Sneha Kanaujia

Software Design Final Paper

05/02/19

Creating A Magic Starter Pack Generator using the Builder Pattern

The Builder Pattern

In this software design final project I implemented the Builder Pattern in an application that allows a user to produce descriptions of magic items one might receive in a magic starter pack if they were playing a witch or wizard character in a role-playing game (RPG). The application allows the user to click buttons that randomly produce a structured description of either a magic wand, magic potion or magic artifact depending on the button(s) clicked. The Builder Pattern is meant to encapsulate the construction of a complex object and break it down into several parts. This allows for the object to be constructed in a variety of ways, which is one of the most useful attributes of the Builder Pattern, and keeps the product's construction process a bit more discrete and behind the scenes.

The structure of the Builder Pattern includes a Director, a Builder interface, a ConcreteBuilder, and a Product, displayed in Figure 1 below. The Product is the complex object under construction. The Builder is an abstract interface that is responsible for outlining the parts of the Product object. The ConcreteBuilder is responsible for constructing and assembling the parts of the Product in specific sections by implementing the Builder interface and its BuildPart() methods, along with defining and maintaining the Product representation created, and making sure the Product is retrievable. The Director constructs the Product by calling the BuildPart() methods of the ConcreteBuilder in the correct order and is able to retrieve the finalized Product representation through the ConcreteBuilder's GetResult() method..

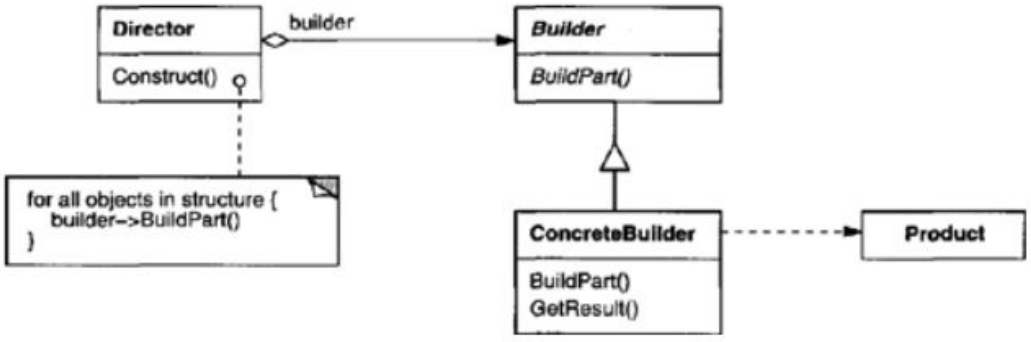

Figure 1: Builder Pattern UML [1, page 98].

Typically, the Builder Pattern is employed by a Client class which contains an instance of the Director and configures its creation by passing in a specific Builder object. This Director then calls the BuildPart() methods of the given Builder, a ConcreteBuilder, in a specific order. The ConcreteBuilder then constructs the Product from the various BuildPart() calls and is able to return a fully constructed Product to the Client either from a getter method in the Director class or from the ConcreteBuilder itself. These interactions between objects/classes can seen in Figure 2 below. It's also significant to note the Product could be anything from a single class to a combination of interfaces, classes and so on. They don't typically have a single interface or abstract class since different ConcreteBuilders are capable and often meant to construct Products that differ largely in representation and thus no common parent class would be of use [1, page 101].

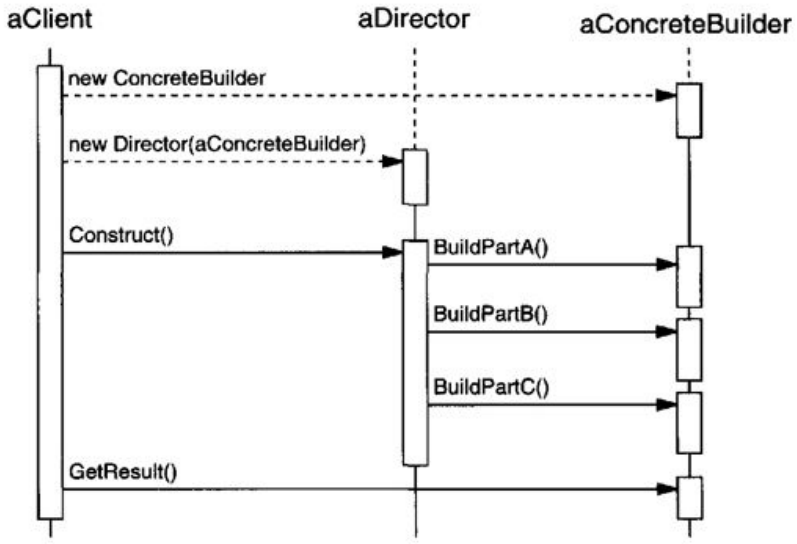

Figure 2: Diagram of the interactions between the Client, Director and ConcreteBuilder [1, page 99].

