BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

REIMPLEMENTATION OF COREDATA IN C++

OBJEKTOVÁ DATABÁZE TYPU COREDATA PRO C++

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MICHAL HORNICKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. MARTIN HRUBÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Zadání bakalářské práce/19873/2016/xhorni14

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Hornický Michal**
Obor: Informační technologie
Téma: **Objektová databáze typu CoreData pro C++**
Reimplementation of CoreData in C++
Kategorie: Databáze

Pokyny:

1. Prostudujte knihovnu CoreData a objektové databáze jako nadstavbu nad relačními DB.
2. Navrhněte systém objektové databáze charakteru CoreData pro implementaci v jazyce C++. Při návrhu se inspirujte koncepcí CoreData.
3. Implementujte navržený systém v podobě knihovny pro C++. Knihovnu přizpůsobte pro snadou implementaci dodatečných databázových připojení. Implementujte napojení na Sqlite3.
4. Knihovnu testujte na jednoduché demonstrační aplikaci. Proveďte výkonnostní testy knihovny.

Literatura:

- Isted, T., Harrington, T.: Core Data for iOS: Developing Data-Driven Applications for the iPad, iPhone, and iPod touch (Core Frameworks Series), Addison-Wesley Professional; 1. edition (May 24, 2011).

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hrubý Martin, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstract

Core Data is important component of software ecosystems on Apple's platforms. The main shortcoming of Core Data is the limited number of platforms, on which it can be utilized. This bachelor's thesis documents the process of analysis of Core Data, subsequent design and implementation of CoreStore library. Implemented library is written in C++, and aims to provide functionality of Core Data on other platforms.

Abstrakt

Core Data je kľučovou súčastou softvérových ekosystémov na platformách od firmy Apple. Hlavný nedostatok Core Data je obmendzené množstvo platforiem, na ktorých je možné túto knižnicu využívať. Táto bakalárska práca sa zaoberá analýzou tejto knižnice, a následným návrhom a implementáciou knižnice CoreStore, ktorej gólom je poskytovať funkcionalitu Core Data pre aplikácie vyvíjane v programovacom jazyku C++.

Keywords

Object database, Core Data, Apple, C++, SQLite

Klíčová slova

Objektová databáza, Core Data, Apple, C++, SQLite

Reference

HORNICKÝ, Michal. *Reimplementation of CoreData in C++*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Hrubý Martin.

Reimplementation of CoreData in C++

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Martin Hrubý, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Michal Hornický
May 14, 2017

Acknowledgements

I would like to thank Ing. Martin Hrubý, Ph.D., the supervisor of this thesis, for extremely valuable feedback provided during frequent meetings.

Contents

Chapter 1

Introduction

Apart from processing data, most software requires a simple way to store structured data. Apple's Core Data aims to solve this problem. Core Data is object graph persistence and management framework, working very much like Object-oriented database for Objective-C runtime. It provides an easy way for applications to store highly object-oriented data. It also provides methods for effectively managing this data and very powerful query system. However, its requirements limit its usability. It requires the Objective-C runtime, and only works on Apple platforms. The goal of this thesis is to provide an open-source alternative, which does not require the heavy Objective-C runtime and is not tied to the specific operating system. The implementation language is C++, which can run on a wide variety of platforms, and its runtime library is much smaller.

The product of this thesis is design and implementation of *CoreStore* library. It roughly follows the design of Core Data, while adapting Core Data concepts to C++ ecosystem. It's implemented using modern C++, and should be easy to incorporate into existing software.

This thesis consists of four chapters. The first chapter describes Core Data and its structure. The second chapter describes design of CoreStore library. The third chapter more closely describes actual library implementation, and explains how the concepts specified in Chapter 2 were translated into working, ergonomic C++ code. It also aims to show basic usage patterns of implemented library. The last chapter deals with verification and testing of the implementation. The library was thoroughly tested, and this Chapter explains how actual testing was performed. It also presents the measured performance of implementation.

Throughout this thesis, several terms that might not be familiar to the reader are used. The meaning of most of these terms is the same as it is in Core Data or similar database systems. The term *User* is used in the following sense: *User* is a programmer, that uses CoreStore library in his/her software, in order to persist some data across multiple executions of developed software.

Chapter 2

Current state

Most frequently used database systems are based on the relational data model. This model is based on the mathematical concept of relation and uses tables with columns and rows to store data[5]. This abstraction does not directly map to objects in object-oriented languages. This gap is commonly solved by introducing object-relational[1] mapping(ORM) layer into developed software. Object-relational mappers serve as translation layer. This translation layer provides tools for modelling object-oriented data structures, and internally translates created object-oriented model into the relational data model.

Object-oriented databases provide abstractions better suited for modelling object-oriented data. They model data as a set of objects belonging to specific class. Each object can be uniquely identified and contains attributes and references to other objects. This abstraction maps directly to object-oriented programming languages and therefore is better suited for modelling object-oriented data[7]. Object oriented databases are commonly very tightly integrated with one language/runtime, and use hosting language constructs for performing individual actions.

These are some object-oriented databases that were at evaluated during the design of the CoreStore library.

ObjectDatabase++ - Is object-oriented database management system for C++ and C#, designed for easy embeddability into existing code, and server usage with minimal maintenance. It provides support for multi-process transactions, and interesting indexing methods.

Objectivity/DB - Is multi-language object database, supporting C++, C#, Java or Python. It is designed as distributed database solution, with highreliability guarantees with high data throughput. It supports multiple platforms and is arguably most popular object-oriented database in the enterprise context.

ObjectStore - Is Object-oriented database for C++, that heavily integrates into C++ language, allowing the creation of database object by using an overloaded *new*

[1] https://en.wikipedia.org/wiki/Object-relational_mapping

operator, and dynamic loading of object by trapping pointer exceptions. It is used heavily in many fields, including telecomunications, GIS[2] and government services.

Core Data - Is object oriented database for Objective-C runtime developed by Apple Inc. that served as basis of this thesis. Its structure is explained in following section.

2.1 Core Data

Apple's Core Data is an object graph and persistence framework developed for their macOS and iOS platforms. It could be classified as ORM, but it more closely resembles object oriented database. It provides tools for storing complex object graphs into a persistent medium. It also provides abstractions for detailed management of individual object lifetimes, monitoring changes to these objects, validating of values stored in them and efficient integration with user interfaces. Other capabilities include effective data migrations after changes to the data model, that can be automatic or manual. One of the most powerful features of Core Data is extremely sophisticated query capability. Programmer working with Core Data can perform complex queries against persistent storage medium, and have the user interface dynamically updated based on changes in object graph[2].

2.1.1 Core Data stack

This section aims to examine Core Data stack in terms of its components. Each component has one specific purpose in this system. Figure 2.1 shows what position in this stack several of the most important components occupy.

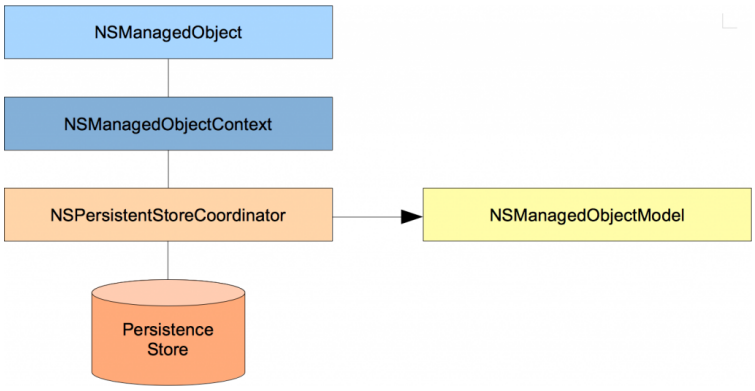

Figure 2.1: Core Data stack [3]

[2]https://en.wikipedia.org/wiki/Geographic_information_system
[2]Image retrieved from: https://blog.codecentric.de/en/2014/11/concurrency-coredata/

NSManagedObjectModel

This class contains detailed data model. It consists of collection of entity models (*NSEntityDescrition*). It is created from a *.mom* file, which is bundled with an application, and contains data model in XML format. It provides methods for retrieving individual entity models.

NSEntityDescription

Contains metadata associated with one particular entity - it name, attributes and relationships. It also contains references to *NSEntityDescription*s of derived and deriving classes.

NSPersistentStoreCoordinator

This component serves as an intermediate layer between *NSManagedObjectContext* and *NSPersistentStore*. It groups different persistent stores, and presents an interface to *NSManagedObjectContext* so that multiple persistent stores can appear as single one. This component allows for storing multiple objects within single object graph in multiple persistent stores.

NSPersistentStore

This component presents an interface to storage medium. The medium can have different forms based on speed, concurrency and safety requirements. It has an interface for loading and saving information to persistent medium, and heavily uses object-oriented class hierarchy. Provided store types include: *XML, Binary, SQLite, in-memory.* XML persistent store uses XML files to persist objects. This type of store does not provide exceptional performance. Binary persistent store serializes objects into binary format and stores them as files. This store provides acceptable performance and has small storage requirements. The SQLite persistent store uses SQLite database to store objects. It provides acceptable performance and very fast queries.The in-memory persistent store is fastest of all persistent stores, but it does not persist data across multiple executions of an application.

NSManagedObjectContext

This is the most important component in Core Data. The programmer mainly interacts with this component. Its primary responsibility is to manage a collection of *NSManagedObject*s that constitute an object graph. Each managed object in this context must be unique, with respect to its *Id*. *NSManagedObjectContext* controls object lifetime, allowing it to load and unload(fault) objects on demand and perform validation of objects and their

attributes/relationships. It is also responsible for proper handling of relationships and ensuring consistency of inverse relationships[1].

