Combining Software Transactional Memory with a Domain Modeling Language to Simplify Web Application Development

João Cachopo and António Rito-Silva

Inesc-ID/Technical University of Lisbon

ICWE 2006
July 14th, 2006
Outline

1. Context
2. Motivating Example
3. Software Transactional Memory
4. Domain Modeling Language
5. Conclusions and Future Work
Outline

1. Context
2. Motivating Example
3. Software Transactional Memory
4. Domain Modeling Language
5. Conclusions and Future Work
Outline

1. Context
2. Motivating Example
3. Software Transactional Memory
4. Domain Modeling Language
5. Conclusions and Future Work
Outline

1. Context
2. Motivating Example
3. Software Transactional Memory
4. Domain Modeling Language
5. Conclusions and Future Work
Outline

1. Context
2. Motivating Example
3. Software Transactional Memory
4. Domain Modeling Language
5. Conclusions and Future Work
Outline

1. Context
2. Motivating Example
3. Software Transactional Memory
4. Domain Modeling Language
5. Conclusions and Future Work
Main Goal

To simplify the development of web applications.

Why?

Because web applications are way too hard to develop with current technologies... even simple ones!

So, how can we make it simpler?
Main Goal

To simplify the development of web applications.

Why?

Because web applications are way too hard to develop with current technologies... ...even simple ones!

So, how can we make it simpler?
Main Goal

To simplify the development of web applications.

Why?

Because web applications are way too hard to develop with current technologies . . .

. . . even simple ones!

So, how can we make it simpler?
Main Goal

To simplify the development of web applications.

Why?

Because web applications are way too hard to develop with current technologies... 

...even simple ones!

So, how can we make it simpler?
The MDA approach

To model the application and then just push a button to generate all of the application's code.

However, that is not the current state of the practice.

Can we change this?

Maybe, or maybe not.
But we are not up to there, yet!
The MDA approach

To model the application and then just push a button to generate all of the application’s code.

However, that is not the current state of the practice.

Can we change this?

Maybe, or maybe not.
But we are not up to there, yet!

- Many proposals are not mature enough.
- It may be appropriate for data-intensive applications, but not for domain-intensive applications.
- It represents a major change to the programmer’s way of doing things.
The MDA approach

To model the application and then just push a button to generate all of the application’s code.

However, that is not the current state of the practice.

Can we change this?

Maybe, or maybe not.
But we are not up to there, yet!

- Many proposals are not mature enough
- It may be appropriate for data-intensive applications, but not for domain-intensive applications
- It represents a major change to the programmer’s way of doing things
The MDA approach

To model the application and then just push a button to generate all of the application’s code.

However, that is not the current state of the practice.

Can we change this?

Maybe, or maybe not. But we are not up to there, yet!

- Many proposals are not mature enough
- It may be appropriate for data-intensive applications, but not for domain-intensive applications
- It represents a major change to the programmer’s way of doing things
The MDA approach

To model the application and then just push a button to generate all of the application’s code.

However, that is not the current state of the practice.

Can we change this?

Maybe, or maybe not.
But we are not up to there, yet!

- Many proposals are not mature enough
  - It may be appropriate for data-intensive applications, but not for domain-intensive applications
  - It represents a major change to the programmer’s way of doing things
The MDA approach

To model the application and then just push a button to generate all of the application's code.

However, that is not the current state of the practice.

Can we change this?

Maybe, or maybe not. But we are not up to there, yet!

- Many proposals are not mature enough
- It may be appropriate for **data-intensive** applications, but not for **domain-intensive** applications
- It represents a major change to the programmer’s way of doing things
The MDA approach

To model the application and then just push a button to generate all of the application’s code.

However, that is not the current state of the practice.

Can we change this?

Maybe, or maybe not.
But we are not up to there, yet!

- Many proposals are not mature enough
- It may be appropriate for **data-intensive** applications, but not for **domain-intensive** applications
- It represents a major change to the **programmer’s way of doing things**
A more pragmatic approach

We want to help current developers by improving on how they do their work, rather than to replace it entirely.

So, we propose simple and programmer-friendly extensions to the technologies that programmers currently use.

In particular, we propose two simple extensions to the Java programming language.

In this work, we target specifically the domain model and the service layer of a web application. Therefore, we will not talk about the many other parts of a web application.
A more pragmatic approach

We want to help current developers by improving on how they do their work, rather than to replace it entirely.

So, we propose simple and programmer-friendly extensions to the technologies that programmers currently use.

In particular, we propose two simple extensions to the Java programming language.

In this work, we target specifically the domain model and the service layer of a web application. Therefore, we will not talk about the many other parts of a web application.
A more pragmatic approach

We want to help current developers by improving on how they do their work, rather than to replace it entirely.

So, we propose simple and programmer-friendly extensions to the technologies that programmers currently use.

In particular, we propose two simple extensions to the Java programming language.

In this work, we target specifically the domain model and the service layer of a web application. Therefore, we will not talk about the many other parts of a web application.
A more pragmatic approach

We want to help current developers by improving on how they do their work, rather than to replace it entirely.

So, we propose simple and programmer-friendly extensions to the technologies that programmers currently use.

In particular, we propose two simple extensions to the Java programming language.

In this work, we target specifically the domain model and the service layer of a web application. Therefore, we will not talk about the many other parts of a web application.
Domain-intensive web applications

Domain-intensive applications have rich and complex domains, not only structurally, but also in terms of behavior.

These applications can certainly benefit from using object-oriented languages’ features such as inheritance, polymorphism, late binding...

Add to this the fact that we need both:

- to ensure that domain entities are persistent
- to process requests concurrently
Domain-intensive web applications

Domain-intensive applications have rich and complex domains, not only structurally, but also in terms of behavior.

These applications can certainly benefit from using object-oriented languages’ features such as inheritance, polymorphism, late binding. . .

Add to this the fact that we need both:

- to ensure that domain entities are persistent
- to process requests concurrently
Domain-intensive web applications

Domain-intensive applications have rich and complex domains, not only structurally, but also in terms of behavior.

These applications can certainly benefit from using object-oriented languages’ features such as inheritance, polymorphism, late binding...

Add to this the fact that we need both:

- to ensure that domain entities are persistent
- to process requests concurrently
Domain-intensive web applications

Domain-intensive applications have rich and complex domains, not only structurally, but also in terms of behavior.

These applications can certainly benefit from using object-oriented languages’ features such as inheritance, polymorphism, late binding...

Add to this the fact that we need both:

- to ensure that domain entities are persistent
- to process requests concurrently
Domain-intensive web applications

Domain-intensive applications have rich and complex domains, not only structurally, but also in terms of behavior.

