2017

Generating a Normalized Database Using Class Normalization

Daniel Sushil Sudhindaran
University of North Florida

GENERATING A NORMALIZED DATABASE USING CLASS NORMALIZATION

by

Daniel Sudhindaran

A thesis submitted to the
School of Computing
in partial fulfillment of the requirement for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

April, 2017

The thesis “Generating a Normalized Database Using Class Normalization” submitted by Daniel Sudhindaran in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

_____ _____

Dr. Robert Roggio
Thesis Advisor and Committee Chairperson

_____ _____

Dr. Sherif Elfayoumy

_____ _____

Dr. Sandeep Reddivari

Accepted for the School of Computing:

_____ _____

Dr. Sherif Elfayoumy
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

_____ _____

Dr. Mark Tumeo
Dean of the College

Accepted for the University:

_____ _____

Dr. John Kantner
Director of the Graduate School

ACKNOWLEDGMENTS

I want to thank my family for all the support and love throughout my master's program. I also thank my friends and co-workers for encouraging and motivating me to complete my thesis. A special thanks to Dr. Roggio for his vital support throughout the development of this thesis.

CONTENTS

TABLES

FIGURES

ABSTRACT

Relational databases are the most popular databases used by enterprise applications to store persistent data to this day. It gives a lot of flexibility and efficiency. A process called database normalization helps make sure that the database is free from redundancies and update anomalies. In a Database-First approach to software development, the database is designed first, and then an Object-Relational Mapping (ORM) tool is used to generate the programming classes (data layer) to interact with the database. Finally, the business logic code is written to interact with the data layer to persist the business data to the database. However, in modern application development, a process called Code-First approach evolved where the domain classes and the business logic that interacts with the domain classes are written first. Then an Object Relational Mapping (ORM) tool is used to generate the database from the domain classes. In this approach, since database design is not a concern, software programmers may ignore the process of database normalization altogether. To help software programmers in this process, this thesis takes the theory behind the five database normal forms (1NF - 5NF) and proposes Five Class Normal Forms (1CNF - 5CNF) that software programmers may use to normalize their domain classes. This thesis demonstrates that when the Five Class Normal Forms are applied manually to a class by a programmer, the resulting database that is generated from the Code-First approach is also normalized according to the rules of relational theory.

Chapter 1

INTRODUCTION

1.1 Importance of Relational Databases

West and Fowler state that "the data model is the base that supports systems and business" [Fowler92]. Almost every business application has the need to store its data somewhere. This might involve user preferences information, contact information, business process results, financial information, etc. Most of this data needs to be stored permanently so that it can be queried and retrieved for use later. Business applications usually use a database as their storage medium to store their data.

Among the different types of databases, relational databases are the most popular kind. Enterprise applications rely on relational databases because of the structure, performance and "*queryable*" nature that relational databases bring to the table. Also, relational databases provide data independence—the ability for application programs to grow independently even when there are changes to the data types and data structures in the database [Codd70]. Data independence is critical in a business application because any change or improvement to the application regarding data storage should not cause critical changes to the database. Since relational databases deal with relationships between related entities, a new feature added to a business application could very well be just

another table added to store the data related to the new feature without affecting the rest of the data ecosystem.

1.2 Importance of Good Database Design

Now that it has been established why business applications prefer relational database systems, it is important to discuss why a good database design is important particularly within a relational database. A good database design is crucial for a robust, scalable, and a high-performance application. Without optimized relationships in the database, the database will not perform as efficiently as possible.

Normalization refers to the process of structuring data to minimize duplication and inconsistencies. Kent says that the normalization rules are specifically designed to prevent and even eliminate update anomalies and data inconsistencies found in databases [Kent83]. The underlying goal is to make certain that the same data is not stored in duplicate locations in a database. This ensures that when data is inserted, updated or deleted, it does not become inconsistent.

To accomplish this goal, Edgar F. Codd, the inventor of the relational model, in 1970, introduced the concept of normalization, and what we now know as the First Normal Form (1NF). Codd later defined a Second Normal Form (2NF) and later, a Third Normal Form (3NF) in 1971. Codd and Raymond F. Boyce defined the Boyce-Codd Normal Form (BCNF) in 1974. Given these normal forms, a relational database table is often

described as “normalized” if it is in the Third Normal Form because most 3NF tables are free of insertion, update and deletion anomalies. Thus, this standard normalization to 3NF is paramount in building a good relational database [Codd70].

1.3 Database First Approach in Business Applications

One of the techniques used in building business applications is the use of Database Driven Design or “Database First” approach. In such a database driven design, the business requirements are translated into entities and their relationships among these business entities. Thus the database schema is clearly developed first to support the business model. After a database schema is established, a programming technique known as “Object Relational Mapping” (ORM) [1] is used to translate the database schema into a form such that programming objects can understand them and interact with them. To put ORM into perspective, it is important to note that there are two ways in which the programming language can interact with the database. The first way is for the programming language to issue database commands directly from the business logic layer of the application to the database. This is done by issuing commands to the database in a native language such as SQL to which the database can react. Programming in this way can quickly become tedious and lead to a maintainability nightmare. Imagine a contact manager application wanting to insert, update, delete and select a contact from an

[1] Object-relational mapping (ORM) technique involves extracting metadata information about a database schema and storing them in an XML or JSON format. This metadata information can later be used by ORM tools to generate programming entities (classes) and their relationships in essence creating a virtual object database with which t an object-oriented program can interact.

existing database. The SQL commands that the programming language might need to issue to the database might be:

Selection:

```
SELECT ContactID, FirstName, LastName, Email, PhoneNumber
FROM Contact
WHERE LastName = 'Marley' AND FirstName = 'Bob'
```

Insertion:

```
INSERT INTO Contact (FirstName, LastName, Email, PhoneNumber)
VALUES ('Bob', 'Marley', 'bob@marley.com', '904-234-5556')
```

Deletion:

```
DELETE FROM Contact
WHERE LastName = 'Marley' AND FirstName = 'Bob'
```

Update:

```
UPDATE Contact
SET FirstName = 'Bobby',
PhoneNumber = '904-222-2222'
WHERE LastName = 'Marley' AND FirstName = 'Bob'
```

Now if a couple of new attributes need to be added to the contact manager application in addition to the attributes ContactID, FirstName, LastName, Email, and PhoneNumber, such as MiddleName and BusinessPhoneNumber, all of the SQL commands above may potentially need to be changed. For a simple application such as a contact manager application, this might not seem as too difficult. However, for an enterprise application that can have hundreds of tables and fields, this will be a very tedious and error-prone process. That is why enterprise application programmers prefer a second way of communicating with the database using a programming technique known as Object Relational Mapping (ORM).

1.3.1 Object Relational Mapping (ORM)

ORM is a technique in which the metadata such as table names, column names, relationships (sometimes called foreign keys), indexes and more are extracted out of a database and stored in an XML or JSON file [Torres17]. This created metadata is then fed to a tool which understands both the type of database used as well as the type of programming language used. This tool (usually built in-house or third party) generates object-oriented programming classes[2] for the database tables, columns, relationships and more so that data can be persisted from creating, reading, updating, and deleting application code to the database. This generation of programming classes from the database schema is possible due to the relationship between a relational database schema and an object oriented programming class. In object-oriented programming, a class is a blueprint to create objects. Objects are instances of a class, and they have attributes that define the state of that object. If an analogy was drawn between databases and programming classes, database tables could be compared to classes, columns to class attributes, and rows (which contain data in a database) to instances of objects.

With this analogy in mind, ORM techniques evolved in the recent years which gave birth to a model called "Database First" approach. This technique enables business applications to just re-generate the programming classes (domain classes) whenever a

[2] For the remainder of the thesis prospectus, any reference to "class" is always to be understood as referring to object-oriented programming class

new field or table needs to be added to the database so that a tedious change to all SQL statements issued from the application code to the database is not necessary.

There are additional methods that are generated by the ORM tool as part of each generated domain class to insert, update, delete and select data from the database. These methods know how to issue SQL statements to insert, update, delete and select records from the database based on the state of the domain classes. For example, if a Contact record (row in a Contact relation that contains attribute values for each contact) from the previous example needs to be inserted into the database, all that the business logic of the application needs to do is to instantiate a new Contact domain class (which was generated by the ORM tool), set the properties of that class to the desired values, and call the Insert method on the Contact class. The Insert method in the Contact class will issue an insert SQL statement with the correct field names (mapped from attribute names) and values so that the attribute values from the Contact class can be persisted to the right columns in the database. In this approach, however, when the database schema is being built, developers must ensure the database is normalized by applying the different normal forms (1NF - 5NF) of database design. Normalization of a relational database is used to ensure there is no redundancy or update/insert/delete anomalies in the data.