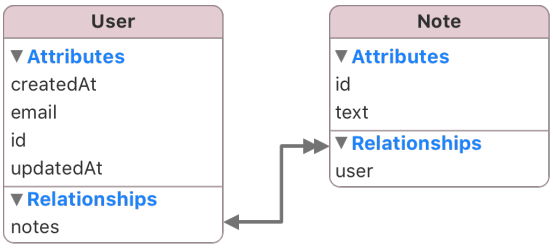

Figure 2.2: Example inverse relationship[4]

Figure 2.2 shows simple data model with an inverse relationship. In this model, entity **User** has a relationship with entity **Note** named **User**.notes. This relationship is mirrored by relationship from **Note** to **User** named **Note**.user. These 2 relationships denote same information from 2 different places. In simple ORM, updating one relationship would require a manual update of inverse relationship, or one relationship to be dynamically generated from the other one. In Core Data it is *NSManagedObjectContext*'s job to ensure these types of relationships are kept consistent. By removing one object from **User**.notes, the removed object should also have its **Note**.user set to null. Core Data ensures this consistency only when inverse relationships are constructed and marked as such in data model editor.

NSManagedObjectId

Is unique identifier of *NSManagedObject* in its *NSManagedObjectContext*. It contains fields that identify *NSPersistentStore* in which this object is stored, *NSEntityDescription* to which object belongs, and unique per-instance identification. *NSManagedObjectId* must be unique.

NSManagedObject

This class is the root of the persistable class hierarchy. It contains several attributes crucial for Core Data. These attributes include *NSManagedObjectId* and reference to *NSManagedObjectContext*. Object values can be retrieved and modified by *setValue:forKey* and *getValue:forKey* methods. Upon the first retrieval of an object from the persistent store, this object does not contain its attributes or relationships - it is in the *fault* state. Actual values are loaded only on demand. This can be altered by using fetch request with eager loading of object attributes and specific relationships.

[4]Image retrieved from: https://cocoapods.org/pods/Sync

2.1.2 Core Data usage

Probably the first step of integrating Core Data into an application is the creation of data model. For this purpose, Apple provides graphical editor, that is included with XCode integrated development environment. After data model is created, it is stored in *.xcdatamodel* or *.xcdatamodeld* file. The latter allows for storing multiple data models, and versioning of these models. This file is compiled into *mom* or *momd* file, that is included with software, and is loaded during execution time into an instance of *NSManagedObjectModel* class. *NSManagedObjectModel* is then in turn used to create core component of Core Data stack: *NSManagedObjectContext*

```
guard let modelURL = NSBundle.mainBundle().URLForResource("DataModel",
 ↪  withExtension:"momd") else {
    fatalError("Error loading model from bundle")
}
guard let mom = NSManagedObjectModel(contentsOfURL: modelURL) else {
    fatalError("Error initializing mom from: \(modelURL)")
}
let psc = NSPersistentStoreCoordinator(managedObjectModel: mom)
managedObjectContext = NSManagedObjectContext(concurrencyType:
 ↪  .MainQueueConcurrencyType)
managedObjectContext.persistentStoreCoordinator = psc
let urls =
 ↪  NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory,
 ↪  inDomains: .UserDomainMask)
let docURL = urls[urls.endIndex-1]
let storeURL = docURL.URLByAppendingPathComponent("DataModel.sqlite")
do {
    try psc.addPersistentStoreWithType(NSSQLiteStoreType, configuration:
     ↪  nil, URL: storeURL, options: nil)
} catch {
    fatalError("Error migrating store: \(error)")
}
```

Listing 1: Creation of Core Data stack in swift

Code listing 1 shows initialization of Core Data stack. The first step in this process is the loading of *.momd* file and creation of *NSManagedObjectModel*. Then, new *NSManagedObjectContext* is created, and *NSPersistentStoreCoordinator* is attached to this context. The last step is creation of *NSPersistentStore*, and attachment of this store to *NSPersistentStoreCoordinator*. In this example, the data model is loaded from application resources, and the database is created in document directory.

```
let employee =
↪  NSEntityDescription.insertNewObjectForEntityForName("Employee",
↪  inManagedObjectContext: managedObjectContext) as! EmployeeMO
```

Listing 2: Creating new persistent object in swift

Code listing 2 shows the creation of new managed object in Core Data. This action requires *NSEntityDescription* of the entity, to which created object should belong. *NSEntityDescription* is provided internally, based on first argument provided to *insertNewObjectForEntityName*.

```
let employeesFetch = NSFetchRequest(entityName: "Employee")
employeesFetch.predicate = NSPredicate(format: "firstName == %@", "Trevor")
do {
    let fetchedEmployees = try moc.executeFetchRequest(employeesFetch) as!
    ↪  [AAAEmployeeMO]
} catch {
    fatalError("Failed to fetch employees: \(error)")
}
```

Listing 3: Fetching objects, and filtering

Code listing 3 shows creation of fetch request, application of predicate to this request, and then execution of created fetch request on *NSManagedObjectContext*.

Chapter 3

System Design

This chapter describes conceptual design of CoreStore library. Components and algorithms defined in this section were inspired by their counterparts in Core Data, and adapted to for easier implementation in C++. Designed library does not provide all functionality found in Core Data. It can properly manage, save and query for objects in an object graph. It does not, however, support some of the advanced features of Core Data, such as change notification and undo/redo management.

3.1 Component definitions

This section defines individual components of CoreStore stack. Each component is defined as a tuple of attributes and implemented as C++ class. Each component provides a specific set of functionality, some of which is described later in this thesis.

ObjectContext - (Store, DatabaseModel, Prototypes, Cached, ToSave, ToDelete)

Store - Instance of PersistentStore class, used to interface with persistent store

DBModel - Instance of DatabaseModel, which itself is set of EntityModels, must conform to Algorithm 1.

Prototypes - Set containing instances of entity prototypes(For each storable class one instance)

Cached - set of cached Objects

ToSave - set of references to objects, which were modified/added, and must be saved, $ToSave \subseteq Cached$

ToDelete - set of ObjectIds, that reference objects, which must be deleted, $\forall id \in ToDelete : \nexists o \in Cached : o.Id = id$

ObjectContext is most important part of CoreStore stack. This component is solely responsible for managing object graph. Almost all actions, the user might want to perform

are implemented as methods on this class. It contains multiple sets of objects, which are implemented as C++ maps with *ObjectId* as key for fast search.

During creation of ObjectContext, user must provide list of persistable classes and PersistentStore that will be associated with this context throughout its lifetime. List of provided objects, which represent persistable classes/entities, is checked for consistency (specified in Algorithm 1). After data model is verified to be valid, provided PersistentStore is initialized. Provided persistent store is responsible for creating and maintaining its internal metadata structures.

During runtime, the purpose of ObjectContext is to serve as a repository of objects. It holds storable objects, tracks deleted and modified objects. It also provides methods for querying the persistent store for a specific set of objects, based on their class and optionally a predicate.

Changes to objects are not saved immediately. ObjectContext holds set of modified objects(*ToSave*), and set of deleted objects(*ToDelete*). When *save* method is invoked ObjectContext will save all objects from the *ToSave* set, and delete all objects from the *ToDelete* set of its persistent store. This action is also performed upon the destruction of ObjectContext.

No persistable object is allowed to outlive this component. If persistable object outlives ObjectContext, it is considered in an implicit error state, and all actions on this object will throw an exception since this is a serious mistake on user's part. Objects in the *Detached* state have limited capabilities.

ObjectId - (EntityId,InstanceId)

EntityId - Name of the Entity/Class to which this object belongs

InstanceId - String describing specific instance of this object

ObjectId is a unique identifier of an object attached to one *ObjectContext*. It provides identification of Entity to which object belongs for purposes of retrieving EntityModel, and identification of a specific instance of an object.

In future, this class could be extended to contain identification of persistent store, in which the identified object is stored for multi-store capabilities. This functionality is provided by *NSPersistentStoreCoordinator* in Core Data. Every *ObjectId* is generated by persistent store, and upon detachment can no longer be used. Subsequent attachments to same or different *ObjectContext*s will require the generation of new *ObjectId* to prevent a collision. This ensures one of the primary invariants in this system, which is uniqueness of *ObjectId* in one *ObjectContext*.

Object - (Id,State,Values,Context)

Id - ObjectId of this object

State - ObjectState in which the object currently is

Values - Set of pairs(Name,Value), where Value might be primitive language value or single/multiple references to other objects, each *name* must be unique, $\forall(n1, v1) \in Values, \nexists(n2, v2) \in Values : n1 = n2 \wedge v1 \neq v2$

Context - Reference to context to which the object is currently attached

This component represents a persistable object in the system. It has a unique identifier, its current state, storage for persistable values and reference to ObjectContext, to which object is attached. This object is implemented as a pure virtual base class, from which all storable classes must be derived.

State of the object is only manipulated by its context. Action that requires modification of object's state might be initiated by the user on an object, eg: reading attribute, but is always carried out by context, since each state requires also a change in internal state of the context. This is one of the reasons, that object manipulation outside context is very limited, and object can't outlive its context while remaining in a valid state.

