INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

On Preserving Domain Consistency for an Evolving Enterprise Application

João Coutinho dos Santos Roxo Neves

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Prof. Doutor Mário Rui Fonseca dos Santos Gomes
Orientador:	Prof. Doutor João Manuel Pinheiro Cachopo
Vogal:	Prof. Doutor David Manuel Martins de Matos

Julho de 2011

Acknowledgements

The work of this master thesis dissertation was carried out with the support and inspiration of several people. I would like to thank everyone in the ESW Software Engineering Group, especially my advisor, Professor João Cachopo, to whom I owe his guidance and determination. Much of the quality of this work was achieved thanks to the sharing of his knowledge, and his decisive thoughts on the subject.

I would also like to thank my current work colleagues at the Fénix Project: Luis Cruz, Susana Fernandes, Ricardo Rodrigues, Pedro Santos, Sérgio Silva, Artur Ventura, and former colleagues Paulo Abrantes, and João Figueiredo. For their encouragement, a few friends from younger years deserve a special cheer: Bruno Almeida, Cláudio Diniz, Sérgio Gomes, Carlos Jacinto, João Lemos, Diogo Oliveira, Filipe Paupério, Inês Perdigão, João Reis, and David Seixas.

Last but never least, a special mention for their understanding and unending confidence goes to Luisa and Paulo, without whom none of this work would have been possible.

Lisboa, July 14, 2011
João Coutinho dos Santos Roxo Neves

Resumo

Muitos dos sistemas de informação existentes são aplicações empresariais, orientadas ao domínio, que manipulam um grande volume de dados complexos. Tipicamente, estes sistemas mantêm os dados organizados segundo um denso grafo de objectos que devem satisfazer complexas regras de consistência. A Fénix Framework permite ao programador definir regras de consistência, através de predicados de consistência que os objectos devem satisfazer. A framework usa estes predicados para verificar a consistência dos dados à medida que mudam ao longo do tempo.

Apesar dos predicados de consistência já serem suportados, eles ainda não têm em conta a persistência dos dados nem o desenvolvimento incremental, que são aspectos comuns em aplicações empresariais. Estas aplicações evoluem ao longo do tempo, por vezes com alterações incompatíveis nas suas regras de consistência, e contudo, precisam sempre de manter os seus dados. É portanto crucial verificar a consistência dos dados à medida que o código muda ao longo do tempo.

Esta dissertação descreve algumas extensões feitas à Fénix Framework, que se baseia em acções atómicas ao nível da linguagem de programação para implementar predicados de consistência. Este trabalho mostra como a persistência e o desenvolvimento incremental tornam tão difícil a tarefa de verificar regras de consistência, e propõe soluções pragmáticas para contornar estas dificuldades.

Em particular, com esta proposta, a framework torna-se ciente de alterações a regras de consistência: detecta regras novas e antigas, e toma acções adequadas com base nas mudanças. Para além disto, a framework suporta tolerância a inconsistências, para ajudar a estabelecer novas regras sobre dados já existentes, e possivelmente inconsistentes. Por último, a framework fornece ao programador informação estruturada sobre as inconsistências existentes nos dados.

Abstract

Many of today's existing information systems are domain-intensive enterprise applications that manage and store vast volumes of complex data. Typically, these systems keep their data organized as a graph of highly interrelated objects, which must satisfy complex domain consistency rules. The Fénix Framework allows the programmer to define consistency rules, by implementing consistency predicates that objects must satisfy. The framework uses these predicates to verify the consistency of the data, as it changes over time.

Even though the Fénix Framework already supported consistency predicates before, it did not take into account the aspects of data persistence and incremental development, both of which are common in enterprise applications. In fact, these applications evolve over time, often with incompatible changes on their behavior and consistency rules, and still always need to preserve their data. Thus, it is crucial for the framework to verify the consistency of the data, as the code changes over time.

This dissertation describes a few extensions to the Fénix Framework, which relies on atomic actions at the programming language level to implement consistency predicates. It shows why persistence and incremental development make the task of verifying these rules so difficult, and proposes pragmatical solutions to address these issues.

In particular, with this proposal, the framework becomes aware of changes within consistency rules: it is able to detect both new and old rules, and takes action accordingly. Also, it supports inconsistency tolerance to help in establishing new consistency rules on already existing, and possibly inconsistent data. Finally, it gives the programmer explicit and structured information about the existing inconsistencies within the data.

Palavras Chave

Consistência dos Dados
Regras de Consistência
Invariantes
Persistência dos Dados
Tolerância a Inconsistências
Alterações de Regras
Análise de Impacto de Alterações

Keywords

Data Consistency
Consistency Rules
Invariants
Data Persistence
Inconsistency Tolerance
Rule Changes
Change Impact Analysis

Contents

List of Figures

Chapter 1

Introduction

To satisfy their clients' needs, software companies design enterprise applications that manipulate large volumes of complex data. This data is composed of many small interrelated parcels of important information, which only make sense together, as a consistent whole. To guarantee a good quality of service, it is crucial for the companies to ensure that their clients always perceive a consistent view of the data.

Many modern enterprise applications [Fowler, 2002] must implement a rich, complex domain model. Their domain is typically implemented with a dense graph of classes and relationships in some object-oriented programming language. Developing such a complex domain model is often a challenging task for most development teams.

To tackle this problem, the Fénix Framework[1] provides a Domain Modeling Language (DML) and a Software Transactional Memory (STM) [Cachopo, 2007, Cachopo and Rito-Silva, 2006]. Its goal is to simplify the development of rich domain models that follow a domain-driven design approach [Evans, 2003]. It separates their **structure**, which is implemented in DML, from their **behavior**, which is implemented in Java. Also, to keep the data consistent, it separates the implementation of **consistency rules**, which are implemented with consistency predicates.

The consistency predicates allow programmers to implement consistency rules that the application's data must satisfy, independently of the implementation of the application's behavior. They are checked automatically at the end of each business transaction to ensure that all the consistency rules were followed, in which case the transaction can commit. Otherwise, if any consistency rule was violated, the transaction aborts to prevent introducing inconsistencies in the application's data.

However, the original proposal of this approach assumed an execution model where the application keeps all of its data in transient memory. The application would start from an empty state, and run forever without updates to its code. Clearly, this is not the common case for enterprise applications, which must persist their data and are incrementally developed over a long period of time.

Under these conditions, the application's **structure**, its **behavior**, and its **consistency predicates** are all subject to code changes. Still, each new version of the code must deal with the previously persisted data. New consistency rules are often added for already existing data, which may be inconsistent according to the new rules. Therefore, it is important to deal with these rule changes as pragmatically as possible.

During the course of this work, I have implemented an extension to the Fénix Framework, which is used by several large-scale web applications [Fernandes and Cachopo, 2011]. With this work, the consistency predicates take into account the pragmatic needs of a programming team that incrementally

[1] `http://fenix-ashes.ist.utl.pt/trac/fenix-framework`

develops an enterprise application in Java. They support applications where the data is persisted and where the domain code evolves.

This document describes how to keep information about the existing consistency predicates across different executions of an application. It shows how to detect changes in the implementation of consistency predicates, and how to deal with these changes. Moreover, this work introduces the concept of *inconsistency tolerance* to handle existing inconsistent data, and describes how the consistency predicates support it.

The remainder of this document is organized as follows:

- Chapter 2 describes the context of this work and the components of the Fénix Framework and the original proposal of the consistency predicates.
- Chapter 3 discusses the limitations of the original consistency predicates when used in an evolving application.
- Chapter 4 shows the existing related work of consistency rules and evolving applications.
- Chapter 5 proposes a new domain model to deal with the code changes that involve the consistency predicates.
- Chapter 6 shows how the new domain of the Fénix Framework can support inconsistency tolerance.
- Chapter 7 validates the new implementation with examples of evolving applications.
- Chapter 8 describes a few ideas for future work to further improve the consistency predicates.
- Chapter 9 draws some conclusions from this work.
- Appendix A contains a few guidelines that establish an initial set of best practices, to help the programmer in using the consistency predicates.

Chapter 2

Components of the Fénix Framework

This chapter contains a detailed description of the major components of the Fénix Framework. Section 2.1 describes the Fénix Framework's Domain Modeling Language - the DML, which fully declares the application's domain model structure. Section 2.2 describes the Fénix Framework's Software Transactional Memory - the JVSTM, which allows the framework to execute the application's operations atomically in a concurrent setting. Section 2.3 describes the Fénix Framework's consistency predicates that allows the developer to implement consistency rules. The framework will automatically enforce these predicates. This last section will also explain how and when are the predicates checked.

2.1 DML - Domain Modeling Language

This work is aimed at enterprise applications with rich domain models that are incrementally developed using an object-oriented programming paradigm. In such applications, the domain generally consists of a complex graph of classes, representing entities with relations among them. All of these entities and their relationships represent the structure of the domain.

The Fénix Framework provides a domain specification language to allow the programmer to specify the structure of his applications' domain. This language is called DML - Domain Modeling Language. The goal of the DML is to separate the structure of the application's domain from its behavior.

Inside a DML file, a developer specifies a set of domain classes in a simple syntax that is similar to Java. Each class can contain several slots of certain value types or primitives. A domain class may extend another class, so domain class hierarchies are supported. If a domain class does not specify an explicit superclass, it will extend the AbstractDomainObject by default. The AbstractDomainObject is the top abstract superclass of all domain class hierarchies. It is not declared in the DML file; it is a regular abstract Java class.

Figure 2.1 and Figure 2.2 provide examples of DML declarations of two domain classes of a hypothetical banking application. This application will serve as an example throughout this dissertation.

```
class Client {
  String name;
}
```

Figure 2.1: Declaration of a domain class Client with a name slot.

Apart from declaring domain classes, the developer can also specify relations between two classes. Each relation has a multiplicity, that indicates if it is a one-to-one, one-to-many, or many-to-many

```
class Account {
  int balance;
  boolean closed;
}
```

Figure 2.2: Declaration of a domain class Account with a balance and a closed slots.

relation. Independently of the multiplicity, relations are bidirectional. None of the two entities owns the other, but either entity can use the relation to obtain the other.

Figure 2.3 illustrates a DML relation between a Client and all of its Accounts. The relation contains two entries, one for each domain entity, on each side of the intended relation. On each entry, the text on the left side of the *playsRole* keyword specifies the class name of the entity. The text on the right side specifies the name of the role that the entity has in that relation. The code generator will use this role name to create getter and setter methods for each entity. Also, the default multiplicity is 1. So, each client may have several accounts, and each account may have only a single client.

```
relation ClientAccounts {
  Client playsRole client;
  Account playsRole accounts {
    multiplicity *;
  }
}
```

Figure 2.3: Relation between the Client and its Accounts. The default multiplicity that is used for the Client is 1.

Figure 2.4 presents an UML class diagram of the banking domain example created so far in DML.

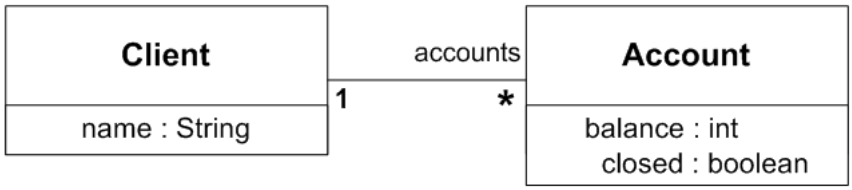

Figure 2.4: UML class diagram of the domain model of the banking example with the Client and the Account.

The Fénix framework also implements a code generator to create the application's structure in Java. The code generator processes the DML file containing the domain classes and their relations. It creates a Java class for each DML class. It also generates getter and setter methods for each slot inside the domain classes, and for each relation between classes. Figure 2.5 provides an example of the use of the generated getter methods for the Client and the Account.

Note that, in this figure, the getTotalBalance() method is part of the application's behavior, and must be implemented by the developer. However, the code generator created the Java class of the Client as an empty class for the developer to use. It also created the superclass Client_Base that contains framework code, and is not meant to be edited manually. Thus, the DML file contains only the high-level description of the domain's structure and contents. It is separated from the operations of the domain that are still implemented in plain old Java classes.

The generated code inside the Client_Base class includes the getter and setter methods for all the domain slots and relations of each class. Apart from these methods, the base classes contain the code

```
public class Client extends Client_Base {
  // (...)
  public int getTotalBalance() {
    int totalBalance = 0;
    for (Account account : getAccounts()) {
      totalBalance += account.getBalance();
    }
    return totalBalance;
  }
}
```

Figure 2.5: Example of the use of generated getter methods inside the Client class. The getAccounts() method of the Client and the getBalance() method of the Account were generated according to slots and relations of the DML file.

that deals with the transactionality and persistence of the domain objects, their slots, and relations. Overall, the DML is one of the central parts of the Fénix Framework that is integrated with the rest of its features. The developer only needs to provide a correct modeling of the application domain in a separate DML file, and implement the domain behavior in Java. The framework will guarantee that the resulting runtime is transactional, concurrent-friendly, and persistent.

2.2 JVSTM - Java Versioned Software Transactional Memory

The Java Versioned Software Transactional Memory (JVSTM) [Cachopo, 2007] is a Java-based Software Transactional Memory. Software Transactional Memory (STM) in general will be described in greater detail in Chapter 4. In short, STM systems provide transactions as an efficient and easy to use synchronization approach for highly concurrent applications. Transactions are an alternative to the traditional use of locks, which induce implementation complexity and deadlocks. Transactions avoid these problems by allowing an all-or-none, atomic execution of a sequence of operations, with the added benefit of fault recovery.

The Fénix Framework includes the JVSTM, which provides transactions with ReadSets and WriteSets to track the operations they performed, and on which parts of the data. To identify the parts of the data, the JVSTM also provides versioned memory locations, which are called versioned boxes (VBoxes). A VBox stores a history of values, one for each version of the world.

Each transaction starts at a certain moment in time, and acquires a version number to use. Afterwards, it reads only values that existed at this version of the world, so that it can perceive a snapshot of the application state. This allows other concurrent transactions to write to the same VBoxes simultaneously (on higher versions) without generating conflicts.

The previous section has shown that the code generator creates code inside the base classes of each DML domain class. This code will already contain VBoxes in the correct locations to make sure that the contents of each domain class are transactional. Therefore, the framework assures that every single piece of domain data is contained inside VBoxes. Moreover, every single operation that the system executes is included inside a transaction.

In summary, the previous section has shown that the Fénix Framework contains the application's domain structure in DML. Moreover, thanks to the JVSTM, it also has the most relevant aspects of the application's behavior under surveillance. The JVSTM contains a well-known set of VBoxes, which contain all the domain data. It is also aware of all the operations that are performed inside transactions,

to be able to control concurrent accesses to the shared memory locations. Thus, the framework can observe the evolution of the application's data, which is the ideal environment to define and to enforce the application's consistency rules.

2.3 Consistency Predicates

The domain entities specified in DML contain data, and behavior that is responsible for operating on the data. In a domain-driven design, this behavior is at the core of the application [Evans, 2003]. It is defined by a set of operations that read from and write to the data of these many entities.

These operations, however, must guarantee that the data remains consistent throughout the application life-cycle. They must abide by certain rules when writing to the domain data. These rules restrain the domain operations, and are often called domain constraints.

In general, consistency rules denote what conditions must the data always satisfy. These rules have much in common with invariants in a Design By Contract approach, which will be discussed in detail in Chapter 4. Typically, these rules are only verified on each state transition; after each operation has finished.

The Fénix Framework allows the programmer to define consistency predicates as plain Java methods within the domain classes. These methods must return a boolean value, receive no arguments, and contain the @ConsistencyPredicate annotation. Figure 2.6 shows the implementation of a simple consistency predicate in the banking domain example.

```
public class Account extends Account_Base {
  // (...)
  @ConsistencyPredicate
  public boolean closedAccountHasNoMoney() {
    return !isClosed() || getBalance() == 0;
  }
}
```

Figure 2.6: Code of the consistency predicate responsible for checking that closed accounts have no money.

A consistency predicate invoked on a domain object should return false if the object is inconsistent, and return true otherwise. But the consistency predicates are not called explicitly by the programmer. Instead, they are automatically identified and executed by the Fénix Framework. Thanks to the transactions provided by the JVSTM, the framework can execute the predicates at the end of each write transaction that creates or changes any domain object. If any consistency predicate returns false, the domain is inconsistent and the transaction aborts. Otherwise, the transaction can commit.

2.3.1 Traditional Defensive Programming

This work is aimed at the intensive use of consistency predicates. I argue that they provide the best implementation technique to implement domain constraints, independently from the rest of the application's behavior. To understand why, it is important to compare the consistency predicates to the traditional way of implementing domain constraints.

Development teams that do not use consistency predicates still need to implement their domain constraints. Arguably, defensive programming is the most common technique used nowadays to implement these constraints. It is common practice to have the programmers defensively protect the beginning of each operation with numerous checks on the domain state.

Most of these constraints indicate under what conditions can an operation take place (Figure 2.7). The checks at the beginning of an operation are an attempt to ensure the consistency of the domain at the end of the operation. In the example of Figure 2.7, the intent of the programmer is to prevent the total balance of the client to become negative after a withdraw from an account.

```
public class Account extends Account_Base {
  // (...)
  public integer withdraw(int amount) {
    if (getClient().getTotalBalance() < amount) {
      return 0;
    }
    setBalance(getBalance() - amount);
    return amount;
  }
}
```

Figure 2.7: A domain method that withdraws a certain amount from an Account. Before doing so, it checks some conditions.

These kinds of verifications express the conditions that the domain state must satisfy prior to the operation's changes, such that it will not get inconsistent after the operation's changes. This approach forces the programmer to reason backwards, to predict the necessary conditions for a proper execution of the operation. Unfortunately, as operations become more complex, it gets harder to reason backwards and to implement these conditions correctly. Besides, even if these conditions are met, they do not give a proven guarantee that the domain will indeed be consistent after the operation's effects are done.

One way to formally guarantee that the effects will not break the consistency, is by providing a formal specification of these effects and by proving the operation's implementation correctness. This approach is called Design by Contract and will be discussed in Chapter 4.

Another way to guarantee the consistency at the end of the operation is by actually executing the operation and then checking the consistency by its definition. If the domain is no longer consistent, the system needs a way to undo the effects of the operation. This approach can be implemented by STM systems that include transactions with an all-or-none execution mode.

Moreover, the approach of using defensive programming presents several other disadvantages. It limits composition, because a small operation that verifies rules may need to break the consistency temporarily inside a larger operation. Also, it results in code scattering, because the code of each rule is split throughout all the operations that change state. Moreover, it results in code tangling, because an operation contains the code of each rule and the code of its core behavior. Last but not least, it fails to define and designate the rules that exist in the system.

Without an explicit designation, the development team is unaware of which consistency rules exist and when should they be applied. The defensive programming approach leaves onto the programmers the responsibility of keeping the domain consistent, for every operation that they create. They are responsible for knowing every rule that might be involved in each operation. Moreover, acquiring this knowledge is especially difficult if these rules are not even a defined artifact in the development environment.

Thus, I argue that the separation of the consistency rules from the rest of the application's behavior is very important. The consistency predicates are an alternative to most cases of defensive programming, and remove several responsibilities from the programmer. They provide a distinct definition of each consistency rule, and allow the programmer to precisely designate each rule with a semantic meaningful name. The programmer is encouraged to place each rule inside the domain class that is responsible for

enforcing it.

The framework can automatically enforce these rules, and the programmers no longer need to check the state of the application on the beginning of each operation. Therefore, by introducing such a centralized implementation of the domain consistency, the developer can avoid using defensive programming in most cases.

However, there are some cases where the use of defensive programming is still encouraged. Indeed, the consistency predicates do not attempt to completely replace all traces of defensive programming. The goal of introducing these predicates is simply to use a mechanism that will thoroughly and automatically enforce the domain consistency.

One of the main downsides of totally replacing defensive programming with consistency predicates is that error detection would become lazy. A write transaction verifies the predicates involved at the end of all operations; just before the commit phase. Only at the end of such a transaction will it detect any potential errors and report them as feedback to the user. This detection is needed, for instance, to remind the user that a required field was not correctly provided on some web interface. It is generally preferred to have eager error-detection; to detect errors as soon as possible. Thus, the use of defensive programming is still recommended, for instance, in a presentation layer.

However, even in the domain layer, the developer cannot avoid several hand-written defensive checks. The predicates have a no-parameter signature and its in their nature to represent a restriction only on the data contained in domain objects. So, they are incapable of checking whatever parameters were passed to regular domain methods. This verification is necessary, for example, to check that a parameter is not null, in order to avoid NullPointerExceptions later. At the beginning of such methods, programmers still need to check the received parameters for whatever validation they need. This work does not address this issue, neither does it introduce any method pre-conditions or other contract-driven development artifacts.

2.3.2 Which Consistency Predicates to Verify?

In the previous example of Figure 2.7, the intent of the programmer is to prevent the total balance of the client to become negative. Instead of implementing this rule with defensive programming, I argue that the programmer should use a consistency predicate. An implementation of this predicate is illustrated in Figure 2.8.

```
public class Client extends Client_Base {
  // (...)
  @ConsistencyPredicate
  public boolean checkTotalBalancePositive() {
    return (getTotalBalance() >= 0);
  }
}
```

Figure 2.8: A consistency predicate that uses an auxiliary method to read all the Accounts of a Client, and to check that his total balance is positive. The implementation of the getTotalBalance() auxiliary method was shown in Figure 2.5.

The Client enforces a rule called checkTotalBalancePositive(). It is important to note that this consistency predicate uses the auxiliary method getTotalBalance(), which was illustrated back in Figure 2.5. The getTotalBalance() method reads all of the client's accounts to calculate the total balance. So, the consistency predicate of the Client will depend on the data of the Accounts.

An interesting characteristic of the consistency predicates is that the consistency of one domain object can indeed depend on the state of other related domain objects. A consequence of this property is that when a write transaction changes an entity (e.g., an Account), it may need to verify the consistency of a different entity (e.g., the Client). However, this transaction only contains an account in its WriteSet, and is not aware that there is a client that depends on it.

Recall that the main goal of the consistency predicates is to keep the data consistent, as it changes over time. In practice, the consistency predicates should be checked on each state transition. Only then can the data have changed, and possibly have become inconsistent.

In the Fénix Framework, state transitions are represented by the commit of a write transaction. This transaction's WriteSet contains the VBoxes that represent all of the changes performed on the data. These VBoxes are placed within domain objects, to contain the domain data of the slots and relations defined in DML. So, whenever a transaction attempts to commit changes to a set of VBoxes, the framework knows exactly which objects are affected by these changes.

Before the commit phase, the write transactions perform a consistency check phase, during which they execute consistency predicates. If all the consistency predicates that they execute return true, then the changes they have performed keep the data consistent. In this case, the transaction proceeds to the commit phase.

But which predicates should each write transaction verify? Verifying all the predicates for every single existing domain object is a way to guarantee that the domain data is still consistent. Clearly, however, this approach does not scale with the amount of domain data of the application.

Alternatively, the transaction could verify only the predicates of the domain objects that it modified directly. Still, this approach would not assure that the domain remains consistent, because an object's consistency can depend on the data of other objects. In the previous example of Figure 2.8, a transaction modifies an Account, but must verify a predicate of the Client

Another alternative is to check all objects that have a direct relation with the one being modified. This naive alternative would work for the Client-Account case, but still presents a few drawbacks. This alternative would fail when two objects that are not directly related depend on each other. Also, in a rich domain model, the Account has potentially many direct relations to other objects, most of which may not define their consistency depending on the Account. This alternative would cause a large amount of false positives and a waste of computational resources for each write transaction that needs to commit.

A **false positive** happens when the framework evaluates a consistency predicate without need. This mishap may cause performance and scalability problems. In general, the framework should reduce the number of false positives whenever possible.

A **false negative**, however, would happen if the framework would **not** execute a predicate that needed to be executed. This mishap can cause the domain data to become inconsistent. So, to keep the framework's correctness, it is critically important to guarantee that there are no false negatives.

It should be clear by now that, without more information, it is not possible to determine exactly which consistency predicates should a write transaction verify. This complexity is due to the fact that the programmer is allowed to define the consistency of an object depending on the data of other objects. For an efficient verification of the consistency predicates, the Fénix Framework creates and maintains a dependency network, implemented with Dependence Records.

2.3.3 Dependence Records

During the consistency check phase of a write transaction, a special nested transaction executes the predicates and collects the ReadSet of that execution. The VBoxes collected in that ReadSet are the values that were read during the execution of the predicate. The execution of that predicate for one domain object depends on those VBoxes. In other words, the object's consistency depends on these values.

The DependenceRecord stores these data dependencies; i.e., the VBoxes read on the last execution of a predicate for one object instance (the dependent), as illustrated in Figure 2.9. In other words, a DependenceRecord indicates what happened the last time that the framework executed the predicate for this object. Each time the data changes and a transaction needs to reexecute the predicate, it will discard and rebuild the DependenceRecord, thereby keeping it up-to-date.

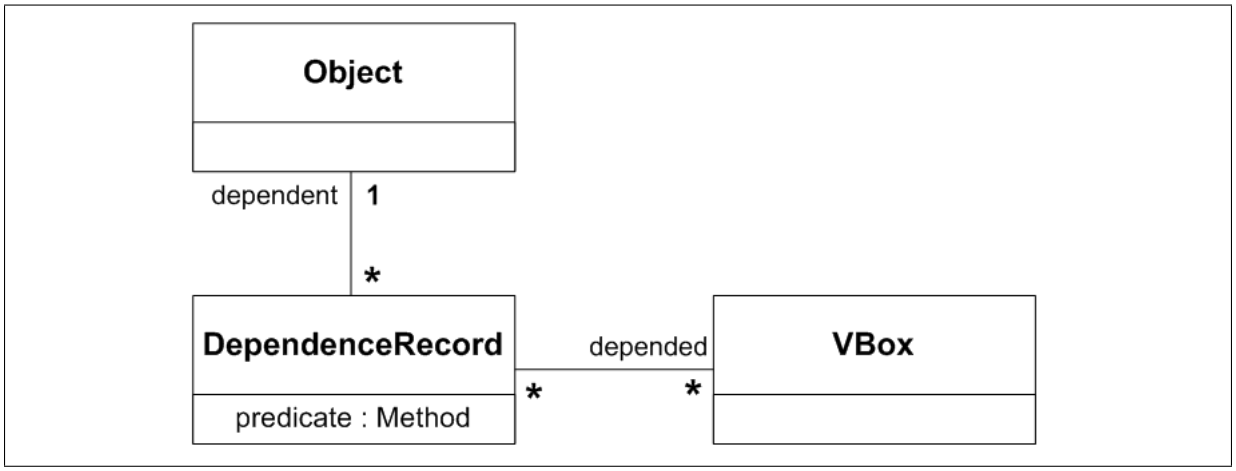

Figure 2.9: Original DependenceRecord model and the relation to several VBoxes.

Note that this model is part of the Fénix Framework's implementation and is not implemented in DML. The DependenceRecord and the VBox are plain old Java classes.

In the banking example, the DependenceRecords hold the information that the client's consistency depends on the accounts, as seen in Figure 2.10. When a transaction changes an account, it uses the DependenceRecords to know that there is another entity (the client) whose consistency must be rechecked.

Imagine a transaction attempting to modify Account C from Figure 2.10 for Sophie. With the DependenceRecords, the transaction knows that the modified Account C's state has an influence in the consistency of Client Sophie. With this knowledge, it can recheck the predicate checkTotalBalancePositive() (illustrated in Figure 2.8).

So, the DependenceRecord is the construct that keeps each dependent domain object consistent. It is crucial to keep the DependenceRecords always available and up-to-date, which is the only way to keep the domain consistent, and never add inconsistencies.

By keeping the DependenceRecords updated, the framework has access to a set of VBoxes. All of these VBoxes determine the predicate's result, and they are the only pieces of data that may have an influence on the predicate's execution flow. A value stored inside a VBox can influence the execution flow, for example, when it is compared inside an if condition.

With this information, the predicate is not concerned about transactions that write to other VBoxes. It remains silent, as long as no write transaction changes any of the VBoxes in its DependenceRecord. In

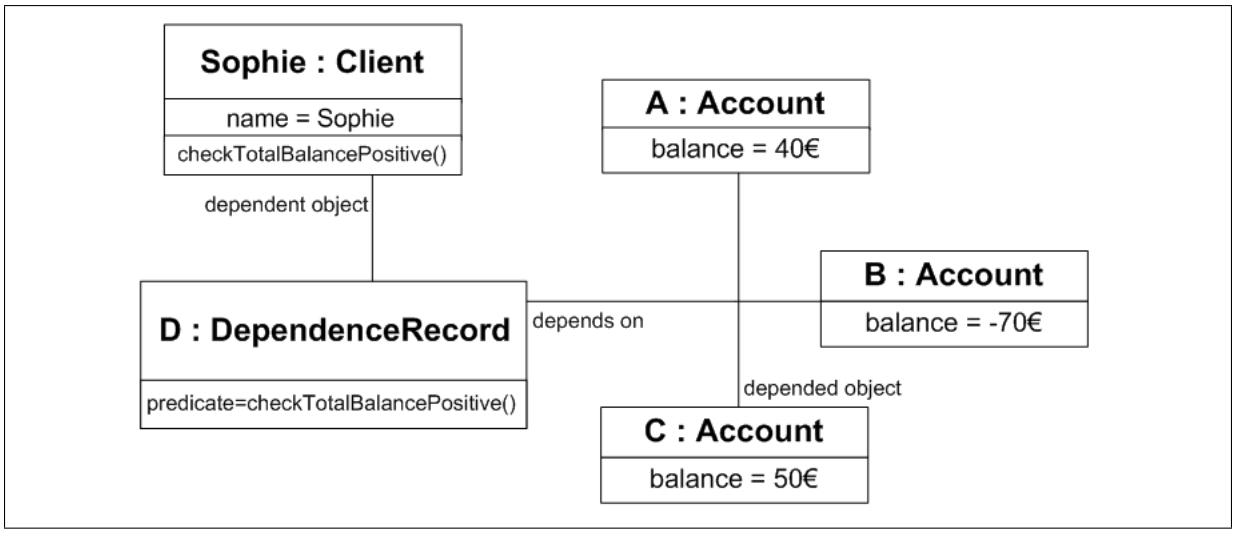

Figure 2.10: DependenceRecord of a Client, and the balance of the Accounts that it depends on. Note that the client is consistent because her total balance is 20€.

this case, no transaction has changed the result of this predicate. Moreover, no transaction has changed the execution flow either.

So, if no transaction changed any of these VBoxes, it is pointless to reevaluate the predicate, because the result will be the same. The reevaluation is not needed because the only changeable input of the consistency predicate are the VBoxes it reads. This theory assumes that the implementation of the predicates is deterministic, and helps in keeping a low amount of **false positives**.

Only the changes made to previously read VBoxes will trigger the reexecution of the predicate. This reexecution has two goals. First, it checks if a transaction has inserted an inconsistency, in which case the transaction aborts. Second, it updates the DependenceRecords for this predicate, whose execution flow may have changed. The goal of keeping the DependenceRecords updated helps in preventing **false negatives**.

To better illustrate the need to update the DependenceRecords, consider the case of the consistency predicate in the banking domain illustrated back in Figure 2.6 The class Account has a consistency predicate to verify that closed accounts have no money. This verification is done by checking that an account marked as closed has a balance that is equal to zero. In this case, assume that there are only two VBoxes involved: one containing a boolean closed, and another containing an integer balance.

If an account is closed, the execution of this predicate will read the closed (which should be true) and balance VBoxes. The DependenceRecord will then register them as depended. When a transaction attempts to change the boolean closed or the balance, at first, it knows only that it has written to either of these VBoxes. Yet, the relation between them and the DependenceRecord is bidirectional. So, the transaction can obtain the DependenceRecord that depends on these VBoxes, and recheck the predicate for this account.

If an account is open, however, the short-circuit evaluation of Java (Figure 2.6) will make this predicate read only the boolean closed (which should be false). Thus, the DependenceRecord will not register the balance VBox as depended. Therefore, if the balance of an open account changes, the framework does not need to recheck this predicate, in favor of a better system performance with less **false positives**.

Now, if a transaction attempts to close this account, it will zero out the balance and change the boolean closed to true. The JVSTM will then reevaluate this predicate to check that the balance is indeed zero. It is important to notice that this reexecution is also needed to update the DependenceRecord,

which must now register the balance VBox as depended. Afterwards, further changes to the balance will recheck this predicate and avoid increasing the balance above zero.

In this example, failing to update the DependenceRecord would not inform the framework that it has to constrain additional changes to the balance. A transaction would write to the balance, which is not depended by any record, because the DependenceRecord is outdated and only depends on the boolean closed. This situation represents a **false negative**.

Put differently, having outdated DependenceRecords would mean that the framework could become obliviously unaware of which predicates to reexecute for each VBox. Then, write transactions would not reevaluate the correct set of predicates, and could add new inconsistencies. I emphasize that it is imperative to guarantee that these DependenceRecords never get outdated.