These applications can certainly benefit from using object-oriented languages’ features such as inheritance, polymorphism, late binding...

Add to this the fact that we need both:
- to ensure that domain entities are persistent
- to process requests concurrently
Outline

1. Context
2. Motivating Example
3. Software Transactional Memory
4. Domain Modeling Language
5. Conclusions and Future Work
The (in)famous PetStore

The users of the PetStore application can browse an online catalog of pets, which is organized by categories of animals such as fish, dogs, and reptiles.

The user can browse any of the categories to see the available products.

So, some key elements on the PetStore domain are the entities Category and Product, and the relationship between them.
The (in)famous PetStore

The users of the PetStore application can browse an online catalog of pets, which is organized by categories of animals such as fish, dogs, and reptiles.

The user can browse any of the categories to see the available products.

So, some key elements on the PetStore domain are the entities Category and Product, and the relationship between them.
The (in)famous PetStore

The users of the PetStore application can browse an online catalog of pets, which is organized by categories of animals such as fish, dogs, and reptiles.

The user can browse any of the categories to see the available products.

So, some key elements on the PetStore domain are the entities Category and Product, and the relationship between them.
public class Category {
    String name;
    String description;
    Set<Product> products = new HashSet<Product>();
    ...
}
public class Category {
    String name;
    String description;
    Set<Product> products = new HashSet<Product>();
    ...
}

public class Product {
    String name;
    String description;
    ...
}

What are the multiplicities of the Category/Product relationship?
Category and Product implementations

```java
public class Category {
    String name;
    String description;
    Set<Product> products = new HashSet<Product>();
    ...
}

public class Product {
    String name;
    String description;
    ...
}
```

What are the multiplicities of the Category/Product relationship?
What are the multiplicities of the Category/Product relationship?
What are the multiplicities of the Category/Product relationship?
protected String doExecute() throws Exception {
    Session s = getHibernateSession();
    try {
        category = (Category)s.load(Category.class, categoryId);
        /*
         * Since category.product is lazy loaded,
         * traverse the collection to load all the products to display
         */
        Iterator it = category.getProducts().iterator();
        while (it.hasNext()) {
            Product p = (Product) it.next();
            p.getName();
        }
        return SUCCESS;
    } finally {
        s.close();
    }
}
protected String doExecute() throws Exception {
    Session s = getHibernateSession();
    try {
        category = (Category)s.load(Category.class, categoryId);
    
        /*
        * Since category.product is lazy loaded,
        * traverse the collection to load all the products to display
        */
        Iterator it = category.getProducts().iterator();
        while (it.hasNext()) {
            Product p = (Product) it.next();
            p.getName();
        }
    
        return SUCCESS;
    } finally {
        s.close();
    }
}
protected String doExecute() throws Exception {
    Session s = getHibernateSession();
    try {
        category = (Category)s.load(Category.class, categoryId);

        /*
         * Since category.product is lazy loaded,
         * traverse the collection to load all the products to display
         */
        Iterator it = category.getProducts().iterator();

        while (it.hasNext()) {
            Product p = (Product) it.next();
            p.getName();
        }

        return SUCCESS;
    } finally {
        s.close();
    }
}
protected String doExecute() throws Exception {
    Session s = getHibernateSession();
    try {
        category = (Category)s.load(Category.class, categoryId);

        /*
        * Since category.product is lazy loaded,
        * traverse the collection to load all the products to display
        */
        Iterator it = category.getProducts().iterator();

        while (it.hasNext()) {
            Product p = (Product) it.next();
            p.getName();
        }

        return SUCCESS;
    } finally {
        s.close();
    }
}
protected String doExecute() throws Exception {
    Session s = getHibernateSession();
    try {
        category = (Category)s.load(Category.class, categoryId);

        /*
         * Since category.product is lazy loaded,
         * traverse the collection to load all the products to display
         */
        Iterator it = category.getProducts().iterator();

        while (it.hasNext()) {
            Product p = (Product) it.next();
            p.getName();
        }

        return SUCCESS;
    } finally {
        s.close();
    }
}
protected String doExecute() throws Exception {
    Session s = getHibernateSession();
    try {
        category = (Category)s.load(Category.class, categoryId);

        /*
         * Since category.product is lazy loaded,
         * traverse the collection to load all the products to display
         */
        Iterator it = category.getProducts().iterator();

        while (it.hasNext()) {
            Product p = (Product) it.next();
            p.getName();
        }

        return SUCCESS;
    }
    finally {
        s.close();
    }
}
protected String doExecute() throws Exception {
    Session s = getHibernateSession();
    try {
        category = (Category)s.load(Category.class, categoryId);
    
        /*
        * Since category.product is lazy loaded,
        * traverse the collection to load all the products to display
        */
        Iterator it = category.getProducts().iterator();

        while (it.hasNext()) {
            Product p = (Product) it.next();
            p.getName();
        }

        return SUCCESS;
    } finally {
        s.close();
    }
}
An action with some bizarre code

protected String doExecute() throws Exception {
    Session s = getHibernateSession();
    try {
        category = (Category)s.load(Category.class, categoryId);
        /*
         * Since category.product is lazy loaded,
         * traverse the collection to load all the products to display
         */
        Iterator it = category.getProducts().iterator();
        while (it.hasNext()) {
            Product p = (Product) it.next();
            p.getName();
        }
        return SUCCESS;
    } finally {
        s.close();
    }
}
The problems we are trying to solve

The previous examples illustrate two of the problems that we address with our work:

- Mainstream programming languages lack the constructs to represent correctly a domain model.
- The support for persistence interferes with the programming model.

Moreover:
The problems we are trying to solve

The previous examples illustrate two of the problems that we address with our work:

- Mainstream programming languages lack the constructs to represent correctly a domain model.
- The support for persistence interferes with the programming model.

Moreover:

- Implementing a bi-directional relationship is even harder.
- Relying on the database for transactional support incurs into performance and scalability costs.
The problems we are trying to solve

The previous examples illustrate two of the problems that we address with our work:

- Mainstream programming languages lack the constructs to represent correctly a domain model.
- The support for persistence interferes with the programming model.

Moreover:

- Implementing a bi-directional relationship is even harder.
- Relying on the database for transactional support incurs into performance and scalability costs.
The problems we are trying to solve

The previous examples illustrate two of the problems that we address with our work:

- Mainstream programming languages lack the constructs to represent correctly a domain model.
- The support for persistence interferes with the programming model.