ObjectState - $\in$ {New,Fault,Loaded,Modified,Detached,Error}

$\mathbf{o}$ = instance of Object

New - $\mathbf{o}$.Context $\neq$ null, $\nexists Persistent(\mathbf{o})$

Fault - $\mathbf{o}$.Context $\neq$ null, $\mathbf{o}$.Values = $\emptyset$, $\exists Persistent(\mathbf{o})$

Loaded - $\mathbf{o}$.Context $\neq$ null, $\mathbf{o}$.Values = Persistent($\mathbf{o}$)

Modified - $\mathbf{o}$.Context $\neq$ null, $\mathbf{o}$.Values $\neq$ Persistent($\mathbf{o}$)

Detached - $\mathbf{o}$.Context = null

Error - Special state that only occurs if Object outlives its ObjectContext

Defines what state object is in, and what actions may be performed on this object. This restriction of possible actions only applies to primitive functions of ObjectContext. Some of these primitive functions are private to ObjectContext, and are used in public functions that perform validation of object's state and may invoke several other functions, to get to the desired state eg. detachObject not only detaches the object from its context but it also

loads this object if it is in the *Fault* state. *Error* state has a special purpose. This state cannot be reached by normal actions, and object can't leave this state.

AttributeModel - (Name,AttributeType)

Name - Name of the attribute

AttributeType - type of the attribute

AttributeModel stores necessary metadata associated with an attribute. This component may be duplicated in the system but is not modified at any point in time, so this duplication will not create any inconsistencies. This duplication is also present for *RelationshipModel*.

RelationshipModel - (EntityName,Name,RelationType,TargetEntity,TargetName)

EntityName - Name of entity, on which this relationship was defined

Name - Name of the relationship

RelationType - type of the relationship, one of {ToMany,ToOne}

TargetEntity - name of entity that is target of this relationship

TargetName - name of inverse relationship

RelationshipModel stores metadata associated with a single relationship. Since this component is used to generate names of tables used to store relationships, it must contain names of both source, and target entities. It also contains the name of inverse *RelationshipModel*, that can be found in target *EntityModel*.

EntityModel - (Name,Attributes,Relationships)

Name - Name of entity, referenced by ObjectId.EntityId

Attributes - set of AttributeModel

Relationships - set of RelationModel

This component is most important part of data model definition. Each persistable class must override *createModel* function, which returns *EntityModel* of that class.

EntityModel holds set of attribute and relationship models. These sets contain definitions of all storable attributes and relationships for this class and all classes from which this class is derived from. Inheriting of attribute and relationship models is crucial for correct handling of inheritance hierarchies in the object-oriented data model.

Both *Attributes* and *Relationships* are implemented as C++ maps. The name of each element is the key into the map. This name must also be unique between these two sets, and cannot repeat in inheritance hierarchies.

3.2 Object lifetime

This section describes object lifetime. By this term, author means the set of states which an object can occupy, how an object can transition between these states, and what changes to the object and its context will be performed by these transitions.

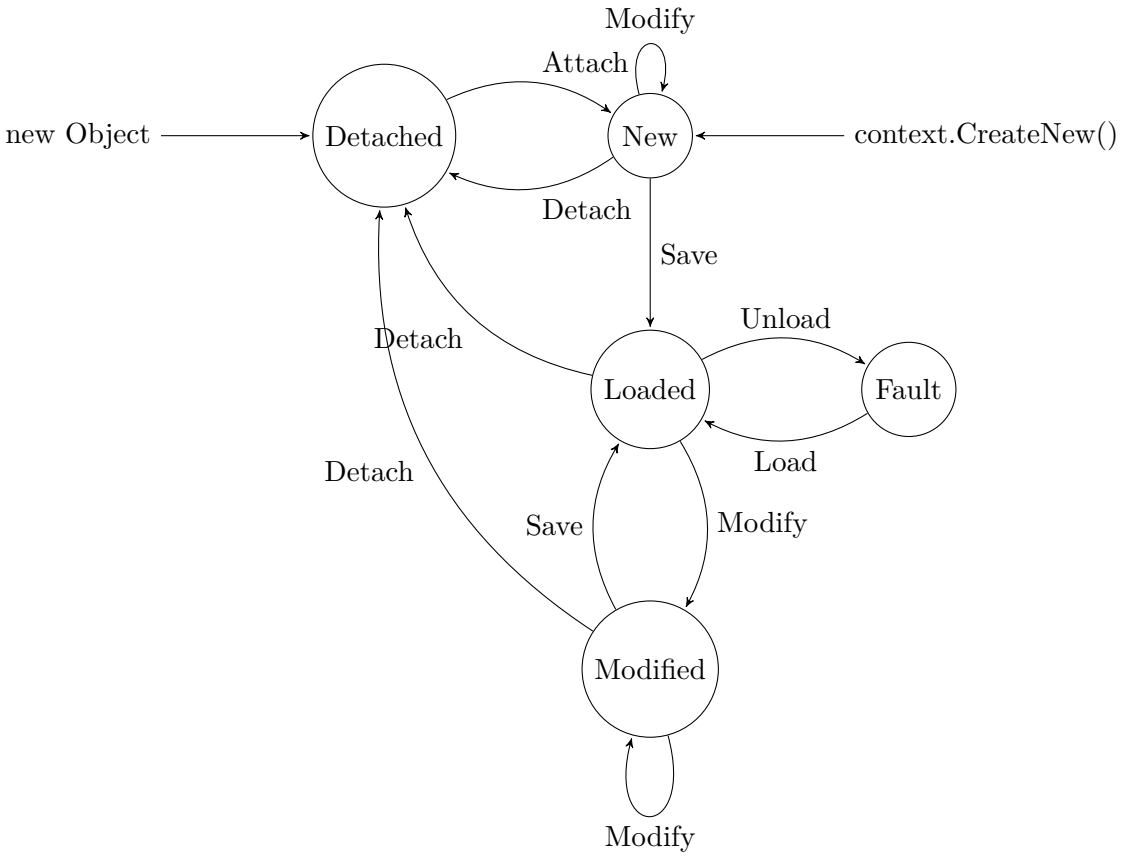

Figure 3.1: Object lifetime

	States				
Actions	New	Fault	Loaded	Modified	Detached
Save	✓ → *Loaded*	×	✓	✓ → *Loaded*	×
Load	×	✓ → *Loaded*	✓	✓ → *Loaded*	×
Unload	×	×	✓ → *Fault*	✓ → *Fault*	×
Detach	✓ → *Detached*	×	✓ → *Detached*	✓ → *Detached*	×
Attach	×	×	×	×	✓ → *New*
R. Attribute	✓	×	✓	✓	✓
R.Relation	✓	×	✓	✓	×
W. Attribute	✓	×	✓ → *Modified*	✓	✓
W. Relation	✓	×	✓ → *Modified*	✓	×

R. Attribute = Read attribute
W. Attribute = Write/modify attribute
✓ → *State* = Operation can proceed, object will be in *State* after operation
✓ = Operation can proceed, and won't change state
× = Operation can't proceed, however, there might be compound functions, which perform multiple transitions to perform specified operation.

Table 3.1: State transitions

Figure 3.1 presents a state diagram, which describes the lifetime of an object. It shows which particular primitive actions must be executed in order to get an object from one state to another. Table 3.1 contains a more detailed view of which actions can be performed in which states, and how they affect the state of an object.

Since requiring the exact sequence of actions from user in order to get an object into the specific state would pose serious usability problem, most of these actions are implemented as private functions. These private functions are in turn used by more complex public functions, which perform validation and modify object state as necessary. Thanks to this division, primitive functions are easily defined, and the user still has an intuitive interface to use.

Definition 1 (Map Notation) *Throughout this document notation* ***map[n]*** *will be used. It denotes insertion/retrieval from standard map/dictionary object. This notation is used on* Object.values *field, and* EntityModel.attributes *and* EntityModel.relationships*. While Object.values is defined as a set of pairs with unique first element in each pair and other fields are defined as sets of elements, where each element has a unique name, All of these fields are implemented as C++ maps. In the case of* Object.values *the first element of stored pair is used as a key, and in other cases, the* name *field is used as a key for lookup/insertion.*

Definition 2 (EntityModel notation) *Throughout following algorithms, the* ***EM(Object,ObjectContext)*** *notation will be used. This notation denotes retrieval of correct* EntityModel *for object* ***o****. This model is obtained by searching ObjectContext's DatabaseModel, and looking for EntityModel that has same Name as* ***o****.Id.EntityId.*

EM(o,oc) = EntityModel e: e ∈ oc ∧ e.Name = o.Id.EntityId *. Shorter variant may be used:* ***EM(o) = EM(o,o.Context)****.*

3.3 Core functions

This section presents detailed definitions of some of the core functions in order to exactly define most important parts of designed system. Implementation aims to follow these definitions as closely as possible while translating concepts presented here into an effective C++ code.

3.3.1 ObjectContext functions

ObjectContext is the main building block of the system. Its responsibilities include tracking and managing object state, creating new objects and persisting them using provided instance of *PersistentStore*.

These are primary functions available on ObjectContext that manage object lifetime & state. Function definitions are roughly split into two categories. Functions following naming convention *functionInternal* perform only primitive actions and are private to class on which they are implemented. Other functions might perform more complex actions, and are public - available to the user.

```
Input  : DatabaseModel model
forall EntityModel e ∈ model do
    forall RelationModel r ∈ e do
        if ∃ EntityModel ie: ie ∈ model ∧ ie.Name = r.TargetEntity then
            if ∃ RelationModel ir: ir ∈ ie.Relationships ∧ ir.Name = r.TargetName
             then
                continue;
            else
                return false;
            end
        else
            return false;
        end
    end
end
return true;
```

Algorithm 1: DatabaseModel.isValid()

Algorithm 1 defines a predicate that verifies whether data model is valid. This predicate is checked upon the construction of *ObjectContext*, and if the created *DatabaseModel* fails this predicate, construction of *ObjectContext* is not permitted. The purpose of this predicate is to check, whether all relationships in data model have valid inverse relationships.