In summary, the motivation for this work is based on the need to have updated DependenceRecords available at all times. This availability is essential to guarantee that there are **no false negatives**, so that new inconsistencies are never created. The next chapter will provide further insight into the limitations of this model, when applied to a typical enterprise application setting.

Chapter 3

Limitations of the Fénix Framework

This chapter gives a more detailed description of the major limitations of the Fénix Framework that will be addressed in the remainder of the document. It explains why some of the typical aspects of development inside enterprise applications are problematic for the model of the consistency predicates and the DependenceRecords presented so far. Section 3.1 describes the problems induced by the aspect of data persistence. Section 3.2 describes how incremental development is not yet fully supported by the consistency predicates. Section 3.3 summarizes the context involved, and provides the objectives of this work.

3.1 Data Persistence

The first problem with the implementation of the consistency predicates in the Fénix Framework is that DependenceRecords are transient, unlike the application's data. The consistency predicates presented where originally introduced with the JVSTM, which was not designed to deal with the persistence of the application data. The JVSTM was designed for applications whose data life-cycle begins when they start, and ends when they stop. Their data is created in memory, under the surveillance of the JVSTM, and ceases to exist when the applications stop.

In an application that uses only the JVSTM, each of these objects exists only in memory. Likewise, the JVSTM needs to store the DependenceRecords only in memory. In the banking example, as long as a client exists, it must be kept consistent, so the DependenceRecords associated with it must also exist. After it creates a client in memory, the JVSTM executes its consistency predicate for the first time, and builds the DependenceRecord. Eventually, someone decides to delete the client and remove all references to it, so that it can be garbage-collected afterwards. At that point, the JVSTM can also delete the DependenceRecord.

In this work, however, I am concerned with the use of the JVSTM in the Fénix Framework, which automatically persists all the slots and relations declared in DML. It creates the domain objects of the application in memory, but also persists them (usually in a database). So, after a transaction creates one client and a few accounts, it executes the client's predicate for the first time, and builds the DependenceRecord. Afterwards, the application runtime can be terminated without losing any of its domain objects.

However, the framework does not originally persist the DependenceRecords. Later, when the application restarts, no records exist. It will no longer need to create new domain objects; rather, it will fetch existing objects from the database. Since it fetches the client and does not create it, no DependenceRecord is created either. Without the DependenceRecord, there's no way to know that the client's consistency

depends on the accounts. Thus, changes made to an account balance may obliviously make the client inconsistent.

Because this work intends to guarantee that no operation adds new inconsistencies, the application needs to check the effects of all write transactions. Values changed in write transactions must be traced back to the DependenceRecords depending on them. Then, the framework rechecks the predicates of these DependenceRecords to keep the involved objects always consistent.

However, to make this workflow possible, the framework needs the DependenceRecords available and updated as soon as the application starts. But rebuilding all the DependenceRecords from scratch after every restart is very time-consuming and presents scalability problems. Imagine a class Account that has 100 existing objects, and defines 5 consistency predicates. The framework would check each predicate on each object for a total of 500 executions, and would need 500 DependenceRecords to keep those objects consistent.

At the moment of this writing, instead of persisting the DependenceRecords, the framework limits the expressiveness of the consistency predicates. Currently, a predicate can refer only to the state of its own object. Thus, the developer is unable to define the rules of an object that depend on the data of other related objects. This approach is a work-around that completely removes the need for DependenceRecords.

I argue that, in a rich domain model, it is very common that the consistency of one object depends on the data of other objects. Thus, the goal of this work is to give back to the programmer the ability to define such consistency rules. To do so, DependenceRecords should be persisted, along with the rest of the application's data.

The goal is to keep the DependenceRecords accessible and updated at all times. Their accessibility assures that changes on a certain object maintains the consistency of any other objects that might depend on it. So, the first step to keep consistent an application that persists its domain objects, is to persist the DependenceRecords as well.

With the DependenceRecords persisted, the application runtime may fetch an account from the database and change it freely. The framework will also fetch the depending DependenceRecord and recheck the consistency of the client.

3.2 Incremental Development

The decision to persist the DependenceRecords does, however, involve a subtle problem. The DependenceRecords store information about the execution of the consistency predicates, which are defined in the code of domain classes. However, during large-scale incremental development, programmers may change this code between every two deploys of the application. The records assume that the only thing that can change is the application's data. But this statement is not true when the application undergoes incremental development.

Incremental development is an iterative software engineering technique, commonly used to develop large scale enterprise applications. The programming team is encouraged to produce several versions of the application over time. Typically, the application is also incrementally delivered [Graham, 1989]. At each deployment phase, the customer is provided with a usable version of the application that will be upgraded later.

A direct consequence of employing incremental development is that the application will start to be used in a production-like environment much earlier than any other development technique. In turn, the application will start producing and storing domain objects midway through the development process. At the time of their creation, these objects are consistent according to the definition of the consistency

rules that had been introduced until then.

However, as newer versions of the application are deployed, objects are created and handled in different ways. Also, the development team can change the existing consistency rules. These rules are defined in classes that may already have existing objects, which were stored in the previous version of the application. If the changed rules were applied to the existing objects, most likely, the rules would not consider the objects consistent any longer.

Traditionally, developers are further responsible for correcting the existing data before the new version is delivered to the customer. But this task is generally difficult and time-consuming in any relatively large and complex domain. Thus, it would be of value if the framework that deals with the application's consistency could deal with these changes and still be usable by the customer, in spite of existing inconsistencies.

But regardless of whether or not the objects are indeed consistent, they were being kept consistent thanks to old DependenceRecords. Unfortunately, code changes might have made the stored DependenceRecords outdated. So, even if the developers manually correct the data within the domain objects, nothing stops write transactions from making those objects inconsistent again. Clearly, if any write transaction is to commit after a restart, the framework needs to be aware of the code changes, and update the DependenceRecords accordingly.

Most importantly, the development team can introduce brand new consistency rules inside a class with already existing objects. These new rules typically make sense in the model of that domain class. They add new quality standards to all of that class' objects.

Consider the introduction of a new consistency predicate checkTotalBalancePositive that was shown back in Figure 2.8. This predicate is inserted in the Client class, which already has an existing client named Sophie, as illustrated in Figure 3.1. Even though the checkTotalBalancePositive predicate is new, Client Sophie is already consistent; her total balance is 20€.

However, the dependency information of the Fénix Framework does not yet exist because this predicate is a new rule and the client already existed, it was not created. So, Transaction T can withdraw from Account B 50€, and obliviously make the client inconsistent. The transaction does not check the new predicate of the Client, because it would need the DependenceRecord to indicate that the Client depends on the Accounts.

Without the DependenceRecords, the framework cannot assure that it is keeping the objects consistent, because it does not know on what depends their consistency. So, a new rule introduced in a class with already existing objects will need a new DependenceRecord for each of those objects. These new DependenceRecords will keep consistent the existing objects that were already consistent, according to the new rule's implementation logic. To build these DependenceRecords, the framework must be aware of the introduction of new rules, and execute them for all existing objects of that class at startup. This approach will be described in Chapter 5.

Moreover, it may happen that some existing objects are already inconsistent at the moment that the new rule is created. The new DependenceRecords of already-inconsistent objects will prevent transactions from modifying those objects until they are corrected. A transaction that writes to one of these object will reexecute its new predicate. If that transaction did not correct the object, the predicate will return false because it was already inconsistent before.

So, if the data was previously inconsistent (and not corrected), the predicate will return false and the transaction will abort. If the transaction aborts, it will not perform several read/write operations on the data. The problem with transactions aborting due to already existing inconsistencies is that some part

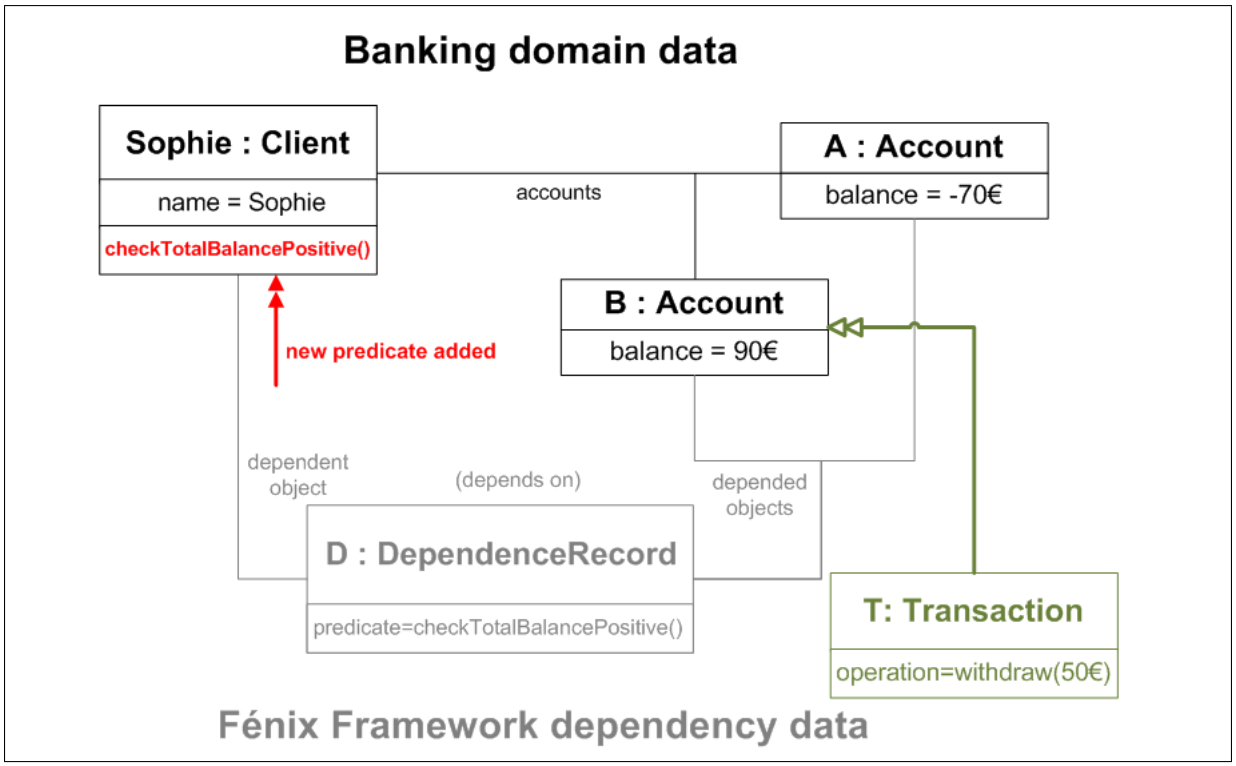

Figure 3.1: An already existing Client Sophie is consistent according to the new rule getTotalBalance-Positive(). Her total balance is 20€. However, because the rule is new, the DependenceRecord shown in gray does not yet exist. Therefore, Transaction T **will commit** and make the client inconsistent. Instead, Transaction T **should abort**.

of the application state may no longer progress in time.

This situation is illustrated in Figure 3.2. The checkTotalBalancePositive predicate is new, but unfortunately Client Natalia is already inconsistent; her total balance is -20€. In this example, assume that the framework is already aware of code changes, and that the DependenceRecord exists. Because the Client is already inconsistent, Transaction T will abort, and will be unable to change the Client's name to Christina. Note that the dependent object is also considered as a depended object. In this case, the client is also used by the predicate's execution, because it contains the list of existing accounts.

However, this transaction did not insert any inconsistencies. It simply attempted to perform an operation on some data that another event made inconsistent before. So, if an object is inconsistent, any operation attempting to manipulate this object will fail, until someone corrects the data. In turn, this may cause users to perceive the application as being faulty, as the application's liveness is compromised.

Therefore, to keep the application liveness as high as possible, the transactions should be able to deal with inconsistencies without halting the entire system. In other words, the transactions need to be tolerant to already existing inconsistencies that they did not create. This approach will be discussed in Chapter 6.

3.3 Main Challenges

In summary, the defensive programming approach is not adequate for implementing rules in a rich domain model. The consistency predicates provide an adequate implementation for consistency rules inside such a rich domain, because they allow the consistency of one object to depend on other objects. However, this approach should support the typical enterprise application setting, where the data is persisted and the application's code is incrementally developed. New versions of the code can change existing consistency

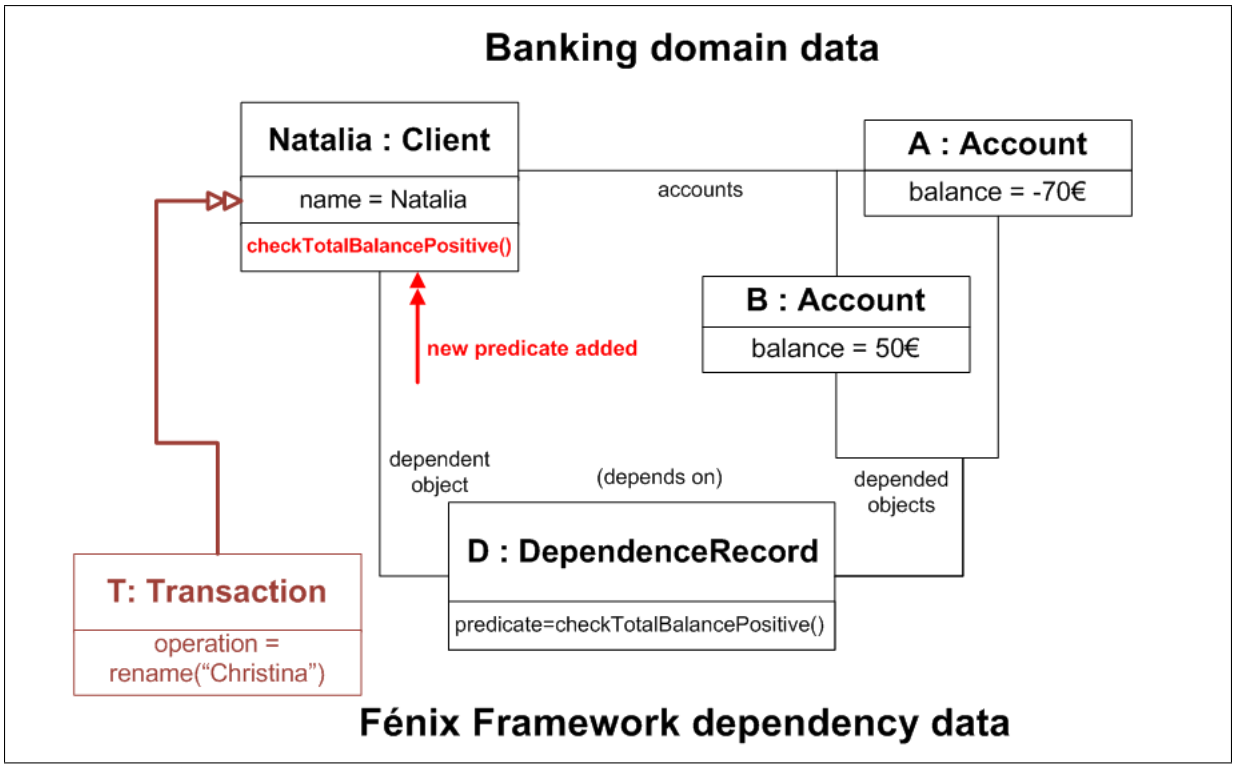

Figure 3.2: An already existing Client Natalia is inconsistent according to the new rule getTotalBalancePositive(). Her total balance is -20€. Unfortunately, Transaction T **will abort**, and will be unable to change her name. Instead, Transaction T **should commit**.

rules, and add new ones on the previously persisted objects.

The context of this work gathers the following conditions:

- The consistency of one object can depend on other objects.
- The application persists its objects.
- The application undergoes incremental development.

Under these conditions, it is crucial to provide solutions to the following two challenges:

1. How will the framework respond to changes made to the consistency rules?
2. How will it deal with already existing inconsistencies in the persisted objects?

Also, it will be important to prevent false negatives as much as possible, and to keep a low number of false positives.

Chapter 4

Related Work

This chapter presents the existing related work involving the definition of consistency in evolving applications. Section 4.1 addresses implementations that follow a Design By Contract approach. Section 4.2 covers several techniques that use Software Transactional Memory to satisfy the need for consistency. Section 4.3 discusses a few interesting implementations of Change Impact Analysis that deal with code changes.

4.1 Design By Contract

The Design by Contract approach is an object-oriented design technique [Meyer, 1988a,b] that specifies a behavior contract for each class. This technique is composed of three distinct design artifacts: pre-conditions, post-conditions and invariants. All of these artifacts follow Hoare's well-known axiomatic pattern [Hoare, 1969]. They are specification rules that are defined by boolean expressions that must be true at certain points in time.

Both pre- and post-conditions are always associated with a method. Pre-conditions must be true at the beginning of the method's execution. They indicate under what conditions it can be executed. Post-conditions must be true at the end of the method's execution. They indicate how it must have made progress after it has executed. Finally, invariants are associated with a class, which contains methods and state. An invariant indicates a condition about the state that the class must guarantee to be true at the beginning and at the end of each method.

As the name indicates, all these types of specification rules are just specification artifacts and, per se, have no effect on the semantics of the application. They serve only as documentation about the contract that each class satisfies. To support the checking of these rules at runtime, Meyer included an implementation in Eiffel [Meyer, 1992] that detects flaws in the code of the application. There have been more implementations based on this design for several other programming languages, including Java [Kramer, 1998, Karaorman et al., 1999, Leavens et al., 2000, Flanagan et al., 2002, Lackner et al., 2002]. Most implementations are either used to make runtime checks for debugging purposes [Cheon, 2003], or combined with formal development methods to prove the program's correctness automatically [Hoare, 1972, Leavens et al., 2003]. They serve the sole purpose of controlling the quality of the application code, and are usually disabled for production environment, after the testing or the proofs are complete. Hence, to provide a correct implementation of the methods, developers still have to resort to defensive programming.

At a first glance, the consistency predicates presented in Section 2.3, and invariants, in a Design by Contract approach, have the same theoretical basis. Both are composed of boolean expressions that

define specification rules, which are separated from the rest of the application logic. However, they have very different goals.

The purpose of the consistency predicate is to define rules that are relative to the application data, rather than to its code. A consistency predicate is a specific implementation artifact that indicates if the data is consistent, rather than if the code is bug-free. Moreover, it is meant to have an active effect on the application semantics during runtime in a production environment. A consistency rule exists to control the consistency of the data, as it changes over time.

Given their purpose, invariants do not address the problems described in Chapter 3: how to handle changes in the consistency rules and how to manage existing persistent data, which may have inconsistencies. Invariants are often called *data invariants* because they specify conditions about the data of their class. The problem is that this *data* is viewed as a direct result of the program computation alone. Given that the program is usually tested and proven to be correct, its output data will be consistent. However, *data invariants* are not flexible enough to deal with already existing data that is read from an external repository (such as a database), and which may have inconsistencies.

Also, if the program has a way to store its own consistent data for future use, it will be particularly difficult to introduce new data invariants or change old ones. The new invariants will establish new consistency rules that the data should follow, but the stored existing data was built according to older rules. The obvious solution is to manually correct the data before the application launches, which generally is an arduous task in any relatively large and complex domain.

4.2 Software Transactional Memory

In this dissertation, I am interested in the work that involves Software Transactional Memory, because STMs keep track of all the changes made to the application's data. They contain a well-known set of the existing memory locations and the operations that are performed, to be able to control concurrent accesses to the shared memory locations. Thus, they can observe the evolution of the application state, which is the ideal environment to define and to enforce the application's consistency rules.

The original idea behind Transactional Memory was to provide an efficient lock-free synchronization approach for highly concurrent systems [Herlihy and Moss, 1993] through hardware support. The goal was to keep the programmers from using locks, the traditional method to deal with concurrent accesses to shared memory. Managing the granularity of locks is complex and error-prone, their use interferes with the composition of operations, and they can lead to deadlocks [Fraser, 2004]. Transactions avoid these problems by allowing an all-or-none, atomic execution of a sequence of operations, with the added benefit of fault recovery.

Much work was, and still is being developed in the area of Hardware Transactional Memory (HTM) [Stone et al., 1993, Hammond et al., 2004, Moore et al., 2006]. Most of these proposals achieve atomicity in shared data access by improving processor cache coherence protocols, which usually yields considerable performance results. However, they require complex hardware support and most of the early work restricts the number of operations that can be performed inside a transaction. Only later in the development of HTMs did Ananian, Asanovic, Kuszmaul, Leiserson, and Lie [2005] propose to support unbounded transactions that can have an arbitrary size and duration.

This work applies to Software Transactional Memory (STM) [Shavit and Touitou, 1995], that uses a common hardware architecture to build a software basis to provide transactions. There have been several recent proposals for STM systems [Harris and Fraser, 2003, Harris et al., 2005, Saha et al., 2006, Adl-Tabatabai et al., 2006, Herlihy et al., 2006, Dragojevic et al., 2009, Bronson et al., 2010]. Most

modern STMs natively provide unbounded transactions [Herlihy et al., 2003] that can run any number of operations.

The following text will describe in greater detail two interesting approaches that use the mechanisms provided by software transactional memory to define the consistency of the application. One of these approaches is the JVSTM [Cachopo, 2007] that was described in Section 2.2. The other is Harris and Jones's STM Haskell.

In general, STMs were introduced to support non-blocking synchronization for multi-threaded applications. They provide transactions to isolate memory operations in atomic units of work. A transaction usually contains a Read Set and a Write Set to register its read and write operations, respectively.

STMs also provide mutable memory locations that can be read from or written to. Thus, they possess the set of the existing memory locations and are aware of the operations that the system performs. Thanks to this awareness, STMs can control concurrent accesses to the shared memory locations. They can overview the progress of the application's data during runtime.

However, few implementations of STMs have used this property to allow the definition of the data's consistency. At the moment of this writing, only the following two implementation of STM include some sort of invariants or consistency rules.

4.2.1 STM Haskell and Data Invariants

STM Haskell [Harris et al., 2005] introduces isolation and atomicity in Concurrent Haskell, a dynamic and purely lazy functional language. This STM defines a set of constructs (atomic, retry, orElse) that allow sequences of operations to be composable within transactions. A transaction performs a sequence of operations and contains a log that registers their reads and writes to memory locations, which are called transactional variables (TVars).

Once finished with its work, the transaction enters a final phase that is responsible for validation and commit. First, it validates its read-log to look for conflicts. If any TVar that was read has changed to a different value, then the transaction restarts because a conflict is at hand. Otherwise, it can proceed to the commit step. To commit, it makes its isolated writes visible to the rest of the application by recording them directly into the corresponding variables.

Later, after this STM was created, a new development was proposed [Harris and Jones, 2006] that introduced data invariants in STM Haskell. This approach is the only one similar to that of the consistency predicates that were originally proposed in the JVSTM, which will be discussed in the next section. The paper presents a new construct (check) to assert that a particular constraint is valid. The construct creates a dynamic invariant that is checked on definition and reads several TVars. Afterwards, when a transaction writes to any of these TVars, it must check the invariant again, before it attempts to commit.

The first evaluation of this invariant is performed on definition by a nested transaction. This evaluation is necessary to verify that the initial data is consistent. If the data is indeed consistent, the transaction commits and registers this invariant in a global invariant set available to any future transactions. This first evaluation is also needed to register this invariant's reads in the transaction log. This way, the STM can know which TVars does the invariant depend on.

The following evaluations of this invariant take into account these dependencies. Future transactions that write to any of these TVars have to respect the invariant. Their validation phase checks if their changes have broken the invariant, in which case they abort.

So far, for incrementally developed applications, it is easy to introduce new TVars, operations and constraints, if the existing data is consistent. However, if a parcel of data is inconsistent, a transaction that

attempts to introduce an invariant on this state will abort on the first evaluation. So, the transaction will discard the invariant, which will never be registered in the global invariant set. This approach is certainly not very code-preserving, and it might not always be desirable, because the data being inconsistent is a common situation.

Moreover, the check construct was designed to be a debugging tool, used during testing and disabled for production environment. It is a temporary, dynamic mechanism used to detect flaws in the code, rather than a permanent, static definition of the domain constraints. The invariants are dynamically created and garbage-collected when not used any longer. Also, even though check can declare temporary domain constraints, it fails to designate them; to give them a name and to make them visible to programmers.

Because Haskell is not an object-oriented language, the domain constraints are not defined inside a class of objects. A single invariant checks only a fixed set of TVars. It is not powerful enough to check all instances of a certain class or abstract data type.

4.2.2 JVSTM and Consistency Predicates

The JVSTM [Cachopo, 2007] is a Java-based system that introduces versions in the STM world. It is especially aimed at read-intensive applications, and succeeds at running read operations with low overheads in a concurrent environment. As described in Section 2.2, the JVSTM provides VBoxes, and transactions with ReadSets and WriteSets to track the operations they performed.

Most importantly, the JVSTM has originally proposed the consistency predicates to define consistency rules. Unlike in STM Haskell, the consistency predicates are a static part of the code. Programmers cannot introduce new predicates on runtime, because Java is not a dynamic language. Unlike any other implementation of invariants for Java, the consistency predicates allow the consistency of one object to rely on the data of other objects.

The consistency predicates were especially designed to be used in object-oriented programming (Java). A single predicate can define the consistency of all the objects of a certain class. It is a fixed part of the code that is never discarded or garbage-collected. It is not a formal development artifact that is used as a debugging tool, but rather a permanent part of the domain behavior. Also, by having access to the name of the predicate and the class in which it is defined, programmers should understand what does each predicate enforce.

Thus, in comparison to the alternatives presented, the JVSTM's consistency predicates are the approach that best declares and designates permanent consistency rules. It allows the programmers to provide a separate implementation of each rule, to give it a semantic meaningful name, and place it inside the class that is responsible for enforcing it. Afterwards, the development team can inquire which rules exist in each class, and what are their names.

Other than the JVSTM, none of the related work presented so far allows the consistency of one object to depend on other objects. None of the approaches presented in this chapter deal with persisted data that may already be inconsistent. Moreover, none of them propose solutions to deal with code changes that influence consistency rules. There is, however, one particular area of research that intends to deal with the impact of code changes.

4.3 Change Impact Analysis

Change Impact Analysis [Arnold and Bohner, 1993] is typically used to manage software change proposals. It determines the consequences of code modifications within parts of the application's implementation.

The main goal is to make an analysis of source code changes at a detailed level, and present it to the programmer.

Ryder and Tip [2001] have presented one of the first impact analysis approaches concerned with object-oriented programming, and with Java in particular. The article discusses how code changes in a Java program can have significant non-local effects in runtime, mostly due to subtyping and dynamic dispatch.

Ryder and Tip have later proposed Chianti [Ren et al., 2004]: an analysis and debugging tool designed for Java, and implemented for the widely used development environment Eclipse. More recently, Wloka, Ryder, and Tip [2009] have proposed a change-aware unit testing tool, especially designed for test-driven development using JUnit.

These impact analysis implementations rely on a series of tests. The tools trace each test's execution and build a call graph. Each call graph allows to determine exactly which parts of the code does each test's execution depends on. These tools can determine the influence of a given set of code changes on the test's execution. They allow to reduce the amount of time needed in running tests, by guaranteeing that the code changes did not affect some test. In fact, this is an approach that attempts to reduce the number of false positives in the execution of tests.

In particular, Chianti's implementation is similar to the code change detection techniques that will be presented in next chapter. It is concerned with the detection of new methods inside a class hierarchy. Because a public method at a subclass can override another at the superclass, Chianti must determine whether a new method influences the tests execution. It is also aware of changes made to a method's modifiers, such as the visibility.

These implementations of Change Impact Analysis are somewhat similar to the code detection techniques designed for the consistency predicates. However, the purpose of these tools is still very distinct from that of this work. The main goal of Change Impact Analysis is to provide the programmer with the analysis of source code changes. It shows the programmer which portions of code have broken which tests, and helps in the detection of bugs.

The goal of the code detection implemented for the consistency predicates is not to detect bugs or analyze code during development time. Rather, the code detection presented in this dissertation is meant to keep the data consistent at runtime.

Chapter 5

Supporting Changes to the Consistency Predicates

"Change should always be balanced by some degree of consistency." - Ron D. Burton

Both consistency and change are necessary. Unstable changes that are left unchecked could disrupt well established and useful quality standards of consistency. But consistency should never be kept in such an intransigent way that it would completely obstruct change.

During incremental development, there is a constant need to incorporate changes. Changing the structure, the behavior, and the rules of the application is often the only way to keep up with inevitable requirement changes. No rule is universal, and although change is necessary, modifying the rules midway through the development process can have unexpected effects on the application's data and behavior. The development team will frequently change the code of the application, whose execution frequently changes the data.

After analyzing a certain part of the domain, a developer can decide to add a new consistency rule to a certain domain class. This new rule typically makes sense in the business model that the class deals with. It adds a new quality standard to all objects of that type.

Traditionally, a developer must correct all old existing objects of that type before introducing the new consistency rule. This correction is a data-related operation that can be extremely complex, time-consuming and irregular. However, in practice, a software developer is usually very busy implementing new features and correcting bugs. From my experience, the complexity of some data corrections usually keeps the developer from inserting the consistency rule in the first place. This situation is unprofitable.

I argue that the developer should always be able to add quality to the code of the application, regardless of the quality of its data. For each new consistency predicate, the framework alone should be able to detect already existing inconsistencies in the data. The framework can even provide inconsistency-tolerant transactions, as Chapter 6 will show. But regardless of the behavior of the transactions, the framework should trace inconsistencies and make them available for the developer. This way, the developer can introduce a new predicate in the code now, and obtain the inconsistencies later to fix them.

Still, the framework must first detect several types of code changes before it can update the DependenceRecords and detect inconsistencies. These code changes may implement a different definition of the consistency predicates in the domain logic. They may introduce new predicates to implement new rules, or remove old ones that no longer make sense. They may also, simply, change the body of a method that

implements a predicate, whose business logic has changed.

5.1 Classification of Code Changes

New predicates that are introduced in classes with already existing objects will need a new DependenceRecord for each of those objects. These new DependenceRecords will keep consistent the existing objects that were already consistent, according to the new predicate's implementation logic. The DependenceRecords of objects that were inconsistent should keep information about this inconsistency with an easy access.

Old predicates that are removed may leave behind old DependenceRecords that were persisted previously. These old DependenceRecords are useless because they no longer enforce any consistency rule. They should also be deleted, regardless of whether their objects were consistent or not.

Also, predicates whose implementation (or signature) is changed may have existing DependenceRecords that are now outdated. These DependenceRecords kept consistency information about existing objects, according to an old implementation that is no longer used. As this work has shown before, the system requires all DependenceRecords to be updated before starting any write operation.

To fulfill this requirement, the framework first needs to be aware of what code changes happened. Once it starts, it will need to have a representation for the existing artifacts of code to store their contents. After it is restarted, it can compare the previous stored representation to the new code of the application, and thus obtain a set of changes. Section 5.2 shall present the new domain model with the representation of the code that allows the detection of code changes. Section 5.3 shall discuss the implementation that responds to the changes after they are detected.

But before moving on to the architecture and implementation, it would be of value to enumerate the possible code changes that a programmer can perform. This section is concerned with the code changes that are related to the consistency predicates and their execution flow. After classifying all the possible changes it will be easier in the following sections to understand how to detect and respond to each change.

5.1.1 Method Modifiers

Most of these changes will involve predicate methods in domain classes. These methods are annotated with @ConsistencyPredicate; they receive no arguments and return a boolean. Each predicate may be **private, protected, or public**, and it may or not be **final**. Package visibility[1] is not supported for the consistency predicates.

Most of the other method modifiers defined in the Java specification do not have an important influence on the consistency predicates. Those that will be left out of the central discussion are the abstract, strictfp, static, native and synchronized modifiers.

Abstract consistency predicates are ignored by the framework because they contain no implementation to verify a specific rule. They can be used at an abstract superclass to force programmers to implement the predicate at the subclasses later. Only at the subclasses can the framework obtain the actual implementation of these predicates, and verify them. However, I believe the incorrect use of abstract predicates might contribute with confusion to the development environment. I generally recommend to implement the predicates directly in the abstract superclass, even if they invoke abstract methods. Although this superclass cannot have any objects, the framework will ensure that objects of the subclasses will check that implemented predicate.

[1]Package visibility is the default visibility for methods that specify no explicit visibility modifier in Java.

Strictfp consistency predicates are treated like regular predicates, although they implement strict floating-point calculations. In other words, a strictfp method truncates all intermediate floating-point calculation results to IEEE single or double precision. This situation is what occurred in earlier versions of the Java Virtual Machine. This modifier is generally used to make methods always have precisely the same floating-point behavior, no matter what platform they are running on. The strictfp modifier can change the execution flow of a predicate that, for example, compares the results of float calculations inside an if condition. Therefore, it is important for the framework to detect the insertion or removal of this modifier from an already existing predicate. As this work will describe later, the framework will detect signature changes in general as a code change in a predicate that must be rechecked.

Static methods are not executed on any object instance, and are limited to access only static data. However, the framework expects to verify, for all objects of a class, a predicate that accesses dynamic domain data. So, although the framework treats static predicates like regular predicates, I believe that it is unconventional to use static predicates. Still, the framework does not forbid the use of static predicates, and the developer is free to implement them.