Moreover:

- Implementing a bi-directional relationship is even harder.
- Relying on the database for transactional support incurs into performance and scalability costs.
The problems we are trying to solve

The previous examples illustrate two of the problems that we address with our work:

- Mainstream programming languages lack the constructs to represent correctly a domain model.
- The support for persistence interferes with the programming model.

Moreover:

- Implementing a bi-directional relationship is even harder.
- Relying on the database for transactional support incurs into performance and scalability costs.
The problems we are trying to solve

The previous examples illustrate two of the problems that we address with our work:

- Mainstream programming languages lack the constructs to represent correctly a domain model.
- The support for persistence interferes with the programming model.

Moreover:

- Implementing a bi-directional relationship is even harder.
- Relying on the database for transactional support incurs into performance and scalability costs.
Outline

1. Context
2. Motivating Example
3. Software Transactional Memory
4. Domain Modeling Language
5. Conclusions and Future Work
The time has come for STM

**Software Transactional Memory** (STM) brings transactions into the realms of a programming language, but discards the **Durability** property.

The fundamental goal is to give us the **Atomicity** property, without using locks.

```plaintext
atomic { instructions }
```

Currently, there are several groups working in Hardware, Software and Hybrid Transaction Memories.
The time has come for STM

**Software Transactional Memory** (STM) brings transactions into the realms of a programming language, but discards the **Durability** property.

The fundamental goal is to give us the **Atomicity** property, *without using locks*.

```
atomic { instructions }
```

Currently, there are several groups working in Hardware, Software and Hybrid Transaction Memories.
The time has come for STM

**Software Transactional Memory** (STM) brings transactions into the realms of a programming language, but discards the **Durability** property.

The fundamental goal is to give us the **Atomicity** property, **without using locks**.

```
atomic { instructions }
```

Currently, there are several groups working in Hardware, Software and Hybrid Transaction Memories.
STM Essentials

Most STMs share a common set of properties:

- In a STM there are transactional objects, which may be read and/or written during a transaction.
- Each transaction keeps track of what was read and what was changed during the transaction.
- If the transaction is valid, it can commit.
- The commit of a transaction makes the changes made during the transaction visible to others.
- If a transaction aborts, none of its changes are visible.
Most STMs share a common set of properties:

- In a STM there are **transactional objects**, which may be **read** and/or **written** during a **transaction**.
- Each transaction keeps track of what was read and what was changed during the transaction.
- If the transaction is **valid**, it can **commit**.
- The **commit** of a transaction makes the changes made during the transaction visible to others.
- If a transaction **aborts**, none of its changes are visible.
STM Essentials

Most STMs share a common set of properties:

- In a STM there are **transactional objects**, which may be **read** and/or **written** during a **transaction**.
- Each transaction keeps track of what was read and what was changed during the transaction.
- If the transaction is **valid**, it can **commit**.
- The **commit** of a transaction makes the changes made during the transaction visible to others.
- If a transaction **aborts**, none of its changes are visible.
Most STMs share a common set of properties:

- In a STM there are **transactional objects**, which may be **read** and/or **written** during a **transaction**.
- Each transaction keeps track of what was read and what was changed during the transaction.
- If the transaction is **valid**, it can **commit**.
- The **commit** of a transaction makes the changes made during the transaction visible to others.
- If a transaction **aborts**, none of its changes are visible.
Most STMs share a common set of properties:

- In a STM there are **transactional objects**, which may be **read** and/or **written** during a **transaction**.
- Each transaction keeps track of what was read and what was changed during the transaction.
- If the transaction is **valid**, it can **commit**.
- The **commit** of a transaction makes the changes made during the transaction visible to others.
- If a transaction **aborts**, none of its changes are visible.
STM Essentials

Most STMs share a common set of properties:

- In a STM there are transactional objects, which may be read and/or written during a transaction.
- Each transaction keeps track of what was read and what was changed during the transaction.
- If the transaction is valid, it can commit.
- The commit of a transaction makes the changes made during the transaction visible to others.
- If a transaction aborts, none of its changes are visible.
An STM based on multiple versions

Use **versioned boxes** as transactional locations.

Given a box $B$, and a current transaction $T$:
- $\text{BoxWrite}(B, v)$ adds $v$ to the local values of $T$.
- When $T$ commits, $v$ becomes the new version of $B$.
- $\text{BoxRead}(B)$ returns the correct version of $B$.
- The commit of $T$ fails iff both:
  - $T$ is a read-write transaction.
  - Some of the boxes read by $T$ changed.

**Key Result**

The commit of read-only and write-only transactions never fails.
An STM based on multiple versions

Use **versioned boxes** as transactional locations.

Given a box $B$, and a current transaction $T$:
- $\text{BoxWrite}(B, v)$ adds $v$ to the **local values** of $T$.
- When $T$ commits, $v$ becomes the **new version** of $B$.
- $\text{BoxRead}(B)$ returns the **correct** version of $B$.
- The commit of $T$ fails iff both:

**Key Result**
The commit of read-only and write-only transactions never fails.
An STM based on multiple versions

Use **versioned boxes** as transactional locations.

Given a box $B$, and a current transaction $T$:

- $\text{BoxWrite}(B, v)$ adds $v$ to the **local values** of $T$.
- When $T$ commits, $v$ becomes the **new version** of $B$.
- $\text{BoxRead}(B)$ returns the **correct** version of $B$.
- The commit of $T$ fails iff both:

**Key Result**

The commit of read-only and write-only transactions never fails.
An STM based on multiple versions

Use **versioned boxes** as transactional locations.

Given a box $B$, and a current transaction $T$:

- $\text{BoxWrite}(B, v)$ adds $v$ to the **local values** of $T$.
- When $T$ commits, $v$ becomes the **new version** of $B$.
- $\text{BoxRead}(B)$ returns the **correct** version of $B$.
- The commit of $T$ fails iff both:
  - $T$ is a read-write transaction.
  - Some of the boxes read by $T$ changed.

**Key Result**

The commit of read-only and write-only transactions never fails.
An STM based on multiple versions

Use **versioned boxes** as transactional locations.

Given a box $B$, and a current transaction $T$:
- $\text{BoxWrite}(B, v)$ adds $v$ to the **local values** of $T$.
- When $T$ commits, $v$ becomes the **new version** of $B$.
- $\text{BoxRead}(B)$ returns the **correct** version of $B$.
- The commit of $T$ fails iff both:
  - $T$ is a read-write transaction.
  - Some of the boxes read by $T$ changed.

**Key Result**
The commit of read-only and write-only transactions never fails.
An STM based on multiple versions

Use **versioned boxes** as transactional locations.