All subsequent algorithms in this thesis assume that this predicate is satisfied - data model is valid.

Input : ObjectContext **oc**, Object **o**
Before : **o**.State = Detached
After : **o**.Context = **oc**, **o**.State = New
if ***o**.State* $\neq$ *Detached* **then**
  return *false*;
end
if $\exists$ *EntityModel* ***e*** = ***EM(o,oc)*** **then**
  o.Id.InstanceId = **oc**.Store.generateInstanceId(**e**);
  o.context = **oc**;
  oc.Cached = **oc**.Cached $\cup$ {**o**};
  oc.ToSave = **oc**.ToSave $\cup$ {**o**};
  o.State = New;
  return true;
else
  return false;
end

Algorithm 2: ObjectContext.attachObject

Algorithm 2 describes function is used for attaching objects to ObjectContext. By attaching an object to context, the context takes ownership of this object, will manage its state and will persist it into database when requested or upon destruction of said *ObjectContext*. Attached object retains its values, but it will have its *ObjectId* reset in order to avoid collisions.

Input : ObjectContext **oc**, Object **o**
Before : **o**.Context = **oc**, **o**.State ≠ Detached
After : **o**.Context = null,**o**.State = Detached
if ***o****.Context* ≠ ***oc*** **then**
  return *false*;
end
if ***o****.State* = *Fault* **then**
  oc.loadObjectInternal(**o**,**EM(o)**);
end
forall *RelationModel* ***r*** ∈ ***EM(o)****.Relationships* **do**
  if *r.type* = *ToMany* **then**
    o.clearObjects(**r**.name);
  else
    o.setValue(**r**.name,null);
  end
end
oc.Cached = **oc**.Cached \ {**o**} ;
oc.ToSave = **oc**.ToSave \ {**o**};
oc.ToDelete = **oc**.ToDelete ∪ {**o**.Id } ;
o.Context = null;
o.State = Detached;
return *true*;

Algorithm 3: ObjectContext.detachobject

Algorithm 3 defines a function that detaches an object from *ObjectContext*. In order to preserve consistency of object graph, detached object's relationships are cleared. Its persistent record will be deleted on next ObjectContext.save(). This function also loads object if it was in the *Fault* state, since detached object with no values would not provide any useful functionality.

Input : ObjectContext **oc**, Object **o**
Before : **o**.Context = **oc**,**o**.State = Fault
After : **o**.State = Loaded
if ***o****.Context* ≠ ***oc*** **then**
  return *false*;
end
if ***o****.State* = *Loaded* **then**
  return *true*;
end
return *oc.LoadObjectInternal(****o,EM(o,oc)****)*;

Algorithm 4: ObjectContext.loadObject

```
Before : o.Context = oc,o.State = Fault
After  : o.State = Loaded
Input  : ObjectContext: oc, Object o, EntityModel e
o.Values = oc.Store.loadAttributes(o.Id,e);
forall RelationModel r ∈ e.Relationships do
    if r.Type = ToMany then
        var ids = store.fetchToManyRelationIds(o.Id,r);
        o.Values[r.Name] = { o.getObject(id) : id ∈ ids };
    else
        o.Values[r.Name] = oc.getObject(oc.Store.fetchToOneRelationId(o.Id,r));
    end
end
o.State = Loaded;
return true;
```

Algorithm 5: ObjectContext.loadObjectInternal()

Algorithms 4 and 5 define functions for loading object attributes and relationships. This moves an object from *Fault* state into the *Loaded* state. This function is divided into private and public parts. The*loadObjectInternal* is private and it does all the actual work, loading attributes and relationships, and is used in several other public functions, including *loadObject*.The *loadObject* function does all necessary checks of arguments. It does not need to check whether the object's *EntityModel* is part of *ObjectContext.DBModel* to which the object is attached because in this condition is checked at the time of creation of the object.

```
Input  : ObjectContext oc, Object o
After  : o.State = Loaded, o.Context = oc
if o.State = Fault ∨ o.State = New then
    oc.saveObject(o);
end
o.Values = ∅;
o.State = Fault;
```

Algorithm 6: ObjectContext.unloadObject()

Algorithm 6 defines a function for unloading object attributes, the inverse of previous 2 functions. It saves modified objects, and then simply removes all values. This function is needed during *ObjectContext* destruction. Since Objects with relationships in our system form a cyclic graph by definition and are stored using reference counted pointers, it is necessary to unload all objects before destroying *ObjectContext*. This action clears all reference cycles and allows automatic deallocation of all objects.

```
Input  : ObjectContext oc, Object o
Before : o.State = New ∨ o.State = Modified
After  : o.State = Loaded
var EntityModel e = EM(o);
oc.Store.saveObjectAttributes(o,e);
forall RelationModel r ∈ e.Relationships do
    if r.type = ToMany then
        oc.storeToManyRelation(o,r,e);
    else
        oc.storeToOneRelation(o,r,e)
    end
end
o.State = Loaded;
oc.ToSave = oc.ToSave \ {o};
```

Algorithm 7: ObjectContext.saveObject()

Algorithm 7 defines function used for saving object attributes and relationships into the persistent store. By performing this action we ensure that all attributes and relationships are properly stored in the persistent store and library will be able to retrieve them on subsequent executions of software.

```
Input  : ObjectContext: oc, ObjectID oid
After  : ∃ Object o ∈ oc.Cached ∧ o.Id = oid
if ∃ Object o; o ∈ oc.Cached ∧ o.Id = oid then
    return o
else
    if oc.Store.idExists(oid) then
        var Object p: p ∈ oc.Prototypes ∧ p.Id.EntityId = oid.EntityId;
        var Object o = p.clone();
        o.Id = oid;
        oc.Cached = oc.Cached ∪ {o};
        return o
    end
end
return null
```

Algorithm 8: ObjectContext.getObject()

Algorithm 8 defines a function for getting an object from *ObjectContext* based on its *Id*. This functions first checks *ObjectContext's* cache for already loaded objects. If it finds an object with matching *Id*, it simply returns it. If searched object is not cached, the

ObjectContext verifies that the object exists in the persistent store, and returns new object created by cloning the prototype from same entity/class.

```
Input  : ObjectContext: oc
After  : oc.ToSave = ∅ ∧ oc.ToDelete = ∅
forall ObjectId oid ∈ oc.ToDelete do
    EntityModel e: e ∈ oc.Model ∧ e.Name = oid.EntityId;
    forall RelationModel r ∈ e.Relationships do
        if r.Type = ToMany then
            oc.Store.deleteToManyRelation(oid,r,e);
        else
            oc.Store.deleteToOneRelation(oid,r,e);
        end
    end
    oc.Store.deleteObjectAttributes(oid,e);
end
oc.ToDelete = ∅;
forall Object o ∈ oc.ToSave do
    oc.saveObject(o);
end
oc.ToSave = ∅;
```

Algorithm 9: ObjectContext.save()

Algorithm 9 defines a function that saves all changes performed on *ObjectContext* into its persistent store. This function first deletes all objects that were detached from the *ObjectContext*, and then saves all objects that were modified. Then it clears sets that hold this information,

While most of the functions in this chapter return their result as boolean value or object pointer, depending on the type of function, This is not reflected in actual implementation. In functions for retrieving EntityModel, C++ Exceptions are used. These types of errors cannot be resolved within the system and usually mean that programmer made a mistake, and would be simply propagated up the call stack, and exceptions denote this behavior perfectly. Raw SQL calls in SQLitePersistentStore also use exceptions. Since system cannot solve database errors internally, returning them as exceptions to application code seemed like the best solution.

This chapter only shows core algorithms for managing objects, that are implemented on ObjectContext. Additional algorithms that define management of values on individual objects can be found in Appendix A.

Chapter 4

Implementation

This chapter describes how components and concepts specified in the previous chapter were implemented in C++.The project uses C++ language with C++11 standard, the C++14 standard was not chosen because it would not provide meaningful improvements to development process or the implemented library. The earlier standard (C++03) was not chosen because smart pointers were first introduced with C++11, and they are a crucial part of the current design of the system. Without smart pointers, the implementation would be much more complex.

Since Core Data operates on objective-C runtime, it does not have to deal with many low-level issues. CoreStore however only relies on C++ runtime which is much smaller and therefore must explicitly implement features that are implicit in objective-C runtime. One example is the prevention of memory leaks. Core Data can make use of automatic reference counting, but CoreStore must rely on manual memory management, usage of RAII[1] and smart pointers.

While CoreStore does not implement all functionality found in Core Data, it implements Core functionality for persisting objects, and managing object graph.

4.1 Core implementation

Each component if the system is implemented as C++ class containing necessary attributes and implementing necessary methods. Inheritance is used to provide extendability of some components. Since the implementation of core components is pretty straightforward, this section does not contain detailed explanations of component implementations. Instead, this section contains detailed explanations of more complex/interesting parts of the implementation.