Native methods allow the programmer to specify methods without implementation, so that their bodies can be filled later with platform-dependent code. The framework usually expects predicates to have a platform-independent and deterministic implementation that only accesses domain data. So, although the framework treats native predicates like regular predicates, I believe that it is also unconventional to use native predicates. Still, the framework does not forbid the use of native predicates, and the developer is free to implement them.

Synchronized methods force the JVM to use locks in their access, to ensure that only one thread at a time is executing synchronized methods on each object. The use of this modifier is discouraged because the Fénix Framework already implements an STM to deal with concurrency issues. The framework already provides atomic transactions and deals with the problem of concurrent access of multiple threads to shared data. The use of a lock in a synchronized predicate might cause performance issues and deadlocks. So, although the framework treats synchronized predicates like regular predicates, I argue that it is very odd to use synchronized predicates. Still, the developer is still free to implement them at his own risk.

5.1.2 Predicates in a Class Hierarchy

For the discussion below, the most important method modifiers are the final, and the visibility modifiers. It is important to distinguish between these modifiers, because predicates may be inserted in a domain class hierarchy. In a domain hierarchy, a single class can have one superclass, and many subclasses. Likewise, a predicate in a domain hierarchy can override a predicate at the superclass, and be overridden by many predicates at the subclasses. This situation only happens with public or protected predicates that are not final.

Private methods can neither override nor be overridden. Final methods can override, but cannot be overridden. Thus, it is redundant to have a private and final method, according to the Java specification.

Furthermore, a predicate will verify a consistency rule, at most, for its own class hierarchy, but never for other classes in other hierarchies. Also, a predicate that is not final can be overridden if it is public or protected. So, for the framework, there will be no semantic difference between a public predicate and a protected predicate. For simplicity purposes, the following discussion will refer to the only three kinds of consistency predicates as follows:

- **Private predicates** are those implemented with a **private** method, which can be final, or not.
- **Public predicates** are those implemented with a **public** or protected method that is **not final**.
- **Final predicates** are those implemented with a **public** or protected method that is **final**.

The following section provides a listing of the possible code changes that involve the consistency predicates. For each of these changes, it describes the pragmatic meanings of the change, and its influence on the rest of the domain class hierarchy. Note that this semantics is general, described without regard to the actual implementation of the methods. The implementation of a method contains the business logic that governs the application's domain. But the following pragmatic meanings will be based only on the semantics that Java provides for methods with certain modifiers in a class hierarchy.

Before proceeding to the possible code changes, it is important to discuss two details about the visibility of consistency predicates. According to the Java specification, a private method is only visible in its declaring class, and not in its subclasses. Nevertheless, the Fénix Framework will verify a private predicate at a superclass for objects of that class and all subclasses. The framework originally uses Java reflection to make all predicate methods accessible in order to invoke them.

This behavior follows a design choice that is based on the idea of a contract invariant at a supertype that should always be followed by all subtypes. Thus, a consistency rule at a superclass will always be verified at all subclasses, independently of the visibility of its method. So, even though a private predicate's method is only visible at its own class, it is executed also at the subclasses. The semantics shown in this work will respect this design choice.

Moreover, a package visibility method at a superclass can only be overridden at subclasses if they are on the same package of the superclass. In other words, the choice of which methods are overridden and where they are overridden, depends on the package of each class. Code changes to the package of a class would change the package visibility predicates that can be executed at the subclasses. Therefore, package visible consistency predicates are not supported.

5.1.3 Possible Code Changes

The first code change to consider is the introduction of a new predicate in a domain class. In general, this predicate represents a brand new consistency rule that objects of that class and all subclasses must follow from now on. This is the case of a new private predicate, which can neither override, nor be overridden. It is applied for all subclasses without exceptions, and allows no refinements.

Much like a private predicate, a new final predicate can also not be overridden, and always applies to all subclasses. However, the new final predicate can override a public predicate at a superclass. Therefore, the new final predicate can be a refinement of an existing public predicate at the superclass. The implementation logic of the predicate at the superclass should no longer be used from this class downwards. It should be replaced by the implementation of the new final predicate.

A new public predicate can also override another public predicate at a superclass. It can represent the refinement of an existing rule, that replaces the superclass' rule for objects of that class and subclasses. But a new public predicate can also be overridden by other (public or final) predicates at the subclasses. In this case, the new public predicate represents a new consistency rule for its class, but not necessarily for all subclasses. Objects of subclasses that already override this new public predicate will not be affected by its insertion.

Figure 5.1 illustrates the introduction of a new public predicate that already overrides and is overridden. This new public predicate is a refinement of an existing rule, which does not necessarily apply to all subclasses. It replaces the existing superclass' predicate for its class, and for the subclasses that do

not already override it. The second possible code change is the removal of an existing predicate from a

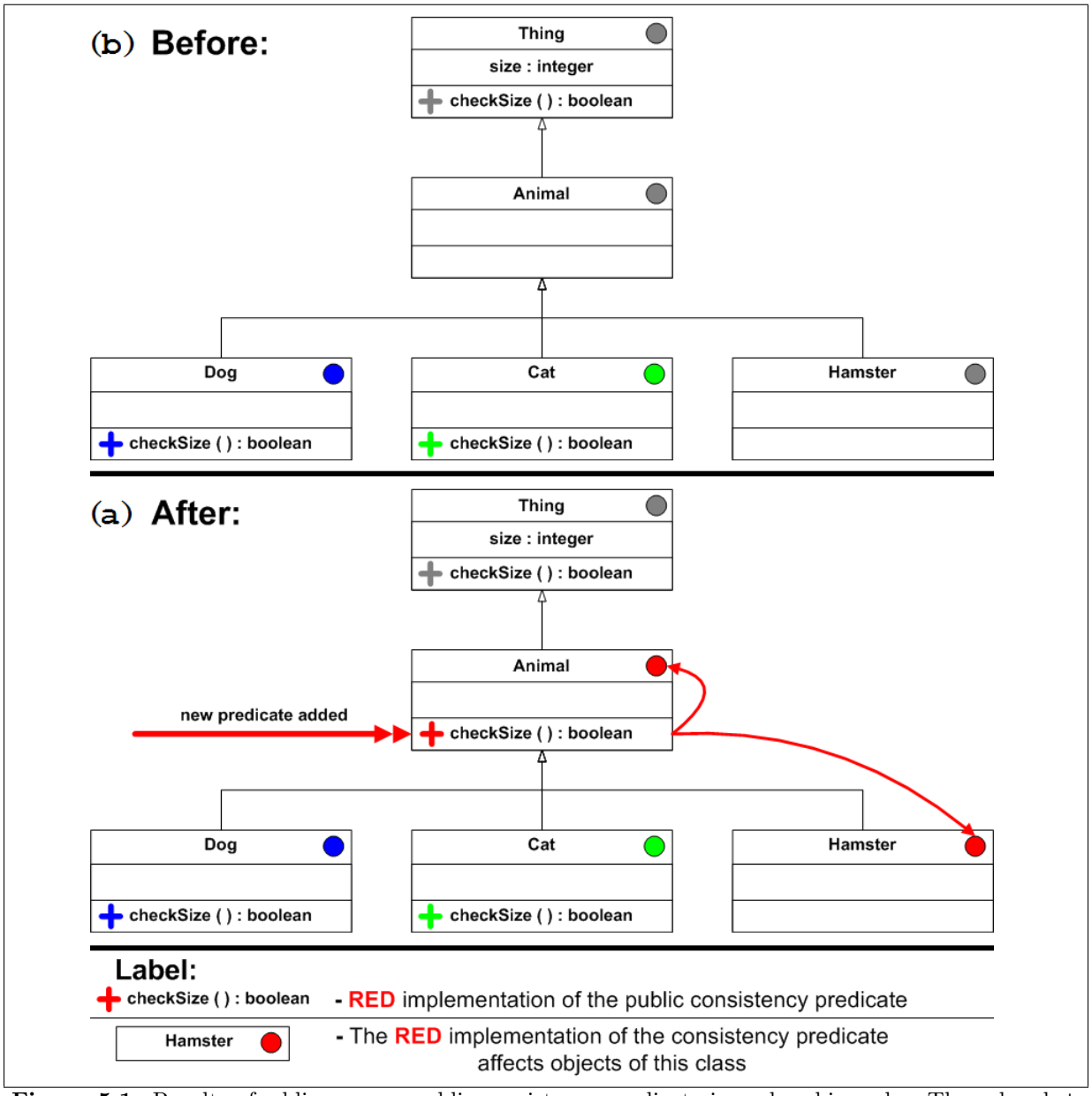

Figure 5.1: Results of adding a new public consistency predicate in a class hierarchy. The colored + signs indicate each predicate's color and its (public) visibility. The colored circles on each class indicate that instances of that class should be checked by the predicate with the same color.

domain class. In general, this predicate represented an old consistency rule that objects of that class and all subclasses do not need to follow anymore. This is the case of an old private predicate, which could neither override, nor have been overridden. All the subclasses used to follow this rule without exceptions, and could not have implemented refinements.

Much like a private predicate, an old final predicate also cannot have been overridden, and used to apply to all subclasses. However, the old final predicate could override a public predicate at a superclass. Therefore, the old final predicate could have been a refinement of an existing public predicate at the superclass. The implementation logic of this old final predicate should no longer be used from this class downwards. It should be replaced by the implementation of the predicate at the superclass.

An old public predicate could also override another public predicate at a superclass. It could have represented the refinement of an existing rule that should be replaced by the superclass' rule for objects of

that class and subclasses. An old public predicate can also have been overridden by other (public or final) predicates at the subclasses. In this case, the old public predicate used to represent an old consistency rule for its class, but that did not necessarily apply to all subclasses. Objects of subclasses that used to override this old public predicate will not be affected by its removal.

Figure 5.2 illustrates the removal of an old public predicate that used to override and be overridden. This old public predicate was a refinement of an existing rule, that did not necessarily apply to all subclasses. The existing superclass' predicate replaces the old predicate for its class, and for the subclasses that did not already override it.

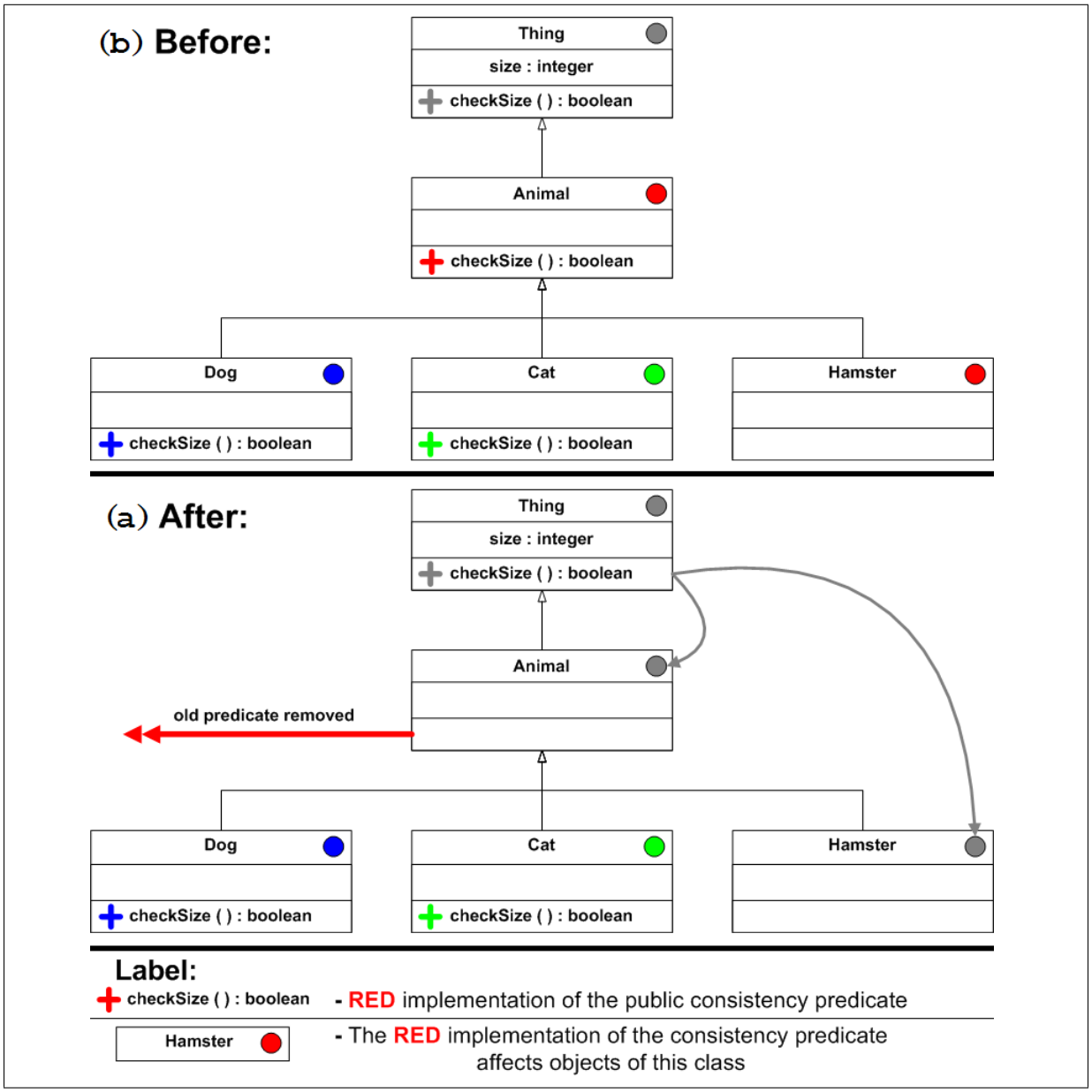

Figure 5.2: Results of removing an old public consistency predicate from a class hierarchy.

The third possible code change is the change of an existing method's signature. The predicate whose signature is changed can have a different execution flow (e.g. with a strictfp modifier). Even if it does not, the programmer may have removed the @ConsistencyPredicate annotation. He may also have modified the visibility or the name of the predicate.

Consider, for instance, a public predicate at a superclass that is overridden at the subclasses, as illustrated in Figure 5.3 b. The implementation of this predicate does not affect the subclasses that

override it. However, if the predicate's name is changed, or if its visibility is changed to private, it will no longer be overridden at the subclasses. After this change, it should be considered as a new predicate to verify the entire class hierarchy.

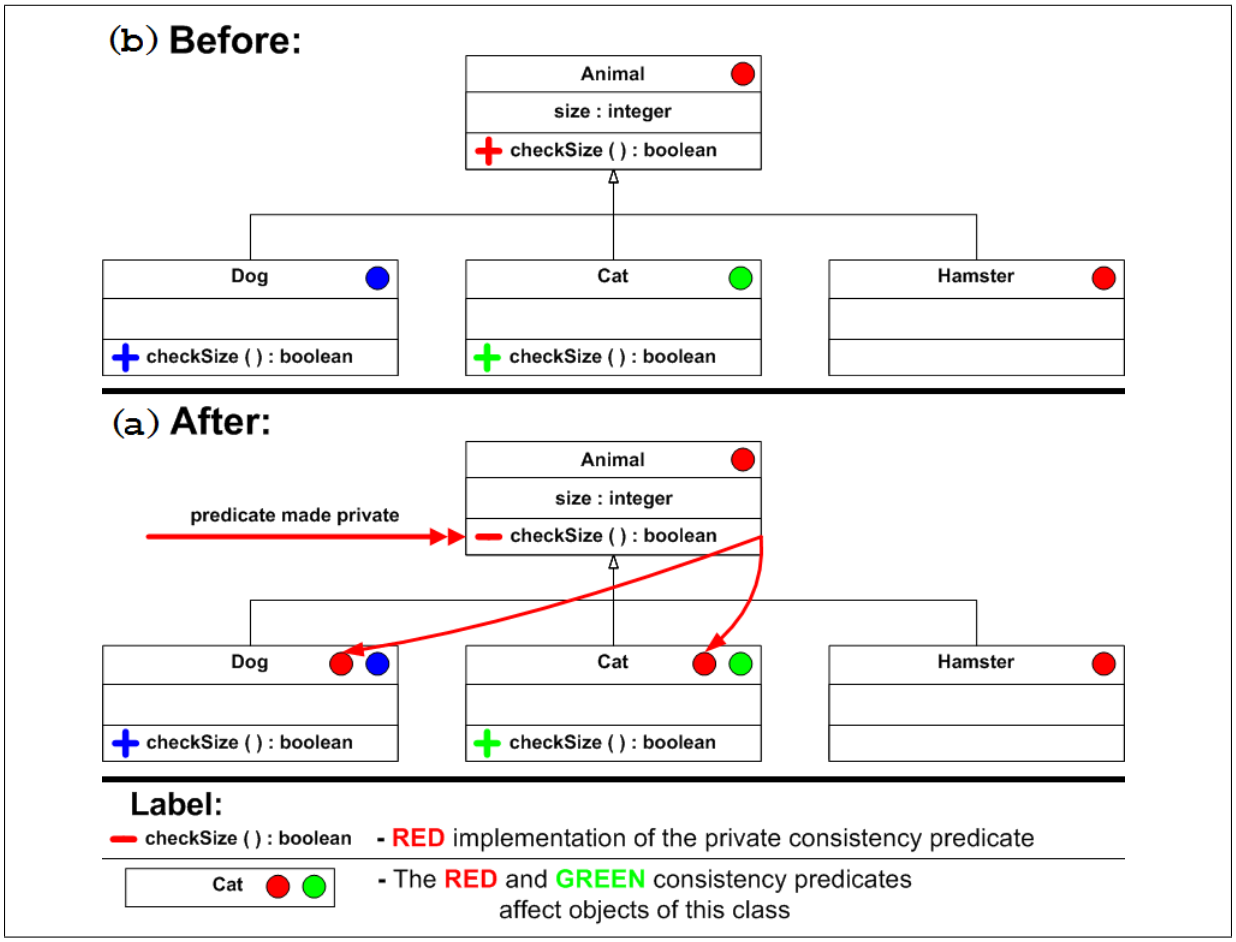

Figure 5.3: Results of changing an existing consistency predicate to private. Because the predicate at the class Animal is private, it is not overridden at the subclasses.

The fourth and final code change is the change of a method's body. The predicate whose implementation is changed will most likely have a different execution flow. The framework should also consider this predicate as a new predicate, and recheck it for the classes that it affects.

It may also happen that a certain predicate invokes an auxiliary method whose implementation is changed. Also, the programmer can introduce a new regular method at a certain class, that overrides an auxiliary method at a superclass. He may even remove a regular method from a certain class, that used to override an auxiliary method at a superclass. All of these changes should be taken into account, even those involving regular (auxiliary) methods, because they can influence the execution flow of the consistency predicates.

This final case concludes the list of code changes that are allowed, and that the framework should expect and react to:

- Inserting a new predicate (public, final, or private)
- Removing an old predicate (public, final, or private)
- Changing the signature of an existing predicate method
- Changing the implementation of an existing predicate method (or an auxiliary method)

There is still one code change that was left out of this discussion, and that the framework does not support. A programmer can insert a new public method with no @ConsistencyPredicate annotation at a subclass, that overrides a public predicate at a superclass. In other words, the programmer introduced a regular method to override a consistency predicate.

There are two possible interpretations for this situation. In a first case, the framework can assume that the developer simply forgot to place the @ConsistencyPredicate annotation. Thus, the framework could provide a warning to alert him to this situation, and treat the new method as a new consistency predicate.

This case makes a blind guess about the developer's intention. In terms of source code, if this situation is erroneous, then it should be detected at compile time and prevented from being used in production. A simple warning can easily go unnoticed, and the framework's decision to treat the method as a predicate may not be welcome.

In a second case, the framework can assume that the developer intentionally omitted the @ConsistencyPredicate annotation. He probably intended to override the method from the superclass, and to declare that this method is no longer a predicate in the subclass. Thus, the intended semantics could be that the consistency rule from the superclass is only verified for objects of the superclass alone. The framework could allow objects from the subclass downwards to no longer follow that rule.

In this case, the developer can allow a collection of objects of a certain type to not follow a rule. However, in terms of source code, I believe this is the incorrect way to provide an exception to a rule. The correct way to do so, is to reimplement the predicate at the subclass, with the @ConsistencyPredicate annotation, and make it always return true. This predicate will then replace the rule from the superclass, and will not verify anything.

Still, neither of these two semantics is appropriate, because both make uncertain assumptions about the intent of the programmer. In fact, in this situation, it is difficult to infer the intent of the programmer because the code is ambiguous. Thus, the framework does not allow to override a consistency predicate with a regular method without the @ConsistencyPredicate annotation. It should detect this situation and present detailed errors during compile-time. This document contains, in Appendix A, a summarized, programmer-friendly list of recommendations on how to properly implement a consistency predicate.

5.2 Detection of Code Changes

This section describes the changes made to the Fénix Framework so that it can fully support the detection of changes to the consistency predicates. The domain will be composed of several new entities that will store important information about the predicates and the rest of the domain. These entities will represent the several code artifacts of the target application. It will be important to persist these entities (which include the DependenceRecord), together with the rest of the application's data.

With these entities persisted, when the application restarts, it will have access to information about the previous version of its code. By comparing this information to the current state of the code (which can be done using Java reflection), the framework can detect changes made to the code. The easiest way to create these persistent entities is to specify them as Fénix Framework domain classes in its own DML file.

The Fénix Framework processes its own DML file, the fenix-framework.dml, like any other DML file of any possible domain model. Just like any DML file, the code generator creates Java classes of the framework's own domain from entities and relations specified in this DML file. These domain classes can be used as a domain-oriented way to enhance the Fénix Framework itself and provide more functionalities

to the developer.

Section 5.2.1 explains the meta objects that will be a general representation of domain objects. Section 5.2.2 describes the persistence of the DependenceRecord. Section 5.2.3 presents the metaclass hierarchy that will represent the target application's domain class hierarchy. Section 5.2.4 introduces the representation of the existing, and known consistency predicates. Section 5.2.5 shows new extra information to support overriding consistency predicates in a class hierarchy. Finally, Section 5.2.6 describes how the system can use all this persisted information to trace already existing inconsistencies.

5.2.1 Persistent Meta Object

Recall from Section 2.3.3 that the framework needs the DependenceRecords accessible and updated at all times. The DependenceRecords assure that their domain objects remain consistent. The first step to keep consistent an application that persists its domain objects, is to persist the DependenceRecords as well.

The simplest way to persist the DependenceRecords along with the application's domain objects, is to declare them as domain objects. In the bank example, the clients and the accounts are domain objects that belong to the banking application's domain model. The DependenceRecord, however, will be declared as a domain object that belongs to the Fénix Framework's domain model.

The original model of the DependenceRecord was presented back in Figure 2.9. In the banking example, the DependenceRecord needs to store the client object on which the predicate was invoked, and that is being kept consistent. Furthermore, it also needs to store the several accounts that were used during the execution of the predicate for that client. So, the framework will need to declare two different relations to domain objects: one for the dependent object, and another for the depended objects.

However, the framework does not yet provide an easy way to create a general bidirectional relation to a domain object of any possible type. Before the DependenceRecord can link to a domain object, the framework needs a generic way to represent a domain object of any type as a DML entity. So, the framework's domain first needs to contain the class PersistentDomainMetaObject, shown in Figure 5.4.

```
class PersistentDomainMetaObject {
}
```

Figure 5.4: Declaration of the PersistentDomainMetaObject.

The PersistentDomainMetaObject allows the framework to reify any possible domain object. Each PersistentDomainMetaObject will be directly associated with exactly one domain object. To create this association, the domain will contain a domain relation to the AbstractDomainObject, which is the top superclass of all domain objects. This relation is a way to create a general link to any possible type of domain object that may exist. It is shown in Figure 5.5.

```
relation PersistentDomainMetaObjectAbstractDomainObject {
  AbstractDomainObject playsRole domainObject;
  PersistentDomainMetaObject playsRole;
}
```

Figure 5.5: Relation between the PersistentDomainMetaObject and the AbstractDomainObject.

Note that, for now, this relation is unidirectional because the PersistentDomainMetaObject entity does not have a playsRole value. This DML relation can only be unidirectional because the AbstractDomainObject is an external entity, which is not declared in any DML file. The framework's code generator cannot

create the code for the AbstractDomainObject class because this is a regular, abstract Java class. The AbstractDomainObject's code is written manually; it is not generated.

The previous relation shown in Figure 5.5 implements only one side of the relation that allows the meta object to obtain its domain object. However, the framework will need this relation to be bidirectional. Each domain object must also be able to obtain its meta object.

In the banking example, a write transaction that modifies an Account (which extends AbstractDomainObject) must obtain its meta object to be able to access all the DependenceRecords that depend on it. Only this way can transactions know which predicates to reexecute after modifying an account.

This bidirectionality is implemented manually in the AbstractDomainObject, much like the code generator would. The implementation is not shown here because it is complex and not central to the present discussion.

The AbstractDomainObject needs to undertake structural changes to support this bidirectionality. In practice, every domain object will now contain a new link to its meta object. However, this structural change will only have to be performed once. The meta object represents a general domain object inside the framework's domain. So, it can be used by any other part of the framework that eventually needs to relate to any domain object. The use of the meta object prevents that these structural changes ever have to be repeated in the future.

Now that the framework has this meta object representation, all further DML relations to domain objects may now go through this meta object. In particular, the DependenceRecords that the framework will declare in DML can now create relations to the PersistentDomainMetaObject.

5.2.2 Persistent Dependence Record

As the previous section has shown, the framework needs to declare the DependenceRecords in DML to persist them. The DML declaration of the PersistentDependenceRecord is shown in Figure 5.6.

```
class PersistentDependenceRecord {
  Method predicate;
}
```

Figure 5.6: Declaration of the PersistentDependenceRecord with a Method predicate slot.

For now, the PersistentDependenceRecord needs to store only a value with the predicate that has executed. In the banking example, the predicate value can contain the checkTotalBalancePositive() method of the Client class. The PersistentDependenceRecord also needs two different relations to domain objects: one for the dependent object, and another for the depended objects. The following relations involve the PersistentDomainMetaObject introduced in the previous section.

The first relation, shown in Figure 5.7, connects the PersistentDependenceRecord to a single dependent object. In the banking example, the dependent is the client for which the consistency predicate has executed, as illustrated back in Figure 2.10. The multiplicity of this relation is many (records) to one (dependent object). So, a PersistentDependenceRecord can only have one dependent object. Each object can have many own PersistentDependenceRecords: one for each consistency predicate that its class defines.

The second relation, shown in Figure 5.8, connects the PersistentDependenceRecord to multiple depended objects. In the banking example, these depended objects are the accounts on which the predicate

```
relation DependentDomainMetaObjectOwnDependenceRecords {
  PersistentDomainMetaObject playsRole dependentDomainMetaObject;
  PersistentDependenceRecord playsRole ownDependenceRecords {
    multiplicity *;
  }
}
```

Figure 5.7: Relation between the PersistentDependenceRecord and the dependent meta object. The dependent object is the one on which the predicate has executed.

depends, as illustrated back in Figure 2.10. The multiplicity of this relation is many (records) to many (depended objects). A PersistentDependenceRecord will depend on many objects: the multiple accounts of the client whose consistency was checked. Each object can have many depending PersistentDependenceRecords: for example if an Account would have multiple owner Clients.

```
relation DependedDomainMetaObjectsDependingDependenceRecords {
  PersistentDomainMetaObject playsRole dependedDomainMetaObjects {
    multiplicity *;
  }
  PersistentDependenceRecord playsRole dependingDependenceRecords {
    multiplicity *;
  }
}
```

Figure 5.8: Relation between the PersistentDependenceRecord and the depended meta objects. The depended objects are those which the predicate's execution has depended on.

Note that, unlike what was discussed in Section 2.3.3, the PersistentDependenceRecord depends on objects, and not on VBoxes. It is not possible, in DML, to specify a relation to a VBox, because it is not a known domain entity. Because the PersistentDependenceRecord now depends on objects, any changed value within that object will trigger the execution of the predicate. Thus, this work has changed the granularity of the dependencies that are recorded.

This change in granularity will have some drawbacks and some advantages. On the one hand, this larger granularity will cause more **false positives**, as was discussed in Section 2.3.2. Consider a certain PersistentDependenceRecord that stores a dependency to an account because it uses its balance. If someone changes the account's date of creation, for instance, the predicate will reexecute needlessly, because the account itself was changed.

On the other hand, this larger granularity will have a smaller memory requirement, because, in general, less dependencies will be stored. A PersistentDependenceRecord that depends on many slots of a single object will not need to store many dependencies to many VBoxes. Instead, it will store one single dependency to the object that contains all those slots. I believe that this case happens more often than not.

Figure 5.9 illustrates the domain declarations introduced so far in this chapter. The PersistentDomainMetaObject allows the framework's domain to have general bidirectional relations to any domain object. The PersistentDependenceRecord stores the information about the execution of a consistency predicate. It is related with the dependent object that was checked and is being kept consistent. It is also related to the several objects on which the predicate's execution depends.

Naturally, the information contained in the relations is also persisted to ensure that the link between the records and the objects is not lost. Thus, after startup, the application runtime can fetch any account

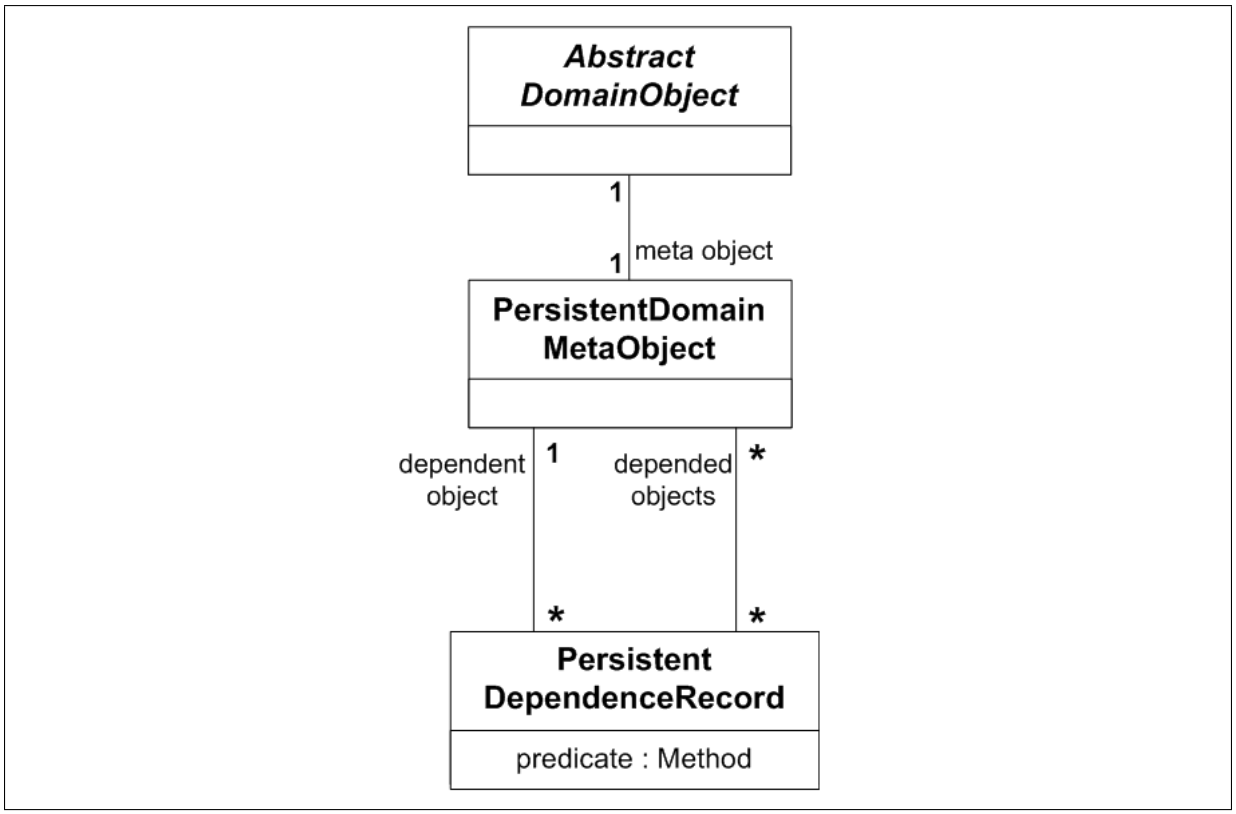

Figure 5.9: Fénix Framework's domain with the PersistentDependenceRecord, PersistentDomainMetaObject, and AbstractDomainObject.

from the database and modify it. The framework will then fetch the depending PersistentDependenceRecord that contains the predicate method and the client. The transaction can then recheck the predicate to maintain the client consistent.

These domain declarations are enough to support the persistence requirement for the consistency predicates in the Fénix Framework, as identified in Section 3.1. However, there are still a few issues to solve. In particular, this work is concerned with the use of the framework on enterprise applications that undergo incremental development and deployment, as discussed in Section 3.2. So, this document must also consider what happens when the development team changes the code of the application between two deploys.

