MET cS665 Summer 2 2018

Abel Gancsos

Abel Gancsos productions | Hawthorne, NJ

AMGPetAgency

2018

Contents

Introduction 2

1 Application Description 3

1.1 Implementation Details 3

1.2 Assumptions 3

1.3 Prerequisites 3

2 Assignment Tasks 4

2.1 Design Goals 4

2.2 Implementation Description 4

2.3 UML Class Diagram 6

2.4 UML Sequence Diagram 7

Bibliography 8

Appendices 9

Appendix A – Setup 9

Appendix B – Building project 9

Appendix C – Running executable 10

# Introduction

With more and more pets being abandoned or removed from an abusive household, leading to more homeless animals, there is a higher need find adoptive parents for these pets. AMGPetAgency thus so happens to be one of these organizations that strives to place animals with the most appropriate adoptive parents based on personality traits, work schedule, family size, and overall lifestyle. However, because of the recent increase in demand, AMGPetAgency’s paper-copy approach isn’t working too well, leading to data loss, ranging from the very minor details to losing documentation. In all honesty, this issue only started recently and the president is looking for an immediate solution.

The president met up with a software developer friend and they came up with a suggestion to build a record management system in order to know which animal is placed in which home as well as which animals are still available for adoption. The end result would be a custom application that can be accessed only on the internal network from any machine and would be scalable to handle the large number of animals needed to be adopted. In order to get a better understanding of the system details, the president requested the development team to build a prototype that demonstrates the overall features that will be available in the initial release.

# 1 Application Description

## 1.1 Implementation Details

*The following details are also available in the README.md file*

This utility, AMGPetAgency, is a prototype system controller that helps keep pet adoption records safe and organized in one central location. The current prototype uses an in-memory database (collections) to store the records and is able to demonstrate the bulk of the functionality of the final product. The controller uses a command-line input mechanism to prompt the user to required details to construct a new person to add to the system or a new pet that is available for adoption. The controller then uses the array indices for the records when prompting the user to adopt a new pet or to return a pet that can no longer be kept in the current home.

During the prompt to adopt, the controller will display a list of available pets to adopt, will validate the input from the user, and will also display a list of all people that are able to adopt a pet. During the return process, the controller will ask for a confirmation if the user really wants to return the pet and will then display a list of adopted pets. If the user input is not valid, either because it’s not in a numeric form or is out of scope of the collection, a message will be displayed and/or the error will be corrected by reapplying a default value.

In the event that the prototype does become a CLI (command-line interface) for the final product, additional feature may be added, such as searching for a specific pet or a specific person. The key aspect of the controller is being able to smoothly build a complete object such as a dog or person since the rest of the operations are just input/output.

## 1.2 Assumptions

* The animal breeds have already been breaded and are available for appropriate animals.
* Animal types are predefined and no new animal can be created via combinations.
* A person can only be either a male, female, or not specified.
* The measurement used for height is in inches.
* The measurement used for weight is pounds.
* Additional nationalities will be added in the future.
* Additional races will be added in the future.
* Additional animal breeds will be added in the future.
* Additional animal types will be added in the future.
* Additional properties will be added and be configurable to people in the future.
* Additional properties will be added and be configurable to animals in the future.

## 1.3 Prerequisites

* C++ compiler with C++11 capability
* CMake version 3.11 or higher
* XCode command-line tools (MacOS only)
* Visual Studio 2017 or higher (Windows only)

# 2 Assignment Tasks

## 2.1 Design Goals

*The following details are also available in the README.md file*

At a high level, whichever design pattern selected to implement the utility, should be able to take user input from the command-line level and produce the complex types of an AMGAnimal or AMGPerson object. Complexity here is referring to the number of properties that an object could contain, which means that the design pattern must be maintainable to handle any attributes that would be added to the complex types. The design pattern should also be able to handle different derivatives of the complex types, for example, it should not only be able to build AMGDog types, but should also be able to build AMGCat types if added to the infrastructure. The components should be portable and able to be applied to other utilities.

## 2.2 Implementation Description

*The following details are also available in the README.md file*

Although not as efficient as other options, the Builder design pattern was primarily used to implement the controller needed for the utility because of its mechanism to construct complex objects. This mechanism that the Builder pattern leverages to construct complex types uses three main components to build or construct the pieces of the object in additional to the final class instance, known as the Product. First, there’s the Director object, which traditionally is the actual object that creates the instance of the object with it’s specific attribute values. In terms of AMGPetAgency, there are two types of products, AMGAnimal and AMGPerson, that the design pattern attempts to build, so two different Directors, AMGAnimalDirector and AMGPersonDirector, are required. Based on the research that was done, it would be possible to build an abstract Product via an Object object (Java) or a void\* object (C++). However, because of the abstraction, it would require additional steps in order to display the information. In addition, each Director object should be able to produce an infinite number of variations of the same Product, so an interface or an abstract object, AMGAnimalBuilder /AMGPersonBuilder, is required for the Director to work directly with. And finally, the object that has the actual configuration must implement that abstract methods as well as set the specific values for the attributes. While researching, this concrete object is typically prebuilt with the attribute values without user input; however, because this utility is strongly driven by user-input to build the complex types, we must first configure the builder and then build the instance.

