Chapter 6

Builder

For those people who prefer building their own houses, they would outsource the project to a contractor. A single contractor doesn't build a whole house for you; he breaks it up into several parts and subcontracts to actual builders who know how to put things together with actual components and parts. A house is composed of parts in different styles, colors, and dimensions. The client tells the contractor what he or she wants in the house. Then the contractor coordinates with house builders about *what* needs to be done. And they build it based on *how* it should be built. Building a house is a complex process. It's very difficult if not impossible to build it with only one pair of hands. The process is a lot easier and more manageable if a contractor (director) coordinates with builders who know how to build it.

Sometimes there are many different ways to build certain objects. The logic of building them can go very crazy if the logic is contained in a single method of a class that builds them (for example, a whole forest of nested `if-else` or `switch-case` statements for different building requirements). If we can break the building process down into client-director-builder relationships, then the process will be more manageable and reusable. The design pattern for these kinds of relationships is called Builder.

In this chapter, we are going to discuss the concepts of the Builder pattern. Later in other sections, we'll also discuss how to use the pattern to create different characters with complex traits in a RPG game.

What Is the Builder Pattern?

Besides a client and the product that it expects, the Builder pattern contains two key roles: `Director` and `Builder`. A `Builder` knows exactly *how* to build the product with some missing information that is specific to particular products (the *what*). The `Director` knows *what* the `Builder` should make by providing it some missing information as parameters in order to build specific products. The *what* and *how* are a little confusing. Even though the `Director` knows what the `Builder` should make, it doesn't mean that the `Director` knows exactly what the concrete `Builder` is. Their static relationships are illustrated in Figure 6–1 as a class diagram.

C. Chung, *Pro Objective-C Design Patterns for iOS*

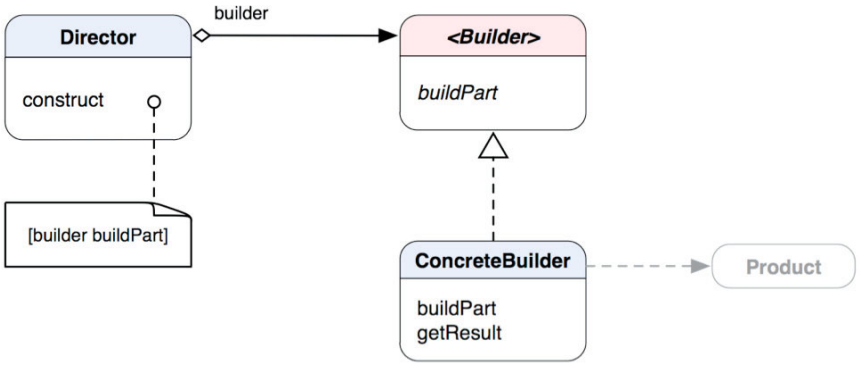

Figure 6–1. *A class diagram of the Builder pattern*

`Builder` is an abstract interface that declares a `buildPart` method, which is implemented by `ConcreteBuilder` to build actual `Product`. `ConcreteBuilder` has a `getResult` method that returns a completely built `Product` to a client. `Director` defines a `construct` method that tells an instance of `Builder` to `buildPart`. `Director` and `Builder` form an aggregation relationship. It means that `Builder` is an integral part that incorporates with `Director` to make the whole pattern work, but at the same time, the `Director` is not responsible for the lifetime of the `Builder`. This whole-part relationship can be made clearer with the sequence diagram in Figure 6–2.