[1] https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

Object storage

Probably the most important implementation detail is storage of persistable objects. Every persistable object in the system is stored using shared pointer. Shared pointer uses reference counting, which ensures memory safety, and shared access to the storable object. When a persistable object is attached to ObjectContext, this ObjectContext takes ownership of this object. The ObjectContext responsible for storing this object, managing its state, and ultimately deallocation. Providing these function while preserving memory safety would be nearly impossible without usage of smart pointers.

However, using shared pointer for this task introduced another problem. Object graph, which library managing, could be thought of as simple graph, where objects are nodes, and references between them are edges. Since designed system requires inverse references in case of relationships, this means that this graph will contain cycles. Reference cycles pose a serious problem for reference counting implementations. If we are using standard shared pointers, without any weak pointers to break the cycle, objects managed will never be deallocated.

This is solved by unloading all objects attached to context before context deallocation. Unloading object removes all its attributes and relationships, breaking all reference cycles. Then, upon the destruction of ObjectContext, the last reference to every object is removed, and it is deallocated safely. If user retains a reference to object even after the destruction of its context, this object is not deallocated, and accessing it will not cause a segmentation fault. However, in the context of this system, this action is illegal, and upon the destruction of ObjectContext, all attached objects transition to *Error* state. If the user accesses object in *Error* state, an exception will be thrown.

Generic Value type

Since Core Data is implemented in a language that provides high-level object-oriented features, such as a common root of inheritance (NSObject). It can be used to implement highly generic algorithms and store any attribute values. In C++, no such root of object hierarchy exists. One solution would be to allocate every value on the heap, and store only a `void` pointer to this value, but this approach would be very unsafe, and prone to mistakes. Another approach is to create a union of all storable types, with information about which type is stored in this union alongside. However, storing classes inside union is not allowed.

Ultimately, chosen solution was to create *Value* class to provide storage for all needed value types. The implementation consists of several classes. *Value* provides interface for querying stored value type and performing proper casting. *Value* contains a pointer to *ValueHolder*, that is allocated on the heap at the time of creation. *ValueHolder* is a pure virtual class, that declares utility methods and overloaded equality and comparison operators. *ValueHolderImpl* is template class that is derived from *ValueHolder*, and its template argument denotes what type is stored inside. There are specialized implementations of *ValueHolderImpl* for strings and numeric types.

Value contains multiple constructors, one for each storable type, which allocate *ValueHolderImpl* with desired type parameter.

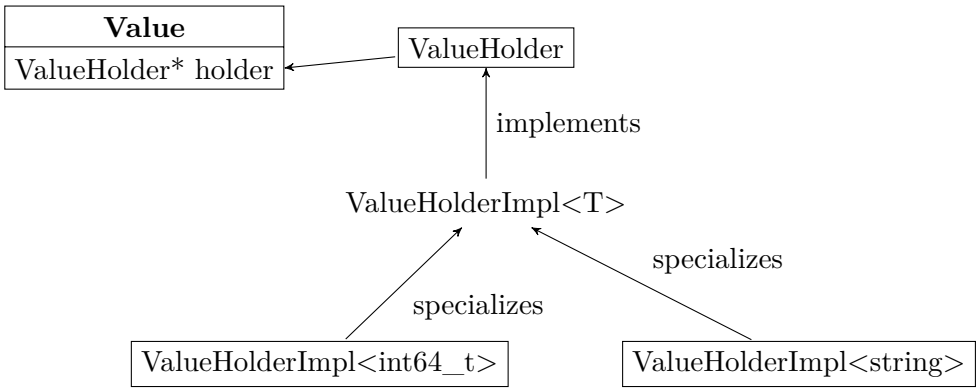

Figure 4.1: Value type hierarchy

Figure 4.1 shows how types used in *Value* implementation interact.

Value storage and access

All attributes and relationships of an object are stored in one *std::map*, using generic value type that is specified in section 4.1. The user defines attributes and relationships, by adding them to EntityModel in pure virtual method *createModel*, that must be implemented on all persistable classes. With this approach, read and write access is provided through *getValue* and *setValue* methods on objects. However, these methods require the name of an attribute in the form of a string, and values must be provided in form of a *Value* type. This is not the most ergonomic way to read and write object attributes/relationships. To provide ergonomic access to object attributes and relationships, accessors must be created. Automatic generation of these methods by macros is explained in subsection 4.3.1.

Predicates

The system provides a method for querying persistent store for specific objects based on predicates. Predicates are implemented using simple class hierarchy. There are specific implementations for simple attribute predicates, expressing equality or specific ordering, and compound predicates for chaining multiple other predicates using logical operators. All predicates are derived from *Predicate* class and override methods on this class.

Primitive and compound predicates can be created by invoking static methods on *Predicate* class. In addition to this method, compound predicates can also be created by utilizing overloaded logical operators:

```
// Simple predicate
Predicate::eq("Name","Peter");
// Compound predicate using static method
Predicate::And(Predicate::gt("age",10),Predicate::lt("age",20);
// or using overloaded "and" operator
Predicate::gt("age",10) && Predicate::lt("age",20);
```

Listing 4: Predicate creation

This system requires compound predicates to hold references to their sub-predicates. To ensure memory safety and prevent any memory leaks, all predicates created by static methods or operator overloading are created as `std::unique_ptr<Predicate>`. By this design, compound predicates own their sub-predicates, while allowing for polymorphic dispatch. Predicates can be used in two ways.They can be used to generate SQLite query, that is then executed on a database, or they can be directly evaluated against objects. This evaluation is performed inside *evaluate* method defined o *Predicate* class.

This creates a predicate tree, which is then used by *SQLitePersistentStore* to generate SQL query, or is evaluated on individual objects to determine whether objects satisfy the predicate.

There are several limitations to this system. Currently, the system only supports attribute predicates and compound predicates. Relationship predicates were not implemented. Queries based on predicates can only look up objects belonging to one Entity at time. These limitations were accepted because in the current state generated SQL only needs to read values from one table, which greatly simplifies SQL generation code, and implementing extended features would provide limited benefit.

4.1.1 Project structure

The project uses CMake build system. This system was chosen because it is currently de-facto standard build system for C++ projects, and is heavily used in many open source projects. CMake configuration files are also easier to write and maintain compared to `make` configuration files, while providing cross-platform compatibility[4].

Directory	Subdirectory	Contains
include+src	core	Implementation of core library classes
	model	Implementation of data model definition classes
	query	Implementation of Predicate classes
	sqlite	Implementation of SQLitePersistentStore
include	value	Implementation of generic value type
deps		Third party dependencies
example		Example applications
test		Implementation of test cases

Table 4.1: Directory structure

Table 4.1 shows directory structure of the implementation.The implementation itself is divided into specific components, close components sharing common directory in include and src.The include directory contains all header files. It also contains *CoreStore.h* file, which itself includes all header files necessary for the use of this library. User only needs to include *CoreStore.h* file.

If user wants to build CoreStore library, he only needs to create makefiles by executing `cmake <CoreStore library directory>`. This action will create makefiles in current directory, which can be used to build the library. CMake is configured to generate specific targets for library, example application, test application and `install` target, which will install header files and build library into appropriate system directories. These specific targets can be built/run using `make <target>`. More information can be found in README file included with the library.

4.1.2 External dependencies

Implementation depends on several other projects. Since C++ ecosystem does not have any widely used dependency management system, these dependencies are included with the library, in the *deps* directory.

SQLite3 wrapper

Since the reference implementation of *PersistentStore* depends on SQLite, it was necessary to interface with this technology. While SQLite provides C API, interfacing with it directly from *SQLitePersistentStore* would add additional complexity. In order to seamlessly interface with SQLite, a small C++ wrapper was included. It provides abstractions built on SQLite C API, and presents set of classes that can be used to safely execute queries, while properly managing memory[3].

SQLite itself is not included in any way. Since most platforms provide their own distribution, with header and library in common directories, its inclusion will probably be automatic.

GoogleTest framework

Another included dependency is a framework for testing from Google[6]. It was used to implement a suite of test cases for at least partially verifying correctness of the library implementation. The testing process is specified in chapter 5.

4.2 Database interface

4.2.1 Persistent Store and extendability

Library implementation is conceptually separated into several parts. *PersistentStore* belongs to the core part of the library. This is pure virtual class used as an interface into persistent store, which is probably some kind of database. By creating a derived class from *PersistentStore*, user can create interfaces into other kinds of persistent stores. The reference implementation of persistent store is *SQLitePersistentStore*, which uses SQLite database as its persistent storage medium.

Every class deriving from the *PersistentStore* has to implement aset of methods. There are management methods used to initialize database(*applyModel*), work with database transactions if supported(*beginTransaction,commitTransaction*), that are used for performance reasons. And there are methods for reading and writing data into persistent storage. These methods are divided into methods working with attribute values(*fetch/storeObjectattributes*), and methods working with relationships (*fetch/storeToOneRelationship*, *fetch/storeToManyRelationships*).

User-implemented persistent stores can be plugged-in directly, without requiring modifications to other parts of the library, by simply providing them during *ObjectContext* creation.

4.2.2 SQlitePersistentStore

As part of the assignment was to write an implementation of PersistentStore, that will interface to SQLite[8] database. This implementation is *SqlitePersistentStore*. The implementation process of this class heavily influenced design of PersistentStore interface. It stores each entity, and each relationship pair as a separate table. This persistent store uses monotonically increasing integer as the instance Id for *ObjectID*s it generates.

It uses C++ standard streams to create SQL statement strings. It does not however serialize any data into these strings. Only the table and column names are serialized. Any occurrence of raw data is replaced with a **?** wildcard. And after preparing the generated string into a statement, actual data is bound to this statement using sqlite binding functions.