One example of why the Builder design pattern may not have been the most efficient to this controller is because the Builder pattern typically has a somewhat prebuilt structure that the Director component of the pattern uses to construct the Product. However, this utility uses user input to build the Builder, depending on the type of the Product was selected, for example the AMGDogBuilder is used to build the AMGDog object, and then the AMGAnimalDirector would call the Build() method of the Builder, to return the instance. Another example of how the Builder pattern may not have been the most efficient is that it could lead to duplicated code. For example, because the controller uses user input/output to set the attributes of the Product through the Builder, getters and setters would be required in multiple classes, not just the AMGDog object. Instead, the utility uses setters in the Builder, which then calls the setters in the AMGDog class and then the getters from the AMGDog class are called to display the information. Although this might not seem exactly like the typical Builder pattern, it can be argued that it is because the Director object is still the object that returns the final instance. This means that the instance can still be manipulated or additional properties can be built before the applications accepts the instance of the product.

So what would have been better alternatives when implementing the AMGPetAgency controller? Based on research, one considered alternative that could have been used in place of the Builder pattern is the Decorator, where instead of building the instance of the complex type by parts, the base product would be created and the Decorator pattern components would be used to enhance the product. The main reason why this design pattern wasn’t used is that it would require a larger amount of small classes, but those smaller classes could be applied to many product types.

As for how flexible the implementation is because of the design chosen, because the builder components were designed to be very much moduler, new builders could be added by just implementing the builder class itself. Another example of how flexible the implementation is, is that new objects (products) can be added by just adding the class to the AMGPetAgencyCommon package, for example, to add an AMGCat product, just add the class to the package. A third example of how flexible the implementation is, is that in order to add a new product type (AMGFood), the base and concrete classes must be added to the AMGPetAgencyCommon package and a builder package must be added to the AMGPetAgencyBuilder container.