5.2.3 Metaclass Hierarchy and Existing Objects

The code and structure changes necessary for incremental development can influence the consistency predicates and the PersistentDependenceRecords. A developer may decide that an already existing class should define a new consistency predicate to guarantee that its objects remain consistent according to a new rule. When a new object of this class is created, it will then trigger the execution of the consistency predicate and build a PersistentDependenceRecord for the new object. Any following changes to the domain will be traced back to the PersistentDependenceRecord and prevent this new object from being made inconsistent.

Still, nothing is done to the old existing objects of this class. They are not created, only fetched from the database. Moreover, in their previous version they did not have any PersistentDependenceRecord, because the consistency predicate did not even exist. Some of these old objects might already be consistent according to the implementation logic of the new consistency predicate. However, they are

not associated with any PersistentDependenceRecord, and nothing prevents the application from making them inconsistent.

To avoid this problem, the system needs to be able to detect and to execute new predicates for all existing objects of a certain class. The execution of new predicates should be performed at startup, during the initialization phase of the Fénix Framework. This initialization phase is executed every time the application starts, either for the first time ever, or after it was restarted. During this phase, for each new predicate to check, the framework will create one PersistentDependenceRecord for each already existing object. As Section 5.3 will show, this initialization phase detects all the code changes and performs the necessary updates.

The first challenge to solve in this section is to allow the execution of a new predicate for all existing objects of a class. The Fénix Framework has no way to obtain all the existing objects of a certain class. One possible way to obtain the existing objects is to query the database for all objects of a certain class. Although this solution seems viable and perhaps quite efficient, generally, it is neither very correct, nor portable.

First, it is not considered correct because it breaks the layered abstraction model of the Fénix Framework. Only the getters and setters of base domain classes should be allowed to read (or write) the data from the database. Second, this solution is not portable because its implementation would be restricted to the persistent medium used. Certainly, the data representation and access on a relational database is different from that of an object-oriented database. The data representation of a big table database is different from that of a file system. All of these alternatives may be used to store the application's domain objects.

To avoid these drawbacks, lets return to the idea of using the domain model that provides an abstraction layer and deals with the persistent medium. I propose to continue extending the domain of the Fénix Framework to keep track of which classes exist. Each existing class can then explicitly store all its existing objects. Whenever a new object is created, it adds itself to the list of existing objects of its class. Whenever an object is deleted, it removes itself from this list.

With this information, the developer can now freely introduce a new consistency predicate on a certain domain class. The framework can then execute the predicate, because it can obtain all the existing objects of that class. It also needs to keep track of any subclasses of that class, because the predicate also applies for all existing objects of those subclasses. Moreover, between each deploy of the application, the development team may create new domain classes, change or delete existing ones. The framework must ensure that the list of existing classes is also updated accordingly.

Essentially, the idea is to extend the Fénix Framework domain to keep persistent information about the application's domain classes and objects. I argue that the correct way to do so is by creating new Fénix Framework domain classes for this purpose. Alike what was done with the PersistentDependenceRecord, I have created a new entity to represent the classes of the application domain.

But first, the framework's domain needs a way to access this new entity. I have created a root domain class to serve as the entry point for the Fénix Framework, and that will allow the framework to obtain any part of its domain.

The class illustrated in Figure 5.10 is a singleton root domain class whose only object is connected to the rest of the persistent domain of the Fénix Framework. The developer can access the PersistenceFenixFrameworkRoot object by using the static getInstance() method. From there, he can access the rest of the domain by traveling through the domain relations.

Still in the Fénix Framework domain, the next step is to create a new entity to contain the repre-

```
class PersistenceFenixFrameworkRoot {
}
```

Figure 5.10: Declaration of the PersistenceFenixFrameworkRoot

sentation of the domain classes of the application. Each object of this new class will represent a class of the target application. Therefore, this new class will be a metaclass, which will contain a list of existing objects of the represented domain class.

The new PersistentDomainMetaClass entity is illustrated in Figure 5.11. Inside this new entity, the domainClass is the slot that holds the representation of the Java Class of the target application. So, in the banking example, the framework's initialization will create PersistentDomainMetaClasses to represent the target applications' banking domain. It will create one PersistentDomainMetaClass object to represent the class Bank, another to represent the Client, another for the Account, and so on.

```
class PersistentDomainMetaClass {
  Class domainClass;
}
```

Figure 5.11: Declaration of the PersistentDomainMetaClass with the domain class slot that it represents.

The Class type in the previous figure is defined in the DML file as a value type. In practice, the value stored is the result of calling the toString() method on a java.lang.Class. This method returns the fully qualified name of the class, which includes the package and the class name. This value is enough to identify a domain class.

These metaclasses are domain objects like any other; they are persisted in the database for future use. Therefore, whenever the application restarts, the next initialization phase detects that each of the bank's domain classes already have metaclasses. These existing metaclasses represent domain classes that are already known. The framework will not need to create these metaclasses again.

The relation shown in Figure 5.12 links the existing PersistentDomainMetaClasses to the PersistenceFenixFrameworkRoot. With this relation, the framework can obtain a list of all existing metaclasses from its root object.

```
relation PersistenceFenixFrameworkRootPersistentDomainMetaClasses {
  PersistenceFenixFrameworkRoot playsRole persistenceFenixFrameworkRoot;
  PersistentDomainMetaClass playsRole persistentDomainMetaClasses {
    multiplicity *;
  }
}
```

Figure 5.12: Relation between the framework's root and the existing metaclasses.

This relation gives the initialization phase a domain-oriented approach to know what classes existed the last time that the application ran. By comparing those previously known classes to the current existing classes, the initialization can detect differences in the target application domain. It can detect new classes that have been created in the domain, and classes that have been removed. It is the goal of this initialization phase to update the list of existing classes according to the latest changes made by the developers.[2] After this task is completed, the application runtime knows that any new domain objects

[2]The actual implementation of the initialization phase is discussed in Section 5.3.1.

that it creates will have a metaclass to be associated with.

However, to associate them, the PersistentDomainMetaClass needs to contain a relation to the list of existing objects. This relation is displayed in Figure 5.13.

```
relation PersistentDomainMetaClassExistingDomainMetaObjects {
  PersistentDomainMetaObject playsRole existingMetaObjects {
    multiplicity *;
  }
  PersistentDomainMetaClass playsRole persistentDomainMetaClass;
}
```

Figure 5.13: Relation between each metaclass and its existing objects.

Whenever a new object is created, the PersistentDomainMetaObject's constructor is invoked. This constructor contains the code to add the object being created to the metaclass' list. The developer does not need to worry about this association, he simply creates new domain objects and the framework will associate them. Likewise, the PersistentDomainMetaObject's delete() method contains the code to remove objects being deleted from this list. Thus, the list of existing objects is kept up-to-date, independently of what code is written by the development team.

The framework will also need a way to access the existing objects of subclasses of these metaclasses. As explained earlier, a new predicate on the superclass must be executed for objects of its class and for those of subclasses too. One possibility to work around this issue is to store all the objects of the class hierarchy in the existing objects list of the top superclass. This way, the superclass' objects and all subclasses' objects would be mixed together in the same domain relation.

As simple as this option may seem, it does not solve the issue entirely. For instance, the developers may choose to insert a new consistency predicate on a subclass, which would apply only to objects of the subclass. Obviously, a consistency predicate method at a subclass could never be executed for objects of the superclass. To support this new predicate, the framework needs to obtain a list of the existing objects from the subclass alone. Thus, it is important that all metaclasses contain only the objects whose actual class is the domain class represented.

I have created a relation between metaclasses to keep the hierarchy information of the domain classes. This relation is shown in Figure 5.14.

```
relation PersistentDomainMetaSuperclassPersistentDomainMetaSubclasses {
  PersistentDomainMetaClass playsRole persistentDomainMetaSuperclass;
  PersistentDomainMetaClass playsRole persistentDomainMetaSubclasses {
    multiplicity *;
  }
}
```

Figure 5.14: Relation between a meta superclass and its meta subclasses.

The multiplicity of this relation allows each metaclass to have one meta superclass and several meta subclasses. The initialization phase will be responsible for keeping this hierarchy information updated. The framework now has a way to obtain the objects of a superclass and, recursively, those of the subclasses of interest. They can now be obtained by using the domain alone, and without resorting to Java reflection.

Moreover, the framework's initialization is now able to detect changes in the domain class hierarchy, and update the metaclass hierarchy accordingly. For instance, the developer may change an existing class that had no superclass, and make it extend another class. In this case, a class whose hierarchy has changed may be affected by new predicates that exist at its new superclass. These and other situations will be examined in greater detail in Section 5.3.1.

In summary, these few domain declarations have introduced the first part of the Fénix Framework domain that deals with code changes. This part of the domain is illustrated in Figure 5.15. The framework now has the basis to support the incremental development of the application, as the developer makes changes to the domain classes. The framework's initialization now detects these changes to the structure of the application's domain.

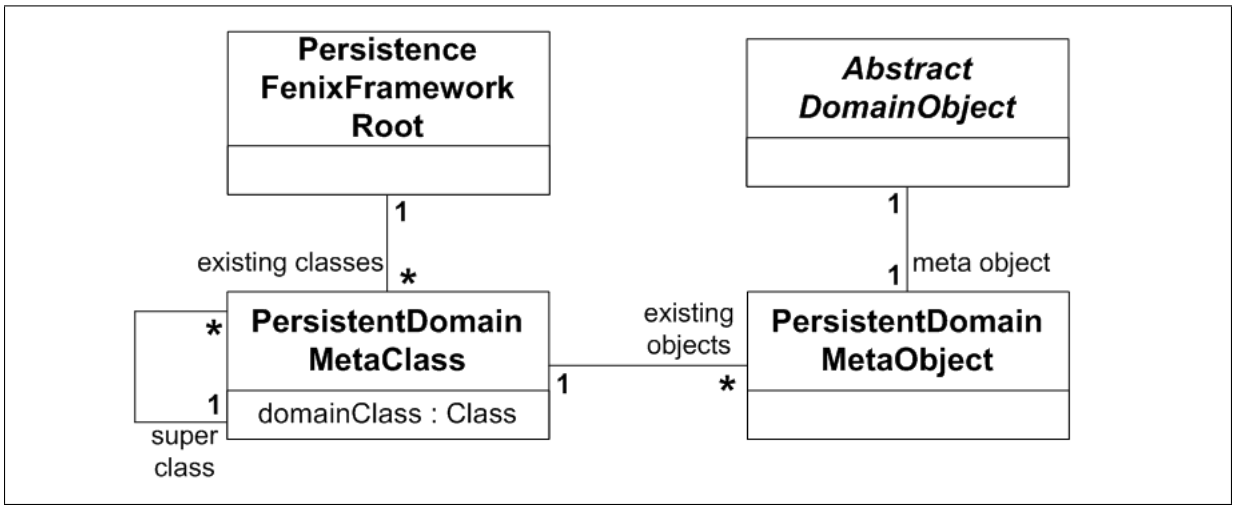

Figure 5.15: Fénix Framework's domain with the PersistenceFenixFrameworkRoot, PersistentDomainMetaClass, PersistentDomainMetaObject, and AbstractDomainObject.

After this initialization, the framework's runtime is able to trace the construction and deletion of objects and keep the existing objects list updated. With these new features, the framework is now aware of the full class hierarchy, and of all existing domain objects of each class.

5.2.4 Known Consistency Predicates

Several sections of this work explain that it is typical for the developers to alter parts of an application's domain model throughout the development process. They can change the domain class hierarchy by creating new classes or by deleting old classes. But they can also change the inner code of a class, without necessarily changing the class hierarchy.

With the work presented so far, the framework has the complete coarse-grained information of the existing domain class hierarchy. This section describes the second part of the Fénix Framework domain that deals with code changes. The framework will need finer-grained information about the individual consistency predicates within each class. Only then will it be able to detect and to respond to changes to the consistency predicates.

The developers may create or remove consistency predicates inside a class. They may even alter a predicate's signature, by changing the visibility of the method to private, to public, or by making it final. All of these changes must be taken into account, especially in a class hierarchy where a predicate at a superclass may be overridden in any subclass.

However, with the PersistentDomainMetaClass described previously, the framework cannot detect any of these changes. These changes do not require any structure modification, neither to the domain model, nor to the existing classes. Therefore, in this section, I have created another new entity to represent, in the framework's domain, the existing consistency predicates.

Similarly to the entities presented in the previous section, this new entity will be persisted. During the initialization, the Fénix Framework will be able to compare the previously persisted predicates to the currently existing ones.[3] This comparison will allow the detection of new predicates created, as well as changes or removals of old predicates.

Basically, the idea is to create another new class in the framework's DML to represent a consistency predicate that is known to exist in a domain class. This new class is the KnownConsistencyPredicate, illustrated in Figure 5.16.

```
class KnownConsistencyPredicate {
  Method predicate;
}
```

Figure 5.16: Declaration of the KnownConsistencyPredicate with the predicate slot that it represents.

The first time that the Fénix Framework initializes, it will create the required metaclasses and introspect every domain class. Within the domain classes, it can obtain the methods that are annotated with @ConsistencyPredicate. For each of those methods, the framework can create a KnownConsistencyPredicate. On the following initialization, the framework will be able to know which predicates existed the last time.

In the KnownConsistencyPredicate illustrated in Figure 5.16, the Method predicate is a value type that is serialized as a string and persisted. This string is the result of calling toString() on a java.lang.reflect.Method, which includes a full textual description of the method's signature. Therefore, if a programmer changes a predicate's signature, the framework will no longer match the persisted method to the existing one. It will delete the old KnownConsistencyPredicate of this predicate whose signature is changed, because a predicate with that signature no longer exists. Then, it will create a brand new KnownConsistencyPredicate to represent this predicate with a new signature. Thus, a predicate whose signature changes is treated like a brand new predicate, and will execute for all objects of its class.

To make this workflow possible, the framework needs a way to access the existing KnownConsistencyPredicates from the domain. I have created another relation; each KnownConsistencyPredicate will be related to the PersistentDomainMetaClass that declares it. That relation is shown in Figure 5.17.

```
relation PersistentDomainMetaClassDeclaredConsistencyPredicates {
  PersistentDomainMetaClass playsRole persistentDomainMetaClass;
  KnownConsistencyPredicate playsRole declaredConsistencyPredicates {
    multiplicity *;
  }
}
```

Figure 5.17: Relation between the PersistentDomainMetaClass and its declared KnownConsistencyPredicates.

Previously, the framework could already obtain the PersistenceFenixFrameworkRoot object, and obtain the existing PersistentDomainMetaClasses. With this relation, the framework can now obtain the

[3] The framework uses reflection to obtain the currently existing consistency predicates

predicates that each domain class declares. For each metaclass, the framework can now compare the KnownConsistencyPredicates to the existing predicate methods in the corresponding domain class. Thus, it now has the means to properly detect changes in the predicates.

There are still a few details left to discuss, concerning the KnownConsistencyPredicates. Recall that, from the previous Section 5.2.2, in Figure 5.6, the PersistentDependenceRecord stores the Method predicate that was executed. This information is now duplicated in the domain, because it appears both in the PersistentDependenceRecord and in the KnownConsistencyPredicates. In terms of implementation, this duplication means that the framework would have twice the management effort to keep this information updated.

I have to placed this information only in the KnownConsistencyPredicate, which is the entity that truly represents the predicate. So, the PersistentDependenceRecord can be refactored. The new, empty declaration of the PersistentDependenceRecord is shown in Figure 5.18.

```
class PersistentDependenceRecord {
}
```

Figure 5.18: Declaration of the PersistentDependenceRecord without the Method predicate.

Instead of storing the predicate, the PersistentDependenceRecord should have a domain relation to the KnownConsistencyPredicate. This new relation is shown in Figure 5.19.

```
relation KnownConsistencyPredicatePersistentDependenceRecords {
  KnownConsistencyPredicate playsRole knownConsistencyPredicate;
  PersistentDependenceRecord playsRole persistentDependenceRecords {
    multiplicity *;
  }
}
```

Figure 5.19: Relation between the KnownConsistencyPredicate and the PersistentDependenceRecords.

With this relation, the PersistentDependenceRecord can obtain the Method predicate as before, even if indirectly. It also creates an indirect relation between the PersistentDependenceRecords and the PersistenceFenixFrameworkRoot. Recall that all of these domain relations are bidirectional. So, this relation also gives the KnownConsistencyPredicate a way to access all the PersistentDependenceRecords that were executed for its predicate.

It is now fairly easy for the developer to obtain all the PersistentDependenceRecords that exist for the application. It is also easy to obtain all the PersistentDependenceRecords for a certain class, or for a single predicate. This easy accessibility can be very useful, because the records can tell exactly which domain objects are being kept consistent by which predicates. It will also be an easy way to look for already-existing inconsistencies, as Section 5.2.6 will discuss later.

This section described the second part of the Fénix Framework domain that deals with code changes. This part of the domain is illustrated in Figure 5.20. The Fénix Framework domain now contains a new persistent class: the KnownConsistencyPredicate, which represents the existing predicates in the code. The KnownConsistencyPredicate is related to the PersistentDomainMetaClass that declares it. It is also related to its existing PersistentDependenceRecords.

With this information, it is now possible to detect new predicates created, and existing predicates changed or deleted. For each new predicate, the framework can execute it for all existing objects of the

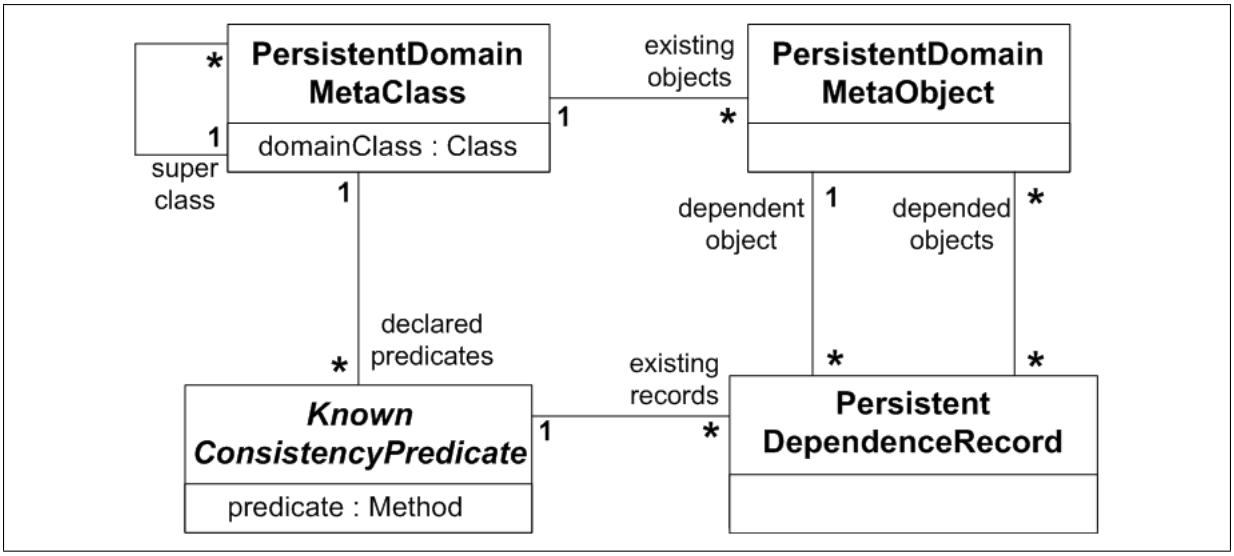

Figure 5.20: Fénix Framework's domain with the PersistentDomainMetaClass, PersistentDomainMetaObject, KnownConsistencyPredicate, and PersistentDependenceRecord.

affected classes, thanks to the PersistentDomainMetaClasses. For each deleted predicate, the framework can delete all the PersistentDependenceRecords that were associated with the KnownConsistencyPredicate. Also, the framework can detect and deal with existing predicates whose signatures were changed, and reexecute them. Thus, it can now keep the PersistentDependenceRecords updated across several versions of the code.

However, it might still not be clear why is a reexecution necessary when, for instance, the programmer changes a predicate from public to private. The following Section 5.2.5 explains this and other situations of signature changes in greater detail. It describes the third, and last part of the Fénix Framework domain that deals with code changes. It extends the framework's domain with three new classes, that will represent the three types of predicates identified earlier in Section 5.1.3. These new classes will provide a better modeling of the relation between predicates that override each other, within a domain class hierarchy. The following section will also provide a more detailed explanation of the implications of changing predicates inside a hierarchy.

Before moving on to the next section, it is important to note that the model proposed in this chapter has a major limitation. The framework does not yet have a way to detect the change of the body of a predicate method. Recall from Section 5.1.3 that this kind of change would require the recheck of the predicate, whose execution flow may have changed. But, for now, changing the code of a predicate is a silent operation that goes unnoticed.

In fact, there are other kinds of silent changes that are not yet detected by the framework, but may have an influence on the predicate's execution. For instance, it may happen that a certain predicate invokes an auxiliary method. If that auxiliary method's code is changed, it is likely that the predicate will have a different execution flow. Still, the framework does not detect this change because it does not have any information about auxiliary methods.

Therefore, to detect these kinds of changes, the framework would first need domain information about the existing methods and their implementation code. Then, the PersistentDependenceRecord would need a link to this information, to represent the parts of code that were used for the predicate's execution. The framework's initialization would be able to detect methods that changed and reexecute the predicates that used them. Even though this work does not implement this solution, the idea is explored in greater

detail in Chapter 8.

However, in spite of this limitation, a code change is only silent when there is no simultaneous structural change. For instance, a developer may change a predicate's code together with its name or signature. In this case, the framework will detect the signature change and reexecute the predicate.

Still, the framework only looks for signature changes in the predicates themselves. Therefore, this workaround is more complex when auxiliary methods are involved. If a developer changes the code of an auxiliary method, he will need to know exactly which predicates are involved that invoke this auxiliary method. He can then force the framework to reexecute the predicates by renaming those predicates manually.

5.2.5 Overriding Consistency Predicates

The previous section has shown the necessary domain model entities to detect changes in the set of predicates. With that information, the framework needs only to execute new predicates and delete the information of the ones that were removed. It also needs to renew the information of predicates whose signatures have been changed.

However, the previous section has only given support to the detection of individual predicates. It did not take into account that a public predicate in a domain class hierarchy can override and be overridden. This section will discuss the implications of changes within a hierarchy where predicates can override each other, as Section 5.1.3 has presented. It will also introduce new entities to provide a better modeling of these situations.

Originally, the Fénix Framework allows the programmer to specify methods with any visibility as consistency predicates. So, both private and public predicate methods at the top of a class hierarchy would execute for all objects of its class and subclasses. Note that a predicate method can also be used as a regular method in the domain of the application, and can be invoked by other parts of the code. For the programmer, changing the visibility of a predicate method would be a way to limit the accessibility of that method to the rest of the application.

Section 5.1.3 has identified three kinds of consistency predicates, and provided a list of possible code changes of predicates in a class hierarchy:

- Inserting a new predicate (public, final, or private)
- Removing an old predicate (public, final, or private)
- Changing the signature of an existing predicate method
- Changing the implementation of an existing predicate method (or an auxiliary method)

The previous Section 5.2.4 has briefly discussed how changes to the implementation of the predicates, or auxiliary methods, are not yet supported. It has also discussed that a predicate whose signature is changed is treated both as an old predicate removed, and as a new predicate inserted. For example, if a private predicate's visibility is changed to public, the framework will detect an old private predicate, and a new public one. However, adding a new public predicate will not have the same effect as adding a new final predicate. Likewise, removing an old private predicate will not have the same effect as removing an old public predicate.

The framework will need to properly implement the different effects of changes in public, final, and private predicates. Before doing so, however, the framework needs to tell them apart. So, in this section, I present three new entities in the Fénix Framework's domain to represent the three kinds of predicates.

These entities allow the framework to separate the implementation of each predicate in different classes.

The PrivateConsistencyPredicate is the entity that represents a consistency predicate whose visibility is private. It is shown in Figure 5.21. This class extends the KnownConsistencyPredicate, which is described in Section 5.2.4. The PrivateConsistencyPredicate inherits a Method predicate and relations to the declaring metaclass and the existing dependence records.

```
class PrivateConsistencyPredicate extends KnownConsistencyPredicate {
}
```

Figure 5.21: Declaration of the PrivateConsistencyPredicate.

This new class will contain the implementation responsible for adding a new private predicate in the application code, and removing it. Because private predicates can neither override nor be overridden, this implementation will be straightforward. At startup, a new PrivateConsistencyPredicate will execute for all objects of its class, and all subclasses.

The PublicConsistencyPredicate is the entity that represents a consistency predicate whose visibility is public or protected. It is shown in Figure 5.22. It also inherits a Method predicate and relations to the metaclass and the dependence records from the KnownConsistencyPredicate. It will contain the implementation responsible for adding a new public predicate in the application code, and removing it.

```
class PublicConsistencyPredicate extends KnownConsistencyPredicate {
}
```

Figure 5.22: Declaration of the PublicConsistencyPredicate.

In a typical domain class hierarchy, each class can have one superclass and many subclasses. A public predicate with a certain name can override another predicate with the same name at the superclass. The following text will refer to the **predicate at the superclass** as the *overridden predicate*. This overridden predicate does not necessarily need to be on the direct superclass of the original class. It can be located as far above the class hierarchy as the developer had decided.

A public predicate with a certain name can also be overridden by many predicates with the same name at the subclasses. The following text will refer to these **predicates at the subclasses** as the *overriding predicates*. These overriding predicates do not necessarily need to be on direct subclasses of the original class. They can be located as far below the class hierarchy as the developer had decided.

The PublicConsistencyPredicate's implementation will need further information about what are the overridden and overriding predicates. This information is needed, for instance, when an old public predicate is removed from a certain class, inside a hierarchy. As Section 5.1.3 has shown, the framework must remove the PersistentDependenceRecords of this old public predicate. After it does, the framework will also need to obtain the previously-overridden predicate at the superclass and execute it from this class downwards.

The task of obtaining the overridden predicate should be implemented in PublicConsistencyPredicate domain class. So, it is important that this implementation uses the domain information of the PublicConsistencyPredicate class. The framework's domain should contain information about the overridden and the overriding predicates. The relation that will keep this information is illustrated in Figure 5.23.

The multiplicity of this relation allows each public predicate to have one overridden predicate, and

```
relation PublicPredicateOverriddenPublicPredicatesOverriding {
  PublicConsistencyPredicate playsRole publicPredicateOverridden;
  PublicConsistencyPredicate playsRole publicPredicatesOverriding {
    multiplicity *;
  }
}
```

Figure 5.23: Relation between the overridden predicate and the overriding predicates.

many overriding predicates. This relation is not needed at the PrivateConsistencyPredicate, and, therefore, it is not placed at the KnownConsistencyPredicate superclass. With this information, the framework can now obtain the overridden predicate of an old public predicate that was removed, and execute the former. Much like the relation between metaclasses shown back in Figure 5.14, the overridden predicate can be obtained by using the domain alone, and without resorting to Java reflection.

The FinalConsistencyPredicate is the entity that represents a predicate that is final and whose visibility is public or protected. It is shown in Figure 5.24. It extends the PublicConsistencyPredicate, and inherits the relation to other overridden or overriding public predicates. It also inherits a Method predicate and relations to the metaclass and the dependence records from the KnownConsistencyPredicate. It will contain the implementation responsible for adding a new final predicate in the application code, and for removing it.

```
class FinalConsistencyPredicate extends PublicConsistencyPredicate {
}
```

Figure 5.24: Declaration of the FinalConsistencyPredicate that extends from the PublicConsistencyPredicate.

In practice, the FinalConsistencyPredicate will never have overriding predicates. It will possibly store one overridden predicate at a superclass, which must be a non-final PublicConsistencyPredicate. The FinalConsistencyPredicate will need to access the overridden predicate when it is removed from the code. Again, in this case, the framework must execute the overridden predicate from this class downwards.

These new declarations have introduced the last classes of the Fénix Framework's domain. The framework now has three new concrete classes to represent the different kinds of consistency predicates that Section 5.1.2 identified. Each of these classes will contain the behaviour that knows how to insert a new predicate and how to remove an old predicate of its type. All other kinds of code changes to the implementation or signature of predicates will be translated into insertions and removals.

Essentially, the framework's initialization phase will only create objects of these three new classes of consistency predicates. It will no longer need to create objects of the KnownConsistencyPredicate class. Therefore, the KnownConsistencyPredicate will be abstract, and contain only the structure and behavior that is common to all predicates. The KnownConsistencyPredicate's subclasses introduced in this section are illustrated in Figure 5.25.

5.2.6 Existing Inconsistencies

There is still one final issue to address in the Fénix Framework's domain. Recall that it will be important for the developer to have easy access to existing inconsistencies. This way, the developer can introduce a new predicate in the code now, and obtain the inconsistencies later to fix them. The model presented

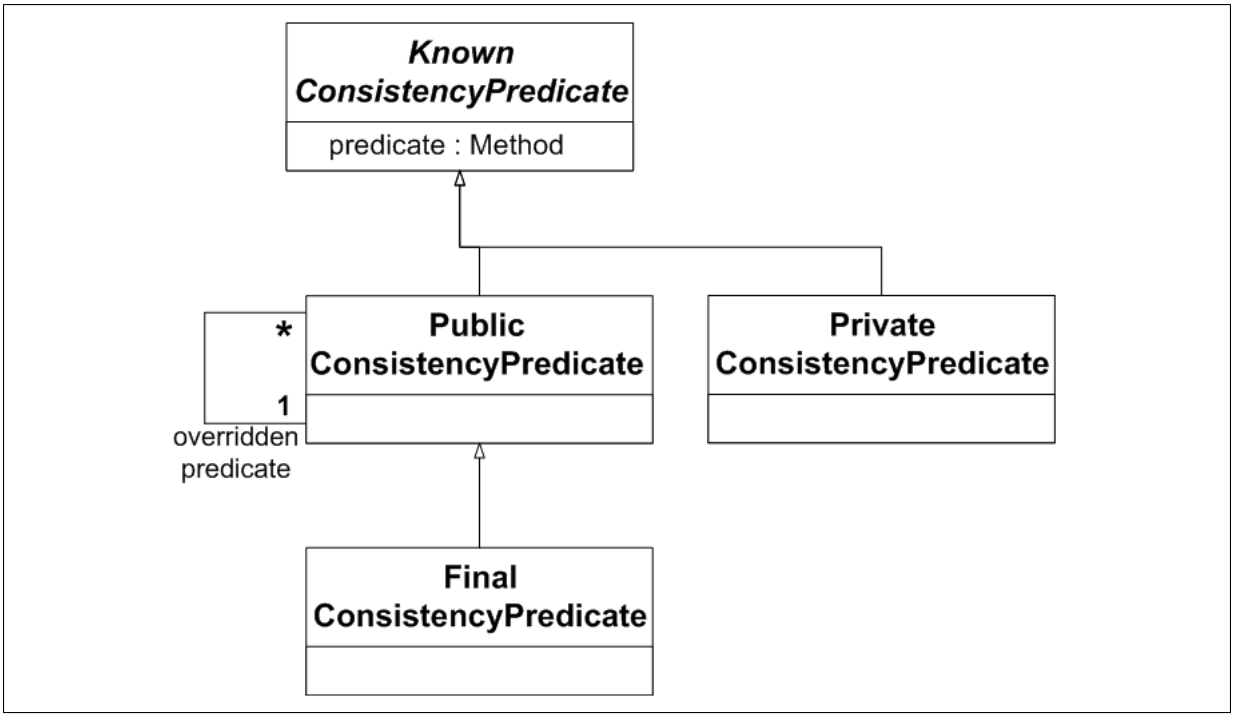

Figure 5.25: Fénix Framework's domain with the KnownConsistencyPredicate and subclasses.

so far does not yet provide this easy access.

Currently, the PersistentDependenceRecord does not even contain information about whether the object that it checks is consistent or not. When a new predicate is executed, each old inconsistent object is associated with a regular PersistentDependenceRecord. It is important for this PersistentDependenceRecord to store explicit information about the inconsistency of its object.

The new slot of the PersistentDependenceRecord, shown in Figure 5.26, is filled out when the record is built. It will store the true or false result of the predicate that executed. If the predicate returned true, the object is consistent. If it returned false, the object is inconsistent.

```
class PersistentDependenceRecord {
  boolean consistent;
}
```

Figure 5.26: Declaration of the PersistentDependenceRecord with consistency information.

This information can be used to implement inconsistency-tolerant transactions, as the next chapter will show. But regardless of the transactions' behavior, the PersistentDependenceRecords now keep track of all the objects that are inconsistent. Furthermore, the framework should also provide easy access to this information.

The developer can use the framework to determine if a certain domain object is inconsistent. He can use the framework's domain to obtain the meta object, and all its PersistentDependenceRecords. These PersistentDependenceRecords can be obtained by using the getOwnDependenceRecords() method generated by the relation illustrated back in Figure 5.7. If any of these dependence records is not consistent, then this domain object is inconsistent according to that record's predicate.

