss field events for those classes. *rprof* will miss field events that occur during JVM initialisation; this is unavoidable as the requisite JVMTI functions are not available. Object arrays do not have fields so *rprof* will not miss any field events for classes tagged using the third approach.

5.4 Tracking methods

Section 5.1 shows an example analysis that tracks constructors and instance methods. *rprof* is capable of generating events for all method calls, returns and exceptional returns (where the method terminates by throwing an exception).

Consider the following Java method:

```
public boolean isHello(String message) {
  return message.equals("Hello!");
}
```

This method takes a string as input and returns true if the message is "Hello!". Compiling this method with Oracle's Java 7 compiler yields the Java byte code presented in Figure 5.6.

rprof can generate three different method events for the byte code method in Figure 5.6: a *'Method call'* event at 0, a *'Method return'* event at 6, and an *'Exceptional return'* event if 3 throws an exception. The method does not check for undefined message parameters, so the method call at offset 3 could throw a NullPointerException.

```
public boolean isHello(java.lang.String);
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=2, args_size=2
       0: aload_1
       1: ldc           "Hello!"
       3: invokevirtual String.equals(..)
       6: ireturn
```

Figure 5.6: The byte code resulting from compiling Section 5.4's example method. The byte code instruction 0 pushes the first parameter onto the stack, then instruction 1 pushes the string constant "Hello!". Instruction 3 invokes the String.equals() method, and the JVM interprets this as a virtual call to equals on the method parameter (bottom of the stack) that takes the string constant as a parameter (top of the stack). The result of the method call to equals will be pushed onto the stack, then returned by instruction 6.

5.4.1 Generating method events

JVMTI can generate events for every method call, but explicitly recommends against using these events as they incur a substantial performance penalty. Most analyses will not require *rprof* to track all methods, so *rprof* uses byte code modification to instrument particular methods as the JVM loads them (this is the approach JVMTI recommends) — this will considerably reduce the overhead of analyses that do not require substantial method tracking. Figure 5.7 shows the modified byte code *rprof* generates for the isHello example method to track method entry, exit, and exceptional return.

For each event, the *Profiler*'s weaver component inserts byte code into the appropriate part of the method using ASM. The inserted code calls a static *rprof* method. Figure 5.7 shows that enabling method tracking causes *rprof* to add byte codes to the beginning of the method. The inserted code pushes class and method identifiers onto the stack, constructs an array containing the arguments to the method, then invokes the static *rprof* method for tracking method entry events. The generated code stores each object parameter to the method, including the this argument where applicable.

rprof handles method returns similarly to method calls. Before each **return** (or equivalent) byte code, the weaver injects byte codes to generate a return event. The inserted code gathers class and method identifiers, the this parameter, and any return value. The code then sends them to the profiler by invoking the static *rprof* method for tracking method returns.

rprof generates exceptional returns by wrapping the whole method body in a

```
public boolean isHello(java.lang.String);
  flags: ACC_PUBLIC
  Code:
    stack=6, locals=2, args_size=2
       0: sipush        classid
       3: sipush        methodid
       6: iconst_2
       7: anewarray     java.lang.Object
      10: dup
      11: iconst_0
      12: aload_0
      13: aastore
      14: dup
      15: iconst_1
      16: aload_1
      17: aastore
      18: invokestatic  enter(..)
      21: aload_1
      22: ldc           "Hello!"
      24: invokevirtual String.equals(..)
      27: sipush        classid
      29: sipush        methodid
      32: aload_0
      33: invokestatic  exit(..)
      36: ireturn
      37: astore_1
      38: sipush        classid
      40: sipush        methodid
      42: aload_1
      43: invokestatic  exception(..)
      46: aload_1
      47: athrow
  Exception table:
    from    to  target type
    0       37  37     Class java/lang/Throwable
```

Enter

Exit

Exception

Figure 5.7: The byte code resulting from modifying Section 5.4's example method to track method entry, exit, and exceptional return. The number of added byte codes are related to the number of method parameters and method returns, and significantly less than the byte codes added by AspectJ for *#prof*. We did not encounter any methods that exceeded the JVM method size limit after *rprof*'s modifications.

try block. It inserts a finally handler at the end of the method that creates an event by passing the class and method identifiers and the exception to *rprof*'s exception tracking method. *rprof* inserts the try block last so that any other catch or finally blocks run first. If a block consumes the exception and returns normally then *rprof*'s exceptional return handler will not run.

rprof's static tracker methods and the resulting JNI calls that map the parameters to object IDs and log the events add a constant-time overhead to modified methods. The overhead's effect on the execution time of a program depends on the nature of that program. Predominantly iterative programs experience insignificant overhead, while programs that rely on recursion for computation become unwieldy. Using *rprof* to instrument every method call resulted in programs that took hours to reach the main method, a process that takes only seconds without method tracking enabled. As a result, although *rprof* can weave all methods, in practice it only tracks methods relevant to the current analysis.

5.4.2 Modifying constructors for profiling

Constructor methods differ from instance methods: the instance parameter (this) remains inaccessible until the super constructor returns. Rather than delay the method entry event, *rprof* generates the event as usual but omits the instance parameter. The analysis can infer the instance parameter from the corresponding method return event, if required. Constructor methods also require special exceptional return handling: Java's method verification algorithm prevents constructor exception handling code from accessing the instance if the handler could have been triggered by the super constructor, or before the super constructor was called.

5.5 Tracking fields

Our running example from Section 5.1 relies on the profiler generating events when the program modifies a field. *rprof* supports generating events for both field writes and field reads of particular fields by registering each field with JVMTI.

When a program reads or writes to a registered field, JVMTI performs a callback to the *Agent*. The callback provides a reference to the object that owns the field, the value of the read or write operation, and the JVM field ID. JVM field IDs are internal offset types used to locate a field slot within an object record.

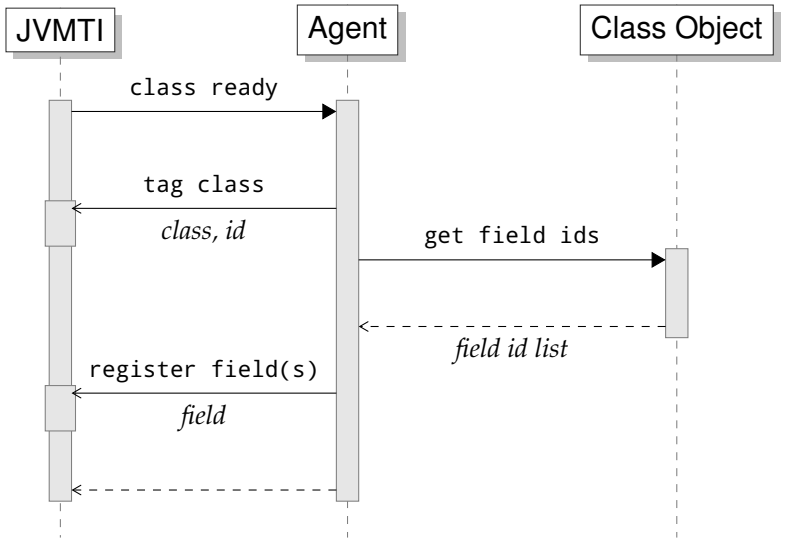


Figure 5.8: A sequence diagram showing field registration in the context of class loading.

They are only useful with the appropriate class reference which is only available while the program is running. *rprof* generates persistent field IDs and the *Agent* maintains a map between JVM-internal field IDs and *rprof*'s persistent field IDs, rather than using the textual field descriptions that JVMTI can provide.

To register a field for tracking, *rprof* must obtain its JVM field ID. *rprof* cannot register a field until after the JVM loads the class and generates its internal field IDs. Section 5.3.4 discussed *rprof*'s mechanisms for detecting when a class is ready. Figure 5.8 shows a sequence diagram for the process *rprof* uses to register fields in the context of class loading.

In addition to the class file modifications described in Section 5.4, *rprof* uses the class weaving process to identify the fields defined by a class and generate their unique IDs. *rprof* stores a mapping of these IDs to records describing the fields in mongodb.

The *Profiler* sends a list of *rprof* field IDs for each class to the *Agent* by adding a special static method to each class that creates and returns a Java array containing the field IDs in the same order that JVMTI presents the fields the *Agent*. This allows the *Agent* to retrieve the field IDs when it is ready to register the fields with JVMTI, causing reads and writes to those fields to generate callbacks.

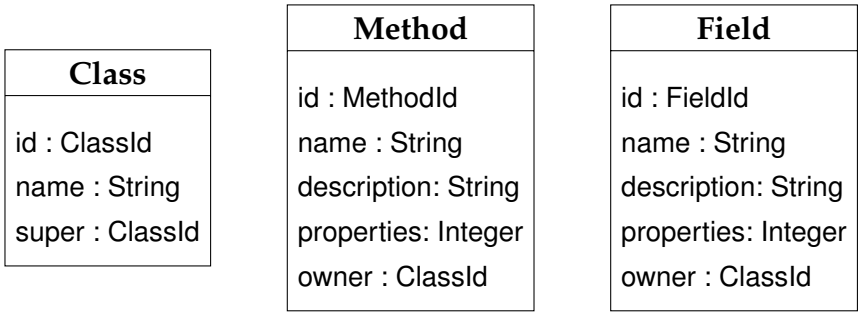
This concludes our discussion of the JVMTI *Agent* and generating the event stream. The final section in this chapter discusses aggregating and storing the event stream in the *Profiler* and *Worker* applications.

5.6 Data aggregation and analysis

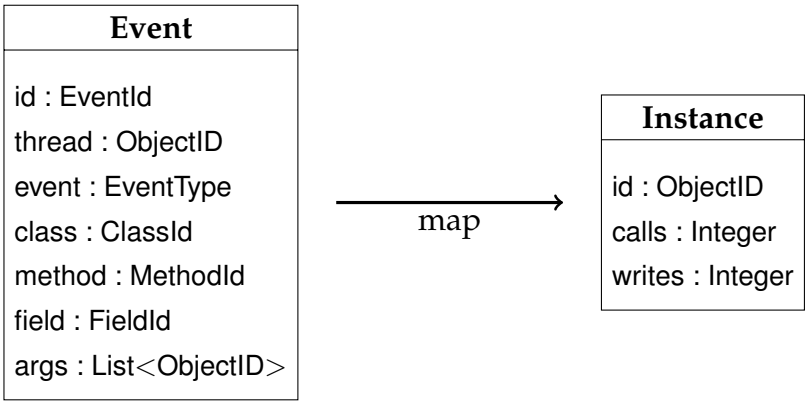
This section discusses the novel use of in-memory MapReduce algorithms for analysing a profiler event stream. It begins by elaborating on the data models for the event and instance records introduced in Section 5.1, then gives an overview of the tasks performed by the framework when running a typical analysis. Finally, it discusses some details of the caching system designed to improve performance by minimising disk writes for partial results.

5.6.1 Analysis data model

While profiling programs, *rprof* builds and maintains a representation of the program's static structure by recording information about the classes it weaves. After modifying each class, the *Profiler* records information about the class, its methods, and its fields to mongodb. The information is available later during the analysis in the following form:



As the profiled application runs, the *Agent* generates event records. *rprof* represents event records as structs within the *Agent*, as binary chunks in the wire protocol, and in the following format when they reach *Worker* processes:



As in Section 5.1, the map operation processes event records and *emits* partial instance records (InstanceId/Instance pairs). reduce combines partial instance

records, ultimately producing a single instance record for each instance. *Worker* processes store complete instance records to mongodb.

The analysis does not require that events are aggregated in order, so aggregation can begin before a profiled program terminates. *Worker*s can process events in parallel and out of order without affecting the results or requiring synchronisation.

5.6.2 Optimising MapReduce: caching partial results

The event streams generated by *rprof* from real-world programs contain billions of events. Storing all events, even compressed, consumes a large amount of disk space and takes time: as much as several days for large programs. For a typical program, the event stream will contain hundreds or even thousands of events for each object. *rprof* can significantly reduce the time and space required to store its output by converting events to instance records as the *Agent* generates them, rather than storing events and performing the conversion later.

Unfortunately, *rprof* cannot directly convert events into instance records: map produces partial instance records that must be combined with the partial records generated from all other events involving that object. Complete records can only be assembled once all events for a particular object have been generated. Running map alone to convert events to partial instance records does not necessarily reduce the amount of data *rprof* must store: every event record can produce one or more partial instance records — potentially resulting in more data to store than simply storing the original event stream. Unlike event records, *rprof* can easily compress partial instance records by aggregating them using reduce. Running both map and reduce eagerly can significantly reduce the amount of data that *rprof* must store, but even after running MapReduce the instance records that *rprof* generated for the largest program we profiled consumed 60GB, far more data than the memory available in a typical computer can store.

Rather than attempting to store all partial instance records in memory, *rprof* maintains a cache of partial instance records that it flushes to mongodb when necessary using a 'least recently used' policy. When *rprof* flushes a partial record to mongodb it first checks whether mongodb already contains a partial instance record for that object and combines the partial records using reduce if necessary.

As shown in Figure 5.2, *rprof* uses *Worker* applications to aggregate events in parallel. Ideally, events for a particular instance would be assigned to a particular *Worker* to avoid duplicating partial records, but in practice this is not possible: events may describe multiple instances and *rprof* cannot predict which instance

the event is for without running map. Instead, *rprof* splits the event stream into batches (to reduce communication overhead) and assigns an entire batch to a single *Worker*. The *Agent* creates batches by accumulating event records as it creates them. It sends complete batches to the *Profiler*, which distributes them to *Worker*s for processing and storage (Section 5.2). *Worker*s iterate over event record batches, running map on each event record to produce partial instance records. *Worker*s store partial instance records in a hash map-based data structure that maps instance IDs to partial instance records. If map emits a partial instance record for an instance ID that is already in the hash map, then the *Worker* aggregates the partial instance records using reduce. If the hash map does not contain a partial instance record for the instance ID, then the *Worker* inserts it. When the *Worker* does not have enough space to store more records it flushes the least recently modified partial instance records from the hash map into mongodb. Once the profiled application has terminated (the *Profiler* receives a *stop* message) the *Profiler* sends a special message to each *Worker*, causing them to flush their cache of partial instance records in the database. When all *Worker*s have completed flushing their caches, mongodb's instance record table is complete.

5.6.3 Optimising MapReduce: flexible caching

rprof uses a custom caching system to keep commonly referenced records in memory, while flushing less frequently used records to disk when memory becomes scarce. It uses the JVM's ability to monitor heap usage to flush the least recently used instance records into the database when heap space is low following a garbage collection.

The caching system uses a priority queue of fixed-size instance record hash maps, where the oldest hash map in the queue contains the least recently modified instances records (we assume that recently modified instance records are more likely to be modified again, so they should be cached in memory). When a cache hit occurs, the *Worker* aggregates the new partial instance record with the existing record using reduce and stores the result in the most recently allocated map. If this insert caused the map to reach its maximum size, the *Worker* allocates a new map and adds it to the queue. When the JVM alerts the *Worker* that heap memory is near exhaustion (90%), the collection flushes partial instance records from the cache into the database. The caching system flushes partial instance records, starting with records from the oldest map, and proceeding until it has flushed half the allocated maps. After the flush has completed the *Worker* triggers JVM garbage collection, then repeats the previous step until the JVM re-

ports that heap is half empty. This caching system allows *Worker*s to maintain a coarse-grained priority ordering of instance records while also providing hash-based access to the records.

5.6.4 Aggregating instance records

As *rprof* profiles a program it generates and stores a table of instance records to mongodb that describes the behaviour of all objects in the program. The example presented in Section 5.1 generates a single instance record. Profiling a real program would produce more instance records than a human can examine individually, so extracting useful information requires aggregating this table.

Consider again the example analysis from Section 5.1. The output table consists of instance records containing the number of field writes and method calls that occurred for each object in the table. Here are some questions that this table could answer:

- How many objects did the program create?
- How many objects had at least one method call?
- How many objects had at least one method call, but no field writes?

Answering these questions requires aggregating the data from the instance table. To do this we employ another MapReduce algorithm:

```
void map(Instance input) {
  Result output = new Result();
  output.instances = 1;
  if (input.calls > 0) output.haveCalls = 1;
  if (input.writes == 0 && input.calls > 0) output.haveCallsNoWrites = 1;
  emit(0, output);
}
```

This map operation emits a partial result for every instance that records:

- the contribution of that instance to the total number of instances.
- whether that instance had any method calls.
- whether that instance had method calls, but no writes.