4.3 Examples and usage

This section contains a number of small examples, whichh illustrate, how implemented library can be used, and how common problems are solved. This section also contains detailed information about example application, that was implemented as part of this thesis.

4.3.1 Class and EntityModel definition

To define storable class, user must create class that inherits from *Object* class, define several utility methods, define createModel method that returns *EntityModel* of the class and create accesors to retrieve and modify object attributes/relationships. All these task can be completed by utilizing several macros defined in *common.h* header file.

```
class Person: public cs::Object{
    DERIVE_CLASS(Person);

    virtual EntityModel createModel() const override {
        auto e = EntityModel("Person");
        ATTRIBUTE(e, firstName, AttributeType::String);
        ATTRIBUTE(e, lastName, AttributeType::String);
        ATTRIBUTE(e, salary, AttributeType::Double)

        RELATION_ONE(e, boss, Person, subordinates);
        RELATION_MANY(e, subordinates, Person, boss);
        return e;
    }
    ACCESSORS(string,firstName);
    ACCESSORS(string,lastName);
    ACCESSORS(double,salary);
    ACCESORS_TO_ONE(Person,boss);
    ACCESSORS_TO_MANY(Person,subordinates);
}
```

Listing 5: Example persistable class

Code listing 5 shows example class that uses several macros. *DERIVE_CLASS* is used to create several basic methods including *clone* method, which is used in class prototypes to create new objects from given class. The *ATTRIBUTE* and *RELATION* macros are used to add attributes/relationships to *EntityModel* of a class. *ACCESSORS* macros are used to define accessors to these attributes and relationships.

Generated accessors wrap low-level methods like *getValue*, *setValue*, and also perform checking and casting of values. Since relationships are stored as `std::shared_ptr<Object>` type, they require casting into shared pointers of specific derived classes.

Neither library design nor implementation currently allow modification of data model. Core Data provides this functionality by multiple Data model versions and data migrations between them. There is currently no such feature in CoreStore, and this area is one of the primary targets for future improvements.

4.3.2 Common usage patterns

This section shows most usage of library classes to perform most common actions like: initialization of library, creation, retrieval of objects and other.

```
ContextConfiguration config = ContextConfiguration()
    .withStore(new SQLitePersistentStore("./people.db",false))
    .withClass(new Person);

auto context = new ObjectContext(config);
```

Listing 6: ObjectContext creation

Code listing 6 shows the creation of *ObjectContext*. *ObjectContext* creation requires *ContextConfiguration*, which contains all necessary information about the created context. It contains reference to *PersistentStore* that will be used, and set of classes that constitute data model. All methods on *ContextConfiguration* return reference to itself, allowing easy chaining of multiple method calls. This was inspired by Fluent interface[2] and Builder[3] patterns.

```
// Creates shared_ptr<Person>
context->createNewObject<Person>();
// Creates shared_ptr<Object>
context->createNewObject("Person");
// Creating Person* and attaching
auto person = shared_ptr<Person>(new Person());
person = context->attachObject<Person>(person);
```

Listing 7: Creating new Object

Code listing 7 shows multiple ways to create new *Object* attached to *ObjectContext*. *createNewObject* is included in 2 variants. One accepts string argument, denoting name of entity, to which the created object should belong. another accepts type argument. Several *ObjectContext* methods are implemented in this way. Methods accepting type parameter always expect and return shared pointers to the derived type, while methods that expect string argument, expect and return a pointer to *Object* type.

[2] https://en.wikipedia.org/wiki/Fluent_interface
[3] https://en.wikipedia.org/wiki/Builder_pattern

```
// returns std::vector<shared_ptr<Person>>
context->getAll<Person>(Predicate::gt("salary",30000.0));
// returns std::vector<shared_ptr<Object>>
context->getAll("Person",Predicate::gt("salary",30000.0));
```

Listing 8: Object retrieval with predicate

Code listing 8 shows how user can query for objects from *PersistentStore* based on predicates. This method is also implemented in 2 variants, providing static, and runtime way to define what entity is queried.

```
// Saves all modified and deleted objects
context->save();
// Does the same + unloads all objects.
context->unload();
```

Listing 9: Object retrieval with predicate

Code listing 9 shows saving of changes performed on *ObjectContext*. All changes are kept in-memory, until user decides to save modified values, or *ObjectContext* is destroyed. At the point of destruction of *ObjectContext*, all changes are automatically saved.

4.3.3 Example application

Part of thesis assignment was to write a small application that would show capabilities of implemented library. Implemented application is extremely simple bank database system, containing a small number of entities, and providing command line interface. Application implementation can be found in *example* directory and can be built and run using `make cs_example`.

Data model

The example application uses very simple data model. This model contains just 3 entities: *Person*, *Account* and *Transaction*. These 3 entities are sufficient to model simple accounting system. All entities contain a minimal number of attributes. Following diagram shows what attributes and relationships this data model contains.

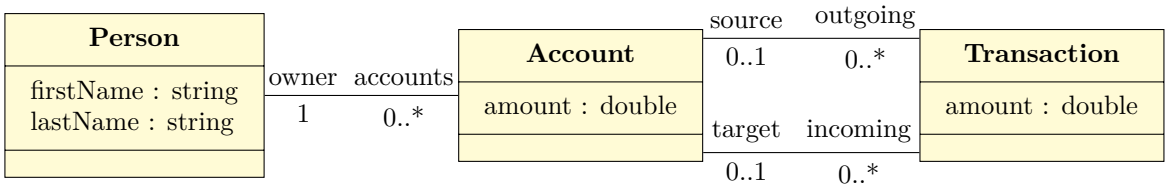

Figure 4.2: Example application data model

Figure 4.2 shows example application data model. In this data model entities *Person* and *Account* fill their standard roles, while entity *Transaction* carries more information. The *Transaction* entity is used to denote usual transaction from account to account, but it is also used to denote withdrawals and deposits to accounts. If particular *Transaction* does not have *source*, it is assumed to be deposit, and if it does not have *target* it is assumed to be a withdrawal.

Application usage

The application provides simple a command line interface using Request-Response style of communication. These are commands and their respective subcommands that can be used to communicate with the application.

Command	Subcommand	Description
help		Shows list of available commands
save		Saves changes to database
quit		Saves changes and exits the application
show	people	Shows list of all people in database
	accounts	Shows list of all accounts in database
	transactions	Shows list of all transactions in database
	<ID>	Shows information about particular object
create	person	Creates new person object
	account	Creates new Account object
	transaction	Creates new transaction object
seed		Seeds database with example data

Table 4.2: Example application commands

The application communicates using standard input and output. Application excepts commands, subcommands and their respective arguments written into the standard input, separated by spaces, and confirmed by line ending character ($\backslash n$). Application then responds in similar fashion, writing its output to the standard output stream, and ending with one empty line.

While this application may not provide a lot of functionality or ergonomic interface, It shows basic *CoreStore* concepts used in a realistic setting.

Chapter 5

Testing and verification

This chapter outlines evaluation process of implemented library. Library was evaluated using 2 main approaches. Testing was performed to ensure implementation correctness, and performance measurements were performed to ensure library performance.

5.1 Testing

Part of provided source code is a small test case library consisting of 20 test cases. The project uses GoogleTest[6] framework for managing tests. Most of the tests were implemented using in-memory SQLite database as a persistent storage medium. Tests focus on proving correctness according to system design (specified in chapter 3), while also trying to uncover all errors that could leave the system in undefined state or even crash the hosting application.

Following table shows what part of the system individual test cases targeted, upon which fixture were they based. The concept of a fixture is presented by GoogleTest framework, and is used to create basic structure for individual test cases.

Test number	Fixture	Target
1	SingleContext	Creation of Data model
2-4		Primitive operations on objects
5-9		Object linking & lifetime
10		ObjectContext lifetime
11		Persistent Storage
12-16		Predicates and queries using them
17-19	MultiContext	Multiple context management
20	InvalidContext	Creation of context with invalid data model

Table 5.1: Focus area of individual test cases

All tests followed a similar structure. The test environment is prepared by GoogleTest with a simple macro invocation. In actual test case body, a sequence of actions is executed, and correct resulting state of the system is verified. Since implemented library consists of several highly dependent components, attempting to unit test each individual component proved very difficult. Because of this difficulty, another testing approach was chosen. Library was tested as a whole, with each test case focusing on verifying a specific property of implementation. Each test performs small actions on *ObjectContext* and then verifies whether the context remains in valid state, and all actions were completed successfully.

Following code listing shows test case number 6 as an example.

```
TEST_F(SingleContextFixture,ObjectDetachedManipulation){
    auto o = context->createNewObject<TestObject>();
    ASSERT_TRUE(context->detachObject<TestObject>(o));

    // Attribute manipulation should work
    ASSERT_TRUE(o->set_strVal("Name"));
    ASSERT_EQ(o->get_strVal(),"Name");

    // Relationship manipulation should work
    ASSERT_FALSE(o->set_parent(o));
    ASSERT_EQ(o->get_parent(), nullptr);
}
```

Listing 10: Sample test: detached object modification

In this test, a new object is created, and immediately detached from its *ObjectContext*. Then, the correctness of attributes and relationships is verified.

After creating a test suite, code coverage of this test suite was also measured using *lcov* tool. Measured coverage is above 90% of all lines of source code in the implementation. Using the lcov tool during implementation of test case library discovered very poor coverage of code dealing with predicates. This issue was fixed by writing more tests to specifically target valid behavior of predicates, and queries using them.