The Builder interface allows the Builder Pattern to have greater reusability of code, and allows for differentiation between Products as each ConcreteBuilder "contains all the code to create and assemble a particular kind of product" and "different Directors can reuse it to build Product variants from the same set of parts" [1, page 100]. The abstract interface also allows the Builder to hide the representation and process of constructing the Product. Thus to change this one simply has to "define a new kind of builder" [1, page 100]. The Builder Pattern also allows for increased modularity as the Client remains unaware of the classes and mechanisms responsible for the construction and representation of the Product due to the encapsulation of such code and the fact that these classes and mechanisms are not present in the Builder interface. This offers a finer control over the Product's construction process where

the Director dictates the construction of the Product step by step and can retrieve it from the Builder once the product is fully completed [1, page 100].

App Implementation of Builder Pattern

The MagicStarterPack app instructs the user to click any of the three buttons, 'Summon your wand!', 'Brew your potion!', and 'Conjure your magic artifact!' to produce descriptions of different magic items: a magic wand, a magic potion, and a magic artifact, respectively. If the user wants a different item they can click the button(s) again to produce a new magic item in accordance to the button(s) clicked. The general build of each magic item description is an introduction, a description of its main visual characteristics, and additional details about the item like its history, contents or additional characteristics.

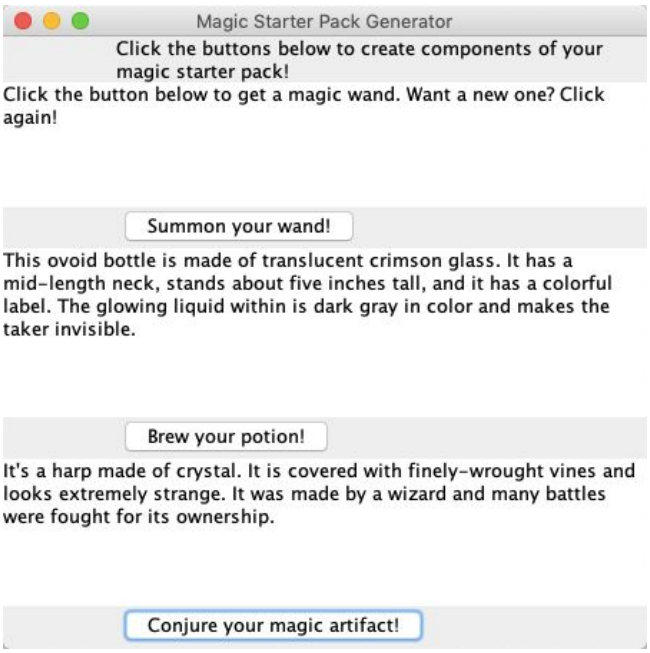

Figure 3: Application appearance after pressing the 'Brew your potion!' and 'Conjure your magic artifact!' buttons.

The MagicStarterPackApp class represents the Client of the program and works as the app's GUI. It also contains the main method, which allows it to call the right methods to create the GUI, and has an instance of a MagicStarterPackDirector. The MagicStarterPackDirector takes in a concrete implementation of the MagicItemBuilder interface so the correct magic item Product can be constructed and provided to the Client. There are three ConcreteBuilders, the MagicWandBuilder, the MagicPotionBuilder, and the MagicArtifactBuilder, each of which implements the MagicItemBuilder interface and thus its Product getter method getMagicItem() and its BuildPart() methods: buildIntroduction(), buildCharacteristicDescrip(), and buildDetails().

MagicWandBuilder	MagicPotionBuilder	MagicArtifactBuilder
The wand is made of [WOOD TYPE] and has a core of [MAGIC ANIMAL FUR/FEATHER/HAIR]. It is [LENGTH] inches long and is [ADJECTIVE]. It is [COLOR] and is [ADVERB] carved.	This [SHAPE] bottle is made of [GLASS ADJECTIVE] [COLOR ADJECTIVE] glass. It has a [SIZE] neck, stands about [HEIGHT] inches tall, and it has [A SIMPLE OR NO] printed label. This [ADJECTIVE] liquid is [COLOR] in color and [AFFECT].	It's a [NOUN] made of [MATERIAL]. It's [CHARACTERISTIC DESCRIPTION] and looks [CHARACTERISTIC DESCRIPTION]. It [BACKSTORY/ABILITY].