By combining these partial results together, reduce can produce a single result object that describes the behaviour of all instances:

```
Result reduce(List<Result> inputs) {
  Result output = new Result();
  for (Result input : inputs) {
    output.instances += input.instances;
    output.haveCalls += input.haveCalls;
    output.haveCallsNoWrites += input.haveCallsNoWrites;
  }
  return output;
}
```

This reduce operation sums the tallies for each input. Once it has merged all of the partial results, the final Result object will contain the answers to the questions above. Note that map uses '0' as the key for its emits. This is because the algorithm combines all the partial records together, so as long as they all have the same key it doesn't matter what the key is.

This example did not produce any information about classes, or discriminate between different methods and fields. The instance records used by the analyses presented in Chapter 6 contain information about fields, methods, and classes that allow several different instance aggregations (by class, instance, and field). These aggregations follow the same style as the analysis in this section, but are considerably more complex.

Chapter 6

Profiling Initialisation Behaviour

In Chapter 4 we observed that many classes and objects either do not settle, or settle after the constructor. *#prof* only detected constructor-settled fields, assuming that all other fields were mutable. Many of the classes and objects that entered collections and had mutable fields also had collection-settled equality. Perhaps these objects' fields were actually collection-settled but *#prof* couldn't tell us?

This chapter uses *rprof*, the profiler described in Chapter 5, to measure the settling behaviour of field declarations and objects. It classifies field declarations and objects using the settling behaviour definitions from Chapter 3. We begin this chapter by introducing the programs whose behaviour we examine in this chapter in Section 6.1 then detail our experimental method in Section 6.2. This chapter presents the following contributions:

- Section 6.3 presents a study of field declaration settling behaviour. This study complements the work of Unkel and Lam, who presented a static study of field declaration initialisation that concluded that final fields are infrequently used but often applicable to field declarations in Java programs, and that a read-settled field annotation would be applicable to an even greater proportion of field declarations than final is [109]. Our study verifies their results using dynamic analysis and we additionally identify constructor-settled field declarations and compare them to the classifications used by Unkel and Lam. By confirming Unkel and Lam's established results, this section also validates *rprof*'s design and implementation.

- Section 6.4 presents a study of object settling behaviour, motivated by the observation made in Chapter 4 that many objects that have settled equality do not have constructor-settled fields. We postulate that many of these objects do in fact settled, but after the constructor returns. In this section

we will identify and compare constructor-settled, equals-settled, collection-settled and read-settled objects. In addition, we will compare the behaviour of settled objects to settled classes to determine whether settling is a predominantly class-specific property, or whether it is independent of class (object-specific).

This is the last contribution chapter in this thesis, and we will summarise our conclusions from this chapter and the contributions of this thesis in the final chapter, Chapter 7.

6.1 Benchmarks

This chapter presents results obtained by analysing the *DaCapo* benchmark suite, a compilation of non-trivial real world Java applications designed for benchmarking [15]. The DaCapo suite consists of 14 applications that are listed in Figure 6.1. The DaCapo suite packages all benchmarks into a single *jar* file that includes code for extracting, setting up, and executing each benchmark. The DaCapo benchmark runner is capable of running benchmarks several times and provides hooks for profiling tools to monitor each run. We did not use these facilities, instead choosing to profile the whole application including the DaCapo bootstrapping process. Each benchmark was executed one at a time using the default size, a single iteration, and a single thread to drive the benchmark (some benchmarks inherently use many threads — this parameter only affects the number of threads driving the benchmarks). We used the following command to execute benchmarks:

```
java [rprof−opts] −Xint −Xmx1024m −jar dacapo−9.12−bach.jar −n 1 −t 1 [benchmark]
```

Table 6.2 presents some statistics about each benchmark obtained using *rprof*, including the number of classes (including interfaces), methods, fields, the number of objects captured, the number of *rprof* events generated, and the total time (in minutes) required to execute and analyse each benchmark.

Two benchmarks in particular: tradebeans and tradesoap were problematic. These benchmarks both use the Geronimo framework [107] to perform various tasks using remote procedure calls implemented using HTTP. These include hard-coded time-limits in two places that will cause the benchmark to fail if a request does not complete successfully within these time-limits. The first is within DaCapo's bootstrapping: the client worker thread will attempt to contact the server at regular intervals until it becomes available for requests. If the server

avrora	Simulates a number of programs run on a grid of AVR microcontrollers.
batik	Produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.
eclipse	Executes some of the (non-gui) jdt performance tests for the Eclipse IDE.
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file.
h2	Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application.
jython	Inteprets the pybench Python benchmark.
luindex	Uses lucene to index a set of documents; the works of Shakespeare and the King James Bible.
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.
pmd	Analyzes a set of Java classes for a range of source code problems.
sunflow	Renders a set of images using ray tracing.
tomcat	Runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages.
tradebeans	Runs the daytrader benchmark via a Java Beans to a GERONIMO backend with an in-memory h2 as the underlying database.
tradesoap	Runs the daytrader benchmark via a SOAP to a GERONIMO backend with in-memory h2 as the underlying database.
xalan	Transforms XML documents into HTML.

Table 6.1: Programs in the DaCapo benchmarks suite (dacapo−9.12−bach), including a brief summary of functionality [15].

takes too long to 'boot' the client will fail. The second timeout occurs within the axis framework, used by Geronimo. A request that takes longer than 10 minutes will be terminated with a server error. In both cases, the overhead caused by *rprof* triggered the timeouts causing the benchmarks to fail. We were able to increase the timeouts by modifying the appropriate class files directly without recompiling the benchmarks, ensuring the minimum change necessary for the benchmark to execute successfully within *rprof*.

rprof tracks all objects created within the JVM, but it is not able to track all fields. *rprof* does not track fields of the following classes:

- java.nio.charset.CharsetDecoder
  java.nio.charset.CharsetEncoder
  java.util.zip.ZipFile
  These three classes use some form of JVM optimisation that causes memory faults if *rprof* tracks them.

Program	Classes	Methods	Fields	Objects	Events	Time (m)
avrora	999	10685	4649	1596K	843M	95
batik	1814	21710	8410	762K	32M	16
eclipse	2653	37949	18123	24159K	2096M	246
fop	1703	20133	8574	2043K	46M	21
h2	934	14622	5603	66183K	1488M	762
jython	2953	34167	11612	31203K	768M	250
luindex	788	10887	4146	219K	124M	19
lusearch	701	9640	3396	6747K	631M	111
pmd	1328	18281	6788	5170K	152M	48
sunflow	907	12854	5010	60222K	3522M	592
tomcat	2373	32369	12939	4362K	130M	46
tradebeans	8155	96250	38068	46588K	780M	321
tradesoap	8246	97026	38278	48758K	1330M	343
xalan	1125	14260	5549	6395K	486M	93

Table 6.2: Observed properties of the DaCapo benchmarks

- java.lang.Throwable
  The JVM generates an additional field at runtime which causes off-by-one errors in our tracking code.

- java.lang.String
  String is excluded because it is so common: Strings are logically immutable but they use internal fields to track access behaviour and generated properties, resulting in a disproportionate number of events.

rprof can track fields of the following generated classes but we chose not to because they do not conform to the behaviour of regular objects. In particular, generated accessor classes have final fields that are actually initialised before the object's constructor runs:

- sun.reflect.generated*
  JVM classes generated for reflection

- *ByCGLIB*
  Classes generated CGLIB, a byte code modification library used extensively by the day-trader benchmarks

6.2 Experimental setup

This chapter presents results obtained using *rprof* to instrument and monitor a selection of Java programs. Programs were executed on an Opteron 254 (2.8GHz)

dual-CPU machine with 4GB of memory running Ubuntu 10.04.3 LTS (64 bit server) using OpenJDK 1.8.0-ea-b37.[1]

We used the preview Java 8 build because our analysis is not stable on previous JDK versions. OpenJDK 1.8.0-ea-b37 includes a bug fix for a problem with JVMTI which we identified and reported [67]. Although we developed a work-around for this particular bug, *rprof* remains unstable on previous builds, occasionally triggering a memory access violation during JDK garbage collection in long-running programs which suggests that there are additional related issues.[2]

6.3 Field declaration settling behaviour

This section presents a study of field declaration settling behaviour, observing 65,785 fields from the programs in the DaCapo benchmark suite. This study confirms the results obtained by Unkel and Lam who used a conservative static analysis [109].

Unkel and Lam observed *final fields*, *undeclared-final fields*, and *stationary fields*. In Chapter 3 we noted that these terms correspond to final field declarations (F), constructor-settled and write-settled field declarations (excluding final) (CW\F), and read-settled field declarations (R) in our terminology. We present and discuss our runtime measurements of these properties.

6.3.1 Read-settled and final field declarations

Figure 6.1 presents the results of our analysis of the DaCapo benchmarks for all fields. The first column shows the name of the benchmark, the second shows the total number of fields contributed by each benchmark. All other columns show the percentage of the total number of fields in that category. Results are separated broadly into read-settled and not-read-settled field declarations, then into final (F), undeclared-final (CW\F), and other field declarations (¬CW). The last

[1] The batik benchmark relies on a proprietary jpeg class that is not included in the pre-release JDK 8 build. For this benchmark we used Oracle JDK 1.7.0_03-b04.

[2] The bug we reported and has been fixed was due to the Hotspot's non-standard use of registers on x86 and x86_64 architectures. Hotspot does not follow either Microsoft's calling conventions or SystemV conventions. In the particular bug we identified Hotspot did not protect one of the SSE extension registers which is designated callee-save by SystemV 64-bit conventions before calling JVMTI code. The register was being modified by memset, which was called from our code. It is likely that a similar issue triggers the garbage-collection memory access violation, but identifying these issues is extremely time-consuming and the problem does not occur after the wide-spread changes introduced to fix a related issue in Hotspot build 24.0-b08, which is included in OpenJDK 1.8.0-ea-b37.

Program	Total	R (%)			¬R (%)			Total (%)		
		F	CW\F	¬CW	F	CW\F	¬CW	F	CW	R
avrora	938	49	19	18	0	0	14	49	68	86
batik	1,314	2	56	20	0	0	22	2	58	78
eclipse	5,406	8	36	27	0	0	30	8	43	70
fop	2,218	5	44	33	0	0	18	5	49	82
h2	1,009	14	35	29	0	0	22	14	49	78
jython	1,271	10	50	20	0	0	20	10	60	80
luindex	789	17	33	18	0	0	32	17	50	68
lusearch	469	7	46	23	0	0	23	7	54	77
pmd	955	9	41	31	0	0	19	9	50	81
sunflow	506	11	30	31	0	0	28	11	41	72
tradebeans	12,381	26	33	22	0	0	18	26	59	81
tradesoap	12,572	25	33	23	0	0	18	25	59	81
tomcat	4,476	6	40	28	0	0	26	6	46	74
xalan	1,209	6	41	31	0	0	22	6	47	77
Total	45,513	18	36	24	0	0	21	18	54	79

Table 6.3: Read-settled (R) and final (F) field declarations excluding java.* and javax.*

three columns in the table show summary information: the total number of final (F), constructor-settled and write-settled (CW), and read-settled (R) field declarations. The information is also presented in the stacked bar chart at the bottom of the figure, where the total height of each bar corresponds to all read-settled field declarations (R), the lowest two segments together show fields that have final behaviour (both constructor-settled and write-settled (CW)), and the lowest segment shows final (F) field declarations. All percentages have been rounded to the nearest percentage point and segments smaller than 1% are omitted from the chart (we observed a total of 171 CW fields that were not in R).

Table 6.3 shows the same data for all field declarations excluding fields declared in system classes (java/javax).

Discussion

These tables show that between 70% and 86% of the field declarations in DaCapo benchmarks are read-settled (R) and between 41% and 68% of those fields showed final-like behaviour (CW). The number of final field declarations (F) varies between benchmarks but in most cases less than half of field declarations whose behaviour was final (CW) were declared final (F).

Comparing our results to Unkel and Lam's, we see that the proportion of final field declarations in our benchmarks was slightly higher. This is largely because of three specific benchmarks: avrora, which has a disproportionately high number

Program	Total	R (%)			¬R (%)			Total (%)		
		F	CW\F	¬CW	F	CW\F	¬CW	F	CW	R
avrora	1,702	36	31	16	0	0	17	36	67	82
batik	3,272	8	52	17	0	0	23	8	60	77
eclipse	6,779	10	37	25	0	0	28	10	46	72
fop	3,390	9	43	28	0	0	19	9	53	81
h2	2,014	17	39	22	0	0	22	17	56	78
jython	2,501	14	46	19	0	0	22	14	60	78
luindex	1,674	18	38	16	0	0	26	18	57	73
lusearch	1,303	14	43	21	0	0	21	14	58	78
pmd	1,962	14	41	24	0	0	20	14	55	79
sunflow	1,831	14	43	20	0	0	23	14	57	77
tradebeans	15,404	25	35	21	0	0	19	25	60	81
tradesoap	15,611	25	35	21	0	0	19	25	60	81
tomcat	6,279	9	40	25	0	0	26	9	50	74
xalan	2,063	11	41	27	0	0	21	11	52	78
Total	65,785	18	38	22	0	0	21	18	57	78

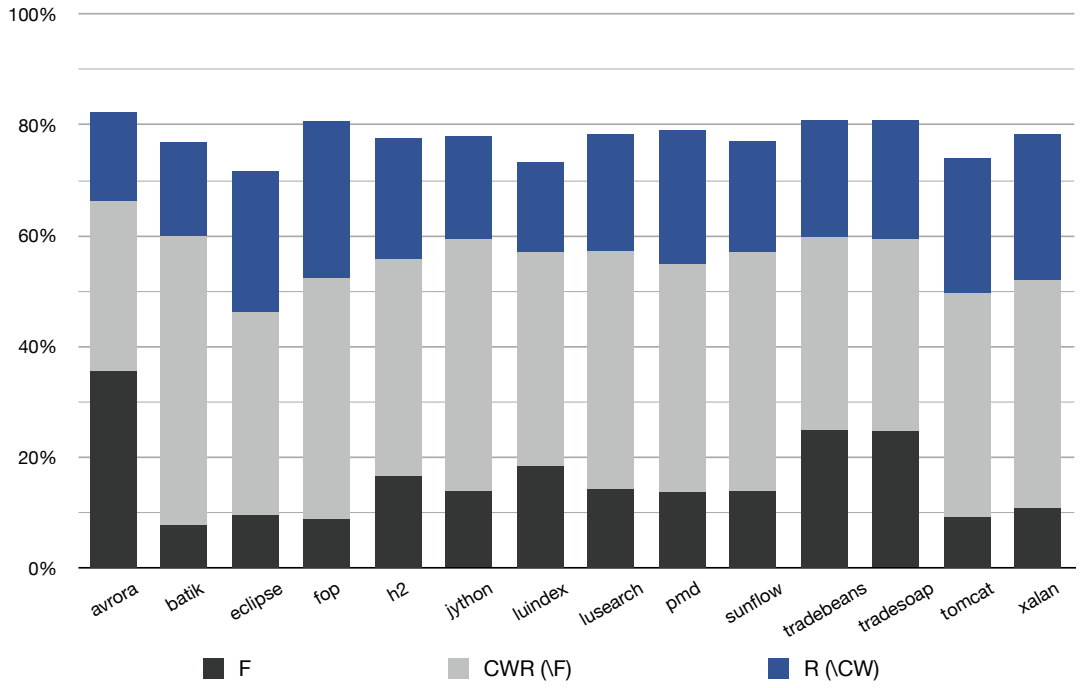


Figure 6.1: Read-settled (R) vs final (F) field declarations, including *undeclared final* – constructor-settled and write-settled but not final (CW\F) – for all field declarations. The table shows each category rounded to the nearest percentage point. This data is summarised in the stacked bar chart. Almost all field declarations that are constructor-settled and write-settled are also read-settled, and all read-settled field declarations are also final so the total height of each bar shows the total number of read-settled field declarations, while the lowest two segments combined show fields with final behaviour (constructor-settled and write-settled (CW)).

of declared final fields (though relatively few fields overall), and tradebeans and tradesoap, which are far larger than any of the benchmarks included in Unkel and Lam's study and have a relatively high proportion of final field declarations. Even allowing for differences between benchmarks, our results show a significantly higher proportion of read-settled field declarations, in particular we detect far fewer fields that are constructor-settled and write-settled but not read-settled (so few that they never reach 0.5% for these tables). This is not surprising because *rprof*'s whole-program behavioural analysis is not conservative, unlike the static analysis performed by Unkel and Lam: their static analysis guarantees behaviour for all possible inputs, whereas we observe the behaviour of a program for a particular set of inputs.