1.4 Code First Approach

In modern application development, instead of using a database-first approach that was just discussed, a popular technique is to build business applications using the Code First

approach. In a Code First approach, the programming classes constituting the business/application domain are created first. Then a tool such as Entity Framework (EF) (a third party Microsoft framework) is used to generate the database from the classes. This is the reverse process of database-first approach where an ORM tool generates the programming classes using metadata extracted from the database schema. The EF is the ORM tool discussed in the next paragraph.

In the Code-First approach, programmers might be designing code and realize they need an entity object such as Order or Supplier. So the programmers would simply define such a class in their code that contains the desired attributes not being overly concerned about where or how this entity object is going to be stored/persisted. This gives much flexibility to the programmers because they can just focus on the business logic and how the programming objects interact with other objects and not worry about how they are persisted eventually to a database. So the database schema is not the primary concern. When the programmers decide which database (for example, relational, object-oriented database, etc.) to use, later on, they can always select an ORM tool that will generate the desired database from the entity classes that they have already developed. Noteworthy, there are many third party ORM tools available that know how to generate the desired database from the programmers' programming language of choice.
The steps by which the ORM tools generate a database from programming classes is given below:

1. Extract metadata information such as class names, attribute names, attribute type names, relationships between classes, etc. from the entity classes.
2. Store this metadata information either in memory or a file such as an XML or JSON file.
3. Using the metadata file and the built-in logic to issue the right database statements (example: desired SQL statements), a database is generated by the ORM tool that maps directly with the entity classes.

Since the database generated is controlled by the domain classes in this approach, care needs to be taken to still make certain that the database generated from the code is normalized. To do that, the classes which generate the database also need to be normalized in some way to ensure the database generated from the class is also normalized. We call this Class Normalization.

1.5 Importance of Normalizing Classes

Note that the discussion is about developing “Class Normal Forms” (CNF) that have rules and definitions that may be applied to programming classes. We call this process Class Normalization. The normalized programming classes can then be used to generate the database using the Code-First approach. This approach is restricted to programming using the Code-First approach.

The goal of this thesis, then, is to use both the theory behind the five normal forms in database design and the theory behind object-oriented design to come up with effective Class Normalization. At the conclusion of the thesis research, development of the five class normal forms (1CNF - 5 CNF) will be produced. It is asserted that if the five class normal forms or rules are taken into account when developing the domain entity programming classes, then the database tables generated from those classes using an ORM tool (such as Microsoft's Entity Framework) will also be normalized.

Why is this important? In object-oriented programming, classes are abstractions created to fulfill a need and to represent a real world object and its associated properties. These are then used to solve a problem. Usually, this might be okay. However, when using tools such as the Code-First approach to generate a database from classes, these classes must follow specific rules to support the generation of an efficient, normalized database. For example, a class that is used to generate a table cannot have repetitive attributes that describe the same attributes, as this would violate one of the primary precepts of database normalization.

In the article published by Merunka, [Merunka13], he provides an example of a class model that doesn't follow any normalization rules.

ORDER	SUPPLY
supplier firstname	supplier firstname
supplier surname	supplier surname
client firstname	client firstname

client address	client address
order date	supply date
payment mode	payment mode
first product name	first product name
first product price	first product price
second product name	second product name
second product price	second product price
...	...

Table 1: Order and Supply Classes

Clearly, if the database tables “Order” and “Supply” are generated from these attributes in these two programming classes, the result would result in a violation of the criteria in satisfying 1NF, which excludes variable repeating fields and groups [Kent83]. Such a condition must be avoided at all cost. C. J. Date, a luminary in database, explains that since databases often stay in a production environment for decades after they are initially developed, it is critical to have a careful design to avoid subtle errors and various processing problems over the course of the database Bad design could have a widespread negative impact [Date12].

Given this statement of exactly what this thesis will be addressing coupled with some key definitions and concepts, and before the methodology of this thesis is presented, it is essential to consider a literature review that addresses these important issues

Chapter 2

LITERATURE REVIEW

Database designers have used normalization rules for years as a de facto standard for designing a good database. However, object-oriented programmers took a different approach to designing object-oriented systems. Instead of using normal forms and rules as seen in relational database theory, design patterns evolved as the clear winner in designing object-oriented systems. Many researchers have tried to bring normalization rules to object-oriented programming so that there is a single way to design object-oriented systems [Ambler03]. However, no attempts have been successful in persuading the object-oriented programming community from using normalization techniques instead of design patterns. The different approaches taken by researchers in the field of using normalization techniques to object-oriented programming is discussed in this literature review.

In an article by Lee [Lee95], Lee deals with normalization of the object-oriented data model. He first explains the concept of object-functional dependency which is an integrated version of both functional dependency and multivalued dependency used in the normalization of relations [Lee95]. Based on the notion of object-functional dependency, he later defines his version of the Object Normal Form. The author only provides one Object Normal Form which he thinks sufficiently eliminates update anomalies in an object-oriented data model. His version of the Object Normal Form is given below:

> "i) Create a referenced class if one does not exist.
> ii) Introduce an object reference if one does not exist.
> iii) Move decomposed attributes to the referenced class. Rename the attributes if necessary." [Lee95]

The basic goal of that work is to eliminate update anomalies in an object-oriented data model. Since relational theory already has a process called normalization to eliminate insert, delete, and update anomalies, the author borrowed that concept, modified it to fit the object-oriented data model, and provided steps for what he calls Object Normalization. This approach is similar to the approach of this thesis where the theory behind relational normal forms is taken and a modified version called Class Normal Forms is provided that can be applied to programming classes in a Code-First development approach.

The idea to take the theory behind data normalization and to try to apply it to classes has also been explored and well documented by Scott Ambler in his book [Ambler03]. Ambler agrees that the rules of data normalization can indeed be applied to object schemas. However, he warns that the rules of data normalization cannot be applied to object schemas directly due to the impedance mismatch between the two different systems - databases and programming objects [Ambler03]. However, there is still enough similarity between the two systems that basic concepts of database normalization can be applied to object-oriented design. Therefore, after a reasonable level of modifications, data normalization rules, or at least the theory behind them, can be applied to objects and classes. Ambler's explains that Class Normalization is the process of reorganizing the structure of an object schema to minimize the coupling between classes while still

increasing the cohesion between them. Ambler also gives definitions for Object Normal Forms (ONF) in his book. The three Object Normal Form definitions found in his book are as follows:

> "**First object normal form (1ONF)**: A class is in first object normal form (1ONF) when specific behavior required by an attribute that is actually a collection of similar attributes is encapsulated within its own class.
>
> **Second object normal form (2ONF)** : A class is in second object normal form (2ONF) when it is in first object normal form (1ONF) and when "shared" behavior required by more than one instance of the class is encapsulated within its own class(es)
>
> **Third object normal form (3ONF)** : A class is in third object normal form (3ONF) when it is in second object normal form and when it encapsulates only one set of cohesive behavior" [Ambler03].

As seen above, Ambler's goal regarding Class Normalization is to increase the cohesion and decrease coupling between the classes. So the author deals with both the attributes of a class as well as its behavior in his normalization rules. Although Ambler provides the definition of Class Normalization as well as the first three Object Normal Form definitions, the goal behind them is different than what might be expected from this thesis. This thesis deals with Class Normalization rules that can be applied to classes so that a normalized database can be generated by using a Code-First approach. Also, the normalization rules that the Ambler provides are more for data professionals who are not familiar with Object-Oriented design patterns. The techniques of Class Normalization, according to Ambler, are important for data professionals because it helps them understand basic object design techniques in a manner that is easily digestible. Although the Class Normalization rules that Ambler provides cannot be used directly in this thesis,

there is still plenty of background provided in the book [Ambler03] that can be used to create our own Class Normalization rules for the purpose of this thesis.