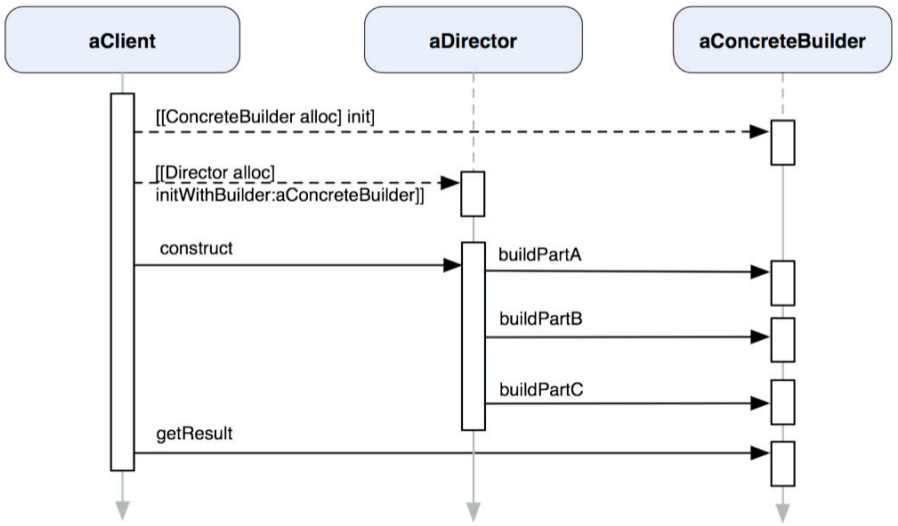

Figure 6–2. *A sequence diagram of the interactions among `aClient`, `aDirector`, and `aConcreteBuilder` at runtime*

`aClient` creates an instance of `ConcreteBuilder` (`aConcreteBuilder`) as well as an instance of `Director` (`aDirector`) with `aConcreteBuilder` as an initialization parameter so they can work together later on. When `aClient` sends a `construct` message to `aDirector`, the method sends messages to `aConcreteBuilder` about what to build (e.g.,

buildPartA, buildPartB, and buildPartC). After the construct method of aDirector returns, aClient sends a getResult to aConcreteBuilder directly to retrieve the completely built product it expected. So the "what" that aDirector knows about is what parts any Builder should be able to build.

THE BUILDER PATTERN: Separates the construction of a complex object from its representation so that the same construction process can create different representations.*

* The original definition appeared in *Design Patterns*, by the "Gang of Four" (Addison-Wesley, 1994).

When Would You Use the Builder Pattern?

You'd naturally think about using this pattern when

- You need to create a complex object that involves different parts. The algorithm for creating it should be independent of how the parts are assembled. A common example is building a composite object.
- You need a construction process that constructs an object in different ways (e.g., different combinations of parts or representations).

BUILDER VS. ABSTRACT FACTORY

We discussed Abstract Factory in the last chapter. You might have realized that both the Abstract Factory and Builder patterns are similar in many ways in terms of being used for abstract object creation. However, they are very different. Builder focuses on constructing a complex object step-by-step, when a lot of times the same type of object can be constructed in different ways. On the other hand, Abstract Factory's emphasis is on creating suites of products that can be either simple or complex. A builder returns a product as a final step of a multiple-step construction process, but the product gets returned immediately from an abstract factory. The following table summarizes the main differences between the Builder and Abstract Factory patterns.

Builder	Abstract Factory
Constructs complex objects	Constructs either simple or complex objects
Constructs an object in multiple steps	Constructs an object in one step
Constructs an object in many ways	Constructs an object in one way
Returns a product object as a final step of a construction process	Returns a product object immediately
Focuses on one particular product	Emphasizes a suite of products

In Figure 6–2, `aClient` needs to know both `aDirector` and `aBuilder` to get the product it wanted from `aBuilder`. You may wonder whether `aBuilder` can be separated from `aClient`'s knowledge if we let `aDirector` return the product from its `construct` method or somehow the `getResult` method is implemented in `aDirector`. In that case, `aDirector` becomes a factory and its `construct` method becomes a factory method that returns an abstract product. Also, `aDirector` will be fixed with what products it supports, which hinders the reusability of the pattern. The whole idea is to separate "what" from "how," so `aDirector` can apply the same "what" (specifications) to a *different* `aBuilder` that knows "how" to build its own *specific* product with the provided specifications and vice versa.

In the following sections, we are going to use an example of building a chasing game that has different types of characters to illustrate how to use the Builder pattern to solve related design problems. The construction process for a game that involves different types of objects, assets, or characters can be quite complex. We can separate the algorithm that knows how to build a character from what characters to build with the Builder pattern.