Consistent with the findings of Unkel and Lam, we find that most programs do not make good use of the final annotation to declare fields that are written once and do not change after the constructor (CW, or *final-like* behaviour). Section 6.3.2 examines constructor behaviour in more detail. Comparing results with and without system classes is consistent with their hypothesis that this is largely due to programmer style: we see that most applications use final annotations less often than system classes even though the proportion of fields with final-like behaviour is similar. Even allowing for 'lazy' programmers not using final where they could, the high proportion of read-settled fields compared with final fields suggests to us that the use of constructors for field initialisation is not supporting programming practices: there are many cases where fields were not initialised during the constructor even though they settled before they were read.

Reference and primitive field declarations

Unkel and Lam compared the behaviour of reference fields – field declarations that store object or array references, and primitive fields – that store primitive types like ints, and observed that reference fields are more likely to settle than primitive fields. Figures 6.2 and 6.3 show our analysis of these field declarations, showing reference fields and primitive fields respectively. Each figure uses the same format as Figure 6.1.

Discussion

These figures show that reference fields are more likely than primitive fields to be final (F), undeclared final (CW\F), and read-settled (R). This is consistent with Unkel and Lam's observations, though the difference is more modest in our re-

Program	Total	R (%)			¬R (%)			Total (%)		
		F	CW\F	¬CW	F	CW\F	¬CW	F	CW	R
avrora	880	47	31	12	0	0	9	47	78	90
batik	1,593	9	54	17	0	1	19	9	64	81
eclipse	3,324	15	43	20	0	0	22	15	58	78
fop	1,760	12	45	28	0	0	14	12	58	86
h2	990	21	41	20	0	0	18	21	62	82
jython	1,354	16	49	18	0	0	16	16	66	83
luindex	812	28	42	16	0	0	14	28	70	85
lusearch	598	20	48	18	0	1	14	20	69	86
pmd	1,048	18	45	22	0	1	15	18	63	85
sunflow	831	18	47	20	0	0	15	18	66	85
tradebeans	9,451	33	34	19	0	0	14	33	67	86
tradesoap	9,619	33	34	19	0	0	14	33	67	86
tomcat	3,393	12	43	26	0	0	18	12	55	82
xalan	1,095	14	46	24	0	1	16	14	60	83
Total	36,748	25	39	20	0	0	16	25	64	84

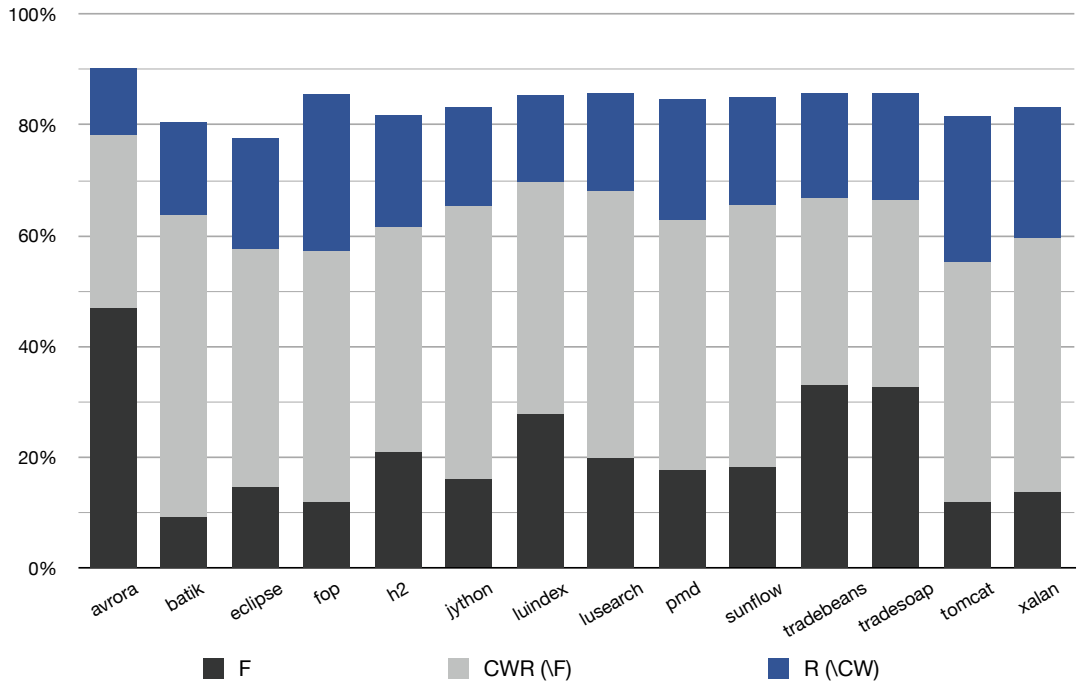


Figure 6.2: Read-settled (R) and final (F) for reference field declarations, including *undeclared final* – constructor-settled and write-settled but not final (CW\F). The table shows each category rounded to the nearest percentage point. This data is summarised in the stacked bar chart. Almost all field declarations that are constructor-settled and write-settled are also read-settled, and all read-settled field declarations are also final so the total height of each bar shows the total number of read-settled field declarations, while the lowest two segments combined show fields with final behaviour (constructor-settled and write-settled).

Program	Total	R (%)			¬R (%)			Total (%)		
		F	CW\F	¬CW	F	CW\F	¬CW	F	CW	R
avrora	882	23	30	20	0	0	26	23	54	74
batik	1,679	6	50	17	0	0	26	6	56	74
eclipse	3,455	5	30	31	0	0	34	5	35	66
fop	1,630	6	42	28	0	0	25	6	47	75
h2	1,024	12	38	23	0	0	26	12	50	73
jython	1,147	11	41	19	0	0	28	11	53	72
luindex	862	10	35	17	0	0	38	10	45	62
lusearch	705	10	38	24	0	0	28	10	48	72
pmd	914	9	36	27	0	0	27	9	46	73
sunflow	1000	11	40	20	0	0	29	11	50	70
tradebeans	5,953	12	36	25	0	0	26	12	49	73
tradesoap	5,992	12	36	25	0	0	27	12	48	73
tomcat	2,886	6	37	22	0	0	34	6	43	65
xalan	968	8	36	30	0	0	27	8	43	73
Total	29,037	10	37	25	0	0	29	10	47	71

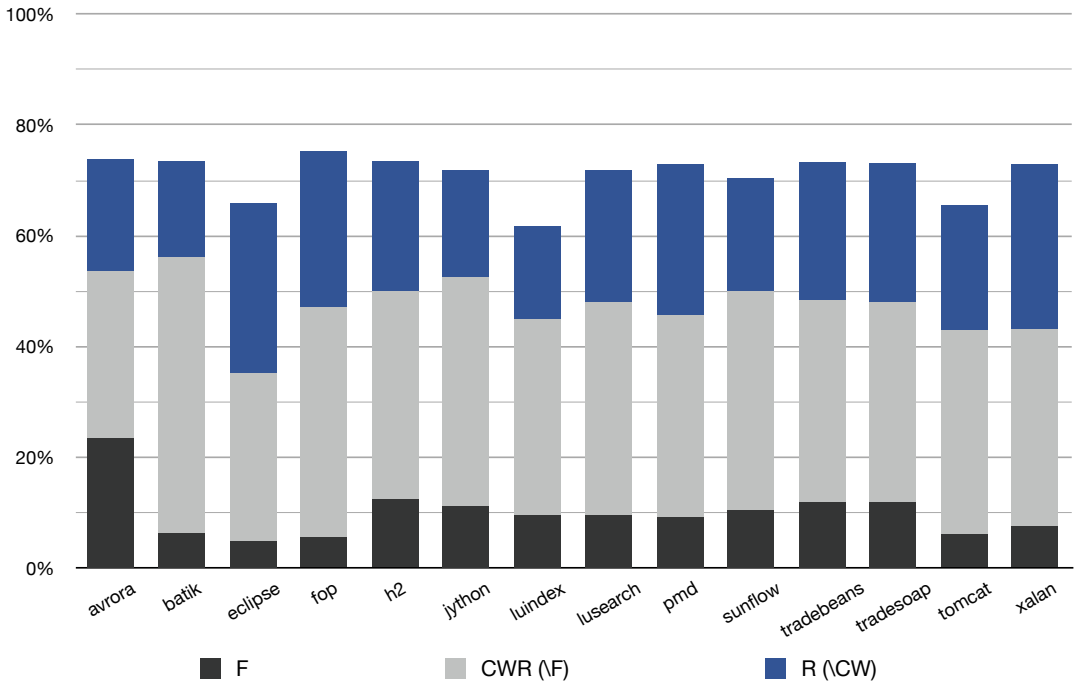


Figure 6.3: Read-settled (R) and final (F) for primitive field declarations, including *undeclared final* – constructor-settled and write-settled but not final (CW\F). The table shows each category rounded to the nearest percentage point. This data is summarised in the stacked bar chart. Almost all field declarations that are constructor-settled and write-settled are also read-settled, and all read-settled field declarations are also final so the total height of each bar shows the total number of read-settled field declarations, while the lowest two segments combined show fields with final behaviour (constructor-settled and write-settled).

sults. This is probably because our dynamic analysis detected more final and read-settled field declarations in general, but could also be due to variations in behaviour of the benchmark suites. The different behaviour of reference and primitive fields suggests that programmers use these fields for different purposes.

It is particularly noticeable that final fields are less common among primitive field declarations; the proportion of final primitive fields is also more consistent between benchmarks than reference fields. This does not have much effect on the overall variation in the number of read-settled fields which range from 62% to 75% for primitive fields, a range of 13% compared to the 10% range observed overall. Read-settled reference fields have a range of 12% across benchmarks: 78% to 90%, even though the number of final reference fields varies between 9% and 47%.

6.3.2 Constructor-settled and final field declarations

If programmers do not make good use of the final annotation perhaps it is simply because the annotation is too restrictive rather than a symptom of constructors themselves being at fault. Final field declarations can only be written once, and only from the constructor methods of the class that declares them. Section 3.2.1 defined constructor-settled field declarations – where any behaviour is allowed until the constructor returns, after that only reads are allowed. This section compares field declarations with *final-like* behaviour (constructor-settled and write-settled (CW)) and constructor-settled field declarations (C) in the DaCapo benchmarks to determine whether a constructed annotation – that allows any behaviour before the constructor returns but does not allow writes after the constructor – in place of final annotations would be more representative of actual program behaviour than final.

Figure 6.4 compares constructor-settled fields by their final behaviour. The format for this figure is similar to the figures in the previous section (e.g. Figure 6.1), but considers constructor-settled in place of read-settled.

Field declarations that are final are not necessarily read-settled (although we observed that they usually are), however final field declarations (F) are a subset of constructor-settled and write-settled field declarations (CW), which are in turn a subset of constructor-settled field declarations (C). The not-constructor-settled column and the constructor-settled and write-settled column are necessarily zero for the data presented in Figure 6.4, but we keep the same format for consistency.

Program	Total	C (%)			¬C (%)			Total (%)		
		F	CW\F	¬CW	F	CW\F	¬CW	F	CW	C
avrora	1,702	36	31	2	0	0	31	36	67	69
batik	3,272	8	53	4	0	0	35	8	60	65
eclipse	6,779	10	37	4	0	0	50	10	46	50
fop	3,390	9	44	5	0	0	42	9	53	58
h2	2,014	17	39	3	0	0	41	17	56	59
jython	2,501	14	46	4	0	0	36	14	60	64
luindex	1,674	18	39	4	0	0	39	18	57	61
lusearch	1,303	14	43	4	0	0	38	14	58	62
pmd	1,962	14	41	3	0	0	41	14	55	59
sunflow	1,831	14	43	6	0	0	37	14	57	63
tradebeans	15,404	25	35	3	0	0	37	25	60	63
tradesoap	15,611	25	35	3	0	0	37	25	60	63
tomcat	6,279	9	41	5	0	0	45	9	50	55
xalan	2,063	11	41	4	0	0	43	11	52	57
Total	65,785	18	38	4	0	0	40	18	57	60

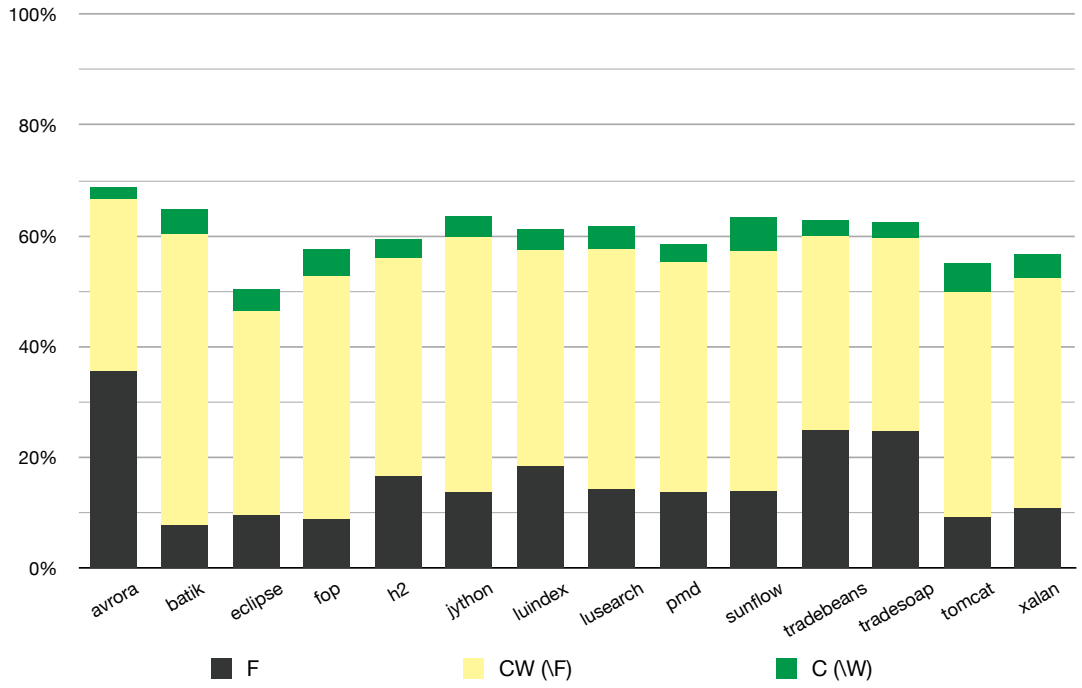


Figure 6.4: Constructor-settled (C) and final (F) for all field declarations. The table shows each category rounded to the nearest percentage point. This data is summarised in the stacked bar chart. Constructor-settled field declarations (C) subsume constructor-settled and write-settled field declarations (CW) and final field declarations (F) so the total height of each bar shows the total number of constructor-settled field declarations, while the first two segments show fields with final behaviour (constructor-settled and write-settled (CW)).

Discussion

This figure shows that only 3% of fields are constructor-settled but not write-settled (5% of constructor-settled fields). This indicates that in almost all cases fields that are initialised during the constructor are only written once. From this data we conclude that there is little value to adding a constructed annotation to Java as it would have very similar applicability to the existing final annotation. New languages might consider using a constructed annotation instead of final as it is slightly more applicable to the field declaration settling behaviour we observed.

6.3.3 Read-settled and constructor-settled field declarations

In Section 6.3.1 we showed that final field annotations (F) are poorly utilised while read-settled field declarations (R) outnumber field declarations that are both constructor-settled and write-settled (final-like, CW). Section 6.3.2 showed that constructor-settled field declarations slightly outnumber field declarations that are both constructor-settled and write-settled. In this section we compare constructor-settled field declarations with read-settled field declarations to determine whether the small number of fields that are constructor-settled but not write-settled are read-settled, or whether read-settled field declarations subsume constructor-settled too.

Figure 6.5 shows a stacked bar chart for constructor-settled and read-settled fields. The bottom segment of each bar shows read-settled field declarations that are not constructor-settled (R\C) (14-24% of all field declarations). These are fields that are never written after they have been read, but are written after the constructor returns. The centre segment of each stacked bar shows fields that are both read-settled and constructor-settled (CR) (50-68% of all field declarations). The top section shows field declarations that are constructor-settled but not read-settled (C\R) (1-2% of all field declarations). These fields are not written after the constructor so they must be read before they are written during the constructor.