In a paper that talks about Normalization of Object Oriented Design, authors Mehdi et al [Lodhi03] attempt to use the theory behind the five normal forms of database design to solve object-oriented design issues. They believe that the object-oriented approach is usually ad-hoc in nature based heavily on the skills of the programmer because there is no formal approach to designing objects. In their work, Mehdi et al outline a normalization mechanism in which the application of normal forms completely removes data redundancy issues as well as functional dependency issues in the object-oriented system. In this paper, the high-level concepts behind relational normal forms (1NF - 5NF) such as decomposition, functional dependency, transitive dependency, and multivalued dependency are applied to object-oriented design. Although an attempt is made to normalize objects in this fashion, no formal object/class normal form definitions are provided at the end of the paper.

Like relational databases, XML databases can contain redundant information as well [El-Sofany09]. El-Sofany explores the possibility of using a modified version of the relational normalization process to normalize XML databases to remove data redundancy and update anomalies. Their goal is to apply the concept of relational database normalization to XML schemas. Like we see here, the normalization process found in relational theory can be used in other domains where data redundancy and update anomalies can occur. This thesis

takes this concept further and tries to take the normalization techniques found in relational theory and apply them to programming classes.

Bura explains that there are three models in the database system such as the conceptual model, the logical model and the physical model [Bura12]. Database normalization is usually done at the logical level (e.g., Relational Database Schema). Bura and his colleagues have explored the possibility of applying normalization at the conceptual level (e.g., Entity-Relationship (ER) model) so that even before the design gets to the logical level, the database schema is already normalized. The authors state the definition of the different normal forms and propose their own custom algorithm (which is based on the normal form definition) with examples to prove that normalization can happen at the conceptual level of database design. This is useful because similar to how the authors here propose applying normalization even at the conceptual level to arrive at a relational schema down the stream that is already normalized, this thesis proposes applying normalization at the programming class level so that a database that is generated from those classes (using Code-First approach) is guaranteed to be normalized. This thesis will take a similar approach taken by these authors in the sense that the thesis will:

1. State the definition of the different relational normal forms
2. Propose new rules derived from relational normal forms but pertaining more towards object-oriented paradigm
3. Prove that the rules derived at Step 2 work by providing examples

Normalization techniques, as we know, are typically used to normalize the relational data model. In the article "Normalization Rules of the Object-Oriented Data Model," Merunka [Merunka13] dives deep into the idea of using normalization techniques to normalize the object-oriented data model. Unlike other authors who try to apply normalization rules to attributes as well as behavior of an object, Merunka clearly explains his normalization rules do not apply to the behavior of a class (i.e. methods of a class) and that they only apply to the data properties of an object. Based on this precondition, the article lays out the following normalization rules for the object-oriented data model:

> "**First Normal Form Rule**: A class is in the first object normal form (1ONF) when its objects do not contain group of repetitive attributes. Repetitive attributes must be extracted into objects of a new class. The group of repetitive attributes is then replaced by the link at the collection of the new objects. An object schema is in 1ONF when all of its classes are in 1ONF.
>
> **Second Normal Form Rule**: A class is in the second object normal form (2ONF) when it is in 1ONF and when its objects do not contain attribute or group of attributes, which are shared with another object. Shared attributes must be extracted into new objects of a new class, and in all objects, where they appeared, must be replaced by the link to the object of the new class. An object schema is in 2ONF when all of its classes are in 2ONF.
>
> **Third Normal Form Rule**: A class is in the third object normal form (3ONF) when it is in 2ONF and when its objects do not contain attribute or group of attributes which have the independent interpretation in the modeled system. These attributes must be extracted into objects of a new class and in objects where they appeared, must be replaced by the link to this new object. An object schema is in 3ONF when all of its classes are in 3ONF.
>
> **Fourth Normal Form Rule**: A class is in the fourth object normal form (4ONF) when it is in 3ONF and when there is no other class in the system, which defines the same attributes. These attributes must be extracted from classes, where they are duplicated, and affected classes must be connected using class inheritance in order to exclude data definition duplicates. If there is no existing class to be reused as an inheritance superclass, a new

> superclass must be added into the system. An object schema is in 4ONF when all of its classes are in 4ONF" [Merunka13].

Merunka and his colleagues approach Object Normalization from a pure data model and storage perspective. That is why they do not concern themselves with accounting for behavior of objects. This is extremely useful for us because we approach Class Normalization the same way by not accounting for behavior/methods of objects because we only deal with data objects.

Although the authors have laid out Object Normal Form definitions from 1ONF through 4ONF, their intent behind those rules are very different. Their goal is to apply their normalization rules to Object-Oriented data models so that the programming objects that are normalized in this fashion can be eventually stored in an object-oriented database. However, the goal of Class Normalization in this thesis is a little different in the sense that the normalization rules for classes that will be produced are intended to be used with programming classes that carry domain data that will eventually be stored in a relational database, and not an object-oriented database. Therefore, Class Normalization rules focus more on normalizing a class so that a table generated from the class in a relational database sense will be fully normalized. Also, the object normal definitions given by Merunka are more at a higher level than what one might expect. The definitions are not broken down into basic rules/steps thus giving room for ambiguity. In this thesis, it is proposed to give very basic rules that can be easily understood and applied to programming classes.

Chapter 3

METHODOLOGY

3.1 Overview

The goal of this thesis is to generate a relationally normalized database that is free from data redundancies and update anomalies using the Code-First approach of software design. This is done by taking the theory behind relational normal forms, creating class normal forms from them, and applying the class normal forms manually to programming classes. After applying the class normal forms to the programming classes, a tool such as Entity Framework is used to generate a database from the normalized classes. After the database is generated, it can be tested to see whether it is normalized. If it is, then the goal of this thesis is met.

To achieve this goal, the following steps will be followed for each of the five database normal forms in relational database theory:

1. State the rules of the database normal form
    a. This step will state the definition of one of the five normal forms in relational database theory. This step will also explain the rules/theory behind the normal form.
2. Provide class normal form rules based on the database normal form

a. This step will take the rules behind the database normal form in step 1 and provide similar rules that can be applied to programming classes. This can be accomplished using the following simple mapping chart between relational database paradigm and object-oriented paradigm.

Relational Database Paradigm	Object-Oriented Programming Paradigm
Table	Class
Column	Property or attribute of a class
Row	Object (instance of a class)
Foreign key between tables	Relationship between objects

Table 2: Relational Database Paradigm Mapping to Object-Oriented Paradigm

3. Apply the class normal form to an example
    a. This step will provide an example of how a programming class will look like before and after applying the class normal form provided in step 3
4. Generate the database and test
    a. This step will use the Entity Framework tool to generate a database from the example normalized programming classes given in step 3.
    b. A screenshot of the generated database output will be provided. The goal is to show that the generated database is normalized.

3.2 Software Used

To fulfill the steps stated in the previous section, the following software is used for the purpose of this thesis:

- Microsoft® SQL Server 2014 and Management Studio
    - SQL Server is the database store and Management Studio is used to manage the server and create database diagrams.
- Microsoft® Visual Studio Professional 2015
    - Visual Studio is the development IDE used to create the programming classes.
- Microsoft® Windows 10 Home Edition
    - Windows 10 is the operating system upon which other software programs run.
- Microsoft® Entity Framework 6.1.3
    - Entity Framework library is installed as a NuGet package into Visual Studio for the purpose of generating the database from domain programming classes. A simple representation of the working of Entity Framework's Code-First approach is given below:

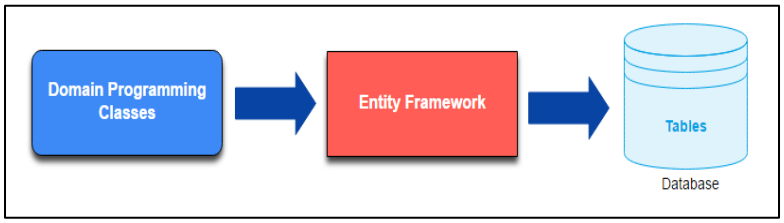

Figure 1: Code-First Approach

Chapter 4

RESULTS

Using the methodology proposed in Chapter 3, here are the Five Class Normal Forms based on the theory behind the Five Database Normal Forms:

4.1 First Class Normal Form (1CNF)

The First Class Normal Form rules are explained below.