But the developer may also want to know which objects of a certain class are inconsistent, according to all of that class' predicates. I believe that he will most often need to obtain the objects that are

inconsistent according to one particular predicate that might have been introduced recently.

With the domain model presented so far, he would first have to obtain all the PersistentDependenceRecords associated with that predicate. Then, he would have no choice but to iterate through each PersistentDependenceRecord, and check if it is inconsistent. However, there will be as many PersistentDependenceRecords as there are existing objects of the predicate's class. Domain classes that are central to the target application will typically have numerous existing objects. Therefore, the performance of this search for inconsistencies might be far from ideal.

The relation presented in Figure 5.27 will help the developer to obtain the inconsistent objects according to a certain predicate. The generated code will now contain a getInconsistentDependenceRecords() method, thanks to which the developer no longer needs to go through all the records.

```
relation KnownConsistencyPredicateInconsistentDependenceRecords {
  KnownConsistencyPredicate playsRole;
  PersistentDependenceRecord playsRole inconsistentDependenceRecords {
    multiplicity *;
  }
}
```

Figure 5.27: Relation between the KnownConsistencyPredicate and the inconsistent dependence records.

Thus, the developer can now obtain inconsistencies more easily and with a better performance. However, note that this new relation is very similar to the one presented in Figure 5.19 from Section 5.2.4. The KnownConsistencyPredicate already had a relation to all its PersistentDependenceRecords. That relation is left unchanged; it will still contain a set of all existing PersistentDependenceRecords of that predicate.

The new relation presented above in Figure 5.27 will contain only the dependence records that are inconsistent. Therefore, the set of inconsistent records will naturally be a subset of the set of all existing records. The framework will need to keep this set of inconsistent records updated as the domain data changes. When an already existing inconsistent object is detected, its dependence record is added to this relation. When an inconsistent object is fixed, its dependence record is removed from this relation.

This last relation concludes the domain changes to the Fénix Framework that this work has performed. Figure 5.28 presents the final, full representation of the Fénix Framework's domain.

In summary, the framework's domain is comprised of 8 entities[4] with the following responsibilities:

- The PersistenceFenixFrameworkRoot is the entry point for the rest of the framework's domain, and detects code changes in the represented entities.
- The PersistentDomainMetaClass represents the application's domain classes.
- The abstract KnownConsistencyPredicate contains the structure and implementation that is common to all consistency predicates.
- The PrivateConsistencyPredicate represents the private consistency predicates.
- The PublicConsistencyPredicate represents the public or protected consistency predicates that are not final.
- The FinalConsistencyPredicate represents the public or protected consistency predicates that are final.

[4]The AbstractDomainObject is not a domain entity of the Fénix Framework. It is simply an abstract superclass for all domain objects.

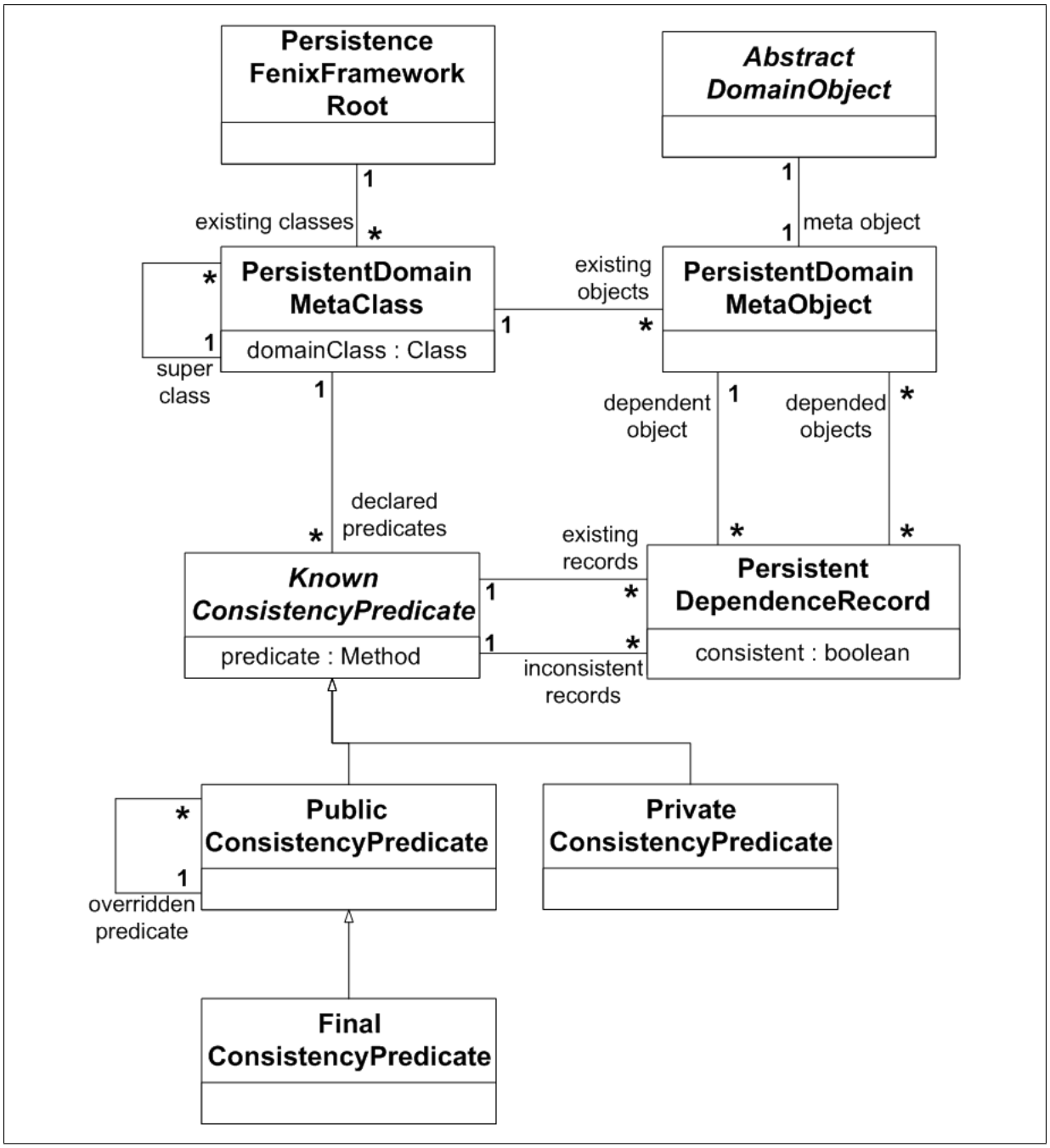

Figure 5.28: Fénix Framework's full domain model.

- The PersistentDependenceRecord stores the dependencies, and the (consistent or not) result of executing a consistency predicate for one object.
- The PersistentDomainMetaObject allows all these entities to relate to any domain object.

Thanks to these entities, the framework's implementation will be able to detect code changes and respond to them correctly. The following section will discuss the initialization of the Fénix Framework. It will explain in detail what is done in response to the detection of each change.

5.3 Responding to Code Changes

This section will discuss the details of the Fénix Framework's initialization. It will present the core implementation that is responsible for the framework's behavior at startup. The framework must correctly initialize the entities presented in the last Section 5.2, to successfully detect and respond to code changes.

5.3.1 Initializing the Metaclass Hierarchy

The Fénix Framework's initialization process is preceded by a small bootstrap phase. This phase is executed if the framework is initializing for the first time ever for a new application with an empty database. During the bootstrap phase, the framework creates the necessary root objects; it creates one object of PersistenceFenixFrameworkRoot for the Fénix Framework's own domain. It creates one object of the application's root class, which is in a separate domain.

After this bootstrap phase is complete, the framework invokes the PersistenceFenixFrameworkRoot's initialize() method. This method will then perform the needed initialization for the metaclasses and the known consistency predicates. This initialization starts by updating the metaclasses, and only afterwards does it update the known consistency predicates.

So, the first part of the framework's initialization has the goal of updating the metaclass hierarchy. The framework first obtains three sets of classes: the old metaclasses to remove, the existing metaclasses to update, and the new classes to add. It can obtain these sets by using the structure in the DML and the PersistentDomainMetaClass model described in Section 5.2.3.

The set of old metaclasses contains those whose domain classes have been deleted from the code. It also contains the metaclasses whose classes are no longer declared in DML, and are now simple Java classes. The initialization invokes the delete() method of each metaclass in this set.

The set of existing metaclasses contains all those whose domain classes still exist in the code, and are still declared in the DML file. The framework is concerned with updating only the existing metaclasses whose superclass has changed in DML. The initialization invokes the delete() method of each metaclass in this set that has changed its superclass.

As Section 5.2.3 has briefly shown, a domain class whose superclass is changed might be affected by predicates of the new superclass. Also, that domain class may no longer be affected by predicates of the old superclass, depending on which were overridden. Altering the superclass of a class is considered as a major change, because usually it also implies large structural and semantic modifications.

So, the idea to deal with changed metaclasses is simply to delete them altogether. Afterwards, the framework will detect these classes as new, because their corresponding metaclasses no longer exist. The framework will then fully process these classes as if they were brand new classes in a new class hierarchy.

The implementation of the delete() operation of the metaclass will delete all declared KnownConsistencyPredicates (which can also be detected later). The delete() method also causes a cascade deletion of all subclasses. If a certain class was deleted, then the subclasses will have either changed superclass, or were deleted too. Therefore, the meta subclasses can be deleted as soon as the meta superclass is deleted. The delete() of each metaclass will invoke the classDelete() of all its declared KnownConsistencyPredicates, which will be described in the next section. After it has deleted the old and modified metaclasses, the framework can start creating the new classes.

The set of new classes contains the classes that are visible in DML, but have no associated metaclass. These classes are either the new classes that the developer added, or the changed classes that the frame-

work had deleted previously. First, the initialization creates a metaclass for each new class in this set by invoking the PersistentDomainMetaClass constructor. Then, after all the metaclasses are created, the framework initializes the superclass of each created metaclass. After the superclasses are initialized, the full metaclass hierarchy is up-to-date and the framework can proceed to the KnownConsistencyPredicates.

5.3.2 Initializing the Known Consistency Predicates

The second part of the framework's initialization deals with the existing consistency predicates declared in domain classes. The framework needs the metaclass hierarchy available before it attempts to detect consistency predicates that possibly override each other. It will iterate the existing metaclasses in top-down order. For each metaclass, the framework will obtain two sets: the new predicates to create, and the old predicates to remove.

During this phase, the initialization process will need to invoke several abstract methods of the KnownConsistencyPredicate class. Recall from Section 5.2.5 that its subclasses are the PrivateConsistencyPredicate and the PublicConsistencyPredicate. The FinalConsistencyPredicate is a subclass of the PublicConsistencyPredicate. Each of these three subclasses implements the abstract methods that they are responsible for.

The set of new predicates of a metaclass contains the declared predicate methods that do not have a corresponding KnownConsistencyPredicate yet. For each new predicate in the class, the framework creates the correct KnownConsistencyPredicate to represent it, based on the method's modifiers. The framework will invoke the constructor of the PrivateConsistencyPredicate, the PublicConsistencyPredicate, or the FinalConsistencyPredicate.

Then, after it has created all the predicates, the framework initializes the overridden predicate of each new predicate created. In practice, only the PublicConsistencyPredicates are concerned with initializing the overridden predicate at a superclass. Its implementation uses the meta superclasses to look for a PublicConsistencyPredicate with the same method name. The FinalConsistencyPredicate also inherits this implementation.

As Section 5.1.3 has shown, a new public consistency predicate should replace an overridden predicate at a superclass, from this class downwards. Therefore, the framework will iterate through the PersistentDependenceRecords of the overridden predicate. It will delete each PersistentDependenceRecord whose domain object's class is equal to, or a subclass of the new predicate's class. Those objects will now follow a new predicate that replaces the existing predicate from the superclass.

After all the predicates are created and the overridden predicates are initialized, the framework executes the new predicates for the affected classes. The Private- and the FinalConsistencyPredicates will execute for all the existing objects of its class and all subclasses, without exceptions. The PublicConsistencyPredicate will execute for all objects of its class, and for the subclasses that do not already have an overriding predicate.

The second set contains the old KnownConsistencyPredicates whose methods no longer exist in the code, or have no @ConsistencyPredicate annotation. Recall from Section 5.2.4 that the KnownConsistencyPredicate represents a predicate by its signature. So, this set of old predicates will also contain those whose signatures have changed, because they no longer match to their KnownConsistencyPredicates.

The framework will simply invoke the delete() operation of each KnownConsistencyPredicate in this set. The PrivateConsistencyPredicate will just delete all its PersistentDependenceRecords. However, the PublicConsistencyPredicate will also obtain the overridden predicate and execute it from its class down-

wards.

Recall from Section 5.1.3, that this situation represents an old predicate being removed, which used to override a predicate at the superclass. So, the predicate at the superclass is no longer being overridden. The framework executes the overridden predicate for the old predicate's class and the subclasses that did not already have an overriding predicate. This situation was illustrated in Figure 5.2. The FinalConsistencyPredicate also inherits this implementation.

However, the classDelete() method of the PublicConsistencyPredicate does not cause the execution of the overridden predicate. The classDelete() method is invoked only when the metaclass that declares this predicate has also been deleted. In turn, the meta subclasses of the declaring metaclass have also been deleted, as the previous Section 5.3.1 has shown. Therefore, the classDelete() method does not need to execute the overridden predicate because the classes that would become affected by that predicate have been deleted.

It is important to note that the developer can deploy a version of the code that has simultaneously deleted two PublicConsistencyPredicates. If the two predicates override each other, the order in which the framework deletes them is relevant. If it deletes the bottom predicate first, it will attempt to obtain the top predicate and execute it.

If the @ConsistencyPredicate annotation was removed from the top consistency predicate, this situation might result in the useless execution of a regular method for a large collection of objects. Worse, if the top method was actually removed from the code, Java might thrown an unexpected error because the reflection runtime cannot find the desired method. These situations are the reason why the framework iterates the metaclass hierarchy in top-down order. This way, the framework always deletes the top predicate first.

Chapter 6

Inconsistency Tolerance

"Errare humanum est."[1] - Lucius Annaeus Seneca

This chapter presents the novel theory and implementation of *Inconsistency Tolerance*. It discusses why tolerating already existing inconsistencies is important to keep an application's liveness as high as possible. This discussion will be based on the assumption that inconsistencies exist inevitably. So, it will be important to deal with this situation in a pragmatic way.

6.1 Theory of Inconsistency Tolerance

The main challenge of this work comes from an attempt to introduce consistency rules on an application that already has existing data. In practice, this work is motivated by the development of the Fénix Project[2], which has incrementally built a large information system in Java. This system has been under intensive use for several years, and has gathered a large amount of information during its lifetime. Yet, the consistency predicates were not applied to enforce the domain consistency. Fortunately, the Fénix team had the discipline of using defensive programming to manage the data. Nonetheless, introducing the consistency predicates can still ease the process of implementing further domain changes.

However, the human error is simply inevitable. Heedless developers are unaware of programming flaws that they might create. Unexperienced users might use the system in a workflow for which it was not designed. Either way, some parts of the data had inevitably become inconsistent over time. The second advantage of introducing the consistency predicates would be to mitigate such human errors.

Yet, despite these already existing inconsistencies, the Fénix system is still online today and is used by thousands of people. In fact, the transactional system successfully commits many transactions on a daily basis. In favor of an outstanding system liveness, the transactions read and write to parts of the inconsistent data, without any problems. Thus, if the developer is to create predicates to enforce the consistency rules, it is important to keep the same liveness quality that has been provided so far.

However, the current framework implementation of the consistency predicates will not guarantee this liveness property. Any transaction that performs a write on some data, will execute the predicates depending on this data. If the data was previously inconsistent (and not corrected), the predicate will return false and the transaction will abort. But this transaction did not necessarily insert new inconsistencies. It may have simply performed operations on some data that another event made inconsistent.

[1] To err is human.
[2] http://fenix-ashes.ist.utl.pt/

This situation was explored back in Section 3.2, and illustrated in Figure 3.2. So, to keep the application liveness as high as possible, the transactions need to deal with inconsistencies without halting the entire system. To do so, the framework needs to be somewhat tolerant to already existing inconsistencies.

Every transaction has a pre-commit phase, whose task is to validate the consistency predicates (see Section 2.3.2). This phase is responsible for the decision to either abort the transaction or proceed to the commit phase. Until now, it based this decision only on the result of the predicate execution. If a predicate returns false (representing an inconsistency) the transaction aborts, otherwise it proceeds.

Basing this decision only on the actual result of the predicate execution is insufficient. The key idea to improve the behavior of the predicate validation phase is to base this decision also on the past result of the predicate. The new PersistentDependenceRecord model presented in Section 5.2.6 contains a consistent slot, which stores the past result.

The value of this boolean variable will reveal inconsistencies existing in the past. It may be false when, for example, a new predicate is inserted in a class that already has many instances. The predicate will be initially executed once for each of these instances, some of which may already be inconsistent. As long as this information is updated, the predicate validation phase can determine which transactions passed through some data that was already inconsistent.

6.1.1 Transaction Operations

So, to make the system inconsistency tolerant, the validation phase must make a finer grain distinction among a few types of transaction operations. In regard to a particular predicate, a transaction may:

- **Corrupt** a consistent object, by making it inconsistent.
- **Fix** an inconsistent object, by making it consistent.
- **Keep** an object **consistent**.
- **Keep** an object **inconsistent**.

As you probably expect, each transaction may write to several objects, and each object may be used by several predicates. Thus, typically, each transaction can perform many of these previous operations.

6.1.2 Inconsistency Tolerance Semantics

With these operation types in mind, this work is now able to define a new desired semantics for the predicate validation phase. The major requirement to fulfill is that new inconsistencies are never added to the system. In the first case, the final outcome of at least one operation results in an inconsistent state, which was not inconsistent before.

*Transactions that **corrupt** any object must **always abort**.*

Fulfilling this requirement is easier said than done, as the previous chapters have shown.

In the second and third cases, the final outcome results in consistency.

*Transactions that only **fix** objects or **keep** them **consistent** should **commit**.*

Finally, what about transactions that **keep** a predicate **inconsistent**? Recall that the goals are to abort any transaction that inserts new inconsistencies, and to keep the system liveness as high as possible. In this final case, the outcome of an operation results in an inconsistent state, which was already inconsistent before. So, the framework should not abort these transactions because they do not insert any new inconsistencies.

*Transactions that only **keep** objects **inconsistent** should **commit**.*

Note that a transaction that performs several operations will check several predicates. If that transaction keeps one object consistent, for instance, it does not necessarily commit immediately. It simply proceeds to the next consistency predicate to check. Therefore, a transaction will only commit if all of its consistency predicates allow it to proceed. If a transaction keeps many objects consistent, but corrupts one single object, it will abort.

6.1.3 Optional Semantics

The change in semantics in this final case is a key difference from most of the related work in this area. Regular invariants are completely intolerant to any operation that results in inconsistency, no matter what may have happened in the past. The argument that supports this new semantics is that the domain will not get more inconsistent over time. However, this argument is only partially true. We can guarantee that the domain does not get more inconsistent in quantity, because no transaction inserts new inconsistencies. But there are some peculiar cases where it may get more inconsistent in quality.

Once again, consider the banking domain example, where an Account uses an inconsistency tolerant predicate checkBalancePositive() to enforce that its balance is always greater than or equal to zero. This predicate is illustrated in Figure 6.1.

```
public class Account extends Account_Base {
  // (...)
  @ConsistencyPredicate(inconsistencyTolerant = true)
  public boolean checkBalancePositive() {
    return (getBalance() >= 0);
  }
}
```

Figure 6.1: An inconsistency tolerant predicate that enforces that the Account's balance is always positive or zero. Arguably, this predicate deals with sensitive information.

Now, consider an application that already has a persisted state, in which an account already has a balance that is equal to -10€. If a transaction attempts to withdraw 500€ from this account, the final balance of the account will be -510€, which is an inconsistent state. But the previous balance was -10€, which was already inconsistent. The transaction does not add new inconsistencies, and will commit. This example is illustrated in Figure 6.2.

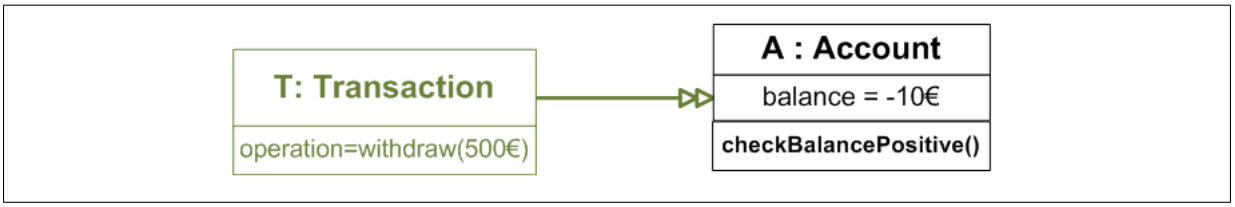

Figure 6.2: An inconsistency tolerant predicate that allows an undesired operation. Because Account A was already inconsistent before, Transaction T will commit. The inconsistency tolerant semantics might not be appropriate for predicates that deal with sensitive information.

Unfortunately, the Fénix Framework does not have any means to detect how serious an inconsistency is. To do so, it would need explicit information about the quality of each inconsistency. Adding and managing such information is beyond the scope of this work.

To help in mitigating this problem, I argue that the inconsistency-tolerant behavior of the transactions should be optional. Each predicate should also have a configuration parameter or annotation that indicates if it tolerates inconsistencies individually, as shown in Figure 6.1. Thus, a developer can specify that an individual predicate is not tolerant, which is particularly useful on predicates that deal with money, or other kinds of sensitive information. These predicates would then behave as regular predicates, while other less critical predicates could be inconsistency tolerant. I believe that this solution allows for a fine tradeoff between intransigent consistency and uninterrupted system liveness.

With optional semantics of inconsistency tolerance, the developer has two different kinds of predicates available: regular predicates, and inconsistency-tolerant predicates. The transactions running these predicates will respond differently whenever they keep an object inconsistent. A transaction running a regular predicate will abort immediately. A transaction running an inconsistency-tolerant predicate will proceed to the commit phase.

6.2 Implementation of Inconsistency Tolerance

Having described the theory behind the inconsistency tolerance, the remainder of this chapter shows how to implement inconsistency tolerance, using the domain presented in Section 5.2. The inconsistency tolerance behavior should be optional, because it might depend on the developer's desired semantics for each predicate. Therefore, the @ConsistencyPredicate annotation now contains a boolean parameter called inconsistencyTolerant that defaults to false. The use of this parameter was shown back in Figure 6.1.

If the developer chooses to set the parameter to true, the annotated predicate will then be tolerant to inconsistencies. A transaction that modifies an already inconsistent object is allowed to commit, even if the object is still inconsistent according to that predicate. The argument to justify this semantics is that the transaction did not create new inconsistencies. It simply modified an object that was already inconsistent.

The implementation that allows this behavior only needs to determine if the previous state of the object was indeed inconsistent. This information is available thanks to the consistent slot of the PersistentDependenceRecord, as seen in Section 5.2.6. Therefore, this slot can be used both to track existing inconsistencies and to allow inconsistency tolerance.

As this work has shown previously, the consistency check phase of the transaction is responsible for verifying the consistency predicates. During this phase, the transaction will obtain its write set that contains all the objects that it modified. Afterwards, the transaction will obtain the PersistentDependenceRecords that depend on those modified objects. For each PersistentDependenceRecord, the transaction will execute the predicate on the dependent object. After this execution, if the predicate returns false, the transaction would traditionally throw a ConsistencyException and abort.

However, it can happen that the predicate is inconsistency tolerant, and the dependence record has a false value in its consistent slot. In this case, the modified object was already inconsistent, and the transaction should not abort, but proceed to the commit phase. Thus, whenever a predicate returns false, the transaction verifies if it meets the conditions to tolerate that inconsistency.

Anyhow, when a new object is created, it must also satisfy the consistency predicates declared in its class and superclasses. In this case, the transaction will execute each predicate for the first time for this new object, which did not yet have any PersistentDependenceRecords. Still, a transaction that attempts to create an inconsistent new object will abort, even if the predicate is inconsistency tolerant. The framework assumes that the previous state of creating a new object is consistent.

Chapter 7

Validation

The work of this dissertation implemented an extension to mitigate the limitations of the Fénix Framework, which were explored in Chapter 3. In particular, this work presents solutions to the challenges presented in Section 3.3. To respond to changes made to consistency rules, this extension provides new entities to represent code artifacts, and a new implementation of the framework's initialization. To deal with existing inconsistencies, this extension modified the semantics of transactions that execute inconsistency-tolerant predicates.

To validate these solutions, it is necessary to demonstrate the response of the framework to several cases of incremental development with complex consistency predicates. As Section 3.1 discussed, it is crucial to show that the DependenceRecords are now persisted together with the application's data. As Section 3.2 discussed, it is important to verify that the framework does indeed detect code changes. Most importantly, the framework must construct and update the hierarchy of PersistentDomainMetaClasses at startup, and manage the list of existing objects at runtime. Also, at startup, the framework must keep the KnownConsistencyPredicates updated, detect new and old consistency predicates, and reexecute the right predicates for the affected objects.

Moreover, it is important to show the reaction of the framework's transactions to regular and inconsistency-tolerant predicates. In particular, the reaction of the transactions must respect the operations and semantics shown in Section 6.1.1. Finally, this chapter will also provide an analysis of the false positives and the false negatives of this extension.

7.1 Validating Code Changes

In respect to the two initialization steps shown in Section 5.3, this section separates in two steps the validation of the framework's awareness about code changes. The first part deals with changes to the domain class hierarchy, and the list of existing objects of each class. This metaclass hierarchy must be updated before the framework can deal with changes to the consistency predicates. Then, the second part can detect new and removed predicates within each class, and execute the correct predicates for the existing objects of the affected classes.

In this section, I am concerned with the correctness of the framework's initialization phase. No matter what code changes the developer performs, it is crucial for the framework to keep the representation of the target domain always updated.

7.1.1 Changing the Domain Class Hierarchy and the Existing Objects

Given the simple model of the metaclasses, illustrated back in Figure 5.15, there are only a few possible changes for each domain class. The developer can:

- Create a new domain class
- Delete an existing class
- Change the superclass of a class

All of these operations are performed in the DML. The framework will detect and respond to these changes according to the implementation described in Section 5.3.1.

However, after a new version of the code is produced, the developer can have made several of these changes simultaneously. The metaclasses are *hard connected* to each other by the structure of their hierarchy. In other words, changes to a certain class may affect the structure of other related classes (especially its subclasses). For instance, the deletion of a class implies that any subclasses must have either been deleted too or have changed their superclass.

Thus, the goal of this section is to view the reaction of the framework to the composition of these 3 types of changes. At the end of the initialization phase, the framework must have a correct representation of the domain class hierarchy. Also, during runtime, the list of existing objects of each metaclass must be kept updated, so that it can be used by the KnownConsistencyPredicates later. Obviously, the application's runtime can also:

- Create a new domain object
- Delete an old domain object

The initialization phase is followed by a debug step that displays a textual description with each existing metaclass, its meta subclasses, and the number of existing objects. In this section, instead of presenting a transcript of this text, I have chosen to illustrate the contents of the metaclass hierarchy in figures. The following series of 6 tests cover the combinations of the 3 elementary class changes, and demonstrate the framework's response. Note that there are only 6 possible combinations, instead of 9. Some combinations do not make sense and would not compile, such as having a new class that extends from a deleted class, or having a deleted class that used to extend from a new class.

These tests cover the following combinations:

1. A new class that extends from another new class
2. A new class that extends from a changed class
3. A deleted class that used to extend from another deleted class
4. A deleted class that used to extend from a changed class
5. A class that used to extend from a deleted class, and now extends from a new class
6. A class that used to extend from a changed class, and now extends from another changed class

In the first test, the developer creates a new class that extends from another new class. The required domain changes and the resulting metaclasses are illustrated in Figure 7.1. First, the framework creates both new metaclasses. Then, it assigns the Animal's meta superclass to the Thing class.

In the second test, the developer creates a new class that extends from a class whose superclass changed. The required domain changes and the resulting metaclasses are illustrated in Figure 7.2. First,

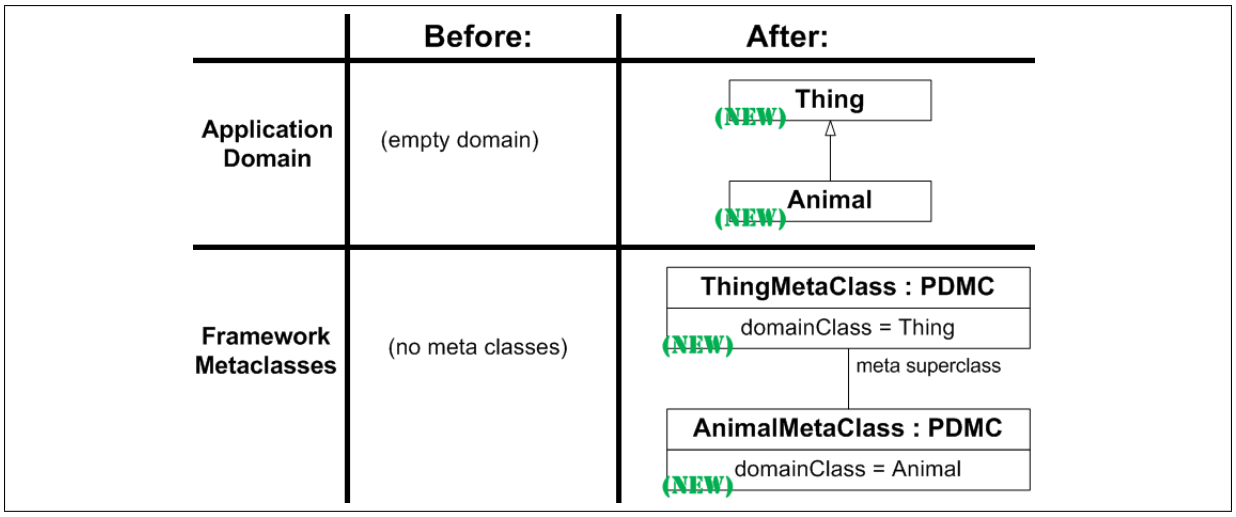

Figure 7.1: The domain changes and the resulting metaclasses of creating a new domain class that extends from another new domain class. **PDMC** stands for PersistentDomainMetaClass.

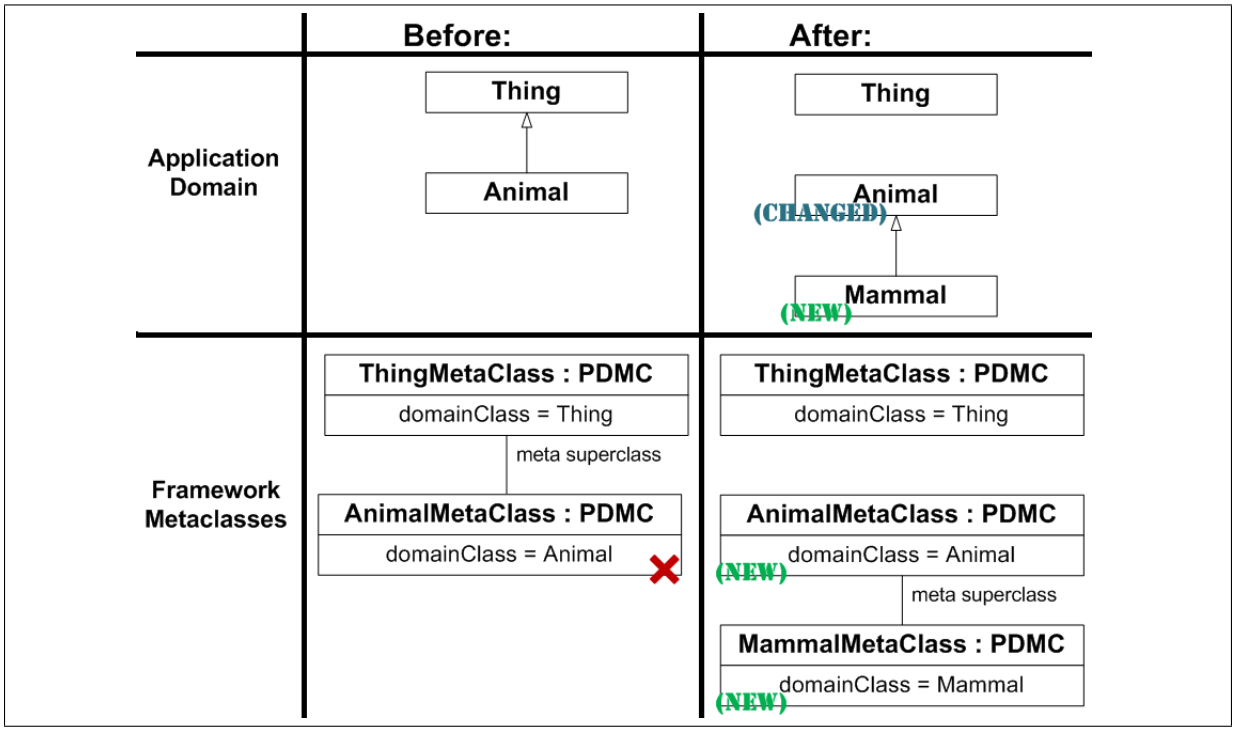

Figure 7.2: The domain changes and the resulting metaclasses of creating a new domain class that extends from an existing domain class whose superclass changed.

the framework deletes the existing metaclass of Animal whose superclass changed. Then, it creates two metaclasses: one for the Mammal, and another for the Animal, whose metaclass was just deleted. Finally, it assigns the Mammal's meta superclass to the Animal class.