Discussion

Figure 6.5 shows that almost all fields that are constructor-settled are also read-settled, but there are a small number of fields that are read during the constructor before they are written. Unkel and Lam refer to a category of fields they call *semi-stationary* — fields that would be read-settled if reads that occur before the objects are 'lost' by their analysis are disregarded. In other words, they allow

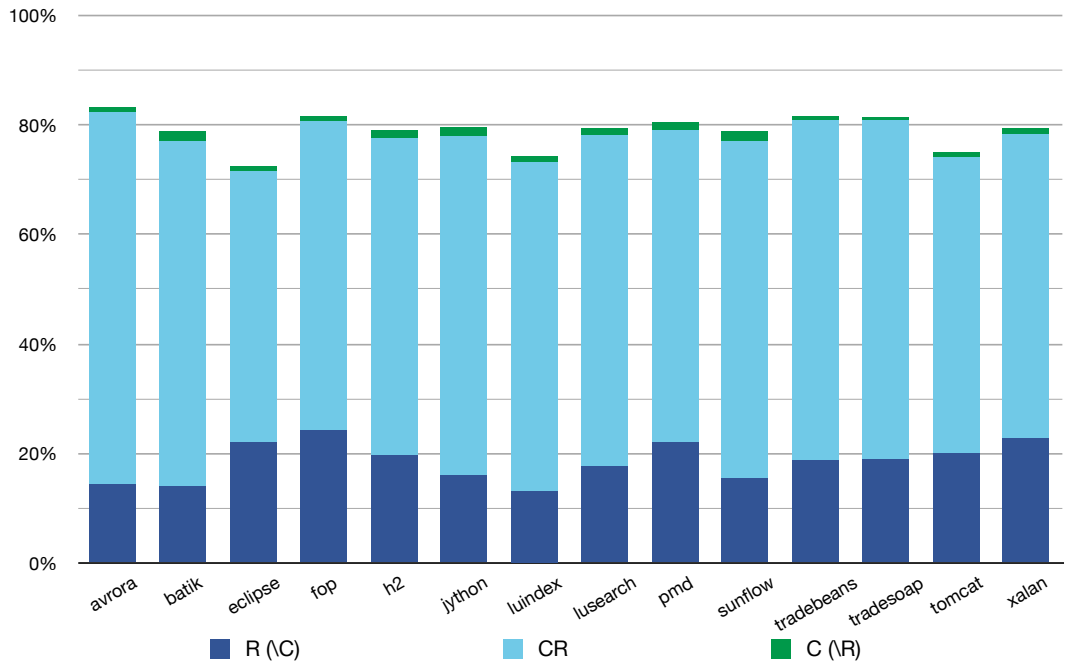


Figure 6.5: Read-settled (R) and constructor-settled (C) for all field declarations. The bottom two segments of each bar show read-settled field declarations (R) while the top two segments show constructor-settled field declarations (C).

reads during and shortly after the constructor, but any subsequent reads must not precede writes. Their definition of semi-stationary corresponds roughly to the total size of the bars in Figure 6.5: the union of read-settled and constructor-settled fields. We can confirm their observation that there are very few fields in this category which are not in the read-settled category — they found up to 7% but we never found more than 2%, probably because their analysis disregards some reads that ours does not.

Figure 6.5 also shows that there is a large percentage of field declarations (14-24%) that are read-settled but not constructor-settled. It also shows that the number of read-settled field declarations is very consistent between benchmarks: the number of final field declarations (F) varies between 9% and 36%, constructor-settled field declarations (C) vary between 50% and 69%, but the number of read-settled field declarations (R) remains between 72% and 82% across a range of programs from different domains, authors, and of significantly different sizes. Our runtime observations together with Unkel and Lam's static analysis suggests that an annotation that enforces read-settled behaviour for field declarations – stationary perhaps – would be substantially more applicable than final and we strongly

encourage future language designers to consider this result.

Unfortunately, even if Java language designers added a stationary annotation we have no evidence to suggest that programmers would utilise it any more than they use final. Read-settled field declarations may be written after the constructor returns and that would certainly allow, for example, cyclical references to be established, but there are many fields that could be final but are not, so perhaps annotating each field declaration is too much effort. Perhaps class or object annotations for documenting classes and objects that settle would be more popular? Section 6.4 considers the initialisation behaviour of objects and classes rather than field declarations to determine whether they show sufficient settling behaviour that language designers could justify whole-class or object annotations.

6.4 Object settling

Section 6.3 confirmed the observations of Unkel and Lam, concluding that most Java field declarations in the programs we observed are read-settled, considerably more than are constructor-settled. This section focuses on whole-object settling behaviour, aggregating data for over 300M objects, to determine whether a significant proportion of Java objects show settling behaviour.

This section does not aggregate objects by class but rather considers each object as a distinct entity; objects of the same class can be classified differently if they exhibit different behaviour. Field declaration aggregations – such as those in the previous section – combine information about different objects (projecting object-fields onto the set of all field declarations). Similarly, object aggregations combine information about different field declarations (projecting object-fields onto the set of all objects). The precise details of these aggregations are described in the object settling behaviour definitions of Chapter 3.

This section is structured as follows:

- We begin with a survey of the settling behaviours of all observed objects, considering specifically read-settled and constructor-settled objects as those behaviours were the most common among fields. These results will show whether or not settling behaviours can be applied to whole objects.

- Section 6.4.2 considers specifically objects that use equality methods (equals and hashCode) and objects that enter collections, categories of objects that we have already observed in Chapter 4. These observations will allow us

to determine whether, as we speculated, there are many objects in these categories that are read-settled, equals-settled, or collection-settled.

- Constructor-settled objects have a single unambiguous program event that marks the end of their initialisation. Read-settled objects, on the other hand, settle one object-field at a time; there is no clear point separating initialisation and use of the whole object. Section 6.4.3 considers object-read-settled objects — objects that are settled as soon as one field is read (writes of all fields must precede the first read of any field). The ratio of read-settled objects to object-read-settled objects will show whether a single initialisation boundary exists for read-settled objects.

- This section concludes by comparing class and object settling behaviour in Section 6.4.4. We aim to determine whether object initialisation behaviour is consistent between classes, or whether there is substantial variation in the behaviour of objects from a single class.

6.4.1 Read-settled and constructor-settled objects

This section examines the settling behaviour of objects in the DaCapo benchmarks by comparing the read-settled objects (R) with constructor-settled objects (C). Figure 6.6 presents the percentages of objects in each category as a table and as a chart in a similar format to the figures in the previous section.

Discussion

Figure 6.6 shows that there is significantly more variance in the observed behaviour of different benchmarks than we encountered when analysing field declaration settling. Like field declarations, constructor-settled objects account for about half of all objects, however the range is much greater: 2-78% of objects are constructor-settled whereas constructor-settled field declarations ranged between 50-69% of all fields across the 14 benchmark applications. The total ratio of constructor-settled objects to all objects is also less than that of constructor-settled fields: 49% of objects compared to 60% of fields. The number of objects in each benchmark varies considerably more than the number of fields so this could skew the object aggregations, but this does not seem to be the case as the mean percentage of constructor-settled objects is 53% vs 60% for fields. We conclude that objects are less likely to be constructor-settled than fields.

		R (%)		¬R (%)		Total (%)	
Benchmark	**Objects**	**C**	**¬C**	**C**	**¬C**	**C**	**R**
avrora	1,596,220	55	1	0	44	55	56
batik	761,841	57	18	1	24	58	75
eclipse	24,159,223	50	18	2	29	53	69
fop	2,042,656	58	15	0	27	58	73
h2	66,183,338	59	2	0	38	59	62
jython	31,203,059	78	10	0	12	78	88
luindex	218,890	73	4	0	22	73	78
lusearch	6,746,880	51	20	0	29	51	71
pmd	5,169,866	48	30	0	22	48	78
sunflow	60,222,278	2	67	10	21	12	69
tradebeans	46,587,901	50	12	0	38	50	62
tradesoap	48,758,437	59	16	0	25	59	74
tomcat	4,362,173	72	10	1	18	73	82
xalan	6,394,758	27	59	0	14	27	86
Total	304,407,520	47	23	2	28	49	70

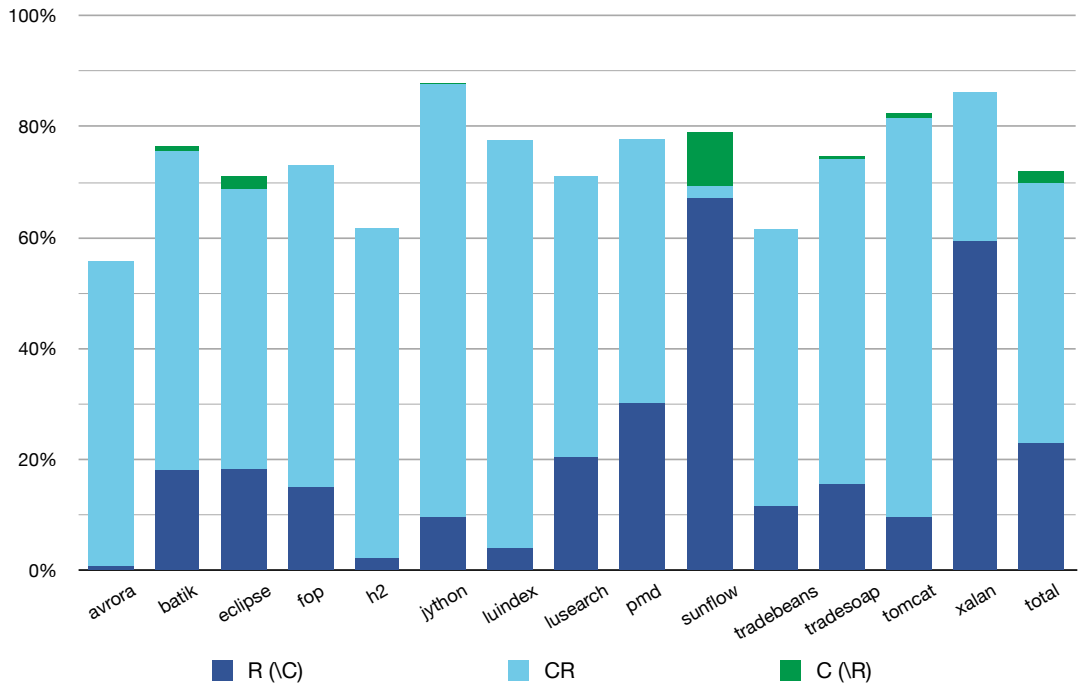


Figure 6.6: Read-settled (R) and constructor-settled (C) objects. The first two columns of the table show the names of benchmarks and the number of objects observed in each benchmark. The four central columns show primary divisions of objects into read-settled (R) and not-read-settled (¬R), then secondary divisions of those objects between constructor-settled (C) and not-constructor-settled (¬C). The final two columns show summary data for constructor-settled (C) and read-settled (R) objects. The chart at the bottom of the figure shows the same data. The bottom two segments of each bar show read-settled objects (R) while the top two segments show constructor-settled objects (C).

Our object settling behaviour analysis also shows more variance in the number of read-settled objects than read-settled field declarations, though the difference is less than between constructor-settled objects and constructor-settled fields. Read-settled objects account for 56-88% of all objects, which compares more favourably with the 72-82% of read-settled fields observed in the previous section than the constructor-settled ratios. Across all benchmarks 70% of objects were read-settled, again lower than the 78% of fields. The mean number of read-settled objects also shows less divergence from field declarations than constructor-settled objects: 73% for objects vs 78% for field declarations.

These results show that both constructor-settled and read-settled behaviours occur for objects, although constructor-settled behaviour varies significantly between benchmark applications.

Correlation between read-settled and constructor-settled objects, and benchmark size

We noted that in the case of both constructor-settled and read-settled object ratios, the ratio of all objects in all benchmarks was lower than the mean ratio of objects

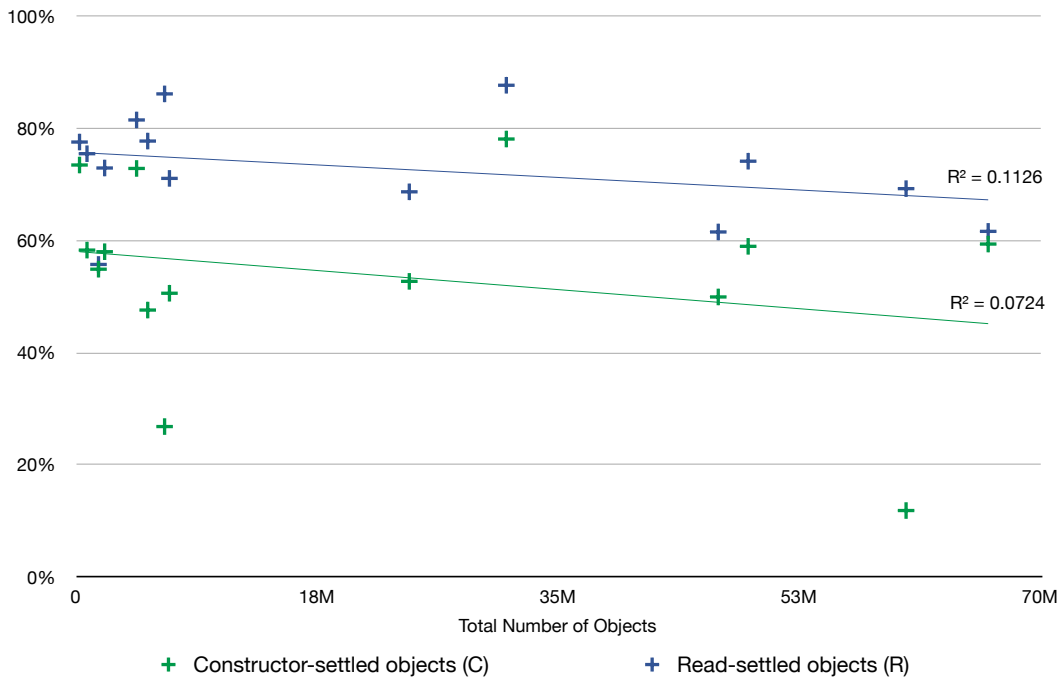


Figure 6.7: Ratios of read-settled and constructor-settled objects to all objects vs benchmark size (total number of objects)

across benchmarks. This could indicate that larger programs are less likely to have constructor-settled/read-settled objects. Figure 6.7 shows a scatter plot of ratios versus total number of objects for each benchmark with generated lines of best fit. This figure indicates that for the 14 benchmarks we observed the larger benchmarks do have slightly fewer constructed and stationary objects. Nevertheless, low correlation coefficients (0.07 for constructor-settled, 0.1 for read-settled) suggest that this is probably an anomaly of the particular benchmarks we observed.

Read-settled and constructor-settled objects excluding system objects

Figure 6.8 presents data for all objects excluding system objects (java/javax) in the same format as Figure 6.6.

Discussion

Figure 6.8 shows that more than half of all objects encountered in these benchmarks were system (java/javax) objects. Excluding these objects results in a modest increase in both constructor-settled (C) and read-settled (R) object ratios. In particular, we see an overall drop of 15% in the proportion of objects which are read-settled but not constructor-settled. This causes a rise of 21% in the number of objects which are both read-settled and constructor-settled.

System objects are implemented by experienced library designers and are available to all Java programmers. The significant increase in proportion of objects that are constructor-settled when we exclude system objects suggests that the programmers implementing system objects used more complex initialisation strategies for system library objects than the programmers who implemented the DaCapo benchmarks. This is particularly surprising because system objects include common logically immutable objects like strings and integers, and Block, author of many classes in the system libraries, tells Java programmers to *"prefer immutability"* [16].

The differences between system and program-specific objects are more pronounced than the differences we observed between system and program-specific fields. This could be for a number of reasons, but it seems likely that it is simply because system objects comprise a far larger proportion of total objects than system fields: 56% vs 31%, probably because system classes are less numerous but more frequently instantiated than program-specific objects.