4.1.1 Rules of First Normal Form

According to Stephens in [Stephens09], the rules behind the first normal form are as follows:

1. Each column must have a unique name
2. Each column must have a single data type
3. No two rows can contain identical values
4. Each column must contain a single value
5. Columns cannot contain repeating groups

4.1.2 First Class Normal Form Rules Based on First Relational Normal Form

1. Each property of a class must have a unique name
    - This is default behavior because any object-oriented language disallows properties with the same name within a class
2. Each property must have a single data type
    - This 1NF rule insists that a property cannot have more than one data type. To satisfy this in object-oriented programming, the type of a property cannot be a collection of basic data types such as int, boolean, string, decimal, etc. (Example: Array<string> PhoneNumbers is not allowed). The only exception to this rule occurs in the instance where the collection type of the property is actually a type of an object within the entity model.
3. No two objects can have identical values
    - This does not come naturally to object-oriented programmers because there is no existing language constraint in object-oriented programming to prevent two objects from having the exact same values. In cases when objects are cloned from an existing object, you now certainly have two objects with the same values. However, if entity objects are to be persisted to the database, care should be taken by the programmer to prevent this scenario. Otherwise, a database error will be thrown when the object is persisted to the database.
4. Each property must contain a single value

- For this to be true, the property can only have valid SQL data types such as int, decimal, date etc. The type of the property cannot be any kind of array or collection type. The only exception to this rule occurs in the instance where the collection *type* of the property is actually a type of an object within the entity model.

5. Properties cannot contain repeating groups
    - This means that different properties in a class cannot represent the same thing. For example, if there is a requirement to store multiple authors for a book, multiple properties called Author1, Author2 and Author3 cannot be part of the same class. This will violate the First Class Normal Form. It is better to split this into a class called "Author" and have a property called "Authors" which is a collection of related Author objects.

4.1.3 Example

Assume a contact manager application to store the following data is being built:

Name	Name	Name	Phone Numbers	Address
Daniel	Sushil	Sudhindaran	9041234321, 6548765678	1234 Falls Dr, Jacksonville, FL 32267

Table 3: Contact Manager Application Data

The code below shows a valid C# class that could be used to store this data before any Class Normal Form rules are applied:

```
public class Contact
{
    public string Name1  { get; set; }

    public string Name2 { get; set; }

    public string Name3 { get; set; }

    public List<string> PhoneNumbers { get; set; }

    public string Address { get; set; }
}
```

This class is not in First Normal Form because it violates rules 3, 4 and 5 of the First Class Normal Form as explained below:

Violation of rule 3: There is no unique identifier for this class. Since there is no unique identifier or a property that guarantees uniqueness, two objects could have the exact same values. Rule 3 of the First Class Normal form doesn't allow this.

Application of First Class Normal Form: Add a property to uniquely identify each instance of the class. A unique integer value could be set to this property at runtime to guarantee uniqueness of the object.

```
public class Contact
{
    public int ContactID { get; set; }

    public string Name1 { get; set; }

    public string Name2 { get; set; }

    public string Name3 { get; set; }

    public List<string> PhoneNumbers { get; set; }

    public string Address { get; set; }
}
```

Violation of rule 4: The PhoneNumbers property does not contain a single value. It contains a list of strings. According to rule d of First Class Normal form, collections cannot be used as valid values for a property unless the type of the property is another object or a collection of objects within the entity model (objects that have a table in the database).

Application of First Class Normal Form: Create a new class called PhoneNumber and have a property called List<PhoneNumber> in the Contact class. The PhoneNumber object can now be a related collection of Contact object.

```
 public class Contact
{
    public int ContactID { get; set; }
    public string Name1 { get; set; }
    public string Name2 { get; set; }
    public string Name3 { get; set; }
    public List<PhoneNumber> PhoneNumbers { get; set; }
    public string Address { get; set; }
}
public class PhoneNumber
{
    public int PhoneNumberID { get; set; }
    public Contact Contact { get; set; }
    public string Number { get; set; }
}
```

Violation of rule 5: The class Contact contains the repeating groups Name1, Name2 and Name3 that mean the same thing. Name1, Name2 and Name3 cannot be differentiated between one another and is ambiguous.

Application of First Class Normal Form: Name the three properties appropriately such as FirstName, MiddleName and LastName. This makes the intent of each property extremely clear and prevents ambiguity.

This is the final version of the class after applying the First Class Normal Form rules.

```
public class Contact
 {
     public int ContactID { get; set; }

     public string FirstName { get; set; }

     public string MiddleName { get; set; }

     public string LastName { get; set; }

     public List<PhoneNumber> PhoneNumbers { get; set; }

     public string Address { get; set; }
 }

 public class PhoneNumber
 {
     public int PhoneNumberID { get; set; }

     public Contact Contact { get; set; }

     public string Number { get; set; }
 }
```

4.1.4 The Generated Output

Using Entity Framework's Code-First approach, the database that is generated from the Contact and PhoneNumber classes are shown below:

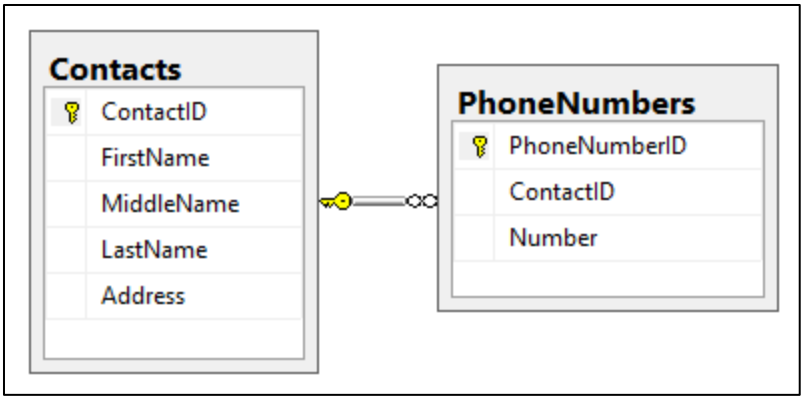

Figure 2: First Class Normal Form Result

The above two tables that are generated from the classes Contact and PhoneNumber are in the First Normal Form of relational database design.

4.2 Second Class Normal Form (2CNF)

The Second Class Normal Form rules are explained below.

4.2.1 Rules of Second Normal Form

According to Stephens in [Stephens09], the rules of the second normal form are as follows. A table is in 2NF if:

1. It is in 1NF
2. All of the non-key fields depend on all of the key fields.

4.2.2 Second Class Normal Form Rules Based on Second Relational Normal Form

A class is in 2CNF if:

1. It is in 1CNF
    - This rule insists that for a class to be in 2CNF, it needs to satisfy all the rules of 1CNF first.
2. All non-primary properties of a class are related to all primary properties of the class.
    - A primary property here means that it is used to uniquely identify an object. A class may have one or more primary properties where all of the primary properties are needed before the object can be uniquely identified from the rest of the objects. If one of the primary properties does not exist, then duplicate objects will exist in the system. Therefore, this class normal form states that any other non-primary property is a class should directly relate to all of the primary properties of the class.

4.2.3 Example

Assume we have to store the following purchase details information:

Customer ID	**Store ID**	**Store Location**
1	1	Jacksonville
2	1	Tampa

Table 4: Purchase Details

The Customer ID and the Store ID are both used to uniquely identify the purchase, which makes them the primary fields. Now examine a valid C# class that could be used to store this data before any Class Normal Form rules are applied:

```
public class Order
{
    public int CustomerID { get; set; }

    public int StoreID { get; set; }

    public int StoreLocation { get; set; }
}
```

This class is not in Second Normal Form because it violates rule 2 of the Second Class Normal Form as detailed below:

Violation of rule 2: The Second Class Normal Form states that, in a class, all the non-primary properties of the class should directly be related to the primary properties of the class. However, the StoreLocation property is only related to the StoreID and not necessarily related to the CustomerID. Therefore, this class violates rule b.