Building Characters in a Chasing Game

We are going to use an imaginary chasing game as an example to show how to implement the Builder pattern. Suppose there are two types of characters, enemy and player. The enemy will be chasing after the player. You will decide where the player character should go. There might be obstacles along the path. Each of them shares some fundamental traits, such as Strength, Stamina, Intelligence, Agility, and Aggressiveness. Each of them can affect the character's ability of Protection and Power. A Protection factor reflects how much a character can protect himself from an attack, while a Power factor indicates how much he is capable of attacking his opponents. Traits are either proportionally or inversely proportionally related to both the Protection and Power factors. A matrix that shows their relationships is illustrated in Table 6–1.

Table 6–1. *Character Traits Matrix.*

	Protection	Power
Strength	↑	↑
Stamina	↑	↑
Intelligence	↑	↓
Agility	↑	↓
Aggressiveness	↓	↑

↑ denotes directly proportional, whereas ↓ denotes inversely proportional between two different character traits.

Strength and Stamina are directly proportional to Protection and Power. A character that has higher Strength and Stamina factors has a better chance of protecting himself and fighting back. Intelligence and Agility are directly proportional to the Protection factor but inversely proportional to Power. Based on our design, if a character is smarter, then he has more ability only to protect himself, not to fight back. Aggressiveness is the reverse of both Intelligence and Agility. If a character is aggressive, then he will have a higher chance of attacking but not of protecting himself during an attack. Of course, the design for the traits of characters is purely fictitious. You can have a totally different design of your own. So we are going to stick to the relationships shown in the matrix for now and use the Builder pattern to build characters for us with the different combinations for character traits.

Food For Thought

How are you going to design a class that creates a character with a combination of traits based on the matrix shown in Table 6–1 without using the Builder pattern?

We will define a class called `ChasingGame` that has two methods to create two types of characters, player and enemy. A `CharacterBuilder` builds characters based on the relationships of different traits as shown in the previous matrix. Each trait factor (value) affects the characteristics of a character being built. A class diagram that shows their static relationships is shown in Figure 6–3.

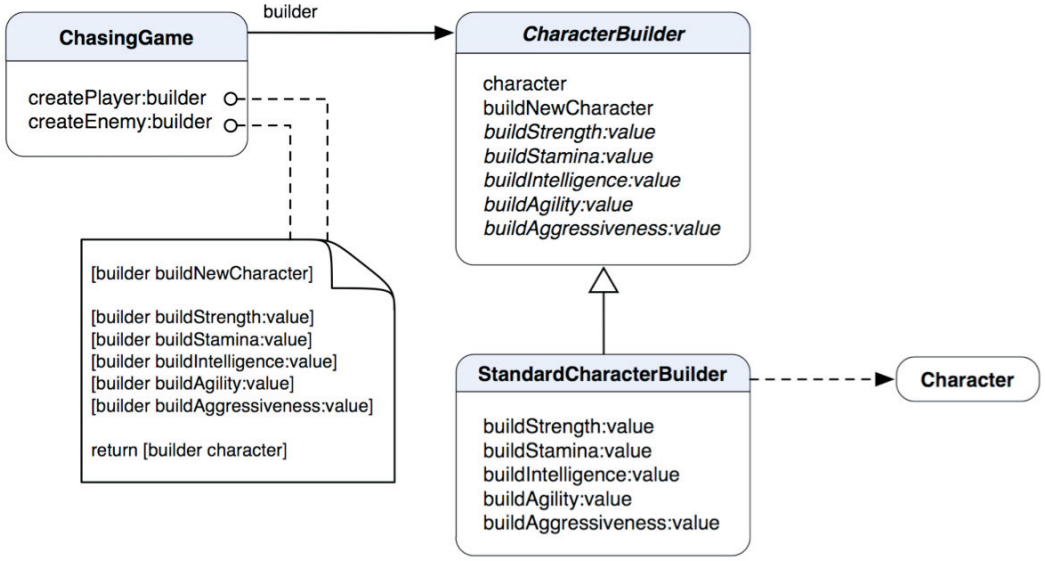

Figure 6–3. *A class diagram of `CharacterBuilder` as an abstract builder, `StandardCharacterBuilder` as a concrete builder, and `ChasingGame` as a director*

`ChasingGame` defines `createPlayer:builder` and `createEnemy:builder` to build player and enemy characters with an instance of `CharacterBuilder`. Each of the methods has a

different set of trait factors that define the characteristics of a character. `StandardCharacterBuilder` is a concrete `CharacterBuilder` that actually builds characters based on different trait factors. When the building process is done, `StandardCharacterBuilder` will return an instance of `Character` as defined in Listing 6–1.