		R (%)		¬R (%)		Total (%)	
Benchmark	**Objects**	**C**	**¬C**	**C**	**¬C**	**C**	**S**
avrora	1M	64	1	0	35	64	65
batik	560K	64	9	1	26	65	73
eclipse	11M	74	3	0	23	75	77
fop	1M	59	8	0	33	59	67
h2	39M	77	1	0	22	77	78
jython	9M	79	6	0	15	79	85
luindex	115K	72	8	0	20	72	80
lusearch	2M	69	1	0	31	69	69
pmd	3M	60	10	0	30	60	70
sunflow	28K	72	7	0	20	73	79
tradebeans	26M	72	3	0	25	72	75
tradesoap	30M	58	14	1	28	58	72
tomcat	3M	72	9	1	18	73	81
xalan	5M	25	72	0	4	25	96
Total	132M	68	8	0	23	69	77

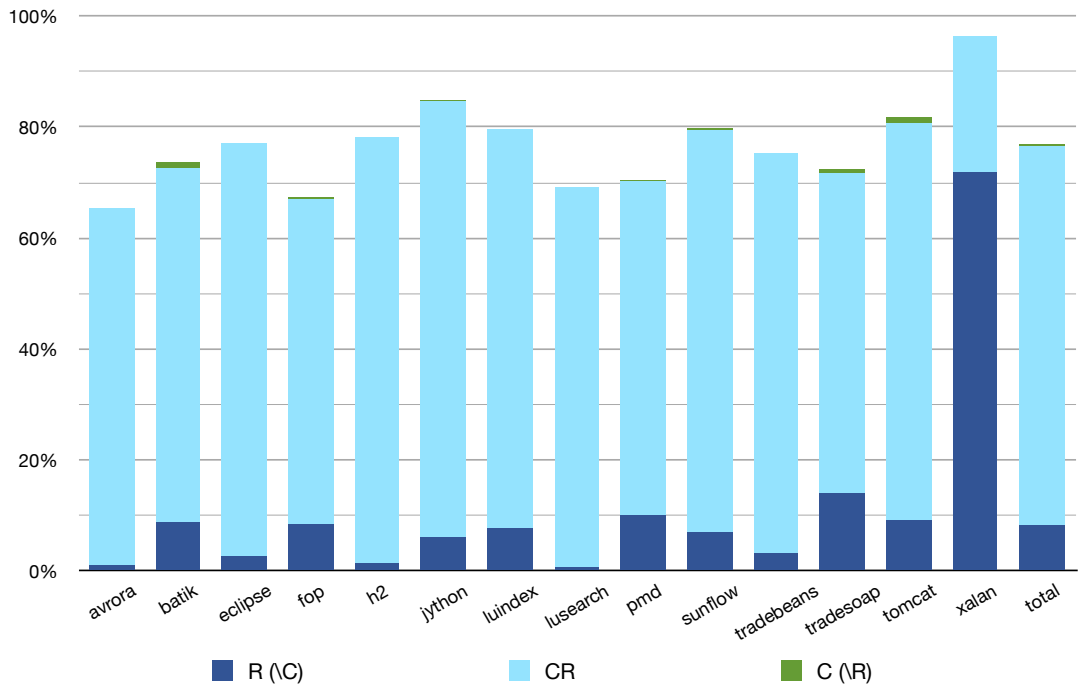


Figure 6.8: Read-settled (R) and constructor-settled (C) objects excluding system objects (java/javax). The data presented in this figure matches the format of Figure 6.6 but excludes system objects.

6.4.2 Equality, collections, and objects

This section inspects the subcategories of objects that had equals or hashCode called and also objects that entered collections. These subcategories of objects are interesting because we have an additional event in each object's lifecycle to determine whether the objects have settled. These subcategories are also interesting insomuch as they behave differently to the general population of objects. The results presented in Chapter 4 suggest that they may: we observed that almost all objects that enter collections had some form of settled equality.

Objects used for equality operations

Figure 6.9 shows objects used in equality operations: objects which have their equals or hashCode methods invoked and objects which are passed as arguments to the equals method of another object. As in previous figures, data is shown both in tabular form and as a chart. The table's first column shows the total number of objects from each benchmark that were used in equality operations. The subsequent four columns show the distribution of these objects between read-settled (R) and equals-settled (E) object categories — where equals-settled (E) requires that objects do not write to fields after an equality operation occurs involving that object. The last three columns summarise the number of these objects that are equals-settled (E) and read-settled (R), as well as the number that are constructor-settled (C). The chart at the bottom of the figure shows the data in the body of the table in a similar format to previous charts: the bottom two segments show read-settled objects (R) while the top two show equals-settled (E) objects. The centre segment shows the intersection (ER).

Discussion

Figure 6.9 shows that objects used in equality operations are far more likely than the general object population to be both read-settled and constructor-settled (i.e. Figure 6.6), with most benchmarks near 100%. This is consistent with the results obtained using *#prof*, presented in Chapter 4. Most objects in this subpopulation that are read-settled are also equals-settled (ER), and there are fewer objects that are equals-settled but not read-settled (E\R) than objects that are read-settled but not equals-settled (R\E). In only one case (batik) do exclusively equals-settled objects exceed the number of exclusively read-settled objects.

Objects that use equality and are equals-settled outnumber those that are use equality and are constructor-settled; in fact the number of objects identified as

		R (%)		¬R (%)		Total (%)		
Benchmark	**Objects**	**E**	**¬E**	**E**	**¬E**	**E**	**C**	**R**
avrora	877	99	1	0	0	99	99	100
batik	15K	83	3	10	5	92	92	85
eclipse	39K	93	1	0	5	93	93	95
fop	43K	79	17	5	0	83	79	95
h2	4M	97	3	0	0	97	97	100
jython	48K	97	3	0	0	97	96	100
luindex	337	93	4	2	1	95	93	97
lusearch	295	96	3	1	0	96	95	99
pmd	24K	100	0	0	0	100	100	100
sunflow	348	94	3	1	1	95	93	97
tradebeans	590K	98	2	0	0	98	89	100
tradesoap	948K	98	2	0	0	98	96	100
tomcat	20K	98	2	0	0	98	45	100
xalan	796	98	1	0	0	99	97	100
Total	6M	97	3	0	0	97	96	100

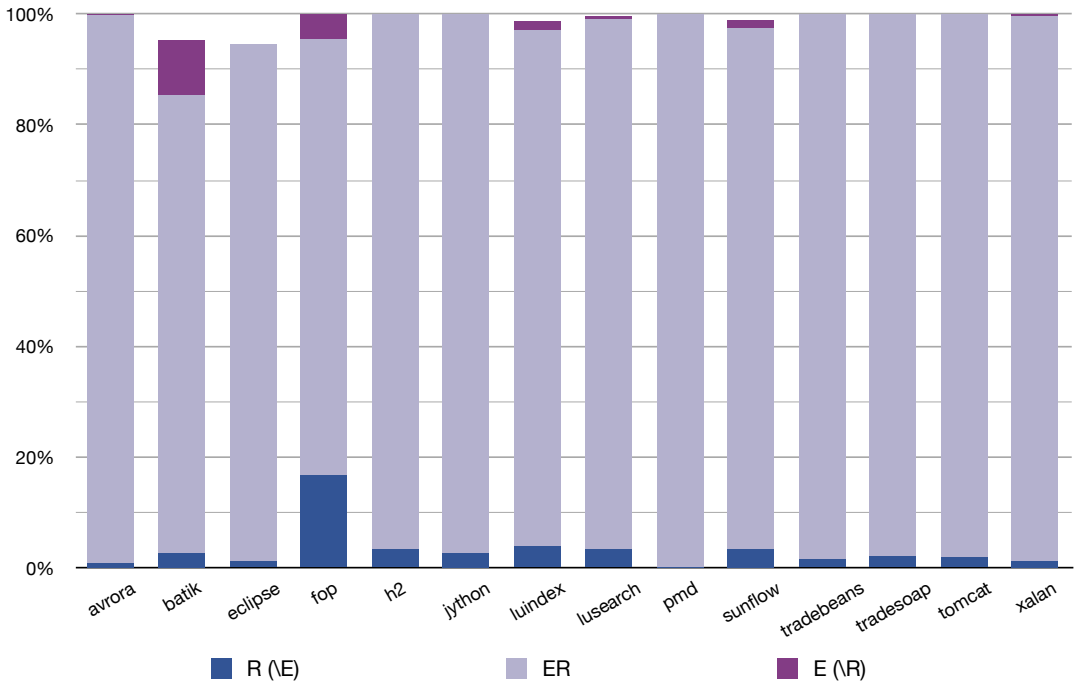


Figure 6.9: Ratios of equals-settled (E) and read-settled (R) objects for the subpopulation of objects used for equality operations. The first two columns of the table show the names of benchmarks and the number of objects used for equality operations observed in each benchmark. The four central columns show primary divisions of objects into read-settled (R) and not-read-settled (¬R), then secondary divisions of those objects between equals-settled (E) and not-equals-settled (¬E). The final three columns show summary data for equals-settled (E), constructor-settled (C), and read-settled (R) objects. The chart at the bottom of the figure shows data for equals-settled and read-settled objects. The bottom two segments of each bar show read-settled objects (R) while the top two segments show equals-settled objects (E).

equals-settled was always between the number of objects that were read-settled and constructor-settled respectively (in batik's case the bounds are inverted). We conclude that while equals-settled is useful for comparing to other settling behaviours, equals-settled is not useful in general as it identifies fewer settled objects than read-settled and constructor-settled, but is only applicable to 2% of the total set of objects observed.

Objects that enter collections

Figure 6.10 presents object categorisations for objects that enter Java collections — objects in this category were observed as arguments to methods belonging to one of Java's collection classes. These objects are interesting because we can observe collection-settled behaviour — objects whose fields are not modified after they enter a collection for the first time. The table at the top of Figure 6.10 summarises objects that enter Java collections for each of collection-settled (K), constructor-settled (C), and read-settled (R) objects, and presents the distribution of objects between collection-settled and read-settled categories in the centre. The chart at the bottom of the figure shows the distribution as a stacked bar chart. The bottom two segments of each bar show read-settled objects (R) while the top two segments show collection-settled objects (the centre segment shows the intersection, KR).

Discussion

These results show that, like objects used for equality operations, objects that enter collections are more likely than the general population to be constructor-settled or read-settled, though the increase in likelihood is modest compared with objects used in equality operations, even excluding xalan, a significant outlier. This is unexpected: we observed in the results of Chapter 4 that objects in collections are actually less likely to be constructor-settled than objects in the general population. However, *#prof* had severe limitations when observing the general population of objects; it was essentially limited to user objects. The results presented here, obtained with *rprof*, track almost all objects created and so give a better overall measure of object behaviour.

Unlike the results for equals-settled objects, these results show several benchmarks where the number of collection-settled objects exceeds the number of read-settled objects in this subpopulation. In addition, while equals-settled objects are always bounded by read-settled and constructor-settled objects, collection-

		R (%)		¬R (%)		Total (%)		
Benchmark	**Objects**	**K**	**¬K**	**K**	**¬K**	**K**	**C**	**R**
avrora	9K	78	4	1	18	79	77	82
batik	149K	87	1	0	12	87	50	87
eclipse	1M	58	21	10	11	68	48	79
fop	315K	66	14	14	6	80	63	80
h2	2M	66	26	0	7	67	66	93
jython	1M	70	22	5	3	75	72	92
luindex	7K	91	2	0	7	92	90	93
lusearch	514K	100	0	0	0	100	53	100
pmd	871K	63	28	3	6	65	61	91
sunflow	8K	75	12	2	11	77	74	87
tradebeans	6M	76	2	5	17	81	75	78
tradesoap	9M	75	14	2	8	78	74	89
tomcat	248K	81	5	10	5	90	82	86
xalan	518K	37	1	0	62	37	37	38
Total	23M	73	12	4	11	76	70	85

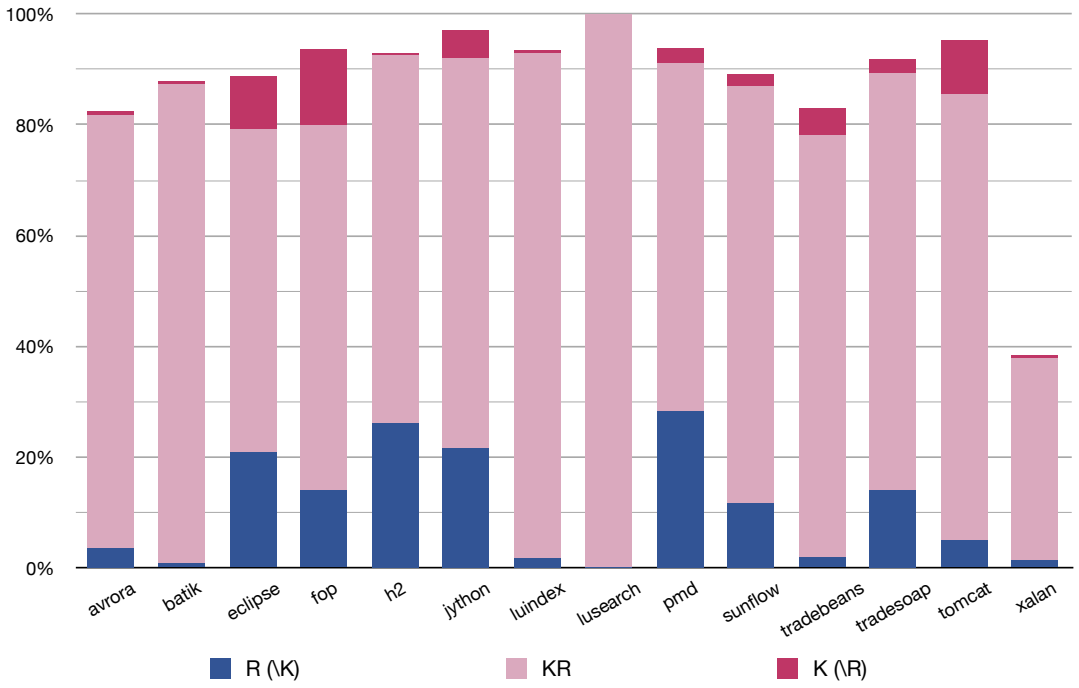


Figure 6.10: Ratios of collection-settled (K) and read-settled (R) objects for the subpopulation of objects that enter collections. The first two columns of the table show the names of benchmarks and the number of objects that enter collections observed in each benchmark. The four central columns show primary divisions of objects into read-settled (R) and not-read-settled (¬R), then secondary divisions of those objects between collection-settled (K) and not-collection-settled (¬K). The final three columns show summary data for collection-settled (K), constructor-settled (C), and read-settled (R) objects. The chart at the bottom of the figure shows the data for collection-settled and read-settled objects. The bottom two segments of each bar show read-settled objects (R) while the top two segments show collection-settled objects (K).

settled objects outnumber the settled objects identified by both general metrics for two benchmarks, though overall it identifies a smaller fraction of objects than equals-settled. The differences, while never dramatic, suggest that programmers use several different initialisation patterns for objects which enter collections. It would be interesting to combine this type of runtime analysis with a static analysis which identifies source-code patterns for objects that enter collections.

Collection-settled is not very useful as a general purpose classifier because, like equals-settled, it is not applicable to most objects (8% of all objects). Even so, it is useful for comparisons and as we observed in Chapter 4, objects in collections do behave quite differently to general objects in some cases.

6.4.3 Read-settled (R) vs object-read-settled (oR)

Section 6.4.1 began by comparing read-settled (R) and constructor-settled (C) objects. Read-settled objects do not necessarily have a single program point where they become initialised as each object-field can become initialised independently. This section considers object-read-settled objects (oR) — objects that are read-settled and have the added condition that all field writes to *every* field occur before the first read of *any* field. Figure 6.11 shows the same data as Figure 6.6, but overlays object-read-settled objects onto the segments that contain read-settled objects. From the ratio of read-settled objects to object-read-settled objects we can conclude whether read-settled objects have single initialisation boundary.

Discussion

Figure 6.11 shows that most objects that are read-settled are also object-read-settled. This number of object-read-settled objects actually exceeds the number of objects that are constructor-settled. This suggests that objects do have single points where they become initialised. Of 300M objects observed, approximately 150K constructor-settled objects were read-settled but not object-read-settled. 7M objects were constructor-settled, but not read-settled.

The data shows two significant outliers: sunflow has relatively few constructor-settled objects, but a similar proportion of read-settled objects to other benchmarks. In addition, of those objects that are constructor-settled, very few are also read-settled. 57% of xalan's objects are read-settled but not object-read-settled, in contrast to other benchmarks where objects that are read-settled but not object-read-settled accounted for 0-8% of all objects.