Application of Second Class Normal Form: The StoreLocation property is not completely related to both CustomerID and StoreID. So it does not really belong in the Purchase class. Having a new class called Store that contains the StoreID (primary property) and StoreLocation (that depends on the StoreID) and having a link back to the Purchase class will satisfy the Second Class Normal Form:

```
public class Purchase
{
    public int CustomerID { get; set; }

    public Store Store { get; set; }
}

public class Store
{
```

```
        public int StoreID { get; set; }

        public string StoreLocation { get; set; }
    }
```

4.2.4 The Generated Output

Using Entity Framework's Code-First approach, the database that is generated from the Purchase and Order classes are shown below:

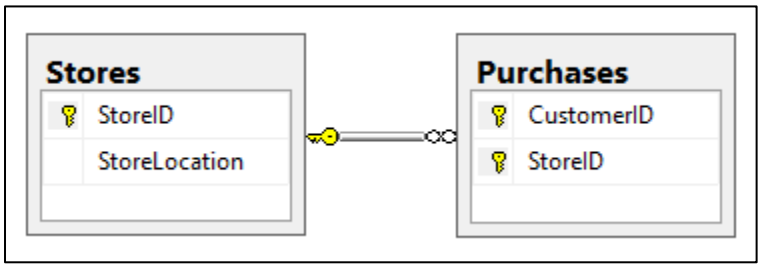

Figure 3: Second Class Normal Form Result

The above two tables that are generated from the classes Purchase and Store are in the Second Normal Form of relational database design.

4.3 Third Class Normal Form (3CNF)

The Third Class Normal Form rules are explained below.

4.3.1 Rules of Third Normal Form

According to Stephens in [Stephens09], the rules of the third normal form are as follows:

A table is in 3NF if:

1. It is in 2NF
2. It contains no transitive dependency.
    - In simple terms, if column A is dependent on column B and column B is dependent on column C, then C is transitively dependent on column A through B. This is called transitive dependency. In other words, all of the columns in a table need to be directly dependent on the primary columns of a table, and there cannot be any transitive dependencies.

4.3.2 Third Class Normal Form Rules Based on Third Relational Normal Form

1. A class is in 3CNF if it is in 2CNF
    - This rule insists that for a class to be in 3CNF, it needs to satisfy all the rules of 2CNF first.
2. Classes should have one, and only one, responsibility. In other words, classes should contain properties that store information about one, and only one entity in the real world.
    - This rule is taken from the well-known object-oriented design pattern called “Single Responsibility Principle” (SRP*)* [Martin09]. This principle focuses more on the responsibility (or reason to change) of a class and

implies that a class should not have more than one responsibility. If a class is used to store information regarding more than one responsibility, it should be split up into two or more classes where each class can then have a single responsibility (or a single reason to change the class). In other words, classes should store information about only one entity. This means that all of the properties of a class should contain direct information about only one entity.

4.3.3 Example

Assume an ordering system is to be built that needs to store the following data:

Customer ID	Store ID	Order Number	Customer Phone Number	Store Location
1	1	34567	904-121-1245	Jacksonville
2	1	7588	766-234-1234	Tampa

Table 5: Ordering System

Now examine a valid C# class that could be used to store this data before any Class Normal Form rules are applied:

```
public class Order
{
    public int CustomerID { get; set; }
    public int StoreID { get; set; }
    public int OrderNumber { get; set; }
    public int CustomePhoneNumber { get; set; }
```

```
    public int StoreLocation { get; set; }
}
```

This class is not in Third Normal Form because it violates rule 2 of the Third Class Normal Form.

Violation of rule 2: The Third Class Normal Form states that a class should not have more than one responsibility. However, the class Order in the above example is trying to store the customer information, the order information as well as the store information, all in the same class. In other words, there are three reasons to change this class:

- When the customer information changes
- When the store information changes
- When the order information changes

According to the Third Class Normal Form, there should be only one reason to change the class.

Application of Third Class Normal Form: Since the Order class has three responsibilities, it needs to be split into three different classes - one to store customer information, one to store order information and one to store information. The following three smaller classes satisfy the Third Class Normal Form:

```
public class Customer
{
    public int CustomerID { get; set; }

    public int CustomerPhoneNumber { get; set; }
}

public class Store
{
    public int StoreID { get; set; }
    public string StoreLocation { get; set; }
}
```

```
public class Order
{
    public int OrderNumber { get; set; }

    public int CustomerID { get; set; }

    public int StoreID { get; set; }
}
```

4.3.4 The Generated Output

Using Entity Framework's Code-First approach, the database that is generated from the Customer, Store and Order classes are shown below:

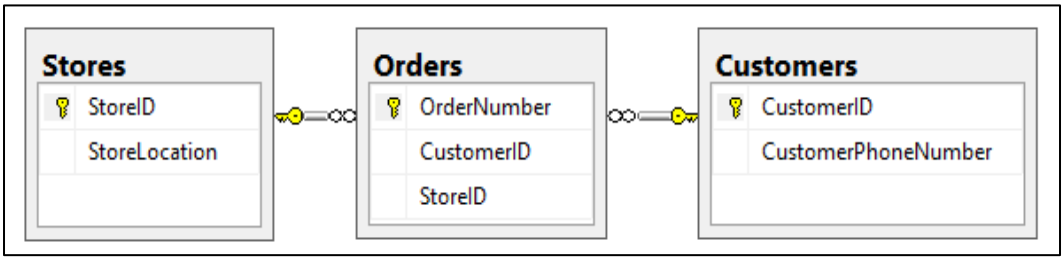

Figure 4: Third Class Normal Form Result

The above three tables that are generated from the classes Order, Store and Customer are in the Third Normal Form of relational database design.

4.4 Fourth Class Normal Form (4CNF)

The Fourth Class Normal Form rules are explained below.

4.4.1 Rules of Fourth Normal Form

According to Kent in [Kent83], the rules of the fourth normal form are as follows:

A record is in 4NF if:

1. It is in 3NF
2. It does not contain two or more independent multi-valued facts about an entity
    a. In simple words, a record should not contain two or more many-to-many relationships in it.

4.4.2 Fourth Class Normal Form Rules Based on Fourth Relational Normal Form

The following are the rules of the Fourth Class Normal Form:

1. A class is in 4CNF if it is in 3CNF
    - This rule insists that for a class to be in 4CNF, it needs to satisfy all the rules of 3CNF first.
2. An entity class A cannot store multiple values of another entity class B without creating a bridge class that stores the relationship between A and B (i.e. AB). The class A may then have a property to hold values of the bridge class AB to represent the multiple values of entity B within entity class A.

4.4.3 Example

Assume we have to store information regarding an employee's skill and language:

Employee ID	Skill	Language
1	Programming, Singing, Writing	French, English, Spanish
2	Running, Writing, Driving	English, Dutch, French

Table 6: Employee Information

Now look at a valid C# class that could be used to store this data before any Class Normal Form rules are applied:

```
public class Employee
{
    public int EmployeeID { get; set; }

    public List<Skill> Skills { get; set; }

    public List<Language> Languages { get; set; }
}

public class Skill
{
    public int SkillID { get; set; }

    public string SkillName { get; set; }
}

public class Language
{
    public int LanguageID { get; set; }

    public string LanguageName { get; set; }
}
```

The class Employee is not in Fourth Normal Form because it violates rule 2 of the Fourth Class Normal Form.

Violation of rule 2: The Fourth Class Normal Form does not allow for multi-valued entity properties such as Skills and Languages to be direct properties of a class.

Application of Fourth Class Normal Form: The following steps need to be taken to conform the Employee class to the Fourth Class Normal Form:

- Create a class called EmployeeSkill to store the relationship between Employee and Skill
- Create a class called EmployeeLanguage to store the relationship between Employee and Language
- Replace the type of Skills property in Employee class from List<Skill> to List<EmployeeSkill>
- Replace the type of Languages property in Employee class from List<Language> to List<EmployeeLanguage>

The following smaller classes satisfy the Fourth Class Normal Form:

```
public class Employee
{
    public int EmployeeID { get; set; }

    public List<EmployeeSkill> Skills { get; set; }

    public List<EmployeeLanguage> Languages { get; set; }
}

public class EmployeeSkill
{
    public int SkillID { get; set; }

    public int EmployeeID { get; set; }
}

public class EmployeeLanguage
{
    public int LanguageID { get; set; }

    public int EmployeeID { get; set; }
}

public class Skill
{
    public int SkillID { get; set; }

    public string SkillName { get; set; }
}
```

```
public class Language
{
    public int LanguageID { get; set; }

    public string LanguageName { get; set; }
}
```

4.4.4 The Generated Output

Using Entity Framework's Code-First approach, the database that is generated from the Employee, EmployeeSkill, EmployeeLanguage, Skill and Language classes are shown below:

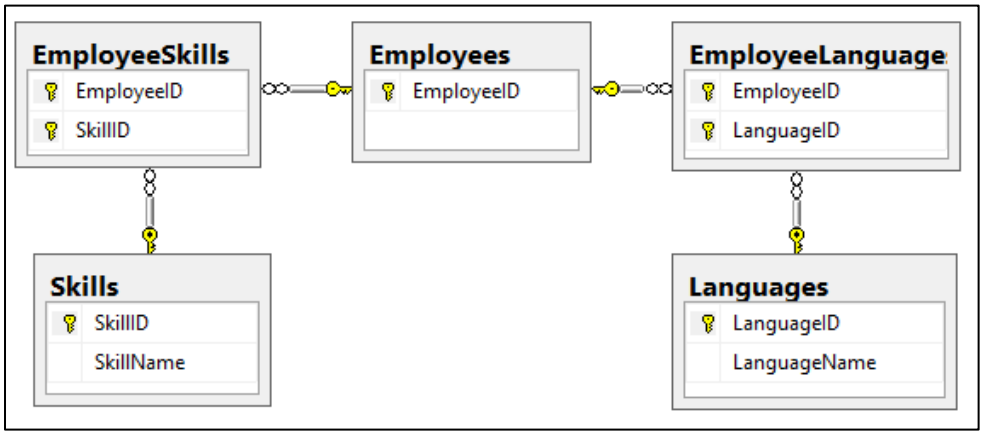

Figure 5: Fourth Class Normal Form Result

The above five tables that are generated from the classes Employee, EmployeeSkill, EmployeeLanguage, Skill and Language are in the Fourth Normal Form of relational database design.

4.5 Fifth Class Normal Form (5CNF)

The Fifth Class Normal Form rules are explained below.

4.5.1 Rules of Fifth Normal Form

According to Stephens in [Stephens09], the rules behind the fifth normal form are as follows:

A table is in 5NF if:

1. It is in 4NF
2. It contains no related multi-valued dependencies
    a. If a table has related multi-valued dependencies, each multi-valued dependency should be separated out into its own table. The individual tables still need to maintain their relationships with each other.

4.5.2 Fifth Class Normal Form Rules Based on Fifth Relational Normal Form

A class is in 5CNF if:

1. It is in 4CNF
    - This rule insists that for a class to be in 5CNF, it needs to satisfy all the rules of 4CNF first.
2. It contains no related multi-valued properties within it.

- Assume class A has two multi-valued properties P1 and P2 where P1 and P2 are related to each other. For class A to be in 5CNF, the following steps need to be taken:
    i. The multivalued properties P1 and P2 need to be separated out into their own classes B and C.
    ii. New classes that store the relationship information between classes A and B, A and C, and B and C needs to be introduced (such as AB, AC, BC).

4.5.3 Example

Assume we have to store information regarding agents, companies and phones. Agents work for companies and companies sell phones. We need to keep track of which agents sell which phones for which companies:

Agent	Company	Phone
Bob	Apple, Samsung	iPhone 6S, Samsung Galaxy S7
Richard	Microsoft, Apple	iPhone 7, Microsoft Lumina 950

Table 7: Agent Information

Below is a valid C# class that could be used to store this data before any Class Normal Form rules are applied:

```
public class Agent
{
```

```
    public int AgentID { get; set; }

    public List<Company> Companies { get; set; }

    public List<Phone> Phones { get; set; }
}

public class Company
{
    public int CompanyID { get; set; }

    public string CompanyName { get; set; }
}

public class Phone
{
    public int PhoneID { get; set; }

    public string PhoneName { get; set; }
}
```

The class Agent is not in Fifth Normal Form because it violates rule 2 of the Fifth Class Normal Form as shown below:

Violation of rule 2: The Fifth Class Normal Form does not allow for related multi-valued entity properties such as Companies and Phones to be direct properties of a class.

Application of Fifth Class Normal Form: The following steps need to be taken to conform the Agent class to the Fifth Class Normal Form:

- Create a class called AgentCompany to store the relationship between Agent and Company so we know which Agent works for which Company.
- Create a class called AgentPhone to store the relationship between Agent and Phone so we know which Agent sells which Phones.
- Create a class called PhoneCompany to store the relationship between Phone and Company so we know which company sells which phone.

The following smaller classes satisfy the Fifth Class Normal Form:

```
public class Agent
{
    public int AgentID { get; set; }

    public string AgentName { get; set; }
}

public class AgentCompany
{
    public int AgentID { get; set; }

    public int CompanyID { get; set; }
}

public class AgentPhone
{
    public int AgentID { get; set; }

    public int PhoneID { get; set; }
}
public class CompanyPhone
{
    public int CompanyID { get; set; }

    public int PhoneID { get; set; }
}

public class Company
{
    public int CompanyID { get; set; }

    public string CompanyName { get; set; }
}

public class Phone
{
    public int PhoneID { get; set; }

    public string PhoneName { get; set; }
}
```

4.5.4 The Generated Output

Using Entity Framework's Code-First approach, the database that is generated from the Agent, Company, Phone, AgentCompany, AgentPhone, and CompanyPhone classes are shown below:

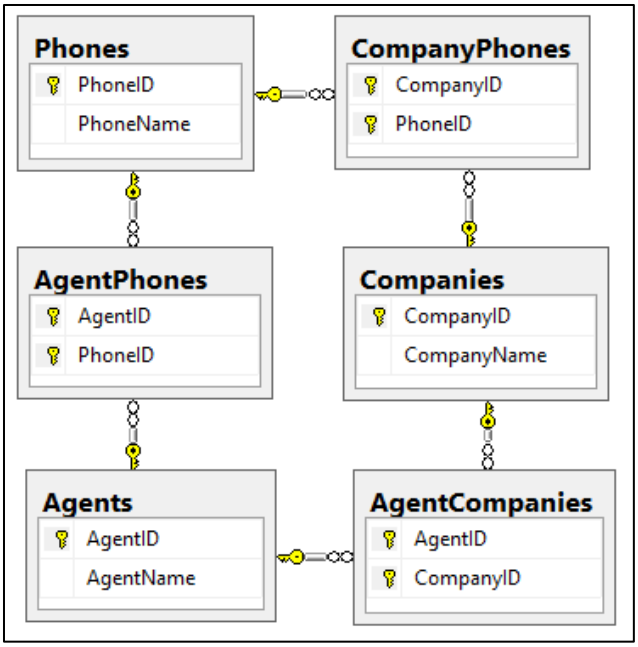

Figure 6: Fifth Class Normal Form Result

The above six tables that are generated from the classes Agent, Company, Phone, AgentCompany, AgentPhone, and CompanyPhone are in the Fifth Normal Form of relational database design.

It has been demonstrated in this chapter that Class Normal Forms can be successfully derived from the theory behind Database Normal Forms. It has also been demonstrated with detailed examples that the Five Class Normal Forms, when applied correctly to programming classes, results in a relational database that is completely normalized.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

The Class Normal Forms provided in this thesis are carefully taken from well-known theories behind database normal forms. Class Normalization in a Code-First approach is important because database normalization is important in a relational database design. The whole point of this thesis is to aid developers from an object-oriented development background to use Class Normalization as a technique to make sure their relational databases are normalized.

Database normalization is an extremely critical process in the development of a business application that relies on a relational database. However, since the introduction of the Code-First approach, database normalization could be missed completely by programmers because the focus has shifted from developing the database first to developing the code first. Since database normalization is an afterthought in the Code-First approach, it can lead to redundancies and update anomalies in the database system if normalization is not accounted for. If a database is not designed correctly, it could cost businesses hundreds to thousands of man hours to fix problems when issues arise. Also, if database normalization is not taken into account, it could lead to redundant data being stored in the database. This can increase the size of the overall database which in turn

increases the storage costs for a business. Therefore, Class Normalization using the Class Normal Forms proposed in this thesis needs to be incorporated into the software development process of enterprise organizations. Just by including a process such as Class Normalization into their software development, enterprises can save a lot of money in the long run.

5.2 Future Work

This thesis has proposed Five Class Normal Forms in the form of rules to be applied to a programming class during software development. These rules can be understood by a software programmer and the programmer could manually follow these rules to make sure a programming class is normalized. However, these rules could potentially be translated into programmatic rules and could be fed into a software system. This software system could then be used to validate a software solution to see if any entity programming classes within the software solution violate any of the Class Normalization rules discussed in this thesis. This type of a software program, once developed, could be used as part of the software build and deployment process of an enterprise. That way, an enterprise can make sure that their database is completely normalized.