Given a box $B$, and a current transaction $T$:

- BoxWrite($B$, $v$) adds $v$ to the **local values** of $T$.
- When $T$ commits, $v$ becomes the **new version** of $B$.
- BoxRead($B$) returns the **correct** version of $B$.
- The commit of $T$ fails iff both:
  1. $T$ is a **read-write** transaction.
  2. Some of the boxes read by $T$ changed.

**Key Result**

The commit of **read-only** and **write-only** transactions never fails.
An STM based on multiple versions

Use **versioned boxes** as transactional locations.

Given a box $B$, and a current transaction $T$:

- $\text{BoxWrite}(B, v)$ adds $v$ to the **local values** of $T$.
- When $T$ commits, $v$ becomes the **new version** of $B$.
- $\text{BoxRead}(B)$ returns the **correct** version of $B$.
- The commit of $T$ fails iff both:
  1. $T$ is a **read-write** transaction.
  2. Some of the boxes read by $T$ changed.

**Key Result**

The commit of **read-only** and **write-only** transactions never fails.
An STM based on multiple versions

Use versioned boxes as transactional locations.

Given a box B, and a current transaction T:

- BoxWrite(B, v) adds v to the local values of T.
- When T commits, v becomes the new version of B.
- BoxRead(B) returns the correct version of B.
- The commit of T fails iff both:
  1. T is a read-write transaction.
  2. Some of the boxes read by T changed.

Key Result

The commit of read-only and write-only transactions never fails.
An STM based on multiple versions

Use *versioned boxes* as transactional locations.

Given a box $B$, and a current transaction $T$:

- $\text{BoxWrite}(B, v)$ adds $v$ to the *local values* of $T$.
- When $T$ commits, $v$ becomes the *new version* of $B$.
- $\text{BoxRead}(B)$ returns the *correct* version of $B$.
- The commit of $T$ fails iff both:
  1. $T$ is a *read-write* transaction.
  2. Some of the boxes read by $T$ changed.

**Key Result**

The commit of *read-only* and *write-only* transactions never fails.
Making a STM Persistent

In STM, transactions already keep record of what they change. . .

. . . so, we just have to make those changes persistent in the commit of the transaction.

That is relatively simple:

1. Using a relational database we just have to write all the changes to the database in one database transaction.

   We need a cache of already loaded objects/boxes to avoid loading the same object/box twice.

   Either the database has all the versions or we cannot allow old values be garbage collected before time.
Making a STM Persistent

In STM, transactions already keep record of what they change... 

...so, we just have to make those changes persistent in the commit of the transaction.

That is relatively simple:

- Using a relational database we just have to write all the changes to the database in one database transaction.
- We need a cache of already loaded objects/boxes to avoid loading the same object/box twice.
- Either the database has all the versions or we cannot allow old values be garbage collected before time.
Making a STM Persistent

In STM, transactions already keep record of what they change...

...so, we just have to make those changes persistent in the commit of the transaction.

That is relatively simple:

- Using a relational database we just have to write all the changes to the database in one database transaction.
- We need a cache of already loaded objects/boxes to avoid loading the same object/box twice.
- Either the database has all the versions or we cannot allow old values be garbage collected before time.
Making a STM Persistent

In STM, transactions already keep record of what they change... 

...so, we just have to make those changes persistent in the commit of the transaction.

That is relatively simple:

- Using a relational database we just have to write all the changes to the database in one database transaction.
- We need a cache of already loaded objects/boxes to avoid loading the same object/box twice.
- Either the database has all the versions or we cannot allow old values to be garbage collected before time.
Making a STM Persistent

In STM, transactions already keep record of what they change. . .

. . . so, we just have to make those changes persistent in the commit of the transaction.

That is relatively simple:

- Using a relational database we just have to write all the changes to the database in one database transaction.
- We need a cache of already loaded objects/boxes to avoid loading the same object/box twice.
- Either the database has all the versions or we cannot allow old values be garbage collected before time.
Making a STM Persistent

In STM, transactions already keep record of what they change. . .

. . . so, we just have to make those changes persistent in the commit of the transaction.

That is relatively simple:

- Using a relational database we just have to write all the changes to the database in one database transaction.
- We need a cache of already loaded objects/boxes to avoid loading the same object/box twice.
- Either the database has all the versions or we cannot allow old values be garbage collected before time.
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}
```

```java
public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}
```

```java
public class Category {
    private VBox<String> name = new VBox<String>(null);
    public String getName() {
        return name.get();
    }
    public void setName(String newName) {
        name.put(newName);
    }
}
```
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}

public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}

public class Category {
    private VBox<String> name = new VBox<String>(null);

    public String getName() {
        return name.get();
    }

    public void setName(String newName) {
        name.put(newName);
    }
}
```
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}

public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}

public class Category {
    private VBox<String> name = new VBox<String>(null);

    public String getName() {
        return name.get();
    }

    public void setName(String newName) {
        name.put(newName);
    }
}
```
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}
```

```java
public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}
```

```java
public class Category {
    private VBox<String> name = new VBox<String>(null);

    public String getName() {
        return name.get();
    }

    public void setName(String newName) {
        name.put(newName);
    }
}
```
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}

public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}

public class Category {
    private VBox<String> name = new VBox<String>(null);

    public String getName() {
        return name.get();
    }

    public void setName(String newName) {
        name.put(newName);
    }
}
```
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}

public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}

public class Category {
    private VBox<String> name = new VBox<String>(null);

    public String getName() {
        return name.get();
    }

    public void setName(String newName) {
        name.put(newName);
    }
}
```
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}

public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}

public class Category {
    private VBox<String> name = new VBox<String>(null);
    public String getName() {
        return name.get();
    }
    public void setName(String newName) {
        name.put(newName);
    }
}
```
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}

public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}

public class Category {
    private VBox<String> name = new VBox<String>(null);

    public String getName() {
        return name.get();
    }

    public void setName(String newName) {
        name.put(newName);
    }
}
```
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}

public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}

public class Category {
    private VBox<String> name = new VBox<String>(null);

    public String getName() {
        return name.get();
    }

    public void setName(String newName) {
        name.put(newName);
    }
}
```
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}
```

```java
public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}
```

```java
public class Category {
    private VBox<String> name = new VBox<String>(null);
    public String getName() {
        return name.get();
    }
    public void setName(String newName) {
        name.put(newName);
    }
}
```
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}
```

```java
public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}
```

```java
public class Category {
    private VBox<String> name = new VBox<String>(null);

    public String getName() {
        return name.get();
    }

    public void setName(String newName) {
        name.put(newName);
    }
}
```
JVSTM implemented as a Java library