Testing process began during implementation and continued until completion of implementation. During this process author's multiple mistakes were uncovered, and subsequently corrected.

5.2 System performance

After completion of implementation and conclusion of the testing process, library performance was measured. Since library will serve as a object-oriented database, most important performance characteristic is the speed of performing individual actions such as

loading or storing objects. This aspect of implementation was measured using a simple application, which is implemented in *examples/performance.cpp*. The application repeatedly executed a set of very simple tasks on increasing number of objects, measured time spent performing these actions, and printed results in human readable format to standard output.

Performance measurements were performed on Thinkpad T460 laptop with, 256 GB Sata3 SSD, 16GB 1600MHz RAM and Intel i5 6200U CPU running Linux 4.10. Measurement on this platform should be reflective of real word performance of implemented library. Following chart shows execution time for most important actions:

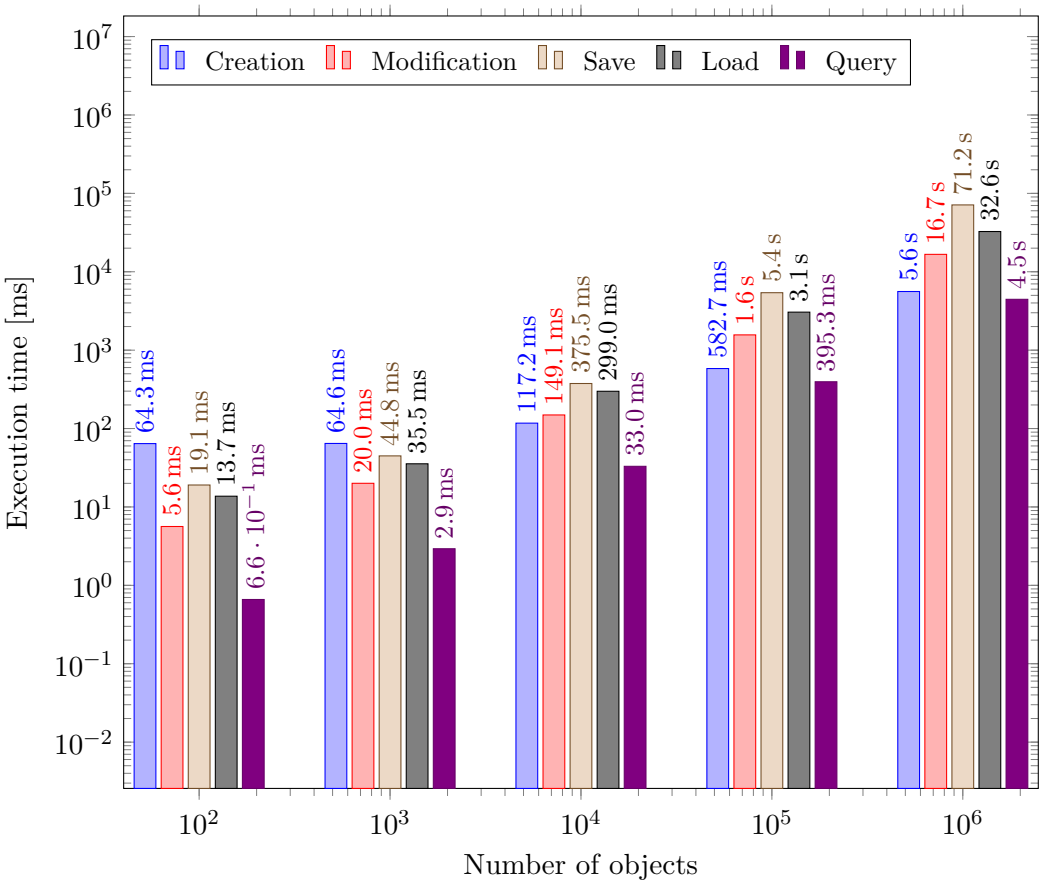

Figure 5.1: Core actions performance

Figure 5.1 presents measured execution time of core actions on a logarithmic chart. Measured values show approximately linear growth of execution time for larger number of objects. However, for a small number of objects, actions like *Save* or *Load* do not show linear growth. For example, saving 100 objects took 19.1 ms, and saving 1000 objects took 44.8 ms, while expected time for 1000 objects is 191 ms. This could be caused by overhead of creating and compiling SQLite query and overhead of SQLite transaction. These actions are executed only once per action, and could result in constant increase of execution time. This increase would be very significant part of measured times for small number of objects, but would be insignificant for large number of objects.

Library throughput for individual actions was calculated from measured times using this simple formula

$$throughput_{action} = \frac{\sum i}{\sum_i time_{action}(i)}; i = 100, 1000, ..., 10^6$$

Calculated values for each measured action along with individual action descriptions are shown below.

Action	Throughput	Description
Create	152.2	Create new objects
Modify	52.4	Modify one attribute and relationship on all objects
Save	11.2	Save objects into persistent store
Load	24.1	Load objects from persistent store
Query DB	396.7	Query SQLite database for objects with predicate
Query Live	163.0	Query SQLite database and reevaluate predicate on all objects, Corresponds to **Query** in earlier chart
Detach	53.2	Detach objects from their ObjectContext
Save detached	73.4	Save objectContext with detached objects, deleting them from SQLite database

Table 5.2: Action throughput and description

Table 5.2 shows throughput of individual actions in thousands of objects per second.

This table and Figure 5.1 were created from extended set of measured values. These values can be found in Appendix B.

Chapter 6

Conclusion

Object oriented databases are a powerful tool for persisting data. Apple's Core Data is exceptional object oriented database, that provides a lot of useful functionality. The goal of this thesis was to bring its functionality into the C++ ecosystem in form of a library. For achieving this goal, Core data was evaluated in detail, analyzed in terms of its components and interactions between them. Based on information obtained during analysis, CoreStore library was designed and implemented.

After implementation was complete, it was evaluated using multiple techniques. Testing was performed to ensure implementation conforms to library design and does not contain critical defects. Few example applications were created to showcase library's capabilities, and whole codebase was packaged using CMake tool for easy distribution and compilation.

The performance of implemented library was measured. Implemented library achieved competitive performance, being able to process several thousands of objects per second.

While implementation achieves remarkable performance, and some degree of correctness thanks to profiling and testing, there are still areas, in which it falls short. One area is data model creation and handling. A graphical editor used to create data model would be a useful addition. Usage of graphical editor could also help with issue of data model upgrades. Current implementation does not allow for modification of data model after database file was created. Apart from this issue, the library probably contains many more small usability issues, that will require extended user testing to uncover and correct.

Bibliography

[1] Apple Inc.: NSManagedObjectContext reference.
Retrieved from:
https://developer.apple.com/reference/coredata/nsmanagedobjectcontext

[2] Apple Inc.: Core Data Programming Guide. Sep 2016.
Retrieved from: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreData/

[3] Beer, D.: Lightweight SQLite3 wrapper for C++.
Retrieved from: http://dlbeer.co.nz/oss/sqlite.html

[4] Cmake developers: Cmake Build system.
Retrieved from: https://cmake.org/

[5] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*. vol. 13, no. 6. June 1970. ISSN 0001-0782. doi:10.1145/362384.362685.
Retrieved from: http://doi.acm.org/10.1145/362384.362685

[6] Google Inc.: GoogleTest testing framework.
Retrieved from: https://github.com/google/googletest

[7] Hughes, J. G.: *Object-Oriented Databases (Prentice-Hall International Series in Computer Science)*. Prentice Hall. 1993. ISBN 0136298745.

[8] SQLite consortium: Sqlite embeddable database engine.
Retrieved from: https://www.sqlite.org/

Appendices

Appendix A

Object modification algorithms

Due to length of primary part of this thesis. Algorithms defining managment of object values are specified in this appendix.

These algorithms describe functions implemented on *Object* class, and they mostly define how said object manipulates it's internal state for safe update and retrieval of it's attributes and relationships. These functions also implement logic needed to keep inverse relationships synchronized, and load object from *fault* state if needed.

Input : Object **o**
if ***o***.*State* = *Error* **then**
  throw *exception(„Object outlived it's Context“)*;
end
if ***o***.*State* = *Fault* **then**
  o.Context.loadObject(**o**);
end

Algorithm 10: Object.ensureLoaded()

Algorithm 10 is called at the start of almost every function that manipulates object values. It ensures that all objects are properly loaded into memory. It's second purpose is to notify programmers about object outliving its context. Since this is very serious error, this methods throws an exception, instead of silently failing to perform any actions on this object.

```
Input  : Object o,name n
o.ensureLoaded();
if ∃ o.Values[n] then
  return o.Values[n];
else
  if o.State == New ∧ n ∈ EM(o).Attributes ∨ n ∈ EM(o).Relationships
   then
    o.Values[n] = getDefaultValue(EM(o),n);
    return o.Values[n];
  else
    return o.getValueForUndefined(n)
  end
end
```

Algorithm 11: Object.getValue()