In spite of the outliers, the large proportion of read-settled objects that are

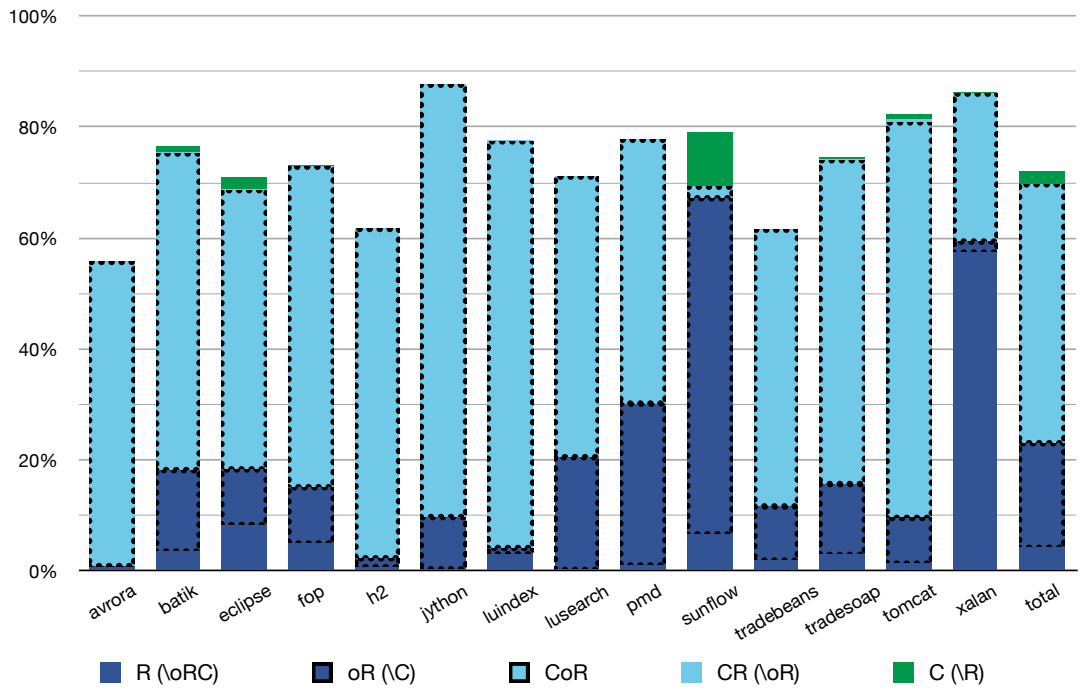


Figure 6.11: Read-settled (R), object-read-settled (oR) and constructor-settled (C) objects. This chart presents exactly the same data as the chart in Figure 6.6 but overlays object-read-settled objects (oR). These are indicated by the black dotted outline in segments that contain read-settled objects.

also object-read-settled objects suggests that programming languages could consider adding *whole-object* initialisation mechanisms. In the following section we will consider whether whole-object initialisation can be extended to classes, or whether it is predominantly an object property.

6.4.4 Object settling behaviour polymorphism

In this section we test the hypothesis that object settling behaviour is a result of the high number of settled field declarations. If this is the case then we would expect to see an inverse relationship between the number of fields in an object and the likelihood that the object is settled.

Figure 6.12 shows a stacked bar chart where the y-axis shows the proportion of objects with a given behaviour, while the x-axis shows the number of fields in each bar. The proportion of objects that have exclusively read-settled object-fields is shown at the bottom of each bar and the proportion of objects that have no read-settled object-fields is shown at the top (in black). The bottom chart in the figure shows relationship between object frequency and number of object-fields.

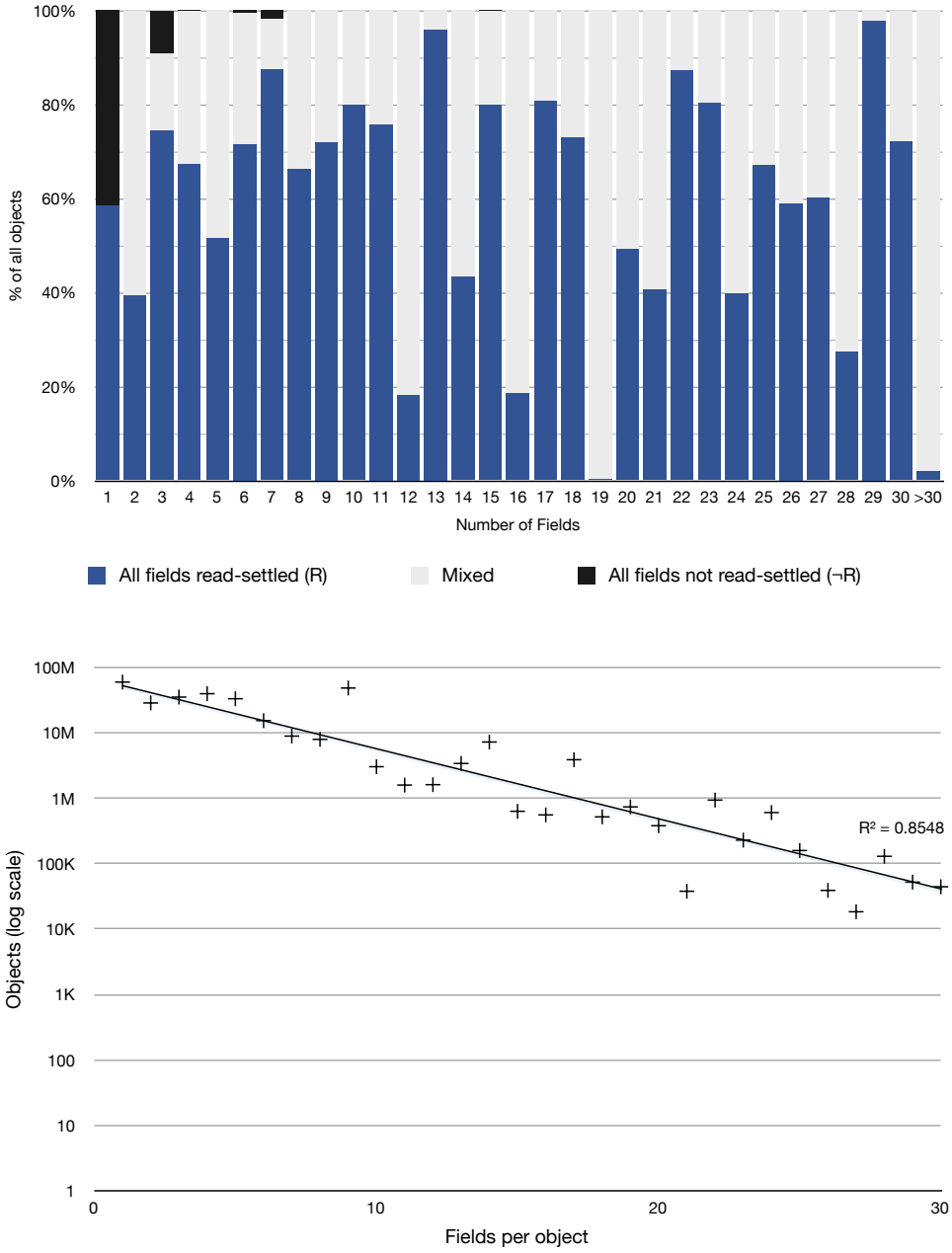


Figure 6.12: Behaviour consistency of whole objects vs number of fields. The top chart splits objects from all benchmarks into categories based on the number of object-fields and shows each category as a bar. Each bar shows 100% of the objects in that category split into three parts: the bottom segment shows objects that have only read-settled object-fields. The central section shows objects with a mix of read-settled and not-read-settled object-fields. The top segment (black) shows objects with exclusively not-read-settled object-fields. The bottom chart shows the number of objects in each bar on a log scale, showing an exponential relationship between the number of objects with a particular number of object-fields and the number of object-fields.

In both cases we have trimmed the graph at 30 fields per object.

Figure 6.12 shows that the number of objects with a particular number of fields decreases with an exponential trend, but there is no significant relationship between the number of fields in an object and the likelihood that the object is read-settled.

Figure 6.13 shows a similar stacked bar chart where the y-axis shows the proportion of classes with a given behaviour, while the x-axis shows the number of fields in each bar. The proportion of classes that have exclusively read-settled objects are shown at the bottom of each bar and the proportion of classes that have no read-settled objects are shown at the top (in black). The bottom chart in the figure shows relationship between class frequency and number of fields. In both cases we have trimmed the graph at 30 fields per object.

From Figure 6.13 we can conclude that the number of classes with a particular number of fields also shows an exponential decay. Figure 6.14 shows the data from the top chart of Figure 6.13 plotted as a scatter plot with lines of best fit and coefficient of determination, showing logarithmic decay in the ratio of both fully read-settled and fully not-read-settled classes as the number of fields in a class increases. This suggests that programmers do not decide to make entire classes that are read-settled: instead they decide for an individual field. The more fields a class has, the less likely it is that the class will show consistent read-settled behaviour.

In conclusion, classes would not benefit from a class-level annotation to document classes with settling behaviour because their settling behaviour is primarily derived from their fields. Objects, however, have independent settling behaviour and consequently programmers may benefit from programming language support for describing immutable objects independent of class (for example, a parametric immutability parameter).

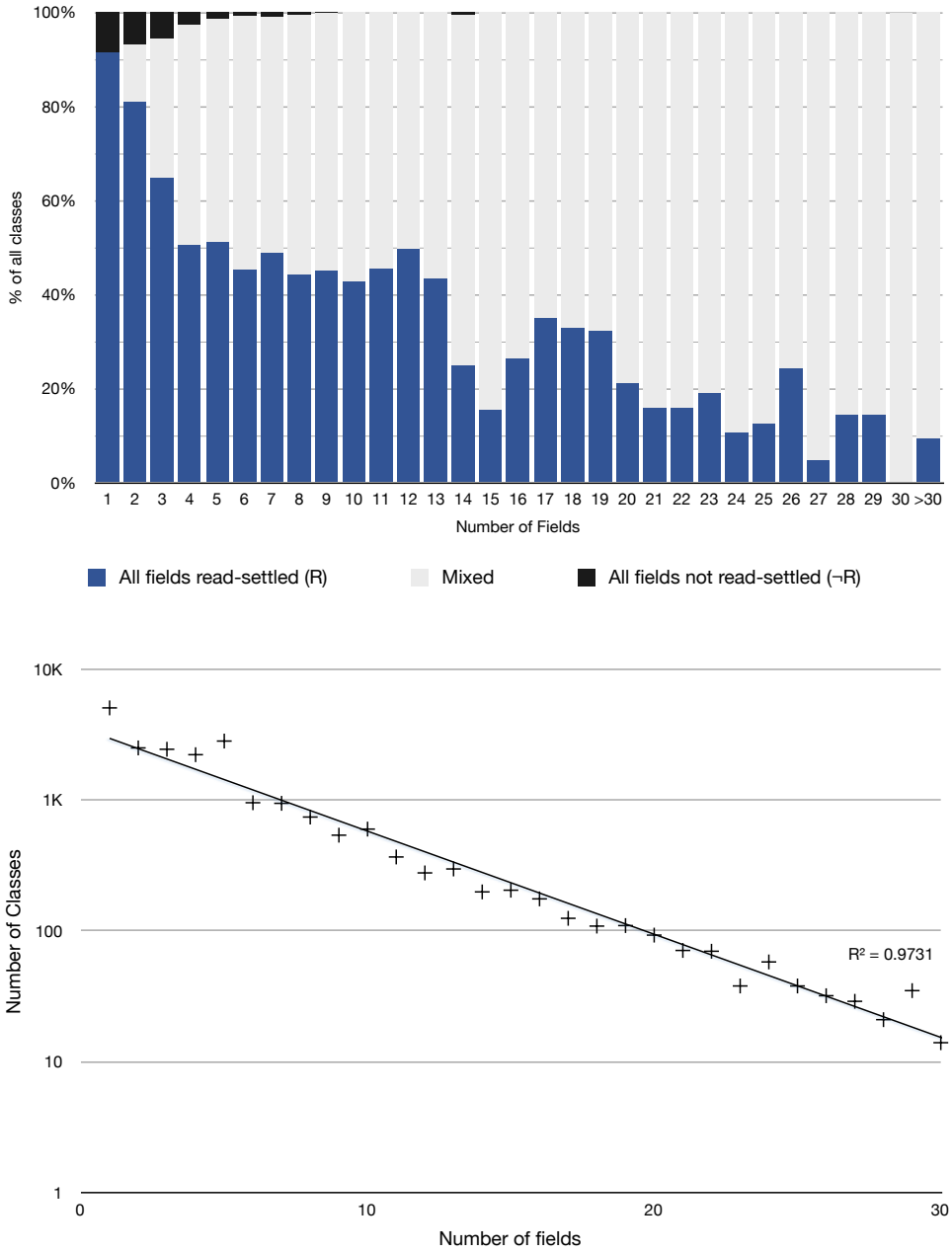


Figure 6.13: Behaviour consistency of classes vs number of fields. The top chart splits classes from all benchmarks into categories based on the number of fields in that class and shows each category as a bar. Each bar shows 100% of the objects in that category split into three parts: the bottom segment shows classes that have only read-settled fields. The central section shows classes with a mix of read-settled and not-read-settled fields. The top segment (black) shows classes with exclusively not-read-settled fields. The bottom chart shows the number of classes in each bar on a log scale, showing an exponential relationship between the number of classes a particular number of field declarations, and the number of field declarations. We do not show information for classes with more than 30 fields — this point was chosen because this is where the number of classes with a particular number of fields drops below 10.

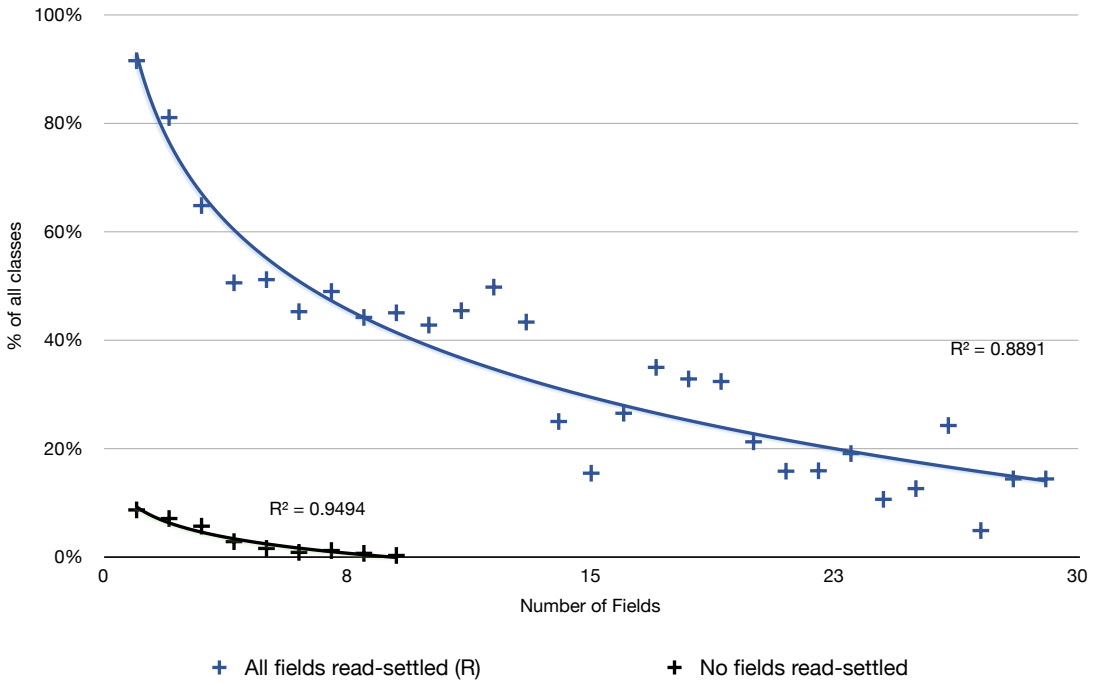


Figure 6.14: Behaviour consistency of classes vs number of fields as a scatter plot with logarithmic trend lines.

6.5 Threats to validity

The results presented in this chapter, like those presented in Chapter 4, are only valid if the assumptions we used to obtain them are correct. This section discusses the assumptions we made that could impact our results if they are incorrect.

6.5.1 Internal validity

One major concern during the development of *rprof* was the difficulty in validating its results. Unlike *#prof*, which only measured the behaviour of user-provided code, *rprof* observes all code running within the JVM including system libraries. Consequently, even the smallest programs are subject to some perturbation from JVM behaviour, so it was difficult to run small tests to ensure consistent behaviour (like we did for *#prof*). To mitigate the risk of incorrect behaviour, we invested considerable effort in testing each component in isolation, developing small end-to-end tests using package filters to examine a subset of the results, and identifying and measuring aggregate behaviours that we could predict with certainty (such as ensuring that there are no fields identified that are both final and not constructor-settled). This was also a significant motivation for beginning this chapter by comparing *rprof*'s results with those of Unkel and Lam [109].

Another significant concern for us was the possibility of the profiler affecting program behaviour. This was partially mitigated for *#prof*, which did not track JVM behaviour, but any use of system libraries by *rprof* would be very difficult to distinguish from program behaviour. Consequently, we minimised the amount of profiler code running within the profiled JVM, calling native code as soon as possible and moving all substantial execution into separate JVMs.

No amount of testing can ensure correct behaviour but we are satisfied that *rprof*'s results are sufficiently accurate to draw the conclusions presented in this chapter. We would welcome any attempts by other research groups to replicate our results using their own techniques.

6.5.2 External validity

The results presented in Chapter 4 were obtained by profiling a selection of programs from the Qualitas Corpus [88]. For the results presented in this chapter we decided instead to profile the DaCapo benchmark suite, partially to make it easier to run large programs without human interaction, but also to improve the reproducibility of our results. However, the DaCapo benchmark suite consists

entirely of non-GUI workloads so it is possible that the results presented in this chapter are not generalisable to GUI Java programs, for example.