```java
public class Transaction {
    public static void start();
    public static void abort();
    public static void commit();
}

public class VBox<E> {
    public VBox(E initial);
    public E get();
    public void put(E newE);
}

public class Category {
    private VBox<String> name = new VBox<String>(null);
    
    public String getName() {
        return name.get();
    }

    public void setName(String newName) {
        name.put(newName);
    }
}
```
Outline

1. Context
2. Motivating Example
3. Software Transactional Memory
4. Domain Modeling Language
5. Conclusions and Future Work
DML: Representing the structure of a domain

DML was designed with the following in mind:

- It should be easy to learn and to use for a Java programmer.
- It should represent **only** the structural part of a domain model.
- It should allow the programmer to add behavior to domain entities.

Did I mention that DML should be easy to learn and to use for a Java programmer?

To satisfy this requirement, we designed the DML syntax to follow Java’s syntax very closely.
DML: Representing the structure of a domain

DML was designed with the following in mind:

- It should be easy to learn and to use for a Java programmer.
- It should represent only the structural part of a domain model.
- It should allow the programmer to add behavior to domain entities.

Did I mention that DML should be easy to learn and to use for a Java programmer?

To satisfy this requirement, we designed the DML syntax to follow Java’s syntax very closely.
DML: Representing the structure of a domain

DML was designed with the following in mind:

- It should be easy to learn and to use for a Java programmer.
- It should represent only the structural part of a domain model.
- It should allow the programmer to add behavior to domain entities.

Did I mention that DML should be easy to learn and to use for a Java programmer?

To satisfy this requirement, we designed the DML syntax to follow Java’s syntax very closely.
DML: Representing the structure of a domain

DML was designed with the following in mind:

- It should be easy to learn and to use for a Java programmer.
- It should represent **only** the structural part of a domain model.
- It should allow the programmer to add behavior to domain entities.

Did I mention that DML should be easy to learn and to use for a Java programmer?

To satisfy this requirement, we designed the DML syntax to follow Java’s syntax very closely.
DML: Representing the structure of a domain

DML was designed with the following in mind:

- It should be easy to learn and to use for a Java programmer.
- It should represent only the structural part of a domain model.
- It should allow the programmer to add behavior to domain entities.

Did I mention that DML should be easy to learn and to use for a Java programmer?

To satisfy this requirement, we designed the DML syntax to follow Java’s syntax very closely.
DML: Representing the structure of a domain

DML was designed with the following in mind:

- It should be easy to learn and to use for a Java programmer.
- It should represent only the structural part of a domain model.
- It should allow the programmer to add behavior to domain entities.

Did I mention that DML should be easy to learn and to use for a Java programmer?

To satisfy this requirement, we designed the DML syntax to follow Java’s syntax very closely.
PetStore domain recast in DML

```java
class Category {
    String name;
    String description;
}
class Product {
    String name;
    String description;
}
class NamedEntity {
    String name;
    String description;
}
class Category extends NamedEntity;
class Product extends NamedEntity;

relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```
PetStore domain recast in DML

```java
class Category {
    String name;
    String description;
}
class Product {
    String name;
    String description;
}
class NamedEntity {
    String name;
    String description;
}
class Category extends NamedEntity;
class Product extends NamedEntity;

relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```

João Cachopo (Inesc-ID)
PetStore domain recast in DML

class Category {
    String name;
    String description;
}
class Product {
    String name;
    String description;
}
class NamedEntity {
    String name;
    String description;
}
class Category extends NamedEntity;
class Product extends NamedEntity;

relation BelongsTo {
    Category playsRole category;  
    Product playsRole product {
        multiplicity 0..*;
    }
}
PetStore domain recast in DML

class Category {
    String name;
    String description;
}

class Product {
    String name;
    String description;
}

class NamedEntity {
    String name;
    String description;
}

class Category extends NamedEntity;

class Product extends NamedEntity;

relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
PetStore domain recast in DML

class Category {
    String name;
    String description;
}

class Product {
    String name;
    String description;
}

class NamedEntity {
    String name;
    String description;
}

class Category extends NamedEntity;

class Product extends NamedEntity;

relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
PetStore domain recast in DML

```java
class Category {
    String name;
    String description;
}
class Product {
    String name;
    String description;
}
class NamedEntity {
    String name;
    String description;
}
class Category extends NamedEntity {
}
class Product extends NamedEntity {
}

relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```
Code generation: The class construct

```java
class Category {
    String name;
    String description;
}

abstract class Category_Base {
    String getName() {...}
    void setName(String name) {...}
    String getDescription() {...}
    void setDescription(String desc) {...}
}

To ensure the round-trip:

class Category extends Category_Base {}
Code generation: The class construct

class Category {
    String name;
    String description;
}

abstract class Category_Base {
    String getName() {...}
    void setName(String name) {...}
    String getDescription() {...}
    void setDescription(String desc) {...}
}

To ensure the round-trip:

class Category extends Category_Base {}
Code generation: The class construct

class Category {
    String name;
    String description;
}

abstract class Category_Base {
    String getName() {...}
    void setName(String name) {...}
    String getDescription() {...}
    void setDescription(String desc) {...}
}

To ensure the round-trip:
class Category extends Category_Base {}
Code generation: The class construct

class Category {
    String name;
    String description;
}

abstract class Category_Base {
    String getName() {...}
    void setName(String name) {...}
    String getDescription() {...}
    void setDescription(String desc) {...}
}

To ensure the round-trip:

class Category extends Category_Base {}
Code generation: The class construct

class Category {
    String name;
    String description;
}

abstract class Category_Base {
    String getName() {...}
    void setName(String name) {...}
    String getDescription() {...}
    void setDescription(String desc) {...}
}

To ensure the round-trip:

class Category extends Category_Base {}
Code generation: The relation construct

```java
relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```

```java
abstract class Category_Base {
    int getProductCount() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}
```

```java
abstract class Product_Base {
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
}
```
Code generation: The relation construct

```java
relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```

```java
abstract class Category_Base {
    ...
    int getProductCount() {...}
    boolean hasAnyProduct() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}

abstract class Product_Base {
    ...
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
}
```

João Cachopo (Inesc-ID) Combining STM with DML ICWE 2006 23 / 30
Code generation: The relation construct

```java
relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```

```java
abstract class Category_Base {
    ...
    int getProductCount() {...}
    boolean hasAnyProduct() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}

abstract class Product_Base {
    ...
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
}
```
Code generation: The relation construct

```
relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```