Algorithm 11 defines function used for retrieving primitive value or references to related objects. If value is not found in Object's internal value map, algorithm first tries to find AttributeModel or RelationModel with provided name. If it exists, it uses helper method to generate default value for provided name, and returns it. If it does not exist, virtual method *getValueForUndefined* is called, that in it's default implementation throws an exception, but can be used to extend Object's functionality.

```
Input  : Object: o,Name n,Value v
After  : o.State = Modified ∨ o.State = New
o.ensureLoaded();
if  ∃ AttributeModel a : a = EM(o).Attribues[n] ∧ type(v) = a.Type then
  o.Values[n] = v;
  o.markModified();
  return true;
else
  if  ∃ RelationModel r : r = EM(o).Relationships[n] then
    var EntityModel iem: iem ∈ o.Context.Model[r.TargetEntity];
    var RelationModel im: im ∈ iem.Relationships[r.TargetName];
    if o.Context = null then
      return false;
    end
    o.markModified();
    if r.Type = ToOne then
      <setRelationshipToOne(r,v)>
    else
      <setRelationshipToMany(r,v)>
    end
  else
    return o.setValueForUndefined(n,v)
  end
end
```

Algorithm 12: Object.setValue()

```
Input  : Object o, RelationshipModel r,Value v
var Object old = o.Values[n];
var Object replace = v as Object;
if old = breplace then
    return true;
end
if (replace ≠ null ∧ replace.Context ≠ o.Context) then
    return false;
end
o.Values[n] = replace;
if im.Type = ToOne then
    if old ≠ null then
        old.setValueInternal(im.Name,null);
    end
    if replace ≠ null then
        var replaceInverse = replace.Values[im.Name];
        if replaceInverse ≠ null then
            replaceInverse.setValueInternal(r.Name,null);
        end
        replace.setValueInternal(im.Name,o);
    end
else
    if old ≠ null then
        old.removeObjectInternal(im.Name,o);
    end
    if replace ≠ null then
        replace.insertObjectInternal(im.Name,o);
    end
end
return true;
```

Algorithm 13: OsetRelationshipToOne

```
Input  : Object o, RelationshipModel r,Value v
var Objects old = o.Values[n] as set of Objects;
var Objects replace = v as set Objects;
if old = replace then
    return true;
end
o.Values[n] = replace;
if im.Type = ToOne then
    forall oldObj ∈ old do
        oldObj.setValueInternal(im.Name,null);
    end
    forall replaceObj ∈ replace do
        var replaceInverse = replaceObj.Values[im.Name];
        if replaceInverse ≠ null then
            replaceInverse.setValueInternal(r.Name,null);
        end
        replaceObj.setValueInternal(im.Name,o);
    end
else
    forall oldObj ∈ old do
        oldObj.removeObjectInternal(im.Name,o);
    end
    forall replaceObj ∈ replace do
        replaceObj.insertObjectInternal(im.Name,o);
    end
end
return true;
```

Algorithm 14: setRelationshipToMany

Algorithms 13 and 14, are not specific functions implemented on object. They are simply parts of algorithm 12, which were extracted due to long length of this function.

```
Input  : Object o, Name n, Value v
o.ensureLoaded();
o.Values[n] = v;
o.markModified();
```

Algorithm 15: Object.setValueInternal()

Algorithms 12 and 15 define functions for modifying object values. Function *setValueInternal* simply loads object if needed, modifies it's internal value map, and marks object as modified. Function *setValue* is more complex. If provided name refers to an attribute, it behaves as *setValueInternal*. However, if the provided value refers to relationship behavior is more complex. This function loads old value stored under said name. If inverse relationship type is ToMany, it removes object on which is this function called from inverse relationship on old objects, and inserts it into inverse relationship on new objects. If inverse relationship type is ToOne, it sets this inverse relationship on old object to null, and on new object, sets inverse value to *this*. This function calls other

internal functions such as *setValueInternal,insertObjectInternal*, by using internal functions that do not perform linking of inverse relationships, this avoids creating infinite cycles. One critical condition is that both target object and object inserted into relationship are from same context.

Input : Object **o**, Name **n**, Object **v**
o.ensureLoaded();
var *Objects* ***objs*** = ***o.getValue(n)***;
objs = **objs** ∪ {**v**};
o.markModified();

Algorithm 16: Object.InsertObjectInternal

Input : Object: **o**,Name **n**,Object **i**
Before : **o**.Context ≠ null
After : **o**.State = Modified ∧ **i**.State = Modified
var EntityModel **e** = **EM(o)**;
o.ensureLoaded();
if ***i*** = *null* ∨ ***o***.*Context* = *null* ∨ ***i***.*Context* ≠ ***o***.*Context* **then**
  return *false*;
end
if ∃ *RelationModel* ***r*** : ***r*** = ***EM(o)***.*Relationships[**n**]* ∧ ***r***.*Type* = *ToMany* **then**
  var EntityModel **iem**: **iem** ∈ **o**.Context.Model[**r**.TargetEntity];
  var RelationModel **im**: **im** ∈ **iem**.Relationships[**r**.TargetName];
  if ***im***.*Type* = *ToOne* **then**
    var *old* = **i**.getValueInternal(**im**.Name);
    if ***old*** ≠ *null* **then**
      old.removeObjectInternal(**name**,**i**);
    end
    i.setValueInternal(**im**.Name,**o**);
  else
    i.insertObjectInternal(**im**.Name,**o**);
  end
  o.insertObjectInternal(**n**,**i**) **return** *true*;
end
return *false*;

Algorithm 17: Object.insertObject

Algorithms 16 and 17 define functions for inserting new objects into ToMany relationships. *insertObjectInternal* as similar functions defined before, do not perform any verification of inputs, and simply perform core action of inserting new object into internal collection holding objects in relationship. *insertObject* on the other hand does perform

verification, and performs linking of inverse relationships, keeping integrity of the object graph. The condition of storing only objects from one *ObjectContext* is also verified.

Input : Object **o**, Name **n**, Object **v**
o.ensureLoaded();
var *Objects* ***objs*** = ***o***.*getValue(**n**)*;
objs = **objs** \ {**v**};
o.markModified();

Algorithm 18: Object.RemoveObjectInternal

Input : Object: **o**,Name **n**,Object **i**
Before : **o**.Context ≠ null
After : **o**.State = Modified ∧ **i**.State = Modified
o.ensureLoaded();
if ***i*** = *null* ∨ ***o***.*Context* = *null* ∨ ***i***.*Context* ≠ ***o***.*Context* **then**
  return *false*;
end
if ∃ *RelationModel* ***r*** : ***r*** = ***EM(o)***.*Relationships[**n**]* ∧ ***r***.*Type* = *ToMany* **then**
  var EntityModel **iem**: **iem** ∈ **o**.Context.Model[**r**.TargetEntity];
  var RelationModel **im**: **im** ∈ **iem**.Relationships[**r**.TargetName];
  if ***i*** ∈ ***o***.*Values[**n**]* **then**
    if ***im***.*Type* = *ToOne* **then**
      i.setValueInternal(**im**.Name,null);
    else
      i.removeObjectInternal(**im**.Name,**o**);
    end
    o.removeObjectInternal(**n**,**i**);
    return *true*;
  end
end
return *false*;

Algorithm 19: Object.removeObject

Algorithms 18 and 19 are relatively straightforward inverses of previous algorithms. They perform removing of objects from ToMany relationships, while preserving integrity of object graph by modifying inverse relationships.

Input : Object: **o**
Before : **o**.State $\neq$ Fault
After : **o**.State = Modified $\vee$ **o**.State = New $\wedge$ **o** $\in$ **o**.Context.ToSave
if ***o***.*Conext* = *null* **then**
  return *false*;
end
o.Context.toSave = **o**.Context.ToSave $\cup$ { **o** };
if ***o***.*State* = *Loaded* **then**
  o.State = Modified;
end
return *true*;

Algorithm 20: Object.markModified()

Function defined in Algorithm 20 is invoked during during all functions that modify object values. This function inserts modified objects into *ToSave* set set in their respective *ObjectContext*, ensuring all changes will be properly saved.

Appendix B

Measured performance data

Following table shows raw measured data. This dataset was obtained by running example application specified in *examples/performance.cpp*. First row contain number of objects that were used to measure durations of individual actions, and subsequent rows contain execution times in milliseconds. Exact meaning of individual actions is best explained by source code of the application used to perform the measurement.

COUNT	100	1000	10000	100000	1000000
CREATE	$6.43 \cdot 10^{1}$	$6.46 \cdot 10^{1}$	$1.17 \cdot 10^{2}$	$5.83 \cdot 10^{2}$	$5.60 \cdot 10^{3}$
MODIFY	$5.63 \cdot 10^{0}$	$2.00 \cdot 10^{1}$	$1.49 \cdot 10^{2}$	$1.57 \cdot 10^{3}$	$1.67 \cdot 10^{4}$
LOAD	$1.37 \cdot 10^{1}$	$3.55 \cdot 10^{1}$	$2.99 \cdot 10^{2}$	$3.06 \cdot 10^{3}$	$3.26 \cdot 10^{4}$
QUERY	$3.80 \cdot 10^{-1}$	$1.32 \cdot 10^{0}$	$1.32 \cdot 10^{1}$	$1.64 \cdot 10^{2}$	$1.95 \cdot 10^{3}$
QUERY-MOD	$6.60 \cdot 10^{-1}$	$2.93 \cdot 10^{0}$	$3.30 \cdot 10^{1}$	$3.95 \cdot 10^{2}$	$4.45 \cdot 10^{3}$
DETACH	$3.11 \cdot 10^{0}$	$1.25 \cdot 10^{1}$	$1.42 \cdot 10^{2}$	$1.65 \cdot 10^{3}$	$1.71 \cdot 10^{4}$
SAVE-DEL	$9.50 \cdot 10^{0}$	$1.88 \cdot 10^{1}$	$1.54 \cdot 10^{2}$	$1.42 \cdot 10^{3}$	$1.27 \cdot 10^{4}$