We chose the DaCapo benchmark suite as a compromise between our concerns about the lack of reproducibility of results obtained by manually interacting with programs, as we did in Chapter 4, and concerns about the generalisability of the results obtained from benchmark suites such as SPEC JVM98 (used by Unkel and Lam [109] and Pechtchanski and Sarkar [82]). As discussed in Section 2.4.1, DaCapo consists of relatively large, general purpose applications, without a particular bias other than for programs with large memory loads, and the lack of GUI tools [15].

6.5.3 Content validity

Finally, it is possible that there are settling events that we did not identify that would produce better results than read-settled. Possible examples of alternative settling events that we did not measure include the assignment of an object to a field or array, or return from the method that constructed the object. These events were out of scope for us to measure during this project but could be measured using the techniques that we developed for *rprof*. It seems unlikely that these events would identify significantly more settled objects/fields than read-settled due to the relatively high number of read-settled fields and objects, and the consistency that we observed between benchmarks.

Chapter 7

Conclusion

This chapter reviews the contributions of this thesis and considers future work.

7.1 Contributions

This thesis has made the following contributions:

7.1.1 Settling classifications

Chapter 3 presented definitions for describing the initialisation behaviour of program entities. We defined *settling* events – observable events in an object's lifecycle where entities may be initialised – and defined five settling events and classifications based on these events for object-fields, objects, field declarations, and classes. We also described how settling behaviour patterns could be observed for equality.

7.1.2 Object equality initialisation

Chapter 4 presented the runtime analysis of 30 open source applications from the Qualitas Corpus. This required the development of a novel profiler that could actively track object initialisation and its effect on object equality. The results presented in this chapter categorised objects based on their interaction with collections and their settling behaviour. In particular, we observed the following properties:

- Most classes and objects use identity-as-equality, only around 6% of classes define `equals` and `hashCode`.

- Many objects and classes settle their fields during the constructor, more than half of all classes and near half of all objects.
- Non-default equals and hashCode are the norm among objects that enter collections but uncommon among objects that do not enter collections.
- Classes and objects that enter equality collections always settle their equality before entering the collections. Most objects that enter identity collections also define and settle their equality. More than half of classes and a quarter of objects in these categories change their fields after their constructors return, but very few changed their equality after entering a collection.

7.1.3 Context-independent object profiling

Chapter 5 presented a second profiler and a data aggregation framework for observing object behaviour at runtime. The profiler presented in this chapter, *rprof*, is more precise than #*prof*: it is capable of tracking all Java objects including system objects; it can monitor reads and writes to all object fields; and it can track method calls, returns and exceptional returns for any method.

rprof represents a novel approach to aggregating profiler data that allowed us to profile and analyse results from all 15 benchmarks in the DaCapo benchmark suite. The DaCapo benchmark suite was designed as a general purpose tool for the programming language, memory management, and computer architecture research communities. Profiling all objects from these non-trivial applications without sampling or otherwise limiting the profiler's scope, and doing so using commodity hardware represents a significant accomplishment that demonstrates the success of our profiler and analysis framework.

rprof is a much more robust profiler than #*prof* but we were not able to re-implement all of the features we explored with #*prof* within the scope of this PhD — in particular the ability to monitor equality settling. Future work with *rprof* could verify #*prof*'s equality settling result for all objects and for the DaCapo benchmark.

7.1.4 Field declaration settling

Using results produced by *rprof*, Section 6.3 presented a runtime study of field declaration initialisation behaviour that confirms the results that Unkel and Lam obtained using static analysis [109]. In particular, we made the following contributions:

- We confirmed that programmers do not use final field declaration annotations as often as they could. 50-68% of fields showed constructor-settled and write-settled (final-like) behaviour while only 8-36% of those fields were declared final by programmer.

- We confirmed that a high proportion of field declarations are read-settled – 62-75% – including almost all fields that could have been declared final. In addition there were a considerable proportion of field declarations that could not have been declared final that were read-settled.

- We considered the case for replacing final field declaration annotations with a constructed annotation that allowed field modification up to but not after constructor return, but concluded that while there were some fields that could be annotated with this modifier but not final, there were insufficient fields to justify adding a new annotation to languages that already have final.

- We compared the number of constructor-settled, and read-settled field declarations, concluding that a stationary annotation for field declarations that allowed fields to be written until the first time they are read would allow considerably more fields to be annotated. We conclude that new languages should strongly consider adding a stationary annotation for documenting fields that settle, while existing languages may also benefit.

From the results presented in this section we conclude that constructor methods are not a good fit for the field declaration initialisation that we observe in real-world programs. Language designers should consider other ways to enable programmers to document field declarations that represent constant properties – field declarations that settle to a particular value after initialisation.

The conclusions we draw from the results in this section are largely consistent with the conclusions of Unkel and Lam. By confirming their static analysis results using dynamic analysis we have not only strengthened their conclusions but validated our approach to observing initialisation behaviour at runtime. However, the results are not directly comparable because Unkel and Lam's static analysis is conservative: their observations correspond to a guarantee that particular code will show exactly the behaviour they identify, no matter what inputs it receives. Our dynamic observations are only relevant to context in which we observed them: a library may show particular behaviour for one set of inputs but different behaviour for another set, or when it is used in another program. Nevertheless,

our runtime study is useful for observing trends and is able determine the actual behaviour of a program rather than what behaviour is possible. These results are complimentary.

7.1.5 Object settling

Chapter 5 also analysed the settling behaviour of objects. Object behaviour is very difficult to capture using static analysis so this type of analysis is best suited to runtime profiling. Section 6.4 measured the frequency of constructor-settled, equals-settled, collection-settled, and read-settled objects in the DaCapo benchmarks. This section presented the following contributions:

- We showed that constructor-settled is an inconsistent metric for measuring object initialisation, suggesting that constructor methods are not well suited to object initialisation in general.

- We showed that read-settled objects account for the majority of objects in the programs we observed and show more consistency between programs than constructor-settled objects.

- We observed that nearly all read-settled objects are object-read-settled, suggesting that entire objects are initialised before any fields are used. This is an important result for language designers and VM implementers because it suggests that there is a point at which the whole object will cease to change, rather than individual fields settling independently.

- We showed that there is no significant correlation between the number of object-fields in an object and the likelihood that an object is settled, however there is significant correlation between the number of field declarations in a class and the likelihood that the class is settled. Together, these results show that programmers would not benefit from a whole-class annotation to document settling behaviour because their behaviour is determined by the aggregate behaviour of their fields. On the other hand, because object initialisation behaviour is not directly correlated with the behaviour of their fields – objects are sufficiently polymorphic that their settling behaviour overwhelms the declared field settling behaviour of their class – programmers may benefit from a language mechanism for documenting immutable objects that is perpendicular to their class.

7.2 Comparison to related work and future work

In our opinion the most unexpected result from this thesis is our observation that object settling behaviour is independent of the number of object-fields an object has. This is surprising because we expected that object behaviour, like class behaviour, would be correlated with the number of fields. There is considerable future work to be done in this area as we have only begun to explore this. In particular, it would be interesting to measure the number of classes whose objects show polymorphic settling behaviour — classes that have some objects that settle and some that do not. It would also be interesting to compare the settling behaviour of objects with object longevity: are most settled objects short-lived, for example?

If object settling behaviour independent of class is an important part of existing programs there are several ways that a language could support this. For example, an "immutable" type parameter could be used to annotate uninitialised objects then a special operator could be used to declare that an object has been initialised and can safely be used. Type systems such as Fähndrich and Xia's *Delayed Types* [35] could be helpful.

The profiling approach used by *rprof* was very successful, but there are significant improvements that could be made. Some of these improvements are largely engineering, such as applying existing research to compress the result streams on the fly, and combining duplicate sequential events to reduce the number of updates. More advanced improvements could enable feedback from the analysis to disable field event generation for fields which are already known to be mutable, for example. *rprof*'s design is inherently parallel and we believe that with sufficient tuning and hardware it could achieve near real-time performance so that graphical and user-interactive programs can be analysed. This could allow *rprof* to be used as a code comprehension tool. A significant shortcoming of *rprof* is its inability to track arrays. *rprof* can already identify array creation events but future work could add the byte code modification support necessary to generate events for array modifications and accesses so that array-aware aggregations could be implemented as array value tracking is not supported by JVMTI directly.

Marinov and O'Callahan's use of runtime profiling to identify and merge indistinguishable objects shows an interesting application of the type of profiling we perform in this chapter [63]. Their results, however, are not applicable to running programs and like ours can only be determined after the fact. Our observation of frequent read-settled object behaviour could be used by VMs, for example, as a predictor of object immutability: when a field read occurs the VM

could tag the object as *initialised*, and in about 60% of cases that object would not subsequently change any fields. A VM could move initialised objects to read-optimised memory regions then move any objects that subsequently change into write-optimised regions.

7.3 Conclusion

In conclusion, this thesis has presented a comprehensive study of initialisation behaviour in existing Java programs. We have examined the initialisation of object equality and field declarations, we have replicated studies showing that final is poorly utilised for field annotations, that *stationary* would be a better annotation, and we have shown that most fields and objects settle by an easily observable program event. We have also shown that object settling occurs sufficiently frequently that it is not directly related to field declaration behaviour. We conclude that *constructors* are a poor match for the types of object initialisation that programmers actually use and future languages should consider alternative techniques for initialising program state.

Glossary

== Java's equivalence operator operator for comparing objects by reference. This operator is shared with other C-like languages and similar operators are available in most object-oriented languages (see Section 2.2.2). 12–14, 35

collection-settled A settling behaviour property defined in Section 3.2.6. All field writes occur before the object enters a Java Collections API collection. vi, x, 32–34, 36, 38, 40, 41, 51, 56, 58–62, 64, 67, 93, 94, 108, 115–117, 128

constructor-settled A settling behaviour property defined in Section 3.2.1. All field writes occur before constructor return. v–vii, ix, x, 26–31, 35, 36, 38, 40, 41, 46, 56, 58–64, 93, 94, 97–118, 123, 127, 128

deep state Any mutable state of an object that is not stored in its fields. This could include state stored in referenced arrays or objects. 41, 59, 67

EGAL An equivalence operator for identity that identifies objects with mutable fields by reference and without mutable fields by their field values. EGAL was first proposed by Baker for use in Common Lisp [8] (see Section 2.2.2). 11, 14, 15, 22

equality Any properties of an object used to compute equals and hashCode. These may include object-fields of the object, object-fields of objects referenced by the object's object-fields, and state stored in arrays in object-fields. Equality is defined in Section 3.3. xi, 31, 32, 35, 36, 40, 41, 132

equals Java's user-defined equivalence operator for objects. Java's API documentation requires that equals method implementations should be *reflexive*, *symmetric*, *transitive*, and *consistent*, as long as an object's state does not change. ix, 12–15, 27, 31–43, 45, 46, 56, 59, 61, 63, 65, 67, 107, 113, 125, 126, 131, 132

equals-settled A settling behaviour property defined in Section 3.2.5. All field writes occur before the first method call to equals or hashCode. v, x, 31–33, 94, 108, 113–115, 117, 128

equivalence operator A comparison operator (binary) that is *reflexive*, *symmetric*, and *transitive*. Equivalence operators in imperative languages like Java may also be *consistent*, that is, given the same input they should return the same result. v, 14, 131

final Java field declarations can be annotated with final and Java will ensure that they are only written once, and only from a constructor. As a settling behaviour, final defers to field declarations and their corresponding object-fields that are annotated with final, defined in Section 3.2.1. vii, ix, xi, 2, 27–32, 97–106, 123, 127

hashCode A Java method for optimising equals comparisons. hashCode returns an integer based on the object's equality, by default derived from the object's memory location. Java programmers who override equals should ensure that if equals returns true for an object comparison then both objects should have the same hashCode value. ix, 12, 13, 15, 31–37, 39–46, 50–53, 56, 59, 61, 63, 65–67, 107, 113, 125, 126, 131, 132

identity *"Identity is that property of an object which distinguishes each object from all others"* [53]. Most object-oriented languages including Java use identity to mean reference equality: if two references are the same the objects they refer to are indistinguishable. See Section 2.2.1. 6, 13, 14, 131

identity-as-equality An object's equality is defined using reference comparisons so its equality settles as soon as the object is created (before the constructor runs). An equality-specific settling property defined in Section 3.3.1. vi, 35, 38, 40, 41, 56, 58–65, 67, 125

mutable In general, a property that can or does change. In the context of settling behaviour, a program entity that does not settle. Defined in Section 3.2.7. vi, 34, 37, 38, 40–42, 46, 56, 58–65, 67, 93

object-field A particular field slot inside a program object that corresponds to a field declaration in the object's class hierarchy. For example, if a Point class declares two fields, x and y, then each Point object will have two object-fields

corresponding to the field declarations. xi, 14, 26–34, 40, 41, 69, 70, 107, 108, 117–119, 125, 128, 129, 131, 132

object-read-settled A settling behaviour property defined in Section 3.2.3. All field writes occur before the first field read of any field for a particular object. vii, x, 108, 117, 118, 128

read-settled A settling behaviour property defined in Section 3.2.2. All field writes occur before the first field read. v, vii, ix–xi, 29, 30, 32, 33, 93, 94, 97–103, 105–121, 124, 127–129

settled A program entity that has stopped changing and will not change again for the duration of a program. Chapter 3 identifies several behaviour patterns for identifying settled program entities. 133

settling See settled. 132

write-settled A settling behaviour property defined in Section 3.2.4. No field writes occur after the first field write, or, the field is written at most once. v, 30, 31, 97–105, 127

Bibliography

[1] AGESEN, O., AND GARTHWAITE, A. Efficient object sampling via weak references. In *Proceedings of the 2nd international symposium on Memory management* (New York, NY, USA, 2000), ISMM '00, ACM, pp. 121–126.

[2] ALDRICH, J., KOSTADINOV, V., AND CHAMBERS, C. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), OOPSLA '02, ACM, pp. 311–330.

[3] ANDERSON, J. M., BERC, L. M., DEAN, J., GHEMAWAT, S., HENZINGER, M. R., LEUNG, S.-T. A., SITES, R. L., VANDEVOORDE, M. T., WALDSPURGER, C. A., AND WEIHL, W. E. Continuous profiling: where have all the cycles gone? In *Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 1–14.

[4] ANDERSON, T. E., AND LAZOWSKA, E. D. Quartz: a tool for tuning parallel program performance. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (New York, NY, USA, 1990), SIGMETRICS '90, ACM, pp. 115–125.

[5] ARNOLD, K., GOSLING, J., AND HOLMES, D. *The Java Programming Language*, fourth ed. Addison-Wesley, 2007.

[6] ARNOLD, M., HIND, M., AND RYDER, B. G. Online feedback-directed optimization of Java. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), OOPSLA '02, ACM, pp. 111–129.

[7] ARNOLD, M., AND RYDER, B. G. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), PLDI '01, ACM, pp. 168–179.

[8] BAKER, H. G. Equal rights for functional objects or, the more things change, the more they are the same. *SIGPLAN OOPS Mess. 4*, 4 (Oct. 1993), 2–27.

[9] BALL, T. The concept of dynamic analysis. In *Software Engineering — ESEC/FSE '99*, O. Nierstrasz and M. Lemoine, Eds., vol. 1687 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1999, pp. 216–234.

[10] BALZER, S., GROSS, T., AND EUGSTER, P. A relational model of object collaborations and its use in reasoning about relationships. In *ECOOP 2007 – Object-Oriented Programming* (2007), E. Ernst, Ed., vol. 4609 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 323–346.

[11] BECK, K., AND CUNNINGHAM, W. A laboratory for teaching object oriented thinking. In *Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1989), OOPSLA '89, ACM, pp. 1–6.

[12] BIERMAN, G., AND WREN, A. First-class relationships in an object-oriented language. In *ECOOP 2005 - Object-Oriented Programming* (2005), A. Black, Ed., vol. 3586 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 733–733.

[13] BINDER, W., AND HULAAS, J. Exact and portable profiling for the jvm using bytecode instruction counting. *Electronic Notes in Theoretical Computer Science 164*, 3 (2006), 45–64.

[14] BIRTWISTLE, G. M., DAHL, O. J., MYHRHAUG, B., AND NYGAARD, K. *Simula Begin*. Studentlitteratur, 1979.

[15] BLACKBURN, S. M., GARNER, R., HOFFMANN, C., KHANG, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 169–190.

[16] BLOCH, J. *Effective Java*. Prentice Hall PTR, 2008.