```
abstract class Category_Base {
    ...
    int getProductCount() {...}
    boolean hasAnyProduct() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}
```

```
abstract class Product_Base {
    ...
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
}
```
Code generation: The relation construct

```
relation BelongsTo {
    Category  playsRole  category;
    Product   playsRole  product {
        multiplicity 0..*;
    }
}

abstract class Category_Base {
    ...
    int getProductCount() {...}
    boolean hasAnyProduct() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}

abstract class Product_Base {
    ...
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
}
Code generation: The relation construct

```java
relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```

```java
abstract class Category_Base {
    ...
    int getProductCount() {...}
    boolean hasAnyProduct() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}
```

```java
abstract class Product_Base {
    ...
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
```
Code generation: The relation construct

```java
relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```

```java
abstract class Category_Base {
...
    int getProductCount() {...}
    boolean hasAnyProduct() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}

abstract class Product_Base {
...
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
}
```
Code generation: The relation construct

```java
relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```

```java
abstract class Category_Base {
    ...
    int getProductCount() {...}
    boolean hasAnyProduct() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}

abstract class Product_Base {
    ...
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
}
Code generation: The relation construct

```java
relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```

```java
abstract class Category_Base {
...
    int getProductCount() {...}
    boolean hasAnyProduct() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}

abstract class Product_Base {
...
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
}
```
Code generation: The relation construct

relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}

abstract class Category_Base {
    ...
    int getProductCount() {...}
    boolean hasAnyProduct() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}

abstract class Product_Base {
    ...
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
}
Code generation: The relation construct

```java
relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}

abstract class Category_Base {
    ...
    int getProductCount() {...}
    boolean hasAnyProduct() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}

abstract class Product_Base {
    ...
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
}
```
Code generation: The relation construct

relation BelongsTo {  
    Category playsRole category;  
    Product playsRole product {  
        multiplicity 0..*;  
    }  
}

abstract class Category_Base {  
    ...  
    int getProductCount() {...}  
    boolean hasAnyProduct() {...}  
    Set<Product> getProductSet() {...}  
    void addProduct(Product product) {...}  
    void removeProduct(Product product) {...}  
}

abstract class Product_Base {  
    ...  
    Category getCategory() {...}  
    void setCategory(Category category) {...}  
    boolean hasCategory() {...}  
    void removeCategory() {...}  
}
Code generation: The relation construct

```java
relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}
```

```java
abstract class Category_Base {
    ...  
    int getProductCount() {...}
    boolean hasAnyProduct() {...}
    Set<Product> getProductSet() {...}
    void addProduct(Product product) {...}
    void removeProduct(Product product) {...}
}

abstract class Product_Base {
    ...  
    Category getCategory() {...}
    void setCategory(Category category) {...}
    boolean hasCategory() {...}
    void removeCategory() {...}
}
```
relation BelongsTo {
    Category playsRole category;
    Product playsRole product {
        multiplicity 0..*;
    }
}

abstract class Category_Base {
    ... 
    int getProductCount() {...} 
    boolean hasAnyProduct() {...} 
    Set<Product> getProductSet() {...} 
    void addProduct(Product product) {...} 
    void removeProduct(Product product) {...} 
}

abstract class Product_Base {
    ... 
    Category getCategory() {...} 
    void setCategory(Category category) {...} 
    boolean hasCategory() {...} 
    void removeCategory() {...} 
}
Automatic relationship management

The code generated for a relationship construct ensures the correct update of both ends of the relationship, independently of how the programmer manipulates it.

```java
// The following are equivalent
// Pick one, or more...

category.addProduct(product);
product.setCategory(category);
category.getProductSet().add(product);

// This
product.setCategory(newCategory);

// is the same as
Category oldCategory = product.getCategory();
oldCategory.removeProduct(product);
product.setCategory(newCategory);
newCategory.addProduct(product);
```
Automatic relationship management

The code generated for a relationship construct ensures the correct update of both ends of the relationship, independently of how the programmer manipulates it.

// The following are equivalent
// Pick one, or more...

category.addProduct(product);

product.setCategory(category);

category.getProductSet().add(product);

// This is the same as
Category oldCategory = product.getCategory();
oldCategory.removeProduct(product);
product.setCategory(newCategory);
newCategory.addProduct(product);
Automatic relationship management

The code generated for a relationship construct ensures the correct update of both ends of the relationship, independently of how the programmer manipulates it.

```java
// The following are equivalent
// Pick one, or more...

category.addProduct(product);
product.setCategory(category);
category.getProductSet().add(product);

// This
product.setCategory(newCategory);
// is the same as
Category oldCategory = product.getCategory();
oldCategory.removeProduct(product);
product.setCategory(newCategory);
newCategory.addProduct(product);
```
Automatic relationship management

The code generated for a relationship construct ensures the correct update of both ends of the relationship, independently of how the programmer manipulates it.

// The following are equivalent
// Pick one, or more...

category.addProduct(product);
product.setCategory(category);
category.getProductSet().add(product);

// This
product.setCategory(newCategory);

// is the same as
Category oldCategory = product.getCategory();
oldCategory.removeProduct(product);
product.setCategory(newCategory);
newCategory.addProduct(product);
Automatic relationship management

The code generated for a relationship construct ensures the correct update of both ends of the relationship, independently of how the programmer manipulates it.

```
// The following are equivalent
// Pick one, or more...

category.addProduct(product);
product.setCategory(category);
category.getProductSet().add(product);
```

```
// This
product.setCategory(newCategory);

// is the same as
Category oldCategory = product.getCategory();
oldCategory.removeProduct(product);
product.setCategory(newCategory);
newCategory.addProduct(product);
```
Automatic relationship management

The code generated for a relationship construct ensures the correct update of both ends of the relationship, independently of how the programmer manipulates it.