In the third test, the developer deletes a class that extended from another deleted class. The required domain changes and the resulting metaclasses are illustrated in Figure 7.3. The framework simply deletes both metaclasses. If it deletes the Animal metaclass first, that deletion immediately causes the cascade deletion of the Mammal metaclass. If the framework deletes the Mammal metaclass first, it individually deletes the Animal metaclass afterwards.

In the fourth test, the developer deletes a class that extended from a class whose superclass changed. The required domain changes and the resulting metaclasses are illustrated in Figure 7.4. The framework

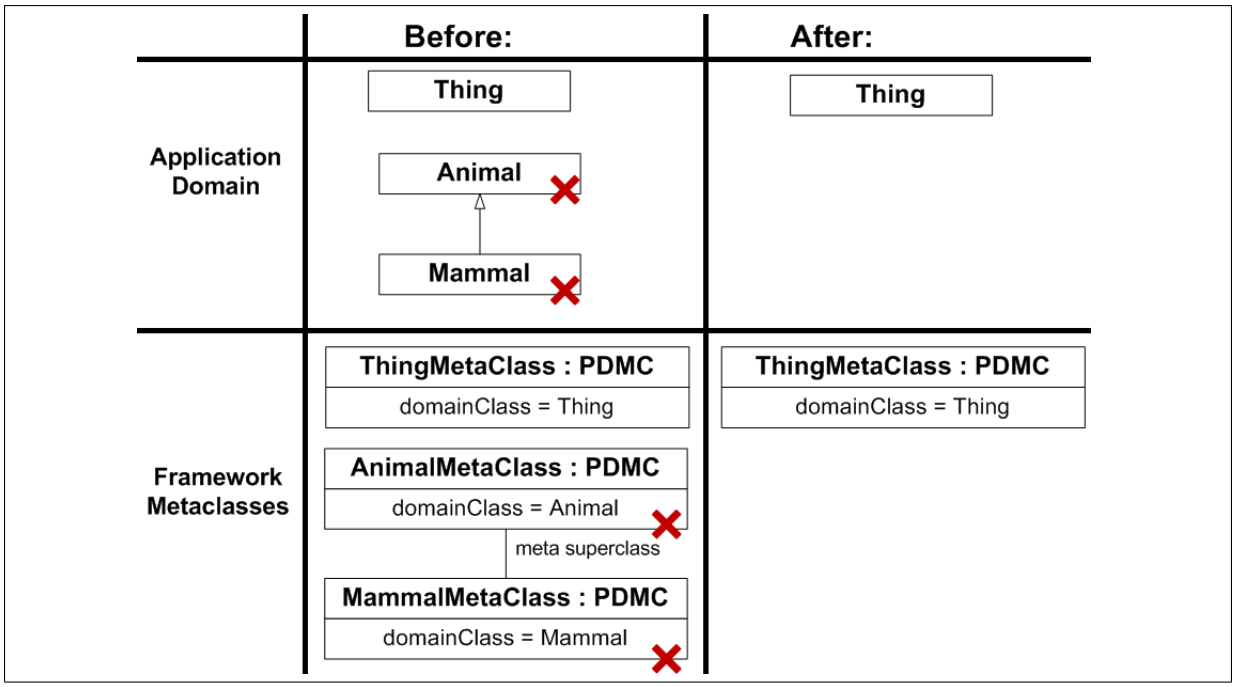

Figure 7.3: The domain changes and the resulting metaclasses of deleting an old domain class that used to extend from another old domain class that was also deleted.

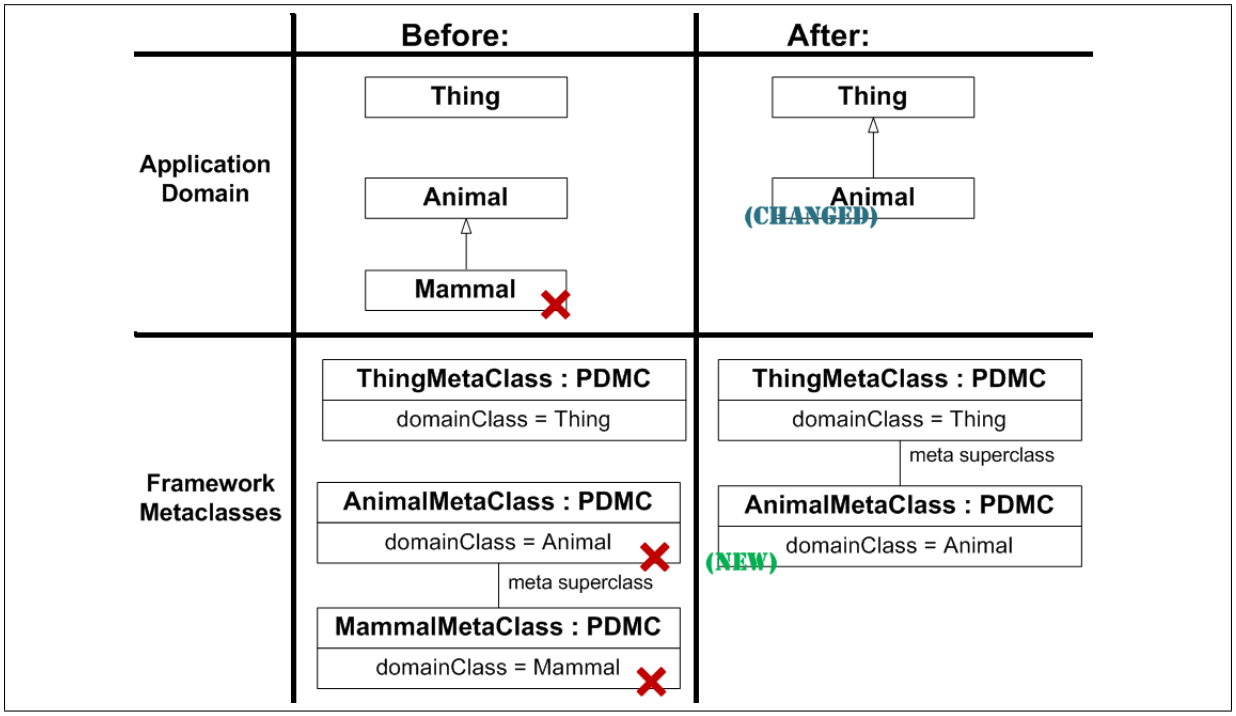

Figure 7.4: The domain changes and the resulting metaclasses of deleting an old domain class that used to extend from an existing domain class whose superclass changed.

first deletes the metaclass of Mammal. Then, it deletes the metaclass of Animal, whose superclass has changed. Afterwards, it creates again the metaclass for Animal. Finally, it assigns the Animal's meta superclass to the Thing class.

In the fifth test, the developer changes the superclass of a class from an old deleted class to a new class. The required domain changes and the resulting metaclasses are illustrated in Figure 7.5. First, the framework deletes the Thing metaclass. In the old hierarchy, the Mammal class used to extend from the

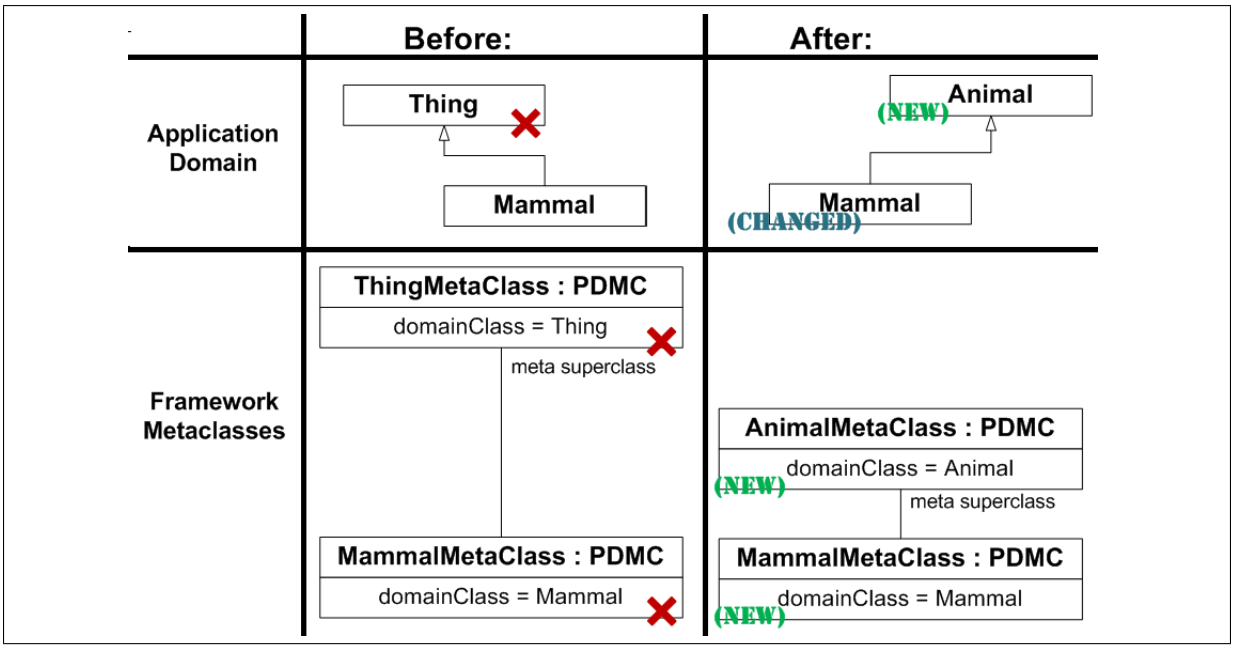

Figure 7.5: The domain changes and the resulting metaclasses of changing the super class of an existing domain class from an old domain class that was deleted to a new domain class.

Thing, so this deletion of the Thing causes the cascade deletion of the Mammal. Then, the framework creates two metaclasses: one for the Animal and another for the Mammal. Finally, it assigns the Mammal's meta superclass to the Animal.

In the sixth and last test, the developer changes the superclass of a class from a class whose superclass changed, to another class whose superclass also changed. The required domain changes and the resulting metaclasses are illustrated in Figure 7.6. First, the framework deletes all three metaclasses, either individually or in cascade. Then, the framework creates the three metaclasses again. Finaly, it assigns the Mammal's meta superclass to the Carnivore, and the Carnivore's meta superclass to the Thing.

No matter what combination of domain class changes the developer might perform, the initialization always keeps its metaclass hierarchy updated. It is also important to test that the list of existing objects of each metaclass is updated during runtime, as the target application creates and deletes its objects.

When the application creates a domain object, it invokes the constructor of the AbstractDomainObject. In turn, this constructor invokes the PersistentDomainMetaObject's constructor and associates the domain object with the meta object. Afterwards, it looks for the metaclass of its domain class and adds the meta object to the list of existing objects of the metaclass. Because the initialization phase has already updated the metaclass hierarchy, the application's runtime is sure to find a metaclass for all of its domain objects.

When the application deletes a domain object, it first disconnects all the relations of that object, and invokes the deleteDomainObject() of the AbstractDomainObject. In turn, this method invokes the delete() of the PersistentDomainMetaObject which disconnects all the relations of the meta object. So, this method also removes the meta object from the relation of existing objects of its metaclass. The initialization phase displays the number of existing objects of each metaclass, which is the result of calling the generated getExistingObjectsCount() method of this relation. So, it displays the number of objects in that relation. This value is always correct, no matter how many simultaneous object creations and deletions each transaction performs.

In the next section, the framework will have to deal with changes to the KnownConsistencyPredicates. The KnownConsistencyPredicates will require the PersistentDomainMetaClasses to be updated, because

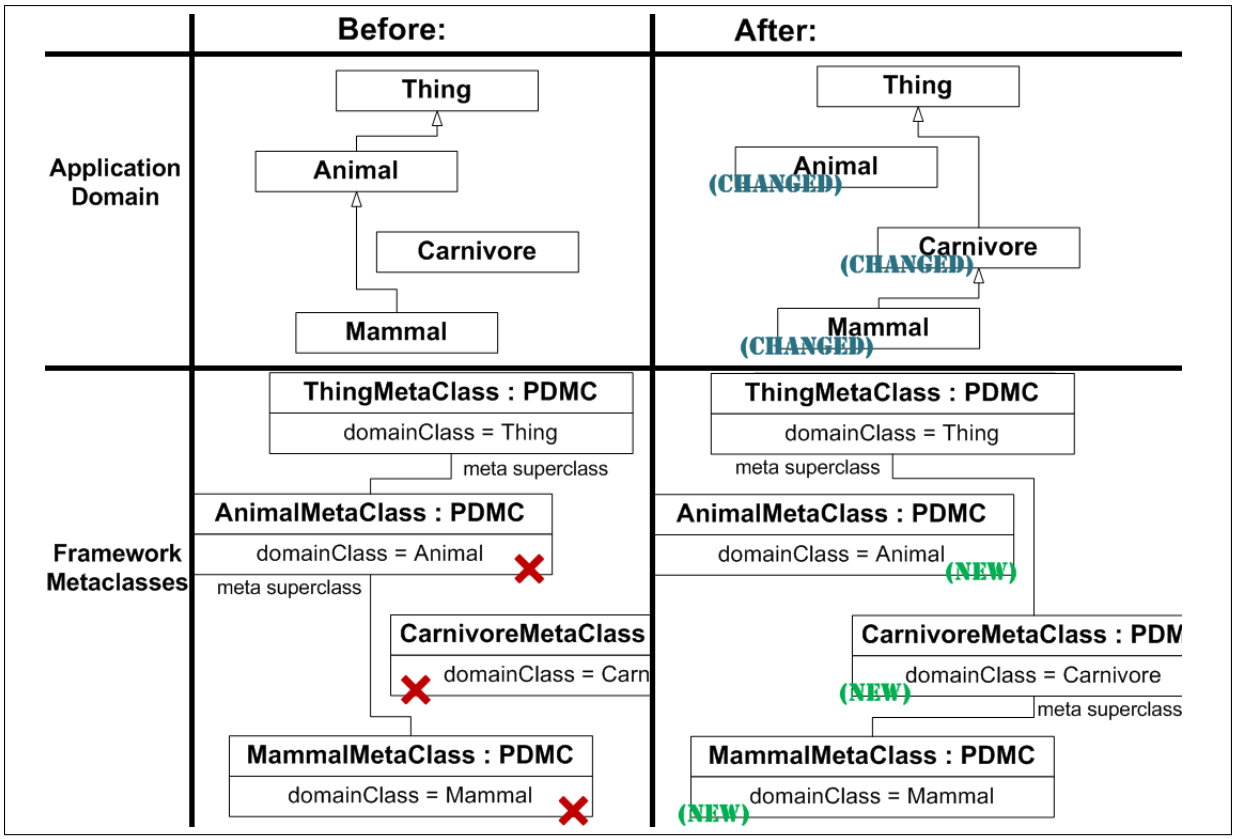

Figure 7.6: The domain changes and the resulting metaclasses of changing the super class of an existing domain class from a changed class to another changed class.

they are declared inside domain classes. Also, they need the list of existing objects updated for a new predicate to be executed at startup for all the objects of its class, so that the framework can build the PersistentDependenceRecords.

7.1.2 Changing the Consistency Predicates

The next step is to update the KnownConsistencyPredicates, whose domain was illustrated back in Figure 5.20, and in Figure 5.25. A classification of code changes was given in Section 5.1. Of these changes, the ones supported are:

- Inserting a new predicate
- Removing an old predicate
- Changing the signature of an existing predicate

All of these operations are performed in the code of domain classes. The framework will detect and respond to these changes according to the implementation described in Section 5.3.2.

However, after a new version of the code is produced, the developer can have made several of these changes simultaneously. Recall that the PublicConsistencyPredicate has a relation to the overridden predicate at a superclass, and the overriding predicates at the subclasses. Still, the PublicConsistencyPredicates are not *hard connected* to each other by this relation. In other words, changes to a certain predicate do not imply structural changes at the overridden predicates at the subclasses. Thus, the goal of this section is to view the reaction of the framework to these 3 types of changes individually. By the end of the initialization phase, the framework must have a correct representation of the KnownConsistencyPredicates. Also, it must have executed the predicates needed, for all the objects of the affected classes.

The initialization phase is followed by a debug step that displays a textual description with each existing KnownConsistencyPredicate, its type, and its overridden predicate (if any). In this section, instead of presenting a transcript of this text, I have chosen to illustrate the contents of the KnownConsistencyPredicates in figures. The initialization also displays which predicates are executed for the objects of which classes. This debugging step allows to confirm that the predicates do execute for the correct classes, and for the total amount of objects of each class.

The most elaborate type of predicate insertion is when the developer creates a new public predicate that has an overridden predicate, and several overriding predicates. The required domain changes and the framework's response are illustrated in Figure 7.7. First, the framework creates a new PublicConsis-

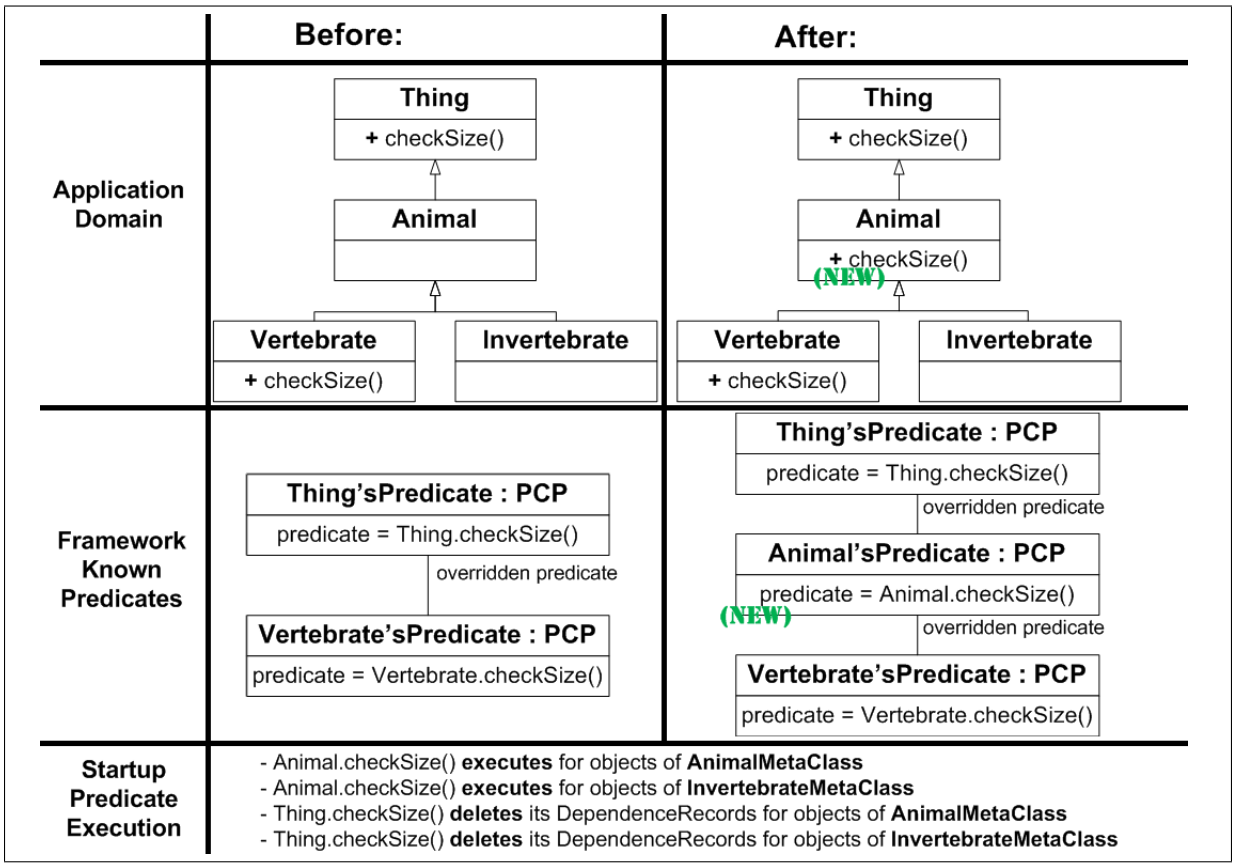

Figure 7.7: The domain changes and the framework's response of adding a new public predicate that already overrides and is overridden. **PCP** stands for PublicConsistencyPredicate.

tencyPredicate for the Animal'sPredicate. Then, it updates the information that the Vertebrate'sPredicate now overrides the Animal'sPredicate, which overrides the Thing'sPredicate. Also, it executes the Animal'sPredicate for the objects of the AnimalMetaClass and the InvertebrateMetaClass. Finally, it deletes the PersistentDependenceRecords of the Thing'sPredicate for the objects of the AnimalMetaClass and the InvertebrateMetaClass.

The developer can also insert a new final predicate that overrides a public predicate. The required domain changes and the framework's response are illustrated in Figure 7.8. First, the framework creates a new FinalConsistencyPredicate for the Animal'sPredicate. Then, it updates the information that the Animal'sPredicate now overrides the Thing'sPredicate. Also, it executes the Animal'sPredicate for the objects of the AnimalMetaClass, the VertebrateMetaClass, and the InvertebrateMetaClass. Finally, it deletes the PersistentDependenceRecords of the Thing'sPredicate for the objects of the AnimalMetaClass, the VertebrateMetaClass, and the InvertebrateMetaClass.

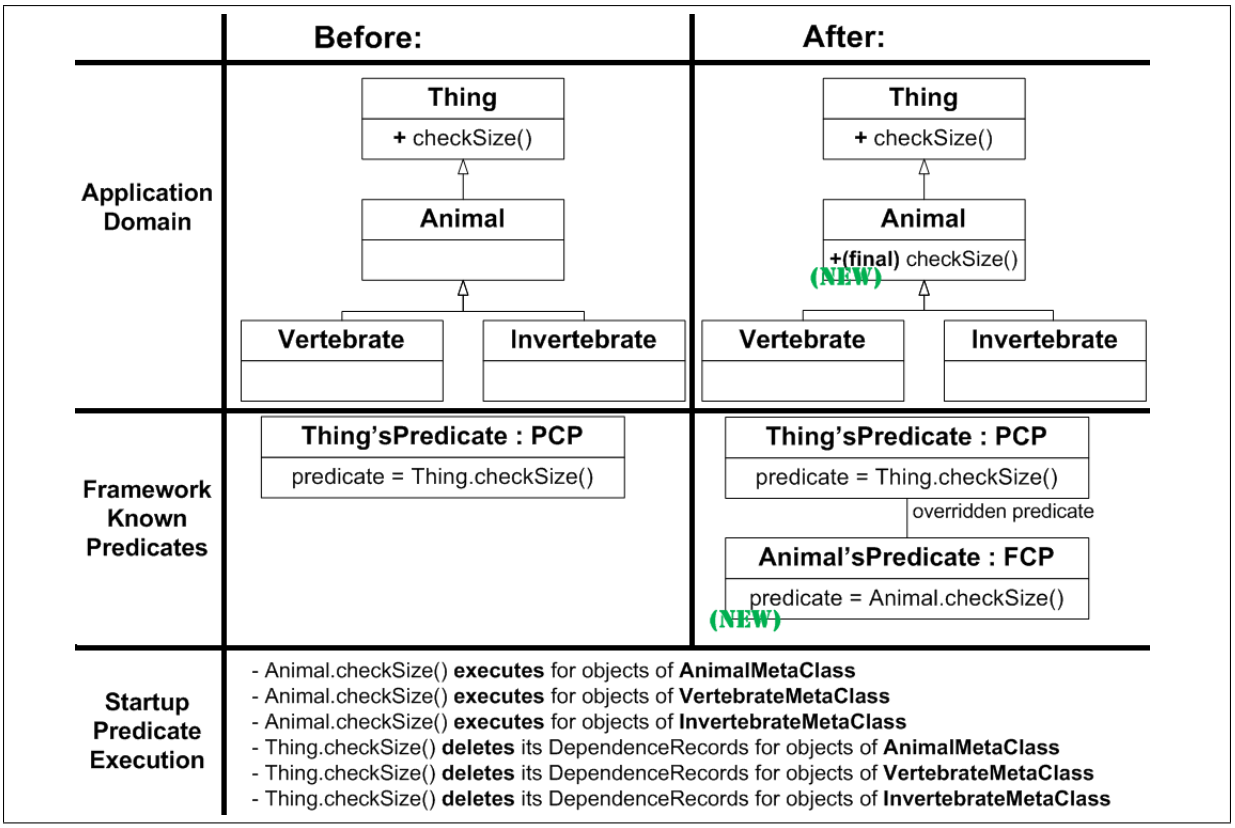

Figure 7.8: The domain changes and the framework's response of adding a new final predicate that already overrides. **FCP** stands for FinalConsistencyPredicate.

The most complex type of predicate removal happens when the developer deletes an old public predicate that used to override and be overridden. The required domain changes and the framework's response are illustrated in Figure 7.9. First, the framework deletes the PublicConsistencyPredicate, and all the PersistentDependenceRecords of the Animal'sPredicate. Then, it executes the Thing'sPredicate for the objects of the AnimalMetaClass and the InvertebrateMetaClass.

Moreover, the developer can change a predicate's signature from public to private. The required domain changes and the framework's response are illustrated in Figure 7.10. First, the framework creates a new PrivateConsistencyPredicate to match the new signature of the Animal'sPredicate. Then, it executes the Animal'sPredicate for the objects of the AnimalMetaClass, the VertebrateMetaClass, and the InvertebrateMetaClass. Finally, the framework deletes the old PublicConsistencyPredicate of the old signature of the Animal'sPredicate, and all the previous PersistentDependenceRecords.

7.2 Validating the Consistency Predicates' Semantics

After the framework's initialization phase is finished, all the metaclasses and consistency predicates are updated. Also, each new predicate has been executed for all the objects that it affect to create the PersistentDependenceRecords to keep them consistent. However, after the initialization is finished, the application starts its normal operation, and transactions start changing the domain data. Because this work has altered the semantics of the consistency predicates, it is important to verify that their behavior coincides with this semantics.

Before this extension, the framework restricted expressiveness of the consistency predicates, which could only access the data of their own object. Now, the developer is free to define the consistency of

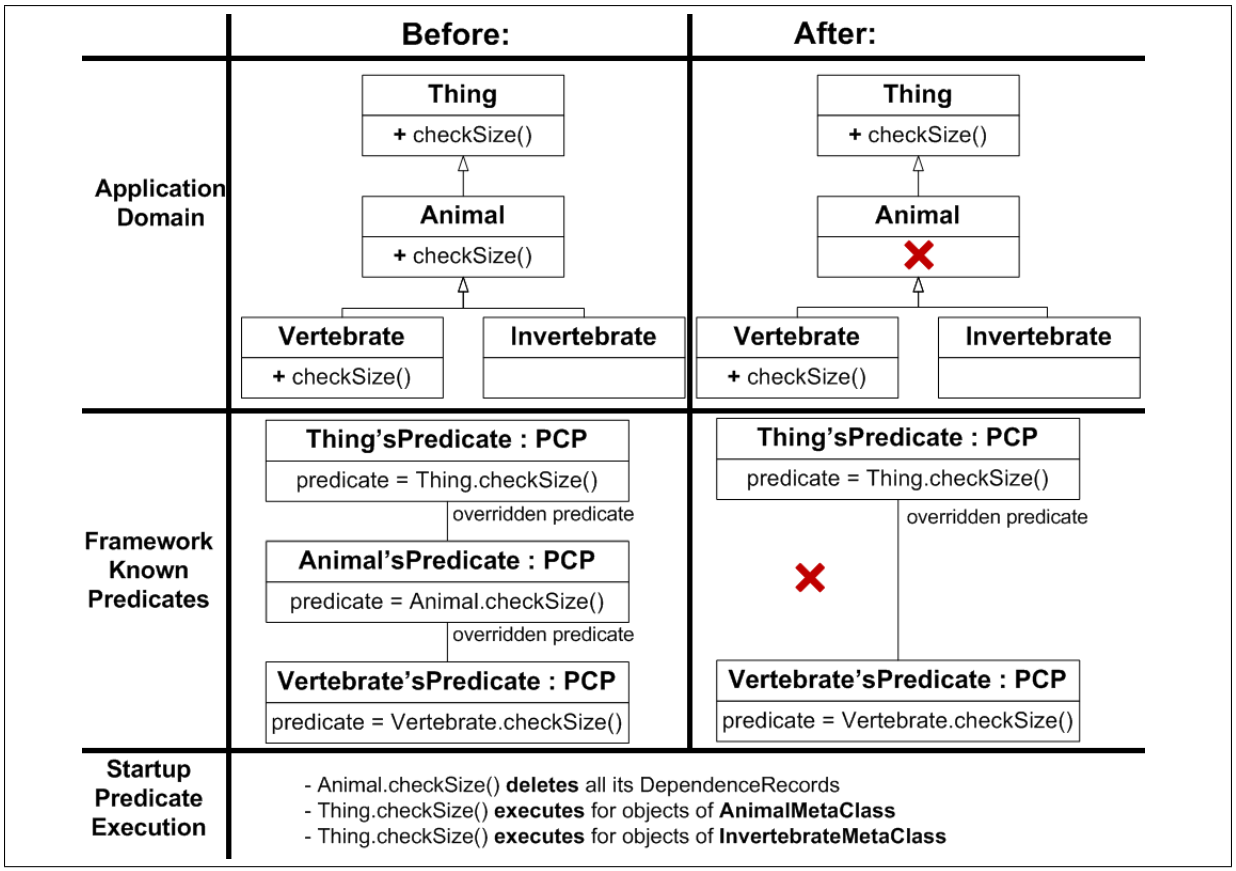

Figure 7.9: The domain changes and the framework's response of removing an old public predicate that used to override and be overridden.

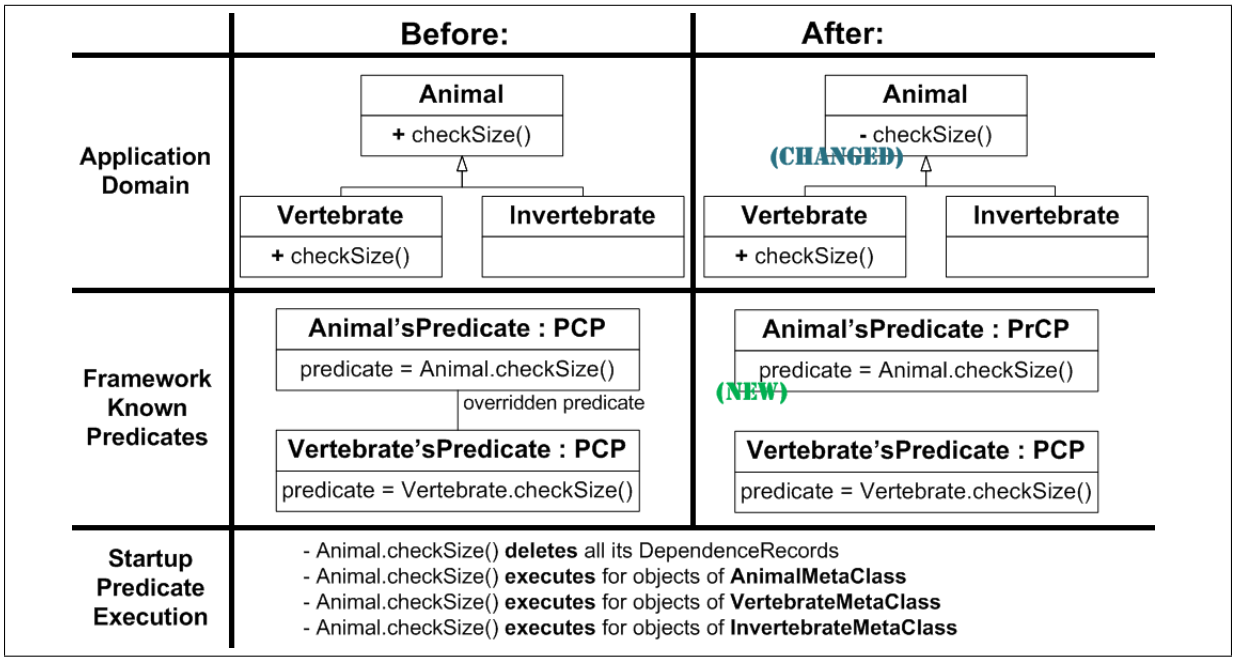

Figure 7.10: The domain changes and the framework's response of changing a public predicate that used to be overridden to private. **PrCP** stands for PrivateConsistencyPredicate.

one object depending on its domain relations and on the state of other objects. Also, before this work, the framework did not persist its DependenceRecords. Now, the framework keeps the PersistentDependenceRecords available even after the application restarts. Finally, the framework now provides optional inconsistency-tolerant semantics. Therefore, there are two kinds of consistency predicates with different

semantics.