Listing 6–1. *Character.h*

```
@interface Character : NSObject
{
  @private
  float protection_;
  float power_;
  float strength_;
  float stamina_;
  float intelligence_;
  float agility_;
  float aggressiveness_;
}

@property (nonatomic, assign) float protection;
@property (nonatomic, assign) float power;
@property (nonatomic, assign) float strength;
@property (nonatomic, assign) float stamina;
@property (nonatomic, assign) float intelligence;
@property (nonatomic, assign) float agility;
@property (nonatomic, assign) float aggressiveness;

@end
```

`Character` defines a set of traits that are common to any type of character (either a player or an enemy), which includes protection, power, strength, stamina, intelligence, agility, and aggressiveness, just like in the matrix.

An implementation for `Character` is nothing more than just defining an `init` method and some synchronization for the properties, as in Listing 6–2.

Listing 6–2. *Character.m*

```
#import "Character.h"


@implementation Character

@synthesize protection=protection_;
@synthesize power=power_;
@synthesize strength=strength_;
@synthesize stamina=stamina_;
@synthesize intelligence=intelligence_;
@synthesize agility=agility_;
@synthesize aggressiveness=aggressiveness_;

- (id) init
{
  if (self = [super init])
  {
    protection_ = 1.0;
    power_ = 1.0;
```

```
    strength_ = 1.0;
    stamina_ = 1.0;
    intelligence_ = 1.0;
    agility_ = 1.0;
    aggressiveness_ = 1.0;
  }

  return self;
}

@end
```

An instance of `Character` doesn't know how to build itself to create any meaningful characters. So that's where a `CharacterBuilder` comes in the picture—to help build any meaningful characters based on the trait relationships we defined previously. In Figure 6–2, we have an abstract `CharacterBuilder` that defines an interface that any character builder is supposed to have. Its class declaration is shown in Listing 6–3.

Listing 6–3. *CharacterBuilder.h*

```
#import "Character.h"

@interface CharacterBuilder : NSObject
{
  @protected
  Character *character_;
}

@property (nonatomic, readonly) Character *character;

- (CharacterBuilder *) buildNewCharacter;
- (CharacterBuilder *) buildStrength:(float) value;
- (CharacterBuilder *) buildStamina:(float) value;
- (CharacterBuilder *) buildIntelligence:(float) value;
- (CharacterBuilder *) buildAgility:(float) value;
- (CharacterBuilder *) buildAggressiveness:(float) value;

@end
```

An instance of `CharacterBuilder` has a reference to a target `Character` that will be returned to a client after it's built. There are a few methods to build a character with specific values of Strength, Stamina, Intelligence, Agility, and Aggressiveness. Those values affect the factors for both Protection and Power. The abstract `CharacterBuilder` defines default behaviors of setting those values to a target `Character`, as shown in Listing 6–4.

Listing 6–4. *CharacterBuilder.m*

```
#import "CharacterBuilder.h"


@implementation CharacterBuilder

@synthesize character=character_;


- (CharacterBuilder *) buildNewCharacter
```

```
{
  // autorelease the previous character
  // before creating a new one
  [character_ autorelease];
  character_ = [[Character alloc] init];

  return self;
}

- (CharacterBuilder *) buildStrength:(float) value
{
  character_.strength = value;
  return self;
}

- (CharacterBuilder *) buildStamina:(float) value
{
  character_.stamina = value;
  return self;
}

- (CharacterBuilder *) buildIntelligence:(float) value
{
  character_.intelligence = value;
  return self;
}

- (CharacterBuilder *) buildAgility:(float) value
{
  character_.agility = value;
  return self;
}

- (CharacterBuilder *) buildAggressiveness:(float) value
{
  character_.aggressiveness = value;
  return self;
}

- (void) dealloc
{
  [character_ autorelease];
  [super dealloc];
}

@end
```

The `buildNewCharacter` method of `CharacterBuilder` creates a new instance of `Character` to build. Every time the method is called, it will `autorelease` any old character before creating a new one. Using `autorelease` is a safer approach than just using