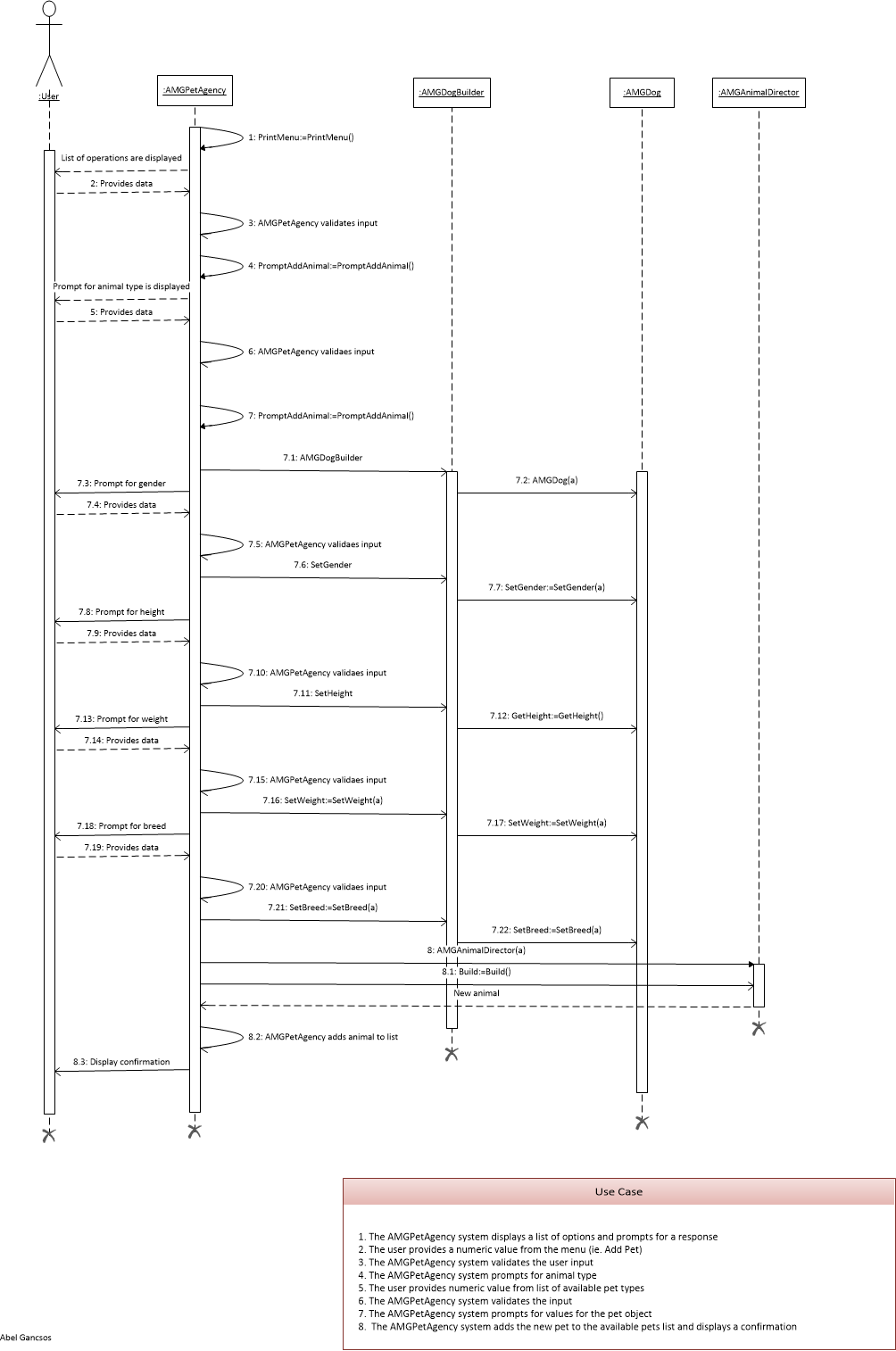
In terms of simplicity, the implementation is broken down into 2 moduler packages, AMGAnimalBuilder and AMGPersonBuilder. This is in order organizes the classes that are specific to certain product types. The packages are then further broken down into what’s currently considered the basic structures to construct the complex product types. There’s the concrete builders, for example the AMGDogBuilder, which contains the values specific for AMGDog products and then the AMGAnimalDirector builds, or in this case produces, the final product. As for the structure of the implementation should be very understandable because the objects are organized in a logical way. First, there are common objects or types that can be used throughout the entire system and they are organized in the AMGPetAgencyCommon package. Then there’s the main system classes that are kept in the AMGPetAgency package, allowing a place to further break down the main system as well as it’s helpers into smaller pieces. Then there’s the main AMGPetAgencyBuilder container, which is used as a section of the implementation to hold all the builder modules. The rest of the system is heavily dependent on user interaction, which is documented as best as possible. Additional documentation might be provided as the CLI becomes more integrated with the final system.

In regards to avoiding duplicate code, although some of the user interaction code required duplicated code, other instances where duplicated code would normally occur was avoided by modifying the traditional Builder design objects to only have the code required for the Builder mechanism. For example, because the AMGDogBuilder and AMGCustomerBuilder objects had the actual AMGDog or AMGPerson object as an attribute, the builders only required setter methods, not getters as the Builder design pattern would typically require. Another example of how duplicate code was avoided is that simple, non-interaction blocks were implemented in common classes. For example, the user input is validated using the AMGSystem::ValidateInput() method.

## 2.3 UML Class Diagram



## 2.4 UML Sequence Diagram



# Bibliography

Döring, A. (2016, 11 07). *minimal\_cmake\_example*. Retrieved 07 03, 2018, from GitHub Web site: https://github.com/krux02/minimal\_cmake\_example

Geeks for Geeks Organization. (n.d.). *Builder Design Pattern* . Retrieved from Geeks for Geeks Web site: https://www.geeksforgeeks.org/builder-design-pattern/

Google Corporation. (2018, 06 19). *Google C++ Style Guide*. Retrieved 07 03, 2018, from Google GitHub styles: https://google.github.io/styleguide/cppguide.html

Kitware Corporation. (n.d.). *Download*. Retrieved 07 03, 2018, from CMake Web site: https://cmake.org/download/

Lucid Software Corporation. (2017). *Class Diagram Tutorial*. Retrieved 7 25, 2017, from Lucid Software Corporation Web site: https://www.lucidchart.com/pages/uml/class-diagram

Microsoft Corporation. (2010). Visio. (2010). Redmond, WA, USA.

Moser, M. (2018, 06 28). *Patterns*. Retrieved 07 04, 2018, from Java Patterns Web site: http://java-design-patterns.com/patterns/builder/

Oracle Corporation. (2017). VirtualBox. (5.1.18). Redwood City, CA, USA.