7.2.1 Regular Consistency Predicates

The regular consistency predicates are the original ones, that are not inconsistency-tolerant. This work has extended the regular consistency predicates, which can now use domain relations and the state of several objects. Also, the predicates now persist their dependency information, which is kept after the application restarts. So, it is important to verify that the dependencies are correctly recorded and persisted.

During the application's execution write transactions change the domain data. It is also important to verify that these transactions check the correct set of predicates. The transactions can:

1. Change the slots of an object
2. Create a new object
3. Delete an existing object

Note that these data changes are not local. For instance, a transaction can delete an object regardless of whether it was consistent or not. However, the deletion of this object may make inconsistent other objects that depended on it.

Before this work, existing objects that were already consistent according to a new predicate could not be kept consistent by the framework's runtime. This situation was illustrated in Figure 3.1. I have reproduced this situation by creating the Client Sophie and the two accounts from the previous figure. The consistency predicate checkTotalBalancePositive() does not yet exist in the code. However, the client is already consistent; her total balance is 20€. Then, the predicate is introduced in the code, and the application is restarted.

During startup, the framework detects this new predicate and executes it for Client Sophie. This execution creates the PersistentDependenceRecord to keep the client consistent. Afterwards, the Transaction T attempts to withdraw 50€ from an account. However, the PersistentDependenceRecord was stored and is now available, and the information presented in gray in Figure 3.1 now exists. So, Transaction T checks the predicate checkTotalBalancePositive(), which returns false. Then the Transaction T aborts, and Client Sophie is kept consistent.

If the application restarts again, the PersistentDependenceRecord is still kept. The initialization phase does not need to perform any operation. Another transaction that attempts to withdraw 50€ from an account verifies the Client's predicate and aborts. Thus, the correct set of dependencies was recorded and persisted. Also, changes to the slots of an object do cause the verification of predicates on other objects.

Furthermore, a transaction can also create a new account with -30€, and associate it with Client Sophie. This situation is illustrated in Figure 7.11. Even though the new created account is consistent (it does not define any predicates), this operation makes the client's total balance negative. Because Transaction T changes the client's set of accounts, it rechecks the Client's predicate. Then, Transaction T aborts, and Account C is not created.

Finally, a transaction can attempt to delete Account A, independently of its state. This situation is illustrated in Figure 7.12. However, the deletion of this object makes the client's total balance negative. Because Transaction T changes the client's set of accounts, it rechecks the Client's predicate. Transaction T aborts, and Account A is not deleted.

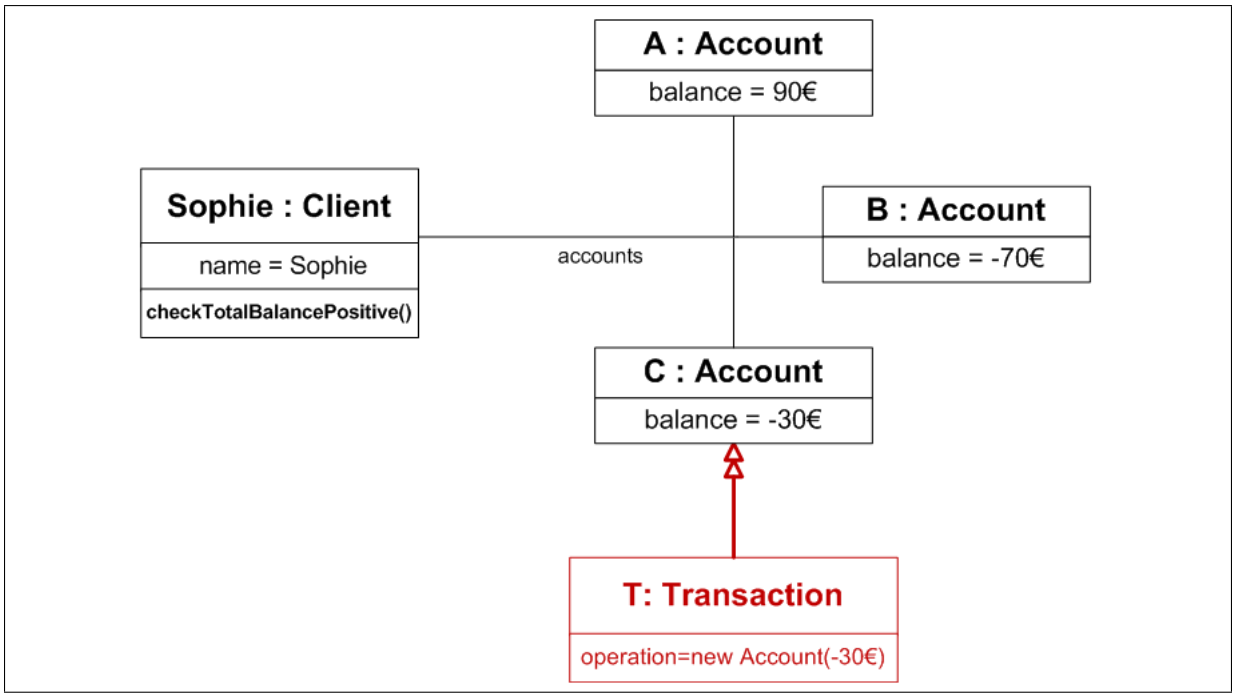

Figure 7.11: A transaction that attempts to create an object, which makes another object inconsistent. Transaction T **aborts**.

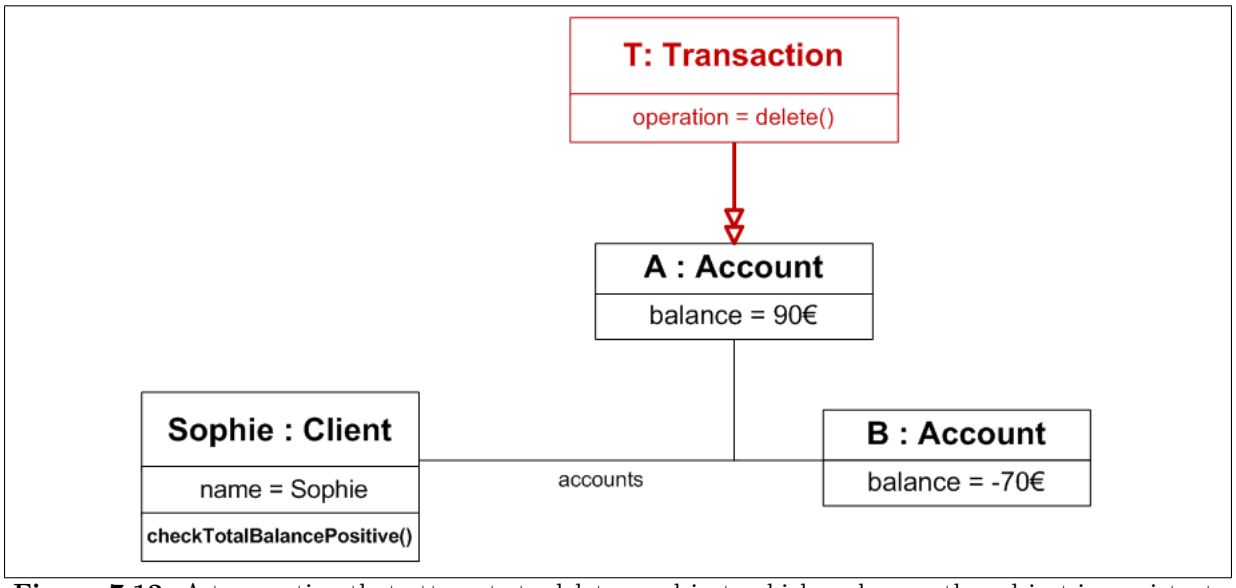

Figure 7.12: A transaction that attempts to delete an object, which makes another object inconsistent. Transaction T **aborts**.

7.2.2 Inconsistency-Tolerant Predicates

It is central to test the difference in semantics from regular predicates to inconsistency-tolerant predicates. To do so, the following discussion will be concerned with the operations identified in Section 6.1.1. In particular, a transaction can:

1. Corrupt a consistent object
2. Fix an inconsistent object
3. Keep an object consistent
4. Keep an object inconsistent

A regular consistency predicate will abort transactions that corrupt objects, or keep them inconsistent. It will commit transactions that fix objects or keep them consistent. These situations can also be verified thanks to the example illustrated in Figure 3.1.

So, I have recreated the Client Sophie and the two accounts from the previous figure. The consistency predicate checkTotalBalancePositive() does not yet exist in the code. However, the client is already consistent; her total balance is 20€. Then, the predicate is introduced in the code, and the application is restarted. During startup, the framework detects this new predicate and executes it for Client Sophie. This execution creates the PersistentDependenceRecord to keep the client consistent.

After reproducing the situation in Figure 3.1, Transaction T that attempts to withdraw 50€ from Sophie's Account B would corrupt the client and make it inconsistent. However, the PersistentDependenceRecord is now available, and Transaction T modifies Account B, so it checks the Client's predicate, which returns false. Transaction T attempts to corrupt the client and therefore it aborts.

Figure 7.13 illustrates a similar situation with an already inconsistent client. Transaction R attempts

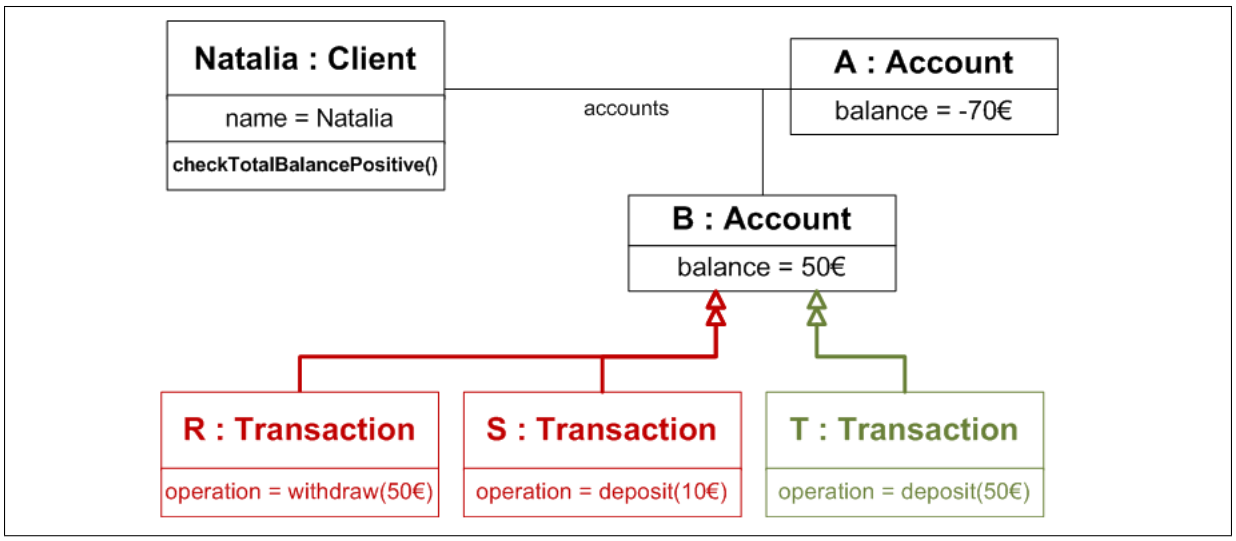

Figure 7.13: Three transactions that attempt to keep a client inconsistent, or correct it. The predicate is **not** inconsistency-tolerant. Transactions R and S **abort**. Transaction T **commits**.

to **withdraw** 50€ from Natalia's Account B, and would keep the client inconsistent. Her total balance was -20€ before, and would be -70€ if Transaction R would commit. But the predicate depends on Account B, which Transaction R has modified. So, Transaction R checks the predicate, which returns false. Because the predicate is not inconsistency-tolerant yet, Transaction R simply aborts. Transaction S also keeps the client inconsistent and aborts.

Transaction T, however, attempts to **deposit** 50€ on Natalia's Account B, and would fix the client and make her consistent. The predicate depends on Account B, which Transaction T has modified. So, Transaction T checks the predicate, which returns true. Therefore, Transaction T commits. Natalia's total balance is now equal to 30€.

Afterwards, if Transaction T is repeated to deposit another 50€ on Natalia's Account B, the transaction would keep the client consistent. Thus, the same situation happens. The transaction checks the predicate again, which returns true, and the transaction commits.

A predicate that is tolerant to inconsistencies will follow the same semantics of a regular predicate, except for a transaction that keeps an object inconsistent. Whereas a regular predicate would abort this transaction, an inconsistency-tolerant predicate will allow the transaction to commit.

To illustrate this semantics, I have reproduced again the situation of Figure 7.13. Client Natalia's total balance is already -20€ , and now an inconsistency-tolerant checkTotalBalancePositive() predicate is introduced in the code. After the application restarts, the framework executes the new predicate for Client Natalia. This execution creates the PersistentDependenceRecord that indicates that the client is already inconsistent.

Transaction S attempts to deposit 10€ on Natalia's Account B and keeps the client inconsistent. Her total balance was -20€ before, and will be -10€ after Transaction S takes effect. The predicate depends on Account B, and the transaction writes to this account. So, Transaction S checks the predicate, which returns false. However, because the predicate is tolerant to inconsistencies, Transaction S reads the consistent slot of the predicate's PersistentDependenceRecord. This slot indicates that the client was already inconsistent before. Thus, Transaction S commits, because it does not create new inconsistencies.

Unfortunately, Transaction R is in the same situation, and also commits. Arguably, the example of Figure 7.13 illustrates a predicate that deals with sensitive information. Transaction R withdraws money from a client whose total balance is already negative, and makes the client even less consistent in quality. In this case, the programmer might not want the predicate to be tolerant to inconsistencies.

However, if the predicate would not be inconsistency tolerant, then Transaction S would also abort. Transaction S deposits money on a client whose total balance is negative. So, even though Transaction S does not deposit enough money to make the client immediately consistent, it does make the client more consistent in quality. This example illustrates a situation where it is not easy to decide whether or not to use inconsistency-tolerance semantics.

7.3 False Positives and Negatives

Two other important aspects of this extension's implementation are the false positives and the false negatives. These aspects were briefly described in Section 2.3.2. A **false positive** happens when the framework evaluates a consistency predicate without need. This mishap may cause performance problems. A **false negative**, however, would happen if the framework would **not** execute a predicate that needed to be executed. This mishap can cause the domain data to become inconsistent.

7.3.1 False Positives

The false positives have a negative effect on the framework's performance. However, they do not cause errors in its workflow, and do not cause the application's data to become inconsistent.

As described in Section 3.3, keeping a low number of false positives is the least critical requirement of this work. During the course of this work, I was first concerned with providing the programmer with new tools in regard to the consistency predicates. Then, I was concerned with avoiding the false negatives at all costs, because they may cause the data to become inconsistent.

The original consistency predicates designed in the JVSTM did not support the automatic detection of code and structure changes, and did not persist the DependenceRecords. With this work, the consistency predicates now support these features, at the cost of an increased number of false positives overall.

Therefore, the reduction of false positives can still be subject to multiple improvements. This text provides an analysis of situations when false positives happen in the current implementation. Hopefully, with a list of these situations, it will be easier to improve this aspect of the framework in the future.

Most of the false positives of this work were induced by the code detection techniques designed in Chapter 5. In particular, the first limitation of the framework in regard to the false positives is due

to the PersistentDomainMetaClasses. When the developer deletes or changes the superclass of a domain class, the framework deletes the entire PersistentDomainMetaClass hierarchy. Most importantly, this deletion also causes the deletion of all KnownConsistencyPredicates declared in that hierarchy, and all their PersistentDependenceRecords.

A class that changes its superclass may be affected by a different set of predicates. More, it may now inherit a different set of domain slots, relations and behavior. It can happen that the old predicates that this class declared used the previously inherited slots and behavior, which may now have different contents and semantics.

However, some of the predicates of that class may not be affected by these changes at all. In particular, this statement is true for those that used only the slots, the relations, and the auxiliary methods of their class alone. This situation is illustrated in Figure 7.14. The implementation of closedAccountHas-

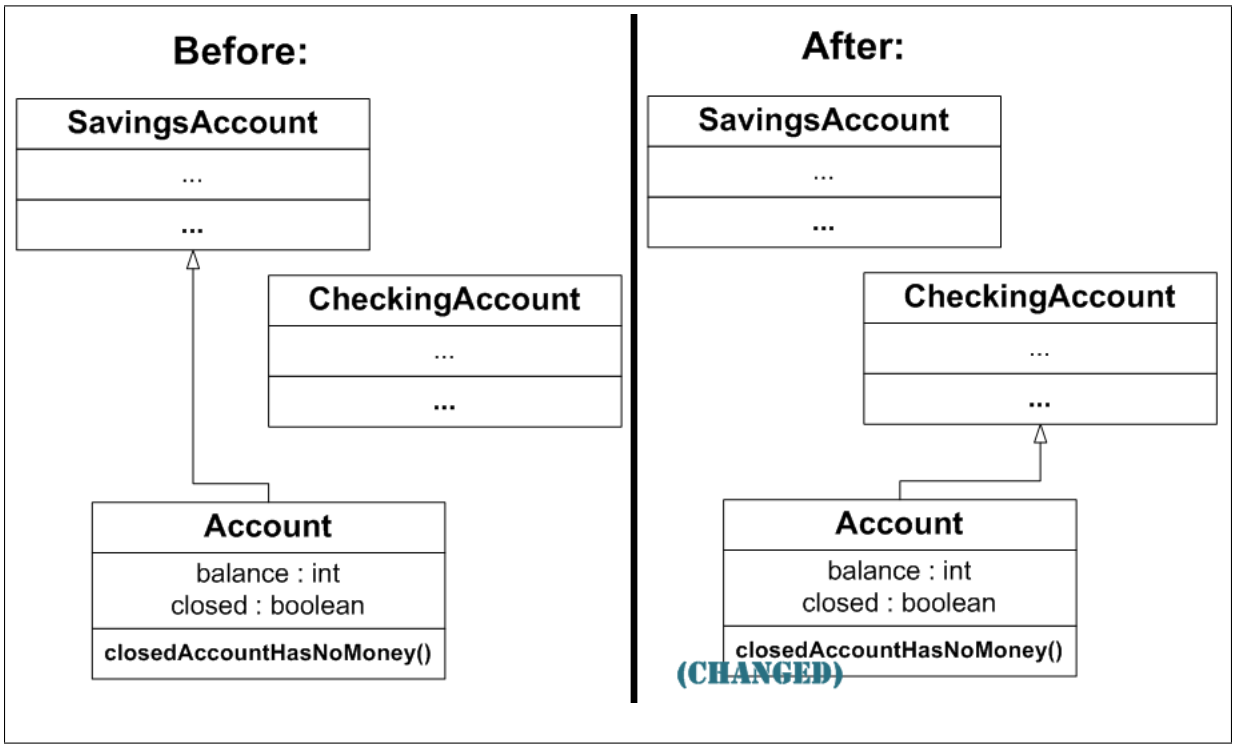

Figure 7.14: Example of a class hierarchy change with a predicate that is not affected by inheritance. The implementation of closedAccountHasNoMoney() was shown back in Figure 2.6.

NoMoney() was shown back in Figure 2.6. This predicate does not use any of the inherited contents of its class, so, it is not affected by these hierarchy changes. In this situation, its KnownConsistencyPredicate could have been kept, along with its PersistentDependenceRecords.

Because in the implementation of this work the whole metaclass hierarchy is discarded, predicates in this situation are also discarded. Then, the framework rebuilds the metaclass hierarchy, and detects the previously deleted predicates as new predicates. These *new* predicates are then executed for every existing object of that class.

The current implementation was designed with this limitation for simplicity purposes. It is difficult to detect predicates in this situation, because the metaclasses do not hold information of which slots they contain, and which are inherited. Also, the framework does not yet have the means to determine which auxiliary domain methods does a predicate use. However, the following Chapter 8 provides an insight in how to detect the use of auxiliary domain methods.

The second limitation in regard to the false positives is due to the model of the KnownConsisten-

cyPredicates. When the developer changes a predicate's signature, the framework treats this situation as an old predicate being removed, and a new predicate being inserted. Most importantly, this situation causes the deletion of the old KnownConsistencyPredicate, and all its PersistentDependenceRecords.

A predicate whose signature has changed may have a different execution flow. This scenario was explained in Section 5.1.1. However, not all method modifiers affect a predicate's execution flow. Note that, as illustrated in Figure 5.3, a predicate whose visibility is changed simply affects a new set of classes.

Still, in the current implementation, the predicate is discarded, together with its PersistentDependenceRecords. Then, the framework detects this predicate as a new predicate, and reexecutes it for the whole class hierarchy. In Figure 5.3, the predicate has already executed for all the objects of classes Animal and Hamster before. After this new detection, the framework will needlessly execute the predicate for those objects again.

Again, the implementation of this work was designed with this limitation for simplicity purposes. The current implementation allows to detect and respond to signature changes in a generic way, without having to identify each method modifier individually. Moreover, the framework cannot yet detect changes to the inner code of methods (see Chapter 8). So, a developer can choose to change the signature of a predicate to manually force the framework to reexecute it. This workaround is described in Appendix A.3.

The third and last known limitation in regard to the false positives is due to the granularity of dependencies of the PersistentDependenceRecord. The original model of the DependenceRecord, shown in Figure 2.9, registers dependencies to VBoxes. If each VBox stores a single domain slot, the consistency predicates used to register fine-grained dependencies to slots. However, the model of the PersistentDependenceRecord, shown in Figure 5.9, registers dependencies to objects. So, the consistency predicates now store coarse-grained dependencies to all the slots of domain objects.

In the current implementation, a predicate that reads a slot of an object will register a dependency to the object. Then, a transaction that modifies a different slot of that object will recheck the predicate needlessly. So, this implementation causes false positives and executes more predicates than necessary.

As Section 5.2.2 has shown, it is not currently possible to create a dependency to a VBox, because it is not a domain entity. However, each PersistentDependenceRecord registers less dependencies overall. If a certain predicate reads several slots of an object, its PersistentDependenceRecord will only store one dependency to that object. Thus, this property reduces the memory requirement of the dependence records, because, in general, less dependencies are stored.

Note that the first two kinds of false positives only affect the performance of the initialization phase. Only the third kind of false positives affect the application's runtime. None of these limitations were dealt with in the present work because they were not considered as a priority. The priority of this work was to avoid the false negatives at all costs.

7.3.2 False Negatives

It may happen that, at a certain point in time, the framework is not able to execute a certain predicate, which needed to be executed. This situation is a false negative, which can cause erroneous workflows and lead to inconsistent data. The execution of predicates is needed for two reasons:

1. To prevent transactions from corrupting domain objects
2. To update the PersistentDependenceRecords with the latest data dependencies

Allowing transactions to corrupt objects is obviously detrimental. Allowing the PersistentDependenceRecords to become outdated may cause more false negatives in the future, which may then corrupt

domain objects.

In this work, there are only two situations that can cause false negatives. The first situation is due to offline data changes. When the application stops running, the developers may make changes directly to the data in the database. Obviously, in this case, the framework has no control over these changes.

This change may have inserted inconsistencies in the data. Obviously, because the application is offline, it is impossible for the framework to check any predicates. But even if offline data changes do not insert inconsistencies directly, the PersistentDependenceRecords may have become outdated. With the PersistentDependenceRecords outdated, when the application restarts, the framework can no longer guarantee that it keeps their objects consistent. Thus, offline data changes can cause further false negatives that create inconsistencies later.

This limitation is a complex situation that the framework cannot easily solve. It is beyond the scope of this work to solve this problem, even though it can cause false negatives. Therefore, currently, it is up to the developer to change the domain data only using the framework, and the generated setter methods.

As several sections of this work have shown, these extensions contain a major limitation that can cause false negatives. This second situation happens whenever a developer changes the inner code of a consistency predicate or an auxiliary method. With the extensions described in this dissertation, the framework detects only changes to a method's signature, but not to its body.

However, if the developer changes a predicate's body, the predicate may have a different execution flow. So, its old PersistentDependenceRecords have become outdated, which may cause false negatives in the future. Because the framework does not yet detect code changes to that predicate's method body, the initialization phase does not know that it should reexecute the predicate.

The developer can work around this problem by manually changing the signature of the predicate whose code changed. This work around was described in Section 5.2.4. It is trivial for the developer to use this work around, except for complex predicates that use auxiliary methods. So, I argue that, in the future, the framework should be able to detect these code changes. Indeed, this limitation is a priority, and is the first and foremost approach described in Chapter 8.

Still, the developer can prevent both of these situations. Assuming that the developer keeps these cases from happening, the Fénix Framework does not allow transactions to corrupt objects. If a transaction attempts to modify an existing consistent object and make it inconsistent, the transaction will abort. Even if it is inconsistency-tolerant, the transaction checks that the object was consistent, and does not allow the object to become inconsistent. Also, if a transaction creates a brand new object that is inconsistent from the start, it will abort and the object will not be created.

Therefore, this semantics have the advantage of providing an overall monotonic progress towards full consistency. In other words, the number of inconsistencies of the application as a whole will either be constant, or decrease over time. So, the number of **inconsistencies monotonically decrease**. The application will not get more inconsistent in quantity.

This property is concerned only about the absolute number of inconsistencies. When the quality of each inconsistency is under discussion, this property does not always hold. If the developer uses inconsistency-tolerance, the application can get more inconsistent in quality. Nevertheless, as discussed in Section 6.1.3, it is not the framework's purpose to measure the quality of inconsistencies.

Also, another apparent exception to this property is related to code changes. Consider, for instance, a certain consistency predicate that is correctly implemented and a collection of objects that are all consistent. If the developers inadvertently introduce a bug in the code of this predicate, it might now

interpret all the objects as inconsistent. Even when they introduce a new predicate that is correctly implemented, the initialization may find several already inconsistent objects.

Thus, code changes can increase the absolute number of inconsistencies. However, the application data still did not get more inconsistent. These inconsistencies already existed; only now is the developer able to detect them. Moreover, it is not the framework's purpose to detect bugs in a new version of the code.

Chapter 8

Future Work

This chapter suggests a few future modifications that could be made to the Fénix Framework either to mitigate some of the limitations of the present work or to further enhance the consistency predicates.

8.1 Method Code Changes

The foremost limitation that several sections of this work describe is the lack of detection of changes to the inner code of methods. The present work is capable of detecting new predicates inserted, old predicates removed, and predicates whose methods' signatures are changed. New predicates are executed for all objects of their classes. This first execution builds the PersistentDependenceRecords that will keep those objects consistent according to the new predicate. The predicates whose signatures have changed, and might have a different execution flow, are reexecuted for all the objects involved. This reexecution keeps the PersistentDependenceRecords up-to-date according to the last version of the domain code.

However, the framework is currently unable to detect changes to a predicate's method body that do not change its signature. Furthermore, if a certain predicate invokes an auxiliary method, changes to that method can influence the predicate's execution flow. The framework does not detect those changes either. So, the most important future work to consider is to enable the detection of implementation changes inside predicates and domain methods.

Recall from Section 5.2 that the framework needs a persistent representation of the domain classes to detect changes in the domain class hierarchy. Likewise, to detect changes in methods, the framework would need a new entity to represent domain methods. Each predicate would also have a representation of its own method, in case its own code changes.

A possible name for this new entity would be KnownDomainMethod. Much like the KnownConsistencyPredicate, the KnownDomainMethod would have a relation to the PersistentDomainMetaClass that declares it. It would have to store the method's signature in a domain slot. Furthermore, it would also need to store a representation of the method's bytecode. The bytecode of a method can be obtained by using tools such as ASM[1]. It could be stored in a byte array domain slot.

The framework's initialization would then iterate through all the metaclasses, and for each domain class, iterate through all the declared methods. For each of those methods, a KnownDomainMethod would be created, which would represent the detection of a brand new method in the code. The following initialization would already see the existing KnownDomainMethods and would not need to create them again.

[1] http://asm.ow2.org/

Whenever the source code of a method would be changed, the framework would no longer match it to the values of the previously persisted method. Thus, the framework could detect which domain methods changed, including both auxiliary and predicate methods. Then, after determining that a certain method changed, it would only need to trace it back to the predicates that used it.

To do so, the PersistentDependenceRecord, which represents the last execution of a predicate for an object, would need to store further information. In the present work, the dependence record stores the data dependencies, and the last (consistent or not) result of the predicate execution. But it would also need to link to the KnownDomainMethods that were used during that execution.

So, for each consistency predicate that executes, the active transaction would need to detect KnownDomainMethods as they are invoked. One way to allow this detection is to modify the bytecode of each KnownDomainMethod processed during the initialization of the framework. At the beginning of each domain method, the framework would inject a call to a special method of the framework's TopLevelTransaction.

Each identified KnownDomainMethod would call a registerUsedDomainMethod() operation at the active transaction. The transaction would collect a complete set of domain methods used as the predicate is being executed. Once the predicate is finished, the transaction creates a PersistentDependenceRecord as before. But now, for each used domain method, the dependence record would contain a relation to the corresponding KnownDomainMethod.

In summary, with these changes, the transaction would detect any methods being invoked during the execution of a predicate. The PersistentDependenceRecord would be linked to all the KnownDomainMethods that it used. The framework's initialization could detect code changes in the KnownDomainMethods and reexecute the predicates involved. Thus, the framework would now support the detection and respond to inner code changes made to the implementation of auxiliary and predicate methods. With this implementation, the framework would cover all the types of code changes identified in Section 5.1.3.

The enhancements to the framework's domain model needed to allow this workflow are illustrated in Figure 8.1. These modifications would detect changes only for methods inside domain classes. I believe that detecting changes to domain methods alone would already cover the most common implementations of predicates.

8.2 Inconsistent New Objects

As this work has shown previously, the Fénix Framework does not allow a consistent object to become inconsistent. A transaction that attempts to corrupt a consistent object will abort. Even if it is inconsistency-tolerant, the transaction will check that the previous state is consistent, and not allow the object to become inconsistent.

Also, if a transaction creates a brand new object that is inconsistent from the start, it will abort and the object will not be created. The transaction creating a new object considers that the previous state is consistent. The framework assumes that the absence of objects is a consistent state.

However, the development team may not always find it desirable to forbid the creation of inconsistent new objects. Typically, for migration purposes, the developers may find it useful to import a large collection of objects even if some are inconsistent. This migration would require the framework to create a few inconsistent new objects. After importing the objects, the developers could then obtain the inconsistent ones and correct them in time.

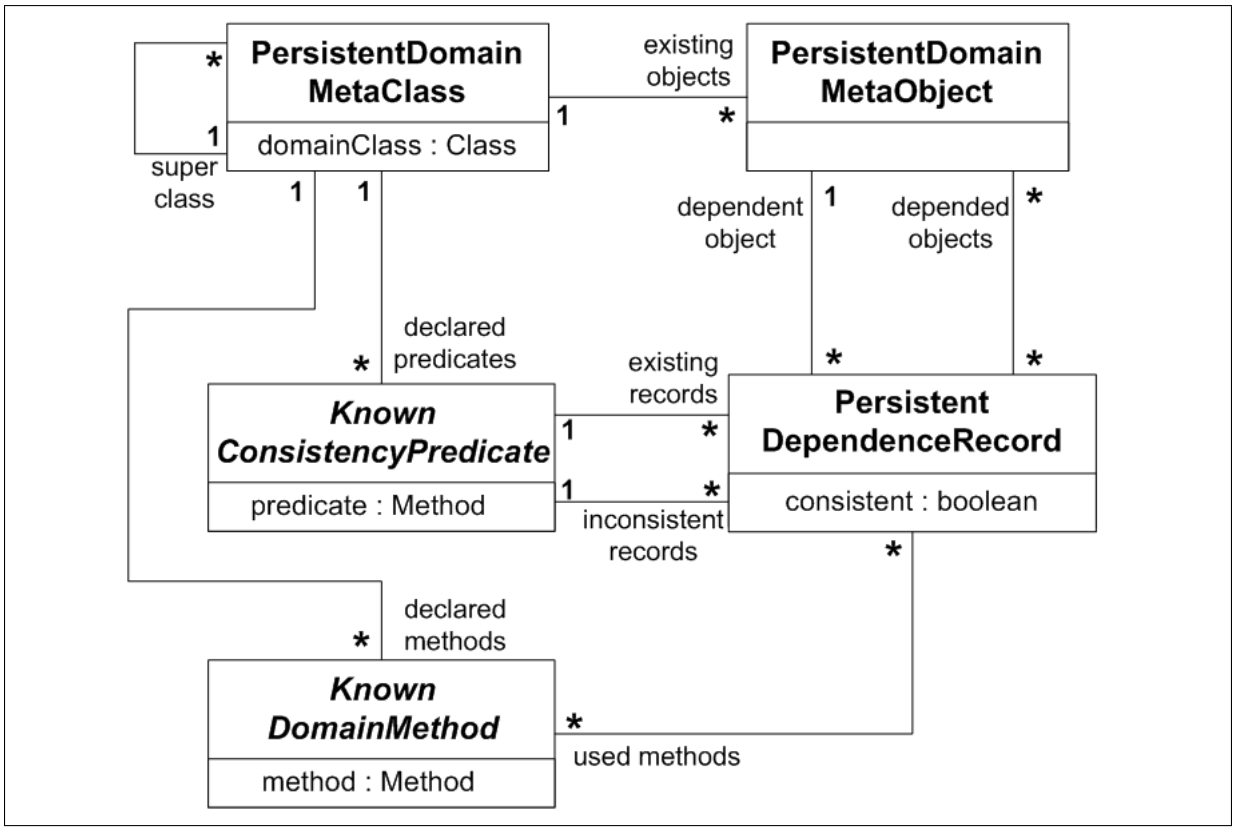

Figure 8.1: Hypothetical Fénix Framework domain with the KnownDomainMethod.