Figure 4: The description structures of each concrete implementation of the MagicItemBuilder

Each sentence in the descriptions of Figure 4 were created by a specific BuildPart() method. The buildIntroduction() method in the MagicPotionBuilder is responsible for forming the first sentence of the magic potion description “This [SHAPE] bottle is made of [GLASS ADJECTIVE] [COLOR ADJECTIVE] glass.” Then the buildCharacteristicDescrip(), and buildDetails() methods are called to add the “It has a [SIZE] neck, stands about [HEIGHT] inches tall, and it has [A SIMPLE OR NO] printed label.” sentence and the “This [ADJECTIVE] liquid is [COLOR] in color and [AFFECT].” sentence. Each bracketed word is meant to be filled in by a String randomly selected from a corresponding String array. Such as the the shape, glassAdj, bottleColor, neckLength, heights, labelDetail, liquidAdj, liquidColor, and affect ‘word bank’ arrays in the MagicPotionBuilder class. These ‘word bank’ String arrays are specific to each ConcreteBuilder and were inspired by online websites that are specified in each class. The Director’s construct method is responsible for calling the builders’ BuildPart() methods in the correct order to properly construct the appropriate Product based on what ConcreteBuilder that’s passed in, which depends on what button the user pressed. The completed magic item or Product is then obtained by the Director called the builder’s getter method which returns its String representation.

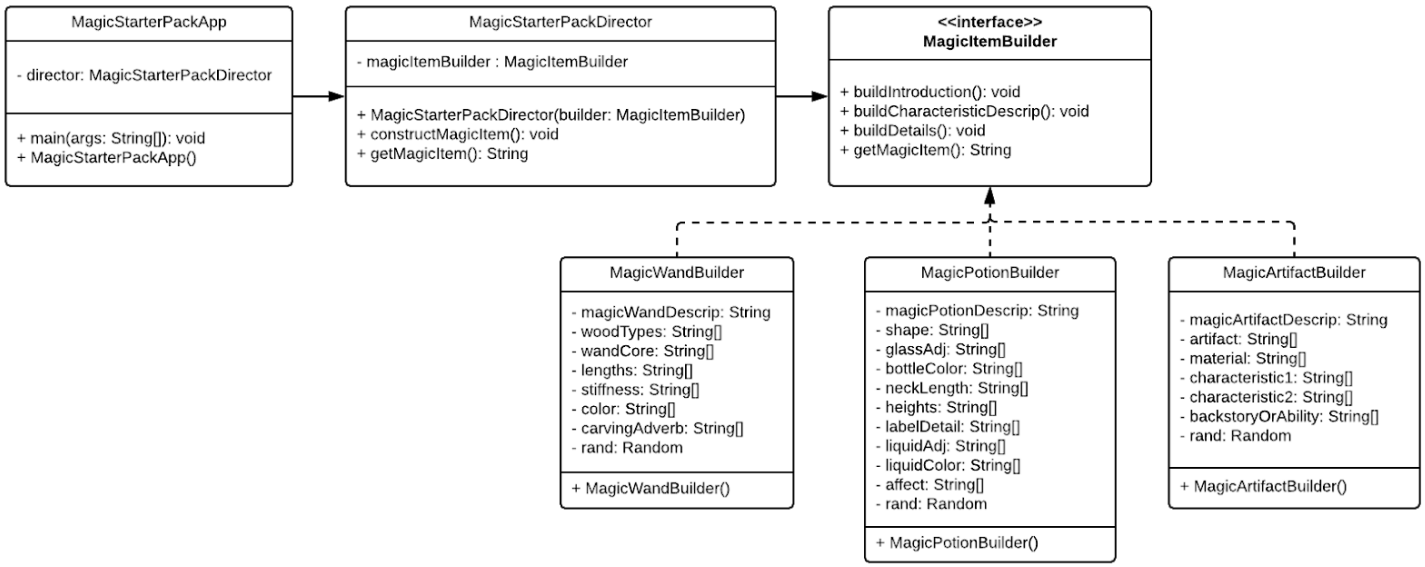

Figure 5: UML of the Magic Starter Pack Generator application that implements the Builder Pattern

Conclusion

I learned a great deal about how the Builder Pattern works as well as its benefits and shortcomings and how to correctly implement it in my own application. I had a lot of fun with the creative description crafting portion of the app and was thoughtful about how it could be applicable to real world situation such as a game. The Magic Starter Pack Generator application effectively implements the Builder Pattern and is capable of creating slightly structured and randomized magic items for a user to obtain, possibly as the initial gear of a hypothetical witch/wizard RPG game.

Acknowledgements

Thank you so much Barbara for an excellent semester and experience in Software Design! I really admire the level to which you were always prepared and put a lot of engagement techniques into practice during class. The effort you put in to develop the curriculum and teach it in the most effective manner was quite apparent. Even the assignments you gave were all different and allowed students to learn topics in new and varied ways. I had a lot of fun in this class and I'm really glad to have met you in this department! Thanks again and I hope you have a lovely summer!

Bibliography

[1] Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. 2014. *Head First design patterns: a brain-friendly guide* (10th. ed.). O'Reilly, Sebastopol, CA.

This design book provided a brief section explaining the Builder Pattern, it's basic principles and the way it functions, as well as an example of how it may be implemented. The example program shows how the Builder Pattern was applied in a "VacationPlanner" program.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA.

This design book provided both basic and advanced explanation and analysis of the Builder Pattern, it's pros and cons, as well as an example of its implementation in a Maze program. The diagram in this book depict a UML of the Builder Pattern and the interaction between the Client, Director and ConcreteBuilder.