```
// The following are equivalent
// Pick one, or more...

category.addProduct(product);
product.setCategory(category);
category.getProductSet().add(product);

// This
product.setCategory(newCategory);

// is the same as
Category oldCategory = product.getCategory();
oldCategory.removeProduct(product);
product.setCategory(newCategory);
newCategory.addProduct(product);
```
Automatic relationship management

The code generated for a relationship construct ensures the correct update of both ends of the relationship, independently of how the programmer manipulates it.

// The following are equivalent
// Pick one, or more...

category.addProduct(product);

product.setCategory(category);

category.getProductSet().add(product);

// This
product.setCategory(newCategory);

// is the same as
Category oldCategory = product.getCategory();
oldCategory.removeProduct(product);
product.setCategory(newCategory);
newCategory.addProduct(product);
Automatic relationship management

The code generated for a relationship construct ensures the correct update of both ends of the relationship, independently of how the programmer manipulates it.

// The following are equivalent
// Pick one, or more...
category.addProduct(product);
product.setCategory(category);
category.getProductSet().add(product);

// This
product.setCategory(newCategory);

// is the same as
Category oldCategory = product.getCategory();
oldCategory.removeProduct(product);
product.setCategory(newCategory);
newCategory.addProduct(product);
Outline

1. Context
2. Motivating Example
3. Software Transactional Memory
4. Domain Modeling Language
5. Conclusions and Future Work
The Fénix project in numbers

We applied this approach to the Fénix web application, which supports the activities of a University campus.

- Fénix is in production since 2001, but always with continuous development of new functionalities.
- The current Fénix team consists of 20 members.
- The service layer has more than 1,000 services.
- The domain is composed of more than 240 domain classes and 320 bi-directional relationships.
- The Java source code consists of \( \sim 400,000 \) LOC.
- The user base consists of more than 10,000 users.
- The application is deployed over a cluster of three servers with two processors each.
The Fénix project in numbers

We applied this approach to the Fénix web application, which supports the activities of a University campus.

- Fénix is in production since 2001, but always with continuous development of new functionalities.
- The current Fénix team consists of 20 members.
- The service layer has more than 1,000 services.
- The domain is composed of more than 240 domain classes and 320 bi-directional relationships.
- The Java source code consists of ~400,000 LOC.
- The user base consists of more than 10,000 users.
- The application is deployed over a cluster of three servers with two processors each.
We applied this approach to the Fénix web application, which supports the activities of a University campus.

- Fénix is in production since 2001, but always with continuous development of new functionalities.
- The current Fénix team consists of 20 members.
- The service layer has more than 1,000 services.
- The domain is composed of more than 240 domain classes and 320 bi-directional relationships.
- The Java source code consists of ∼400,000 LOC.
- The user base consists of more than 10,000 users.
- The application is deployed over a cluster of three servers with two processors each.
The Fénix project in numbers

We applied this approach to the Fénix web application, which supports the activities of a University campus.

- Fénix is in production since 2001, but always with continuous development of new functionalities.
- The current Fénix team consists of 20 members.
- The service layer has more than 1.000 services.
- The domain is composed of more than 240 domain classes and 320 bi-directional relationships.
- The Java source code consists of ~ 400.000 LOC.
- The user base consists of more than 10.000 users.
- The application is deployed over a cluster of three servers with two processors each.
The Fénix project in numbers

We applied this approach to the Fénix web application, which supports the activities of a University campus.

- Fénix is in production since 2001, but always with continuous development of new functionalities.
- The current Fénix team consists of 20 members.
- The service layer has more than 1,000 services.
- The domain is composed of more than 240 domain classes and 320 bi-directional relationships.
- The Java source code consists of ~ 400,000 LOC.
- The user base consists of more than 10,000 users.
- The application is deployed over a cluster of three servers with two processors each.
The Fénix project in numbers

We applied this approach to the Fénix web application, which supports the activities of a University campus.

- Fénix is in production since 2001, but always with continuous development of new functionalities.
- The current Fénix team consists of 20 members.
- The service layer has more than 1,000 services.
- The domain is composed of more than 240 domain classes and 320 bi-directional relationships.
- The Java source code consists of $\sim 400,000$ LOC.
- The user base consists of more than 10,000 users.
- The application is deployed over a cluster of three servers with two processors each.
The Fénix project in numbers

We applied this approach to the Fénix web application, which supports the activities of a University campus.

- Fénix is in production since 2001, but always with continuous development of new functionalities.
- The current Fénix team consists of 20 members.
- The service layer has more than 1,000 services.
- The domain is composed of more than 240 domain classes and 320 bi-directional relationships.
- The Java source code consists of approximately 400,000 LOC.
- The user base consists of more than 10,000 users.
- The application is deployed over a cluster of three servers with two processors each.
The Fénix project in numbers

We applied this approach to the Fénix web application, which supports the activities of a University campus.

- Fénix is in production since 2001, but always with continuous development of new functionalities.
- The current Fénix team consists of 20 members.
- The service layer has more than 1,000 services.
- The domain is composed of more than 240 domain classes and 320 bi-directional relationships.
- The Java source code consists of \( \sim 400,000 \) LOC.
- The user base consists of more than 10,000 users.
- The application is deployed over a cluster of three servers with two processors each.
Changing Fénix to the new approach

We changed Fénix to the approach we described here during the second quarter of 2005, and the new version went into production on September 2005.

The change was performed incrementally:

- We wrote a backend for the DML compiler to generate the (base) domain classes exactly as they existed in Fénix.
- During a period of several weeks, the team converted the existing domain classes into a DML specification, splitting each class into its structure, represented in DML, and its behavior, represented, as before, as Java code.
- When the previous process finished, we changed the backend used to generate the domain classes to use the JVSTM.

Since then, the team has been cleaning up the leftovers of the previous approach that still exist in the code.
Changing Fénix to the new approach

We changed Fénix to the approach we described here during the second quarter of 2005, and the new version went into production on September 2005.

The change was performed incrementally:

- We wrote a backend for the DML compiler to generate the (base) domain classes exactly as they existed in Fénix.
- During a period of several weeks, the team converted the existing domain classes into a DML specification, splitting each class into its structure, represented in DML, and its behavior, represented, as before, as Java code.
- When the previous process finished, we changed the backend used to generate the domain classes to use the JVSTM.

Since then, the team has been cleaning up the leftovers of the previous approach that still exist in the code.
Changing Fénix to the new approach

We changed Fénix to the approach we described here during the second quarter of 2005, and the new version went into production on September 2005.

The change was performed incrementally:

- We wrote a backend for the DML compiler to generate the (base) domain classes exactly as they existed in Fénix.
- During a period of several weeks, the team converted the existing domain classes into a DML specification, splitting each class into its structure, represented in DML, and its behavior, represented, as before, as Java code.
- When the previous process finished, we changed the backend used to generate the domain classes to use the JVSTM.

Since then, the team has been cleaning up the leftovers of the previous approach that still exist in the code.
Changing Fénix to the new approach

We changed Fénix to the approach we described here during the second quarter of 2005, and the new version went into production on September 2005.

The change was performed incrementally:

- We wrote a backend for the DML compiler to generate the (base) domain classes exactly as they existed in Fénix.