[17] BOOCH, G. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[18] BREAR, D. J., WEISE, T., WIFFEN, T., YEUNG, K. C., BENNETT, S. A., AND KELLY, P. H. J. Search strategies for Java bottleneck location by dynamic instrumentation. *IEE Proceedings – Software 150*, 4 (2003), 235–241.

[19] BROWN, S., MITCHELL, A., AND POWER, J. A coverage analysis of java benchmark suites. In *the IASTED International Conference on Software Engineering, Innsbruck, Austria* (2005).

[20] BRUNDEGE, J. Don't let Hibernate steal your identity. `http://onjava.com/pub/a/onjava/2006/09/13/dont-let-hibernate-steal-your-identity.html`, September 2006.

[21] BRUNETON, E. ASM 3.0 a Java bytecode engineering library. `http://download.forge.objectweb.org/asm/asmguide.pdf`, 2007.

[22] BULL, J., SMITH, L., WESTHEAD, M., HENTY, D., AND DAVEY, R. A benchmark suite for high performance java. *Concurrency - Practice and Experience 12*, 6 (2000), 375–388.

[23] CAIN, H. W., MILLER, B. P., AND WYLIE, B. J. A callgraph-based search strategy for automated performance diagnosis. In *Proceedings of the European Conference on Parallel Processing (Euro-Par)* (2001), Springer-Verlag, pp. 108–122.

[24] CHAN, P., AND LEE, R. *The Java Class Libraries, Second Edition, Volume 1*. Addison-Wesley, 1999.

[25] CHEN, G., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. J. Field level analysis for heap space optimization in embedded Java environments. In *Proceedings of the 4th international symposium on Memory management* (New York, NY, USA, 2004), ISMM '04, ACM, pp. 131–142.

[26] CHEN, P. P. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (March 1976), 9–36.

[27] CHENG, P., HARPER, R., AND LEE, P. Generational stack collection and profile-driven pretenuring. In *Proc. of the ACM Conference on Programming Language Design and Implementation* (1998), ACM Press, pp. 162–173.

[28] CHODOROW, K., AND DIROLF, M. *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2010.

[29] CLARKE, D., AND DROSSOPOULOU, S. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), OOPSLA '02, ACM, pp. 292–310.

[30] CONSTANZA, P., AND HASSE, A. The comparand pattern: Cheap identity testing using dedicated values. In *Pattern Languages of Program Design 5*, J. N. Dragos-Anton Manolescu, Markus Voelter, Ed., Software Patterns Series. Addison-Wesley, 2006, ch. 8, pp. 169–188.

[31] COSTANZA, P. *Transmigration of Object Identity*. PhD thesis, Institut für Informatik III, Universität Bonn, 2004.

[32] DAHL, O.-J., AND NYGAARD, K. Simula: an algol-based simulation language. *Commun. ACM 9*, 9 (1966), 671–678.

[33] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[34] DIECKMAN, S., AND HOELZLE, U. A study of the allocation behavior of the SPECjvm98 Java benchmarks. *Lecture Notes in Computer Science 1628* (1999), 92–115.

[35] FAHNDRICH, M., AND XIA, S. Establishing object invariants with delayed types. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications* (New York, NY, USA, 2007), OOPSLA '07, ACM, pp. 337–350.

[36] FILLIÂTRE, J., AND CONCHON, S. Type-safe modular hash-consing. In *Proceedings of the 2006 workshop on ML* (2006), vol. 26, ACM, pp. 12–19.

[37] GAMMA, E., HELM, R., JOHNSON, R. E., AND VLISSIDES, J. M. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP* (1993), pp. 406–431.

[38] GOETZ, B. Java theory and practice: Hashing it out. `http://www.ibm.com/developerworks/java/library/j-jtp05273/index.html`, May 2003.

[39] GOETZ, B. Jsr 335: Lambda expressions for the Java programming language. `http://www.jcp.org/en/jsr/detail?id=335`, June 2012.

[40] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[41] GRADECKI, J. D., AND LESIECKI, N. *Mastering AspectJ : Aspect-Oriented Programming in Java*. Wiley, 2003.

[42] GRAHAM, S. L., KESSLER, P., AND MCKUSICK, M. gprof: a call graph execution profiler. In *ACM Symposium on Compiler Construction* (1982), ACM Press, pp. 120–126.

[43] GROGONO, P., AND SAKKINEN, M. Copying and comparing: Problems and solutions. In *Proceedings of the 14th European Conference on Object-Oriented Programming* (London, UK, UK, 2000), ECOOP '00, Springer-Verlag, pp. 226–250.

[44] GUO, Z., AMARAL, J. N., SZAFRON, D., AND WANG, Y. Utilizing field usage patterns for Java heap space optimization. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research* (New York, NY, USA, 2006), CASCON '06, ACM.

[45] HAACK, C., AND POLL, E. Type-based object immutability with flexible initialization. In *ECOOP 2009 – Object-Oriented Programming*, S. Drossopoulou, Ed., vol. 5653 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 520–545.

[46] HIRZEL, M., HENKEL, J., DIWAN, A., AND HIND, M. Understanding the connectivity of heap objects. In *Proceedings of the 3rd international symposium on Memory management* (New York, NY, USA, 2002), ISMM '02, ACM, pp. 36–49.

[47] HOGG, J. Islands: Aliasing protection in object-oriented languages. In *ACM SIGPLAN Notices* (1991), vol. 26, ACM, pp. 271–285.

[48] HOVEMEYER, D., AND PUGH, W. Finding bugs is easy. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2004), OOPSLA '04, ACM, pp. 132–136.

[49] INOUE, H., STEFANOVIC, D., AND FORREST, S. On the prediction of Java object lifetimes. *IEEE Transactions on Computers 55*, 7 (July 2006), 880–892.

[50] IRELAND, C., BOWERS, D., NEWTON, M., AND WAUGH, K. A classification of object-relational impedance mismatch. In *Proceedings of the 2009 First*

International Conference on Advances in Databases, Knowledge, and Data Applications (Washington, DC, USA, 2009), DBKDA '09, IEEE Computer Society, pp. 36–43.

[51] JAVA GRANDE FORUM. The java grande benchmark suite. `http://www.epcc.ed.ac.uk/javagrande/`, 2006.

[52] JUMP, M., BLACKBURN, S. M., AND MCKINLEY, K. S. Dynamic object sampling for pretenuring. In *Proceedings of the 4th international symposium on Memory management* (New York, NY, USA, 2004), ISMM '04, ACM, pp. 152–162.

[53] KHOSHAFIAN, S. N., AND COPELAND, G. P. Object identity. In *Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1986), OOPLSA '86, ACM, pp. 406–416.

[54] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. An overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming*, J. Knudsen, Ed., vol. 2072 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001, pp. 327–354.

[55] KUMAR, R. V., NARAYANAN, B. L., AND GOVINDARAJAN, R. Dynamic path profile aided recompilation in a Java just-in-time compiler. In *High Performance Computing — HiPC 2002* (2002), vol. 2552 of *Lecture Notes in Computer Science*, pp. 495–506.

[56] LADDAD, R. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.

[57] LEE, W. H., AND CHANG, J. M. An integrated dynamic memory tracing tool for C++. *Information Sciences 151* (2003), 27–49.

[58] LEINO, K., MÜLLER, P., AND WALLENBURG, A. Flexible immutability with frozen objects. In *Verified Software: Theories, Tools, Experiments*, N. Shankar and J. Woodcock, Eds., vol. 5295 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 192–208.

[59] LIANG, S., AND VISWANATHAN, D. Comprehensive profiling support in the Java Virtual Machine. In *Proceedings of the USENIX Conference On Object Oriented Technologies and Systems* (1999), USENIX Association, pp. 229–240.

[60] LISKOV, B., AND GUTTAG, J. V. *Abstraction and Specification in Program Development*. MIT Press/McGraw-Hill, 1986.

[61] MACLENNAN, B. J. Values and objects in programming languages. *SIGPLAN Notices 17*, 12 (December 1982), 70–79.

[62] MAIER, D. Representing database programs as objects. In *Advances in database programming languages* (1990), ACM, pp. 377–386.

[63] MARINOV, D., AND O'CALLAHAN, R. Object equality profiling. *ACM SIGPLAN Notices 38*, 11 (2003), 313–325.

[64] MEYER, B. *Object-oriented software construction*, vol. 2. Prentice hall New York, 1988.

[65] MITCHELL, N. The runtime structure of object ownership. In *ECOOP 2006 – Object-Oriented Programming*, D. Thomas, Ed., vol. 4067 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 74–98.

[66] MORDANI, R. Java servlet specification version 3.0. `http://jcp.org/aboutJava/communityprocess/final/jsr315/index.html`, June 2012.

[67] NELSON, S. JVM hotspot bug report: JVMTI field modification callback side-effect. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7162645`, April 2012.

[68] NELSON, S., PEARCE, D. J., AND NOBLE, J. Understanding the impact of collection contracts on design. In *Proceedings of the 48th international conference on Objects, models, components, patterns* (Berlin, Heidelberg, 2010), TOOLS'10, Springer-Verlag, pp. 61–78.

[69] NELSON, S., PEARCE, D. J., AND NOBLE, J. Understanding the impact of collection contracts on design. Tech. rep., Victoria University of Wellington, 2010.

[70] NELSON, S., PEARCE, D. J., AND NOBLE, J. Profiling field initialization in Java. In *Runtime Verification* (September 2012), S. Qadeer and S. Tasiran, Eds., vol. 7687 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg.

[71] NOBLE, J. Basic relationship patterns. In *Pattern Languages of Program Design 4*, N. Harrison, B. Foote, and H. Rohnert, Eds. Addison-Wesley, 2000, ch. 6, pp. 73–94.

[72] NYGAARD, K. Basic concepts in object oriented programming. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming* (New York, NY, USA, 1986), ACM, pp. 128–132.

[73] ODERSKY, M. *Programming in Scala*. Artima, Inc, 2008.

[74] O'HAIR, K. The JVMPI transition to JVMTI. `http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition/`, July 2004.

[75] ORACLE. Java virtual machine tool interface (JVMTI). `http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/`, June 2012.

[76] ØSTERBYE, K. Associations as a language construct. In *Proceedings of the Technology of Object-Oriented Languages and Systems* (Washington, DC, USA, 1999), TOOLS '99, IEEE Computer Society, pp. 224–.

[77] ØSTERBYE, K. Design of a class library for association relationships. In *Proceedings of the 2007 Symposium on Library-Centric Software Design* (New York, NY, USA, 2007), LCSD '07, ACM, pp. 67–75.

[78] ÖSTLUND, J., WRIGSTAD, T., CLARKE, D., AND ÅKERBLOM, B. Ownership, uniqueness, and immutability. In *Proceedings of the 46th international conference on Objects, Components, Models and Patterns* (2008), R. F. Paige, B. Meyer, W. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, Eds., vol. 11 of *Lecture Notes in Business Information Processing*, Springer Berlin Heidelberg, pp. 178–197.

[79] PATRICIO, A. Hibernate: Equals and hashcode. `https://community.jboss.org/wiki/EqualsandHashCode`, July 2009.

[80] PEARCE, D. J., AND NOBLE, J. Relationship aspects. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development* (New York, NY, USA, 2006), ACM Press, pp. 75–86.

[81] PEARCE, D. J., WEBSTER, M., BERRY, R., AND KELLY, P. H. J. Profiling with AspectJ. *Software-Practice &Experience 37*, 7 (June 2007), 747–777.

[82] PECHTCHANSKI, I., AND SARKAR, V. Immutability specification and its applications. *Concurrency and Computation: Practice and Experience 17*, 5-6 (2005), 639–662.

[83] PORAT, S., BIBERSTEIN, M., KOVED, L., AND MENDELSON, B. Automatic detection of immutable fields in Java. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research* (2000), CASCON '00, IBM Press, p. 10.

[84] POSTGRESQL GLOBAL DEVELOPMENT GROUP. PostgreSQL. `http://www.postgresql.org`, June 2012.

[85] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Generic ownership for generic Java. *SIGPLAN notices 41*, 10 (2006), 311–324.

[86] POTANIN, A., NOBLE, J., FREAN, M., AND BIDDLE, R. Scale-free geometry in OO programs. *Communications of the ACM 48*, 5 (May 2005), 99–103.

[87] QI, X., AND MYERS, A. C. Masked types for sound object initialization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2009), POPL '09, ACM, pp. 53–65.

[88] QUALITAS RESEARCH GROUP. Qualitas corpus release 20080603. `http://www.cs.auckland.ac.nz/~ewan/corpus/` The University of Auckland, June 2008.

[89] RÖJEMO, N., AND RUNCIMAN, C. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 1996), ICFP '96, ACM, pp. 34–41.

[90] RUMBAUGH, J. Relations as semantic constructs in an object-oriented language. In *Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1987), OOPSLA '87, ACM, pp. 466–481.

[91] RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. *Object-oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[92] RUMBAUGH, J. E., BOOCH, G., AND JACOBSON, I. *Unified Modelling Language Specification*. Object Management Group, Framingham, Mass., 1998.

[93] RUPAKHETI, C. R., AND HOU, D. An empirical study of the design and implementation of object equality in Java. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds* (New York, NY, USA, 2008), CASCON '08, ACM, pp. 9:111–9:125.

[94] SASTRY, S. S., BODÍK, R., AND SMITH, J. E. Rapid profiling via stratified sampling. In *Proceedings of the 28th annual international symposium on Computer architecture* (New York, NY, USA, 2001), ISCA '01, ACM, pp. 278–289.

[95] SHAH, A. V., HAMEL, J. H., BORSARI, R. A., AND RUMBAUGH, J. E. Dsm: an object-relationship modeling language. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1989), ACM, pp. 191–202.

[96] SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. Heap profiling for space-efficient Java. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), PLDI '01, ACM, pp. 104–113.

[97] SHANKAR, A., ARNOLD, M., AND BODIK, R. Jolt: lightweight dynamic analysis and removal of object churn. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications* (New York, NY, USA, 2008), OOPSLA '08, ACM, pp. 127–142.

[98] SINGER, J., BROWN, G., LUJAN, M., AND WATSON, I. Towards intelligent analysis techniques for object pretenuring. In *Principles and Practice of Programming in Java* (Lisbon, Sept. 2007), ACM Press.

[99] SPIVEY, J. M. Fast, accurate call graph profiling. *Software-Practice & Experience 34*, 3 (March 2004), 249–264.

[100] STANDARD PERFORMANCE EVALUATION CORPORATION. Spec jvm98 benchmarks. `http://www.spec.org/jvm98/`, 1998.

[101] STANDARD PERFORMANCE EVALUATION CORPORATION. Specjbb2005. `http://www.spec.org/jbb2005/`, 2005.

[102] STANDARD PERFORMANCE EVALUATION CORPORATION. Specjvm2008. `http://www.spec.org/jvm2008/`, 2008.

[103] STEELE, G. *Common LISP: the language*, 2nd ed. Digital Press, 1990.

[104] STENBERG, D. cURL. `http://curl.haxx.se/`, June 2012.

[105] SUN MICROSYSTEMS, INC. *Java Platform, Standard Edition 6 API Specification*, 2009.

[106] TEMPERO, E., ANSLOW, C., DIETRICH, J., HAN, T., LI, J., LUMPE, M., MELTON, H., AND NOBLE, J. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)* (Dec. 2010), pp. 336–345.

[107] THE APACHE SOFTWARE FOUNDATION. Apache geronimo. `http://geronimo.apache.org/`, June 2012.

[108] THE ECLIPSE FOUNDATION. Jetty. `http://www.eclipse.org/jetty/`, June 2012.

[109] UNKEL, C., AND LAM, M. S. Automatic inference of stationary fields: a generalization of Java's final fields. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2008), POPL '08, ACM, pp. 183–195.

[110] VAZIRI, M., TIP, F., FINK, S., AND DOLBY, J. Declarative object identity using relation types. In *European Conference on Object Oriented Programming* (2007), vol. 4609 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 54–78.

[111] WHALEY, J. A portable sampling-based profiler for Java Virtual Machines. In *Proceedings of the ACM Java Grande Conference* (2000), ACM Press, pp. 78–87.

[112] ZIBIN, Y., POTANIN, A., ALI, M., ARTZI, S., KIE, UN, A., AND ERNST, M. D. Object and reference immutability using Java generics. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (New York, NY, USA, 2007), ESEC-FSE '07, ACM, pp. 75–84.

[113] ZIBIN, Y., POTANIN, A., LI, P., ALI, M., AND ERNST, M. D. Ownership and immutability in generic Java. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 598–617.

[114] ZORN, B., AND HILFINGER, P. A memory allocation profiler for C and Lisp programs. In *Proceedings of the Summer USENIX Conference* (1988), USENIX Association, pp. 223–237.