Pazdera, R. (2011). *Example of `builder' design pattern in C++*. Retrieved from GitHub Web site: https://gist.github.com/pazdera/1121152

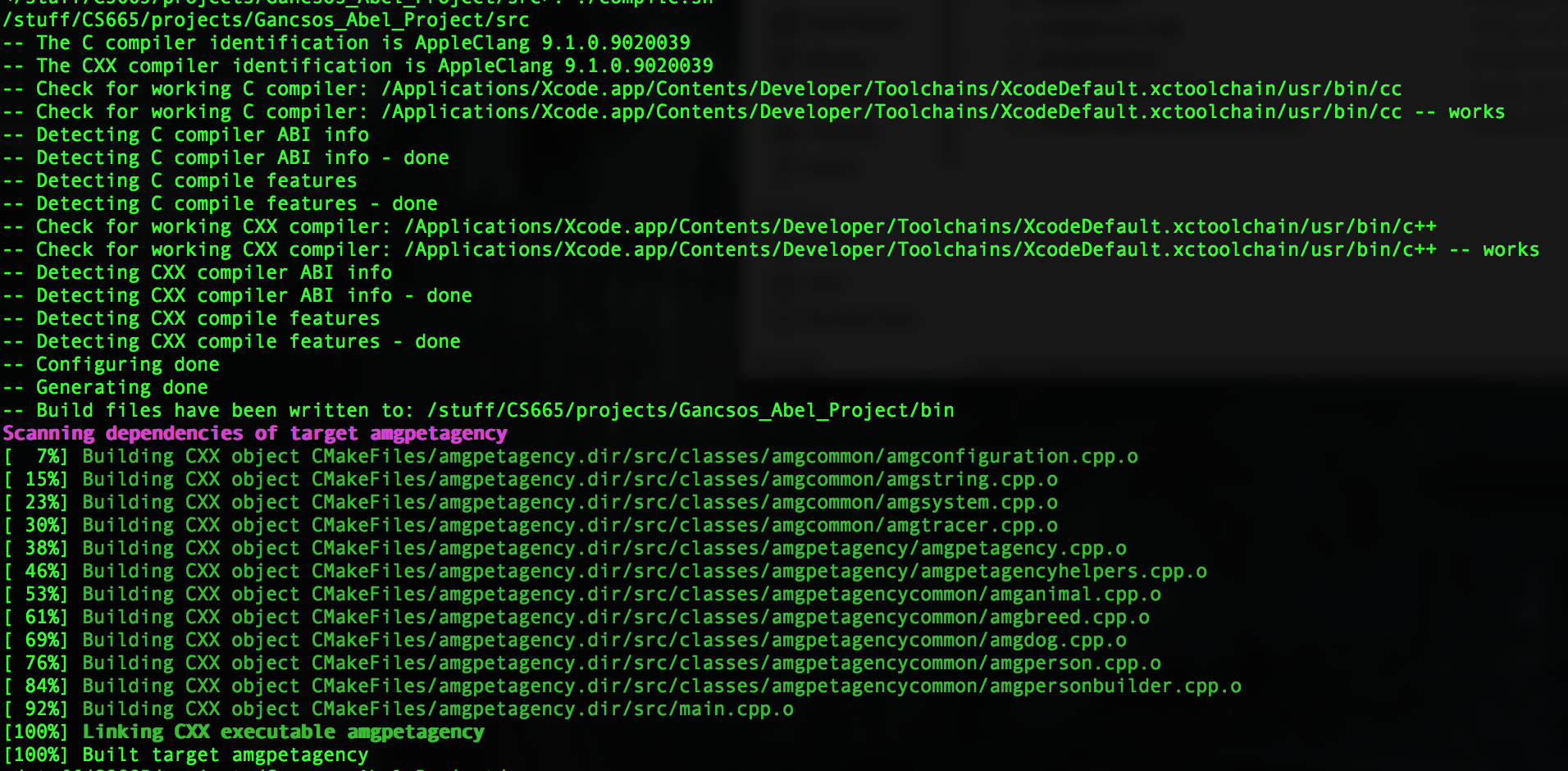
Tutorials Point. (n.d.). *Design Patterns Builder Pattern*. Retrieved 08 01, 2018, from Tutorials Point Web site: https://www.tutorialspoint.com/design\_pattern/builder\_pattern.htm

# Appendices

## Appendix A – Setup

The project can be build using the make.sh or make.bat scripts, depending on the platform. Both of these scripts will run cmake to generate the CMakeFiles and then run the make command in the bin directory. Alternatively, the project can be built manually using the CMakeLists.txt file that comes with the package.

## Appendix B – Building project



## Appendix C – Running executable