A sample prototype program on how this can potentially be implemented is appended to Appendix A of this thesis. The program validates the most common cases of violations of the First Class Normal Form (1CNF) and outputs the error/warning messages found. The program points out exactly which programming entity class has errors and also gives

suggestions on how to fix the errors. This program does not modify the entity classes automatically. Modifying a programming class automatically can cause various issues for a programmer because the classes that are newly modified/created by the automated program can cause new compilation errors due to how they may fit into the rest of the solution. That is why several plugins and helper tools validate a program and let the programmer know about any errors or warnings so that the programmer could surgically fix the errors. This works much like how compilers for several programming languages work. Compilers will cite the errors in syntax, but the repair needs to be done by the programmer.

The example validation program provided in the Appendix serves as an example of how an automated system could be developed to validate all the five Class Normal Forms (1CNF – 5CNF) for a given set of entity programming classes.

The inputs and outputs of the validation program to validate the First Class Normal Form are provided below:

Input entity programming class:

```
namespace ClassNormalization.Sample
{
    public class Contact
    {
        public string Name1 { get; set; }
        public string Name2 { get; set; }
        public string Name3 { get; set; }
        public string[] PhoneNumbers { get; set; }
```

```
        public string Address { get; set; }
    }
}
```

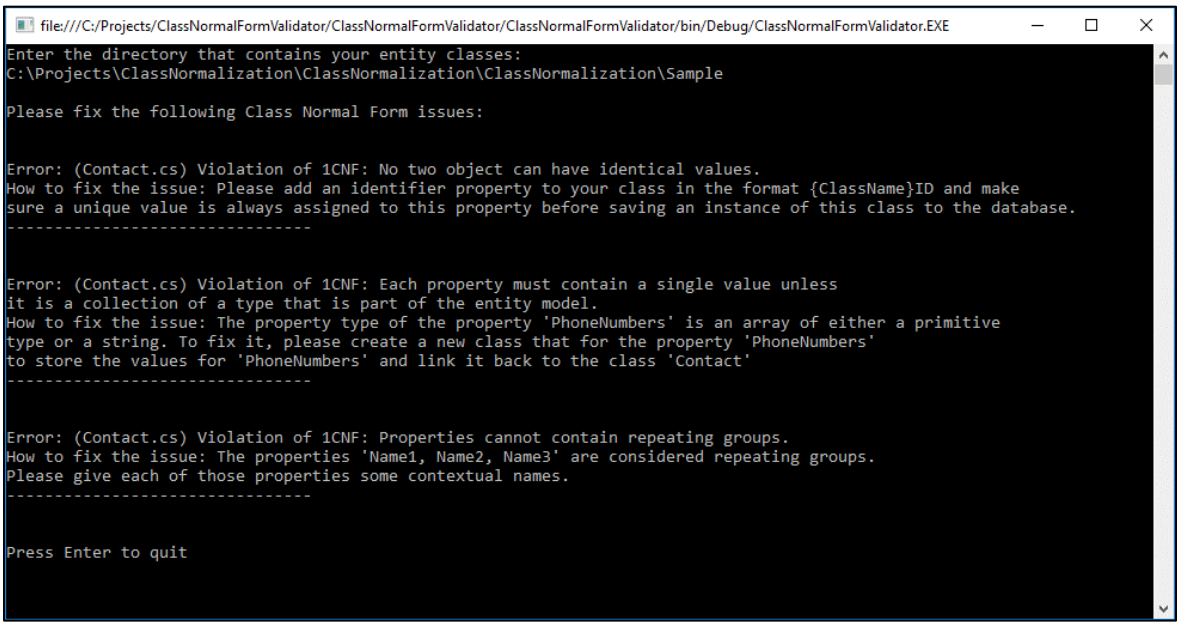

Figure 7: Console Output

The sample program provided concludes its diagnostics at validating the First Class Normal form because validating additional Class Normal Forms (2CNF – 5CNF) is challenging and is beyond the scope of the thesis.

For example, here is a sample class that violates 2CNF:

```
public class Order
{
    public int CustomerID { get; set; }

    public int StoreID { get; set; }

    public int StoreLocation { get; set; }
}
```

From the above program, it can be seen that 2CNF is violated because the StoreLocation property belongs in the Store entity class and does not belong in the Order entity. Any software programmer can easily make that determination by looking at the class and split the Order class as seen below:

```
public class Order

{
    public int CustomerID { get; set; }

    public Store Store { get; set; }
}

public class Store
{
    public int StoreID { get; set; }

    public string StoreLocation { get; set; }
}
```

However, a program cannot automatically detect a violation of 2CNF unless the program has additional information that may include the following:

- Complete knowledge of the domain of the solution.
- Metadata on the semantic meaning of different entity classes in the solution.
- The exact usage data of the classes/attributes in the domain and the entire software solution.

Additional information to detect 2CNF and above might be needed based on the implementation of the solution. It can be done. However, it will have to be its own thesis with a thorough research on all possible solutions and implementation details.

Although it may seem feasible, building a software system sufficiently sophisticated to understand the Class Normal Form rules (2CNF – 5CNF), their semantic meaning, and validating an entity class would be a major undertaking indeed. Complexities associated

with diagnosing and repairing such software (understanding the application domain, related semantics and more) may be some of the reasons why such software is not currently available.

REFERENCES

Print Publications:

[Ambler97]
Ambler S., Building Object Applications That Work: Your Step-By-Step Handbook for Developing Robust Systems with Object Technology, New York: Cambridge University, 1997.

[Ambler03]
Ambler, S., Agile Database Techniques: Effective Strategies for the Agile Software Developer, Wiley, 2003.

[Booch94]
Booch, G., Object-Oriented Analysis and Design with Applications, Redwood City, Calif: Benjamin/Cummings Pub. Co, 1994.

[Bura12]
Bura, D. and R. K. Singh, "Implementing Constraints in Entity-Relationship Models for Enhancing Normalization," IUP Journal of Information Technology 8, 2 (June 2012), pp 46-57.

[Coad92]
Coad, P., "Object-Oriented Patterns," Communications of the ACM 35, 9 (September 1992), pp 152-159.

[Codd70]
Codd, E. F., “A relational model for large shared data banks,” Communications of the ACM 13, 6, (June 1970), pp 377-387

[Date12]
Date, C. J., Database Design and Relational Theory: Normal Forms and All That Jazz, O'Reilly Media, 2012.

[El-Sofany09]
El-Sofany, H. and S. Abou El-Seoud, "Schema Design and Normalization Algorithm for XML Databases Model,” International Journal of Emerging Technologies in Learning 4, 2 (May 2009), pp 11-21.

[Fowler92]
Fowler J. and M. West, “Developing High Quality Data Models,” The European Process Industries STEP Technical Liaison Executive (EPISTLE), 1992.

[Gamma95]
Gamma, E., Design Patterns: Elements of Reusable Object-Oriented Software, Reading, Mass: Addison-Wesley, 1995.

[Gorman90]
Gorman, K. and J. Choobineh, "The Object-Oriented Entity-Relationship Model (OOERM)," Journal of Management Information Systems 7, 3 (October 1990), pp 41-65

[Kent83]
Kent, W., "A simple guide to five normal forms in relational database theory," Communications of the ACM 26, 2 (February 1983), pp 120-125.

[Lee95]
Lee, B. S., "Normalization in OODB design," SIGMOD Record 24, 3 (September 1995), pp 23-27.

[Lerman12]
Lerman, J. and R. Miller, Programming Entity Framework: Code First, O'Reilly Media, 2012.

[Liles12]
Liles, D. and T. Rayburn, Entity Framework 4.1: Expert's Cookbook, Birmingham, UK: Packt Publishing, 2012.

[Martin09]
Martin, R., Clean Code: A Handbook of Agile Software Craftsmanship, Pearson Education, Inc, 2009.

[Merunka13]
Merunka, V. and J. Tůma, “Normalization Rules of the Object-Oriented Data Model,” In Proceedings of the International Workshop on Enterprises & Organizational Modeling and Simulation, 2009

[Nien-Lin09]
Nien-Lin, H., K Jong-Yih and L. Ching-Chiuan, "Object-Oriented Design: A Goal Driven And Pattern-Based Approach," Software & Systems Modeling 8, 1 (February 2009), pp 67-84.

[Stephens09]
Stephens, R., Beginning Database Design Solutions, Indianapolis, IN: Wrox, 2009.

[Torres17]
Torres A, Galante R, Pimenta M, Martins A., “Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design,” Information and Software Technology, 82 (February, 2017), pp 1-18.

APPENDIX A

First Class Normal Form Validator Prototype

```
using ClassNormalizationRules;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ClassNormalFormValidator
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Enter the directory that contains your
entity classes:");
            string dirPath = Console.ReadLine();

            ClassNormalFormsValidator validator = new
ClassNormalFormsValidator();

            try
            {
                List<string> loadErrors = validator.Load(dirPath);

                if(loadErrors.Count > 0)
                {
                    Console.WriteLine("Error encountered during
compilation of the source code:");

                    foreach (string error in loadErrors)
                    {
                        Console.WriteLine(error);
                    }
                }
                else
                {
                    List<ValidationResult> validationResults =
validator.GetValidationResults();

                    if (validationResults.Count > 0)
                    {
                        Console.WriteLine("\r\nPlease fix the following
Class Normal Form issues:");
```

```
                        foreach (ValidationResult result in
validationResults)
                        {
                            Console.WriteLine("\r\n");
                            Console.WriteLine(result.Level ==
ValidationLevel.Waning ? "Warning" : "Error" + ": " +
result.Violation);
                            Console.WriteLine("How to fix the issue: "
+ result.HowToFix);
                            Console.WriteLine("----------------------
--------");
                        }

                        End();
                    }
                    else
                    {
                        Console.WriteLine("No errors found!");
End();
                    }
                }

            }

            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                End();
            }
        }

        private static void End()
        {
            Console.WriteLine("\r\n");
            Console.WriteLine("Press Enter to quit");
            Console.ReadLine();
        }
    }
}


using Microsoft.CSharp;
using System;
using System.CodeDom.Compiler;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ClassNormalizationRules
{
    public class ClassNormalFormsValidator
    {
        private List<Type> _types = new List<Type>();

        public List<string> Load(string dirPath)
```

```
        {
            List<string> sources = new List<string>();
            List<string> errors = new List<string>();

            foreach (string file in Directory.GetFiles(dirPath,
"*.cs"))
            {
                sources.Add(File.ReadAllText(file));
            }

            CompilerParameters parameters = new CompilerParameters();
            parameters.GenerateExecutable = false;
            parameters.GenerateInMemory = true;

            parameters.ReferencedAssemblies.Add("mscorlib.dll");
            parameters.ReferencedAssemblies.Add("System.dll");

            var provider = new CSharpCodeProvider();
            var results =
provider.CompileAssemblyFromSource(parameters, sources.ToArray());

            if (results.Errors.HasErrors)
            {
                foreach(CompilerError error in results.Errors)
                {
                    errors.Add(string.Format("Compilation Error at: {0}
with Text: {1}", error.Line, error.ErrorText));
                }

                return errors;
            }

            var assembly = results.CompiledAssembly;
            var assemblyTypes = assembly.GetTypes();

            foreach (Type type in assemblyTypes)
            {
                _types.Add(type);
            }

            return errors;
        }

        public List<ValidationResult> GetValidationResults()
        {
            var validators = GetValidators();

            var results = new List<ValidationResult>();

            if(_types.Count == 0)
            {
                return results;
            }

            foreach(Type type in _types)
            {
```

```
                foreach (IClassNormalFormValidator validator in
validators)
                {

results.AddRange(validator.GetValidationResults(type));
                }
            }

            return results;
        }

        private List<IClassNormalFormValidator> GetValidators()
        {
            var validators = new List<IClassNormalFormValidator>();

            validators.Add(new FirstClassNormalFormValidator());

            return validators;
        }
    }
}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace ClassNormalizationRules
{
    public class FirstClassNormalFormValidator :
IClassNormalFormValidator
    {
        private Type _type;
        private List<ValidationResult> _results;

        public List<ValidationResult> GetValidationResults(Type type)
        {
            _type = type;
            _results = new List<ValidationResult>();

            ValidateIdentifier();
            ValidateSingleValue();
            ValidateRepeatingGroups();

            return _results;
        }

        private void ValidateIdentifier()
        {
            PropertyInfo[] properties = _type.GetProperties();
            bool idFound = false;

            foreach(PropertyInfo property in properties)
            {
```

```
                // try to find a property with {ClassName}ID, ignpring
case
                if(string.Compare(property.Name, _type.Name + "id",
true) == 0)
                {
                    idFound = true;
                }
            }

            if (!idFound)
            {
                _results.Add(new ValidationResult {
                    Violation = "(" + _type.Name + ".cs) Violation of
1CNF: No two object can have identical values.",
                    HowToFix =
@"Please add an identifier property to your class in the format
{ClassName}ID and make
sure a unique value is always assigned to this property before saving
an instance of this class to the database.",
                    Level = ValidationLevel.Error
                });
            }
        }

        private void ValidateSingleValue()
        {
            PropertyInfo[] properties = _type.GetProperties();

            foreach (PropertyInfo property in properties)
            {
                if (property.PropertyType.IsArray)
                {
                    Type elementType =
property.PropertyType.GetElementType();

                    if(elementType.IsPrimitive ||
elementType.Name.ToLower() == "string")
                    {
                        _results.Add(new ValidationResult
                        {
                            Violation =
"(" + _type.Name + @".cs) Violation of 1CNF: Each property must contain
a single value unless
it is a collection of a type that is part of the entity model.",
                            HowToFix = string.Format(
@"The property type of the property '{0}' is an array of either a
primitive
type or a string. To fix it, please create a new class that for the
property '{0}'
to store the values for '{0}' and link it back to the class '{1}'",
property.Name, _type.Name),
                            Level = ValidationLevel.Error
                        });
                    }
                }
            }
        }
```

```
        private void ValidateRepeatingGroups()
        {
            PropertyInfo[] properties = _type.GetProperties();
            List<string> numberedProperties = new List<string>();

            string numberedProperty = null;

            foreach (PropertyInfo property in properties)
            {
                if (property.Name.EndsWith("1"))
                {
                    numberedProperty = property.Name.Substring(0,
property.Name.Length - 1);
                    break;
                }
            }

            if(numberedProperty != null)
            {
                foreach (PropertyInfo property in properties)
                {
                    string prop = property.Name.Substring(0,
property.Name.Length - 1);

                    if (prop == numberedProperty)
                    {
                        numberedProperties.Add(property.Name);
                    }
                }
            }

            if (numberedProperties.Count > 1)
            {
                string numberedPropertiesString = string.Join(", ",
numberedProperties.ToArray());

                _results.Add(new ValidationResult
                {
                    Violation = "(" + _type.Name + ".cs) Violation of
1CNF: Properties cannot contain repeating groups.",
                    HowToFix = string.Format(
@"The properties '{0}' are considered repeating groups.
Please give each of those properties some contextual names.",
numberedPropertiesString),
                    Level = ValidationLevel.Error
                });
            }
        }
    }
}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace ClassNormalizationRules
{
    interface IClassNormalFormValidator
    {
        /// <summary>
        /// Validates all rules related to this class normal form
        /// </summary>
        /// <returns>A list of error/</returns>
        List<ValidationResult> GetValidationResults(Type type);
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ClassNormalizationRules
{
    public enum ValidationLevel
    {
        Error,
        Waning
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ClassNormalizationRules
{
    public class ValidationResult
    {
        public string Violation { get; set; }

        public string HowToFix { get; set; }

        public ValidationLevel Level { get; set; }
    }
}
```

VITA

Daniel Sudhindaran has Bachelor of Science degree from the University of North Florida in Computer Science, 2007 and expects to receive a Master of Science in Computer and Information Sciences from the University of North Florida, April 2017. Dr. Robert Roggio of the University of North Florida is serving as Daniel's thesis advisor. Daniel is currently employed as a Product Architect at Beeline and has been with the company for ten years.

Daniel has an on-going interest in innovation and startups. Daniel co-founded two companies – TradeSumo and Zwytch. TradeSumo is an online bartering marketplace where people can swap their unused stuff for free. Zwytch is an online car buying platform that helps users chat real-time with multiple dealers around them and negotiate a complete car deal before going to the dealership. Daniel has extensive programming experience in C#, .NET, JavaScript, SQL, HTML and CSS. Daniel's academic work has included the use of Java and C programming languages as well.