- During a period of several weeks, the team converted the existing domain classes into a DML specification, splitting each class into its structure, represented in DML, and its behavior, represented, as before, as Java code.

- When the previous process finished, we changed the backend used to generate the domain classes to use the JVSTM.

Since then, the team has been cleaning up the leftovers of the previous approach that still exist in the code.
Changing Fénix to the new approach

We changed Fénix to the approach we described here during the second quarter of 2005, and the new version went into production on September 2005.

The change was performed incrementally:

- We wrote a backend for the DML compiler to generate the (base) domain classes exactly as they existed in Fénix.
- During a period of several weeks, the team converted the existing domain classes into a DML specification, splitting each class into its structure, represented in DML, and its behavior, represented, as before, as Java code.
- When the previous process finished, we changed the backend used to generate the domain classes to use the JVSTM.

Since then, the team has been cleaning up the leftovers of the previous approach that still exist in the code.
Changing Fénix to the new approach

We changed Fénix to the approach we described here during the second quarter of 2005, and the new version went into production on September 2005.

The change was performed incrementally:

- We wrote a backend for the DML compiler to generate the (base) domain classes exactly as they existed in Fénix.
- During a period of several weeks, the team converted the existing domain classes into a DML specification, splitting each class into its structure, represented in DML, and its behavior, represented, as before, as Java code.
- When the previous process finished, we changed the backend used to generate the domain classes to use the JVSTM.

Since then, the team has been cleaning up the leftovers of the previous approach that still exist in the code.
The benefits for the Fénix development

After the change we identified the following major benefits:

- The data-corruption errors disappeared, because they were caused mostly by misplaced locks.
- The stability and the robustness of the application increased, because errors such as dangling references disappeared.
- There was a perceived increase in the performance of the application after an initial warm-up.
- The development of new functionalities is significantly faster because it requires less coding and less debugging.
The benefits for the Fénix development

After the change we identified the following major benefits:

- The data-corruption errors disappeared, because they were caused mostly by misplaced locks.
- The stability and the robustness of the application increased, because errors such as dangling references disappeared.
- There was a perceived increase in the performance of the application after an initial warm-up.
- The development of new functionalities is significantly faster because it requires less coding and less debugging.
The benefits for the Fénix development

After the change we identified the following major benefits:

- The data-corruption errors disappeared, because they were caused mostly by misplaced locks.
- The stability and the robustness of the application increased, because errors such as dangling references disappeared.
- There was a perceived increase in the performance of the application after an initial warm-up.
- The development of new functionalities is significantly faster because it requires less coding and less debugging.
The benefits for the Fénix development

After the change we identified the following major benefits:

- The data-corruption errors disappeared, because they were caused mostly by misplaced locks.
- The stability and the robustness of the application increased, because errors such as dangling references disappeared.
- There was a perceived increase in the performance of the application after an initial warm-up.
- The development of new functionalities is significantly faster because it requires less coding and less debugging.
The benefits for the Fénix development

After the change we identified the following major benefits:

- The data-corruption errors disappeared, because they were caused mostly by misplaced locks.
- The stability and the robustness of the application increased, because errors such as dangling references disappeared.
- There was a perceived increase in the performance of the application after an initial warm-up.
- The development of new functionalities is significantly faster because it requires less coding and less debugging.
Future Work

There are many interesting things that we are already working on:

- We are extending the DML so that we can specify access control rules for both the domain entities and the relationships.
- We are exploring the idea of specifying composite relationships on the DML.
- We are experimenting with different implementation strategies for the JVSTM, so that we can reduce the space overheads we currently have.
- We are exploring the JVSTM’s ability of identifying all the relevant data for a page to automatically manage a cache of frequently accessed public pages.
- We are exploring the use of long transactions to support the execution of workflows within the application.
- ...
Future Work

There are many interesting things that we are already working on:

- We are extending the DML so that we can specify access control rules for both the domain entities and the relationships.
- We are exploring the idea of specifying composite relationships on the DML.
- We are experimenting with different implementation strategies for the JVSTM, so that we can reduce the space overheads we currently have.
- We are exploring the JVSTM’s ability of identifying all the relevant data for a page to automatically manage a cache of frequently accessed public pages.
- We are exploring the use of long transactions to support the execution of workflows within the application.
- ...
Future Work

There are many interesting things that we are already working on:

- We are extending the DML so that we can specify access control rules for both the domain entities and the relationships.
- We are exploring the idea of specifying composite relationships on the DML.
- We are experimenting with different implementation strategies for the JVSTM, so that we can reduce the space overheads we currently have.
- We are exploring the JVSTM’s ability of identifying all the relevant data for a page to automatically manage a cache of frequently accessed public pages.
- We are exploring the use of long transactions to support the execution of workflows within the application.
Future Work

There are many interesting things that we are already working on:

- We are extending the DML so that we can specify access control rules for both the domain entities and the relationships.
- We are exploring the idea of specifying composite relationships on the DML.
- We are experimenting with different implementation strategies for the JVSTM, so that we can reduce the space overheads we currently have.
  - We are exploring the JVSTM’s ability of identifying all the relevant data for a page to automatically manage a cache of frequently accessed public pages.
- We are exploring the use of long transactions to support the execution of workflows within the application.
- ...
Future Work

There are many interesting things that we are already working on:

- We are extending the DML so that we can specify access control rules for both the domain entities and the relationships.
- We are exploring the idea of specifying composite relationships on the DML.
- We are experimenting with different implementation strategies for the JVSTM, so that we can reduce the space overheads we currently have.
- We are exploring the JVSTM’s ability of identifying all the relevant data for a page to automatically manage a cache of frequently accessed public pages.
- We are exploring the use of long transactions to support the execution of workflows within the application.

...
Future Work

There are many interesting things that we are already working on:

- We are extending the DML so that we can specify access control rules for both the domain entities and the relationships.
- We are exploring the idea of specifying composite relationships on the DML.
- We are experimenting with different implementation strategies for the JVSTM, so that we can reduce the space overheads we currently have.
- We are exploring the JVSTM’s ability of identifying all the relevant data for a page to automatically manage a cache of frequently accessed public pages.
- We are exploring the use of long transactions to support the execution of workflows within the application.

...
Future Work

There are many interesting things that we are already working on:

- We are extending the DML so that we can specify access control rules for both the domain entities and the relationships.
- We are exploring the idea of specifying composite relationships on the DML.
- We are experimenting with different implementation strategies for the JVSTM, so that we can reduce the space overheads we currently have.
- We are exploring the JVSTM’s ability of identifying all the relevant data for a page to automatically manage a cache of frequently accessed public pages.
- We are exploring the use of long transactions to support the execution of workflows within the application.
- ...
The End

Thank you.

Questions?