With the present work, the developers are forced to remove the predicate, import the objects, and then insert the predicate again. This last insertion would force the framework to reexecute the predicate for all the objects of that class, not only the imported ones. Also, this sequence requires two deploys of two different versions of the code. I argue that it would be of value to provide the development team with an easier way to allow inconsistent new objects.

However, allowing inconsistent new objects would disrupt the property of monotonically decreasing inconsistencies discussed in Section 7.3.2. Obviously, if the framework allows their creation, the number of inconsistencies will increase. I argue that the decision of whether or not to allow inconsistent new objects for a certain predicate should be left to the developer.

One possible way to work around this issue is to create another parameter in the @ConsistencyPredicate annotation. This parameter could be called allowInconsistentNewObjects, and, much like the inconsistencyTolerant, it would default to false. If the developer would choose so, he could explicitly make a predicate to allow inconsistent new objects.

With this parameter, a predicate can simultaneously allow inconsistent new objects and be inconsistency tolerant. In this case, the predicate will allow inconsistent new objects to be created and modified freely, even if they are not immediately corrected. After those new objects are corrected and made consistent, they cannot be made inconsistent again.

Also, a predicate can allow inconsistent new objects, but not be inconsistency tolerant. In this case, the predicate will allow inconsistent new objects to be created. But after they are created, any transaction that changes them will abort, unless they are immediately corrected. After those new objects are corrected and made consistent, they cannot be made inconsistent again.

8.3 Automatic Inconsistency Fixes

With the present work, the framework already keeps full information about all the existing inconsistencies, and makes them accessible to the developer. This accessibility, explained in Section 5.2.6, allows the developer to add a predicate now and fix the inconsistencies later. To fix the inconsistent objects, the developer would need to write specific code that would change the values of each object. He would most likely base these changes on the domain information available for each object: the domain slots and relations.

However, there are some cases where a certain predicate detects inconsistent objects that can all be corrected in the same way. If each predicate implements only one consistency rule, the implementation of the correction is often the same for every object of that class. In this scenario, it can be useful for the framework to provide an automatic way to correct objects that are inconsistent according to each predicate.

8.3.1 How To Fix Inconsistent Objects?

As future work, I propose to create yet another parameter in the @ConsistencyPredicate annotation. This parameter could be of type String and called fixingMethod, and it would contain the name of the method to perform the correction. The developer would have to implement that desired method with the common implementation that would fix any object with that detected anomaly. An example of the use of a fixing method is illustrated in Figure 8.2.

```
@ConsistencyPredicate(fixingMethod = "fixTotalBalancePositive")
public boolean checkTotalBalancePositive() {
  return (getTotalBalance() >= 0);
}

public void fixTotalBalancePositive() {
  int totalBalance = getTotalBalance();
  if (totalBalance < 0) {
        new Debt(this, -totalBalance);
        setTotalBalance(0);
  }
}
```

Figure 8.2: Hypothetical use of a fixing method inside the class Client.

The framework would use reflection to invoke that fixing method for objects that are inconsistent according to that predicate. The fixing method would then modify the values of the object (and associated objects) in an attempt to make it consistent. But nothing guarantees that, after the method executes, the object will indeed be consistent. It can happen that after performing the changes, the object is still inconsistent.

The fixing method will run inside a write transaction that will collect a read and write set. During the execution, the transaction will fill the write set, that will store all the objects that were changed. Afterwards, the dependence records depending on the objects in this write set will reexecute their predicates.

If every changed object is consistent, the fixing write transaction obviously commits. In this case,

the fixing transaction successfully corrected the inconsistent object without making any other object inconsistent. Potentially, it may have even fixed other objects.

Nevertheless, if the fixing transaction creates new inconsistencies, it will abort. In this case, the transaction may have fixed the inconsistent object at the cost of corrupting other objects. So, it might have increased the number of inconsistencies and made the application's domain data less consistent overall.

However, it can also happen that some of the changed objects are inconsistent, according to predicates that are inconsistency-tolerant. If those objects were already inconsistent before, the fixing transaction also commits. In this case, the fixing transaction modified objects in an attempt to correct them, and it was not necessarily successful. But the transaction was not unsuccessful either, because it did not create new inconsistencies. Thus, the semantics of the inconsistency-tolerant predicates is preserved, and the fixing transaction commits.

Also, it can happen that the fixing transaction changes inconsistent objects according to regular predicates that are not inconsistency-tolerant. In this case, even if the objects were already inconsistent before, the transaction also aborts. Thus, the semantics of regular predicates that are not inconsistency-tolerant is preserved.

8.3.2 When To Fix Inconsistent Objects?

Lets assume that the fixing methods are correctly implemented and are able to fix inconsistent objects. Obviously, a fixing method is also a regular domain method that the developer can manually invoke. This invocation may be helpful to allow the developer to decide exactly when to fix each object. But, if desired, the development team can make the fixing method private to avoid manual invocations. The original purpose of the fixing methods is for the framework to invoke them automatically. But when should the framework execute a fixing method? The following text describes four possible execution policies, each with its own advantages and drawbacks.

The first execution policy is a lazy approach that is called **fix-on-reads**. The framework would fix inconsistent objects as soon as any transaction attempts to read their values. For instance, if a transaction attempts to read the balance of an account that is inconsistent, it would correct the account before reading the balance. This first policy would guarantee that every transaction would read only consistent data, and every result presented to the user would be correct. Furthermore, no inconsistency would ever be visible, and write transactions would preform changes based on consistent values.

[Cachopo, 2007] describes in detail how the Fénix Framework strives to keep read-only transactions as light-weight as possible. To increase the performance, the framework implements a few optimizations to read-only transactions, that do not even need to collect a read set. However, if inconsistencies are fixed as soon as any transaction reads them, read-only transactions would perform much poorly.

First, read-only transactions would have to collect read objects and look at their dependence records to check if they are reading inconsistent values. Moreover, they would have to wait for a few fixing transactions to complete several writes, and many consistency checks. Note that read-only operations are not even meant to have secondary effects. Then, the fixing transactions would commit, and the read transaction would have to restart, to guarantee that it reads only consistent values from the start. Overall, this first option would provide the best consistency, but it would have the worst performance.

The second execution policy is an even lazier approach that is called **fix-on-exception**. The idea is to correct inconsistent objects as late as possible, i.e. only when a write transaction reads them and uses them to write even more inconsistencies. So, instead of attempting to keep an intransigent consistency

property, the framework could just guarantee that inconsistencies do not propagate.

A write transaction throws a ConsistencyException whenever it creates an inconsistency. At that moment, the framework would start a fixing transaction for each inconsistent object in the write transaction's read set. After the objects are fixed, the write transaction would restart. Hopefully, it would now perform actions based on consistent data, and commit.

This option would have the best performance because it corrects objects as late as possible, and only on write transactions. So, read-only transactions would not be affected. Consequently, a read transaction may present incorrect values to the user, even if there is already a method to fix them.

Worse, an inconsistency would not be fixed on every write transaction, it would only be fixed on those that throw ConsistencyExceptions. This situation may seldom happen, depending on the programming habits of the developers. If the developers implement write operations in a very isolated way, it is unlikely that inconsistent objects from one predicate will influence other predicates. Inconsistencies would rarely propagate, and thus, the framework would rarely fix them. Overall, this option presents the worst consistency with the best performance.

The third execution policy is not as lazy as the previous approach, and is called **fix-on-writes**. This policy would correct the objects whenever a write transaction reads an inconsistency. It would not affect the read-only transactions' performance. It would check all the dependence records of all objects in the read set of write transactions only. The write transactions already needed to collect a read set, and to perform a consistency check phase before commit.

Regardless of whether a write transaction throws an exception or not, all inconsistent objects in the read set would be corrected. After the corrections are made, the write transaction would restart. Hopefully, it would then perform actions based on consistent data, and write the expected output.

The fix-on-writes policy would provide a better consistency than the fix-on-exception policy. All write transactions would gradually correct inconsistencies. Moreover, this policy would provide a better performance than the fix-on-reads policy. Read-only transactions would not be affected. Only write transactions would fix inconsistencies. They were already meant to perform changes, and already had a read and a write set. So far, this option presents the best tradeoff between consistency and performance.

Nevertheless, all of these three first lazy policies assume that the fixing methods are correct, and that they always fix inconsistent objects. In practice, this situation may not always happen. Still, a read or write transaction would invoke the fixing methods in an attempt to fix each inconsistent object. If the object was not fixed, the next transaction that reads this still-inconsistent object will attempt to fix it again, probably without success.

Several transactions would repeatedly call a wrongly implemented fixing method on-demand. This situation would cause transactions to have a suboptimal performance. So, I propose a final policy that is independent from transactions.

The fourth and final execution policy is the only eager approach, and it is called **fix-on-initialize**. The idea is that a consistency predicate with a fixing method should not have existing inconsistencies. Consider a certain consistency predicate that has several inconsistent objects, and a developer that adds a fixing method in that predicate. The framework should be able to detect this new fixing method and execute it for all inconsistent objects during the initialization phase. Even if the objects are not fixed, the framework would not attempt to execute the fixing method again for those objects. In this scenario, inconsistencies would be fixed in the following situations:

- If a predicate allows creating inconsistent new objects, the framework should attempt to fix them

once, immediately after creation.

- When a developer adds a fixing method to a predicate, the framework should correct inconsistent objects during the initialization.
- When a developer changes the code of a fixing method, the framework should correct inconsistent objects again during the initialization.

The method code changes proposed in Section 8.1 would be necessary to detect changes in fixing methods. If the code of a fixing method is changed, then its implementation may have been corrected, and it may now successfully fix inconsistencies. The changes to the framework's domain needed to allow this policy are illustrated in Figure 8.3.

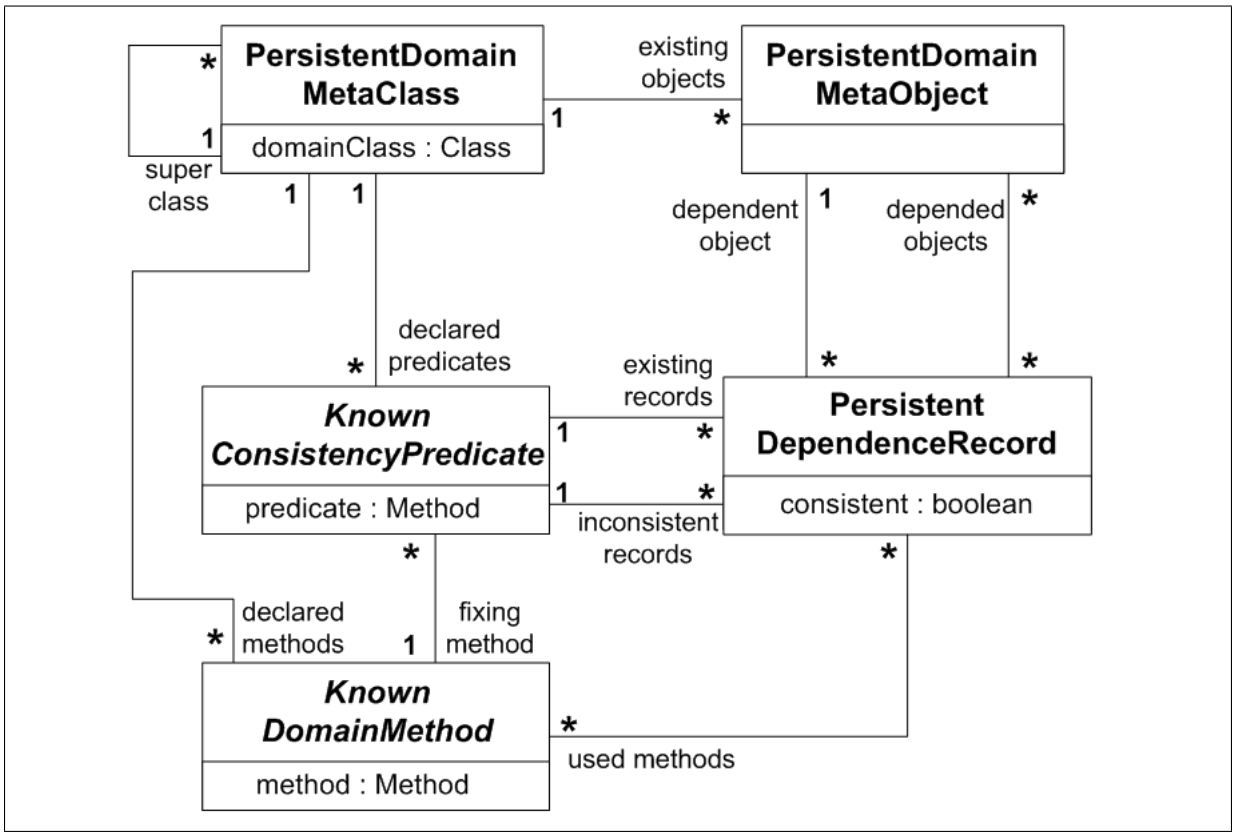

Figure 8.3: Hypothetical Fénix Framework domain with a new relation to support fixing methods.

The fix-on-initialize is the only eager policy that fixes inconsistencies as soon as possible. It would provide as good a consistency property as the fix-on-reads policy. Moreover, it would have a much better performance in runtime, because no transaction is affected. However, this fix-on-initialize policy would probably cause a slower initialization process. Overall, this option presents the best consistency with the best runtime performance, at the cost of the worst initialization performance.

It may seem that implementing automatic corrections is a complex and cumbersome task. However, I believe that this feature would add great value to the consistency predicates. It would not only remove much work from the development team, but it would also help to provide even greater quality assurances to the customer.

At the moment of this writing, I believe that the fourth execution policy is the most adequate. Still, the argumentation provided in this chapter is by no means final. This chapter has merely presented a first hand reasoning of what could be done to further improve the consistency predicates. Certainly, a more thorough analysis of each of the proposed ideas might prove to be even more conclusive.

Chapter 9

Conclusion

The consistency predicates allow the programmer to define, for each class, consistency rules that the framework will automatically enforce. When originally proposed in the JVSTM, the consistency predicates allowed the programmer to define consistency rules of one object depending on the state of other related objects. However, after being included in the Fénix Framework, each consistency predicate was limited to access only the object being checked. This limitation was due to the lack of persistence of the DependenceRecords, and the inability to detect code changes to the existing or new consistency rules.

This work has described an extension to the consistency predicates of the Fénix Framework. This extension has dealt with the typical case of an evolving enterprise application that has persistent data and is developed in an incremental way. In order to ease the incremental insertion of new consistency rules, and to ease their correction and maintenance, this extension detects code changes, and knows how to respond to them. It also tolerates already existing inconsistencies in the data, to help in keeping a high liveness quality for the application runtime. Furthermore, the framework keeps updated information about the existing inconsistencies visible to the programmer, so that he can add consistency rules today, and fix existing inconsistencies tomorrow.

So, with this extension to the Fénix Framework, the programmer can once again implement consistency rules of one object depending on the state of other objects. The PersistentDependenceRecords store data dependencies of each rule applied to each object, and keep each object from getting inconsistent when the data changes.

The programmer can introduce new rules and remove old rules from the application's code. The framework detects new rules and ensures that all existing objects will follow them. Still, it may happen that these existing objects are already inconsistent from the moment that a rule first appears in the system.

So, the framework further provides explicit and structured information to allow the programmer to access inconsistent objects with ease. It is then up to the programmer to correct the inconsistencies whenever he finds it appropriate. Alternatively, as future work, I have proposed to allow the framework to perform automatic inconsistency fixes.

The programmer can also configure some predicates to be inconsistency tolerant. This may be useful to allow the system operations to modify already inconsistent objects freely, especially if these objects do not hold sensitive information. Otherwise, if the objects of a certain class do store sensitive information, the programmer may choose to configure their predicates as intolerant. Intolerant (regular) predicates will prevent the objects from being modified, until they are corrected.

However, one limitation of the current implementation is that it does not detect changes to the inner

code of methods implementing consistency predicates. These changes may influence the execution flow of a predicate, and invalidate the PersistentDependenceRecords. To address this problem, as future work, I propose to enable the detection of method code changes. This detection can be achieved by creating a domain entity to represent regular domain methods, much like what was done with the KnownConsistencyPredicate. The framework could then detect changes in the persisted domain methods, and determine which predicates to reexecute.

There is another limitation to this extension. Problems may occur if someone decides to change the data on the database, while the application is offline. Even if offline data changes do not make the domain state inconsistent directly, the PersistentDependenceRecords may become outdated. With the PersistentDependenceRecords outdated, the framework's runtime can no longer guarantee that it keeps their objects consistent. All of the features described in this document require updated PersistentDependenceRecords, and may misbehave if this assumption fails.

Still, as long as the development team has the discipline to make all data changes within the application's supervision, this extension should work as expected. The framework detects code changes to the structure and the consistency rules of the target application, and responds in regard to these changes. It can also tolerate inconsistencies in non-sensitive data, and provides the programmer with useful information to help in correcting these inconsistencies.

Previously, the Fénix Framework could already maintain the consistency of the data with simple consistency rules. With this work, the framework provides pragmatic solutions to support creating and changing complex consistency rules, while the development team continuously changes and improves its evolving application.

Bibliography

A. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 26–37. ACM, 2006.

C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High-Performance Computer Architecture, HPCA-11 2005*, pages 316–327, 2005.

R. S. Arnold and S. A. Bohner. Impact analysis-towards a framework for comparison. In *Proceedings CSM '93: Conference on Software Maintenance*, 1993.

N. G. Bronson, H. Chafi, and K. Olukotun. *CCSTM: A Library-Based STM for Scala.* PhD thesis, Stanford University, 2010.

J. Cachopo. *Development of Rich Domain Models with Atomic Actions.* PhD thesis, Universidade Técnica de Lisboa, Instituto Superior Técnico, 2007.

J. Cachopo and A. Rito-Silva. Combining software transactional memory with a domain modeling language to simplify web application development. In *Proceedings of the 6th International Conference on Web Engineering*, pages 297–304. ACM Press, Jul 2006. doi: http://doi.acm.org/10.1145/1145581.1145640.

Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language.* PhD thesis, Department of Computer Science, Iowa State University, 2003.

A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2009.

E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2003.

S. M. Fernandes and J. Cachopo. A new architecture for enterprise applications with strong transactional semantics. Technical Report 26, INESC-ID/IST, May 2011.

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. *SIGPLAN Not.*, 37(5):234–245, 2002.

M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2002.

K. Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge Computer Laboratory, 2004.

D. R. Graham. Incremental development: review of nonmonolithic life-cycle development models. *Information and Software Technology*, 31:7–20, 1989.

L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, 2004.

T. Harris and K. Fraser. Language support for lightweight transactions. *Proceedings of the 18th annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, 2003.

T. Harris and S. Peyton Jones. Transactional memory with data invariants. *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. *In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *Proceedings of the 20th annual International Symposium on Computer Architecture (ISCA '93)*, pages 289–300, 1993.

M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. *Proceedings of the twenty-second annual symposium on Principles of Distributed Computing*, 2003.

M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41:253–262, 2006.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12, 1969.

C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

M. Karaorman, U. Holzle, and J. Bruno. jContractor: A reflective java library to support design by contract. In *Meta-Level Architectures and Reflection*, pages 175–196. Springer Berlin / Heidelberg, 1999.

R. Kramer. iContract: the java design by contract tool. In *TOOLS'98: Technology of Object-Oriented Languages and Systems*, pages 295–307. IEEE Computer Society, 1998.

M. Lackner, A. Krall, and F. Puntigam. Supporting design by contract in java. *Journal of Object Technology*, pages 57–76, 2002.

G. T. Leavens, C. Ruby, K. Rustan, M. Leino, E. Poll, and B. Jacobs. JML (poster session): notations and tools supporting detailed design in java. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum)*, pages 105–106. ACM, 2000.

G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of jml accommodates both runtime assertion checking and formal verification. In *Formal Methods for Components and Objects*, pages 262–284. Springer Berlin / Heidelberg, 2003.

B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1994.

B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1988a.

B. Meyer. Applying design by contract. *Computer*, 25:40–51, 1988b.

B. Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.

K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *The Twelfth International Symposium on High-Performance Computer Architecture*, pages 254–265, 2006.

X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA. ACM, 2004.

B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, PASTE. ACM, 2001.

B. Saha, A. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP, 2006.

N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of Distributed Computing*, PODC, 1995.

J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the oklahoma update. *Parallel and Distributed Technology: Systems and Applications, IEEE*, 1:58–71, 1993.

J. Wloka, B. G. Ryder, and F. Tip. JUnitMX - a change-aware unit testing tool. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE. IEEE Computer Society, 2009.

Appendix A

Programmer's Guideline

This appendix is aimed at developers that are interested in knowing how to correctly use the consistency predicates. It is a summarized version of the most important parts of this work that explain the different kinds of consistency predicates that can be used. It is composed of a set of guidelines and good practices on how to implement the consistency predicates properly in an application domain.

A.1 Using Regular Consistency Predicates

The consistency predicates are a tool provided by the Fénix Framework that allows you to define consistency rules inside your application's domain. To use them correctly, all your application's domain entities must be fully specified in DML.[1] The consistency predicates allow you to create rules about the contents of domain slots and relations defined in DML.

Unlike an invariant, a consistency predicate implements a rule that is verified in runtime. The consistency predicates were designed to be used in a production environment. If a certain operation or transaction breaks a rule, the framework will abort the transaction, and revert its effects. The framework will automatically verify the rules when necessary, and prevent your domain objects from getting inconsistent. All you need to do is to provide a correct implementation of the consistency predicates.

To create a consistency predicate, you should place the @ConsistencyPredicate annotation in a method of a domain class. If you place the annotation elsewhere, the framework will ignore it. A consistency predicate method must receive no arguments and return a boolean.

The implementation of a consistency predicate should always be deterministic and based only on the domain. Its implementation should never rely on any random values or on user input. The only changeable input that it should access are the slots and relations of domain objects declared in DML.

On a certain domain class, you can implement a consistency predicate that accesses the object's own slots by using its getter and setter methods. You can also use the relations of that object to access the slots of other related objects. So, the framework supports consistency predicates where an object uses the state of other objects to define its consistency. An example of such a consistency predicate is illustrated in Figure 2.8.

Also, each consistency predicate should be as small and independent as possible. One consistency predicate should implement exactly one consistency rule. If a certain class needs to enforce two rules, then you should implement two distinct consistency predicates. You should avoid having one predicate per class to enforce all the rules in conjunction.

If the object being checked turns out to be consistent, your predicate should return true. If the object turns out to be inconsistent, your predicate should return false. If the object is inconsistent, you can also choose to explicitly throw a ConsistencyException or a subclass of this exception.

Alternatively, you may choose to pass a subclass of ConsistencyException as the value argument to the @ConsistencyPredicate annotation. If you do, and the predicate returns false, the active transaction will throw that exception on its own. The default value of the annotation's argument is the ConsistencyException, that the transaction throws when no explicit argument is provided.

You may, for instance, create a subclass of ConsistencyException with a specific implementation of the getMessage() method. It may be useful to catch that subclass at an exception handler, and provide a

[1] A brief description of DML is presented in Section 2.1.

user-friendly message at the presentation layer of your application.

In summary, a transaction that is executing a predicate will throw an exception that is determined by the following procedures:

- If the predicate returns false, the transaction throws the ConsistencyException passed as argument to the @ConsistencyPredicate annotation.
- If the predicate throws a ConsistencyException or a subclass, the transaction throws that same exception.
- If the predicate throws another exception, the transaction wraps that exception inside the ConsistencyException passed as argument to the annotation.

Obviously, if all the predicates return true, then all the objects involved are consistent. In this case, the transaction simply commits and does not throw any exception.

Cachopo [2007] describes the principles of the consistency predicates in greater detail.

A.2 Using Consistency Predicates Inside a Class Hierarchy

The consistency predicates were designed to work inside a domain class hierarchy. A predicate at a certain class will be verified for objects of that class and those of all subclasses. However, if the predicate method is public, the subclasses may override that method and implement a refinement of that predicate.

You can make a consistency predicate final to prevent other developers from overriding it at the subclasses. You should make it final when you believe that all objects of any subclass should always follow this rule, without exclusions.

You can also make a predicate private to prevent people from overriding it, and from even seeing or invoking it from subclasses. You should make it private when all object of subclasses should always follow this rule, and the method itself should be hidden from other developers.

All other predicates that are public or protected can be overridden at the subclasses. You can make a predicate public if you want to define a normal rule that may have exclusions at the subclasses, depending on the case.

The use of package visible consistency predicates is forbidden. Package visibility is the default visibility for methods that specify no explicit visibility modifier in Java. You must always specify an explicit visibility for every consistency predicate method.

Any method that overrides a consistency predicate must always have the @ConsistencyPredicate annotation. In other words, you cannot override a consistency predicate with a regular (non-predicate) method. The @ConsistencyPredicate annotation itself is not inherited. Therefore, you must manually place that annotation and redefine the value and inconsistencyTolerant[2] arguments if you want predicates at subclasses to define them. For instance, if you do not redefine the value argument, inconsistent objects of subclasses will throw ConsistencyException by default.

If your domain class hierarchy includes an abstract superclass, you may implement a (concrete) consistency predicate at that class. You may invoke other abstract methods inside the implementation of your consistency predicate. Even though the abstract class will never have objects, the framework will ensure that the predicate will be checked for objects of subclasses.

You can also define an abstract consistency predicate at an abstract superclass. The abstract consistency predicate is a way to force other developers to implement that consistency predicate at the subclasses. Although the framework will ignore the abstract predicate because it has no implementation, the framework will verify the concrete predicates at the subclasses.

Finally, note that the Fénix Framework already provides atomic transactions to ensure the correct access of concurrent operations to shared data. The synchronized method modifier forces the JVM to use a lock in the access to the method, which may cause performance issues and deadlocks. Therefore, even though the framework supports them, the use of synchronized consistency predicates is discouraged.

[2]The use of the inconsistencyTolerant argument will be explained in Section A.4.

In summary, the method modifiers of your consistency predicates can influence their behavior. Most importantly, you are allowed to create a public predicate, and to override it in subclasses later. You may, for instance, wish to indicate that this predicate applies for objects of a class, but not for objects of the subclass. To do so, you can create an overriding predicate at the subclass that simply returns true.

However, this use case does not respect the Liskov Substitution Principle (LSP) [Liskov and Wing, 1994]. In object-oriented programming, the LSP states that if a supertype defines a certain rule, all subtypes must also follow that rule. It intends to guarantee the semantic interoperability of classes of the same hierarchy.

Consider a certain method of your application that deals with objects of a class, and receives one as argument. The method will perform operations on this object. It is probably implemented with the assumption that the object follows the rules of its class.

However, an object of a subclass can be passed as argument to that method. If you choose not to follow the LSP, this object may not follow the rules that the method expects it to. Then, the method may have an unexpected behavior. Therefore, if you wish to respect the LSP, you should only override a predicate to make a rule more restrictive, and never more permissive. Alternatively, you can also decide to make all your predicates final, and keep them from ever being overridden.

Section 5.1.2 describes the influence of predicate modifiers inside a class hierarchy in greater detail.

A.3 Tracking Existing Inconsistencies

This section briefly summarizes the behavior of the framework when you create a new predicate in the code of your application. It explains some of the entities that the framework creates and manages, to keep track of your inconsistent objects. You may use these framework entities to easily access all the existing inconsistent objects.

Whenever you introduce a new consistency predicate in a certain domain class, the framework will detect this new predicate. Then, for each existing object of that domain class, the framework will execute the predicate and create a PersistentDependenceRecord. Each of these existing objects may have been consistent or not.

If an object is consistent, its PersistentDependenceRecord will keep it consistent in the future. If an object is inconsistent, its PersistentDependenceRecord will prevent operations from changing it, unless it is corrected. Either way, each PersistentDependenceRecord keeps information about the consistency of its object.

So, if you need to know if an object is inconsistent, you can iterate through all its PersistentDependenceRecords. An object will have one PersistentDependenceRecord for each predicate that its class defines. If any PersistentDependenceRecord has a false value in the consistent slot, then the object is inconsistent according to that predicate. An example of the use of the PersistentDependenceRecord is shown in Figure A.1.

```
for(PersistentDependenceRecord record : myObject.getMetaObject()
    .getDependenceRecords()) {
  if (!record.isConsistent()) {
    System.out.println("The object " + myObject +
        " is inconsistent according to " + record.getPredicate());
  }
}
```

Figure A.1: How to determine if an object is inconsistent.

But you may also want to find, among all the existing objects of a class, which ones are inconsistent according to a certain consistency predicate. You will need to obtain the framework's entity that represents that consistency predicate. To do so, you will need to use the KnownConsistencyPredicate's static readKnownConsistencyPredicate() method. This method receives as argument the class and the name of the consistency predicate. An example of the invocation of this method is presented in Figure A.2.

```
KnownConsistencyPredicate consistencyPredicate = KnownConsistencyPredicate
    .readKnownConsistencyPredicate(Account.class, "checkBalancePositive");
```

Figure A.2: How to obtain a known consistency predicate.

Once you have the KnownConsistencyPredicate that you want, you can call the getInconsistentDependenceRecords() method. This method will provide you with a complete set of PersistentDependenceRecords of all the objects that are inconsistent, according to that predicate. You can then access and correct these objects whenever you see fit. An example of this access is shown in Figure A.3.

```
for (PersistentDependenceRecord record : consistencyPredicate
    .getInconsistentDependenceRecords()) {
  System.out.println("The object " + record.getDependent() +
      " is inconsistent according to " + record.getPredicate());
}
```

Figure A.3: How to obtain the inconsistent objects of a certain predicate. The consistencyPredicate variable was obtained in the previous figure.

Whenever you remove an existing predicate from your code, the framework will detect the missing predicate, and remove its PersistentDependenceRecords. Whenever you introduce a new predicate on a domain class, the framework will detect and reexecute the predicate for all existing objects of that class.

Also, whenever you change the name or signature of an existing predicate, the framework will detect the change and reexecute the predicate. So, the framework will keep the PersistentDependenceRecords, and the list of inconsistent objects up-to-date.

However, the framework does not yet detect code changes to the implementation of a method's body. Therefore, if you change the implementation of a consistency predicate, the framework will not automatically reexecute it. Still, you can always change the signature of the predicate to force the framework to reexecute it.

Section 5.2 describes the framework's domain entities in greater detail.

A.4 Using Inconsistency Tolerance

This section explains how you can make the framework's transactions tolerant to already-existing inconsistencies. It also explains under what circumstances can inconsistency-tolerant transactions be useful.

As seen previously, since the framework detects new predicates in your code, it may happen that existing objects are already inconsistent. Any transaction that changes objects will execute the predicates depending on these objects. If the objects were previously inconsistent (and not corrected), the predicate will return false and the transaction will abort. This means that the operation will not produce any effects, and will simply throw an exception.

However, this transaction does not necessarily create new inconsistencies. It may simply perform operations on some objects that another event had made inconsistent before. The transaction is not responsible for the inconsistencies that it passed through. And yet, the transaction will abort and present an error to the user, because a certain predicate returned false. So, a new predicate inserted on inconsistent objects may prevent many operations from being successful, which may endanger your system's liveness.

To avoid this problem, you can make the predicates of your choice inconsistency-tolerant. Section A.1 has shown that the @ConsistencyPredicate annotation can receive a value argument to specify the type of exception that is thrown. The @ConsistencyPredicate also has a boolean inconsistencyTolerant argument to specify if the predicate is tolerant to inconsistencies, or not. The default value of this last argument is false; by default, predicates are not inconsistency-tolerant. If you explicitly set the argument to true, the annotated predicate will become inconsistency-tolerant.

Consider a certain predicate that is now inconsistency-tolerant, and a transaction that writes to an already-inconsistent object. The execution of the predicate will return false, representing the already-inconsistent object that was not corrected. But the dependence record already stored a false value in its consistent slot. So, the transaction will commit and produce its effects, because it knows that it did not create this inconsistency. Otherwise, any transaction that changes a consistent object and makes it inconsistent will always abort.

Chapter 6 discusses inconsistency tolerance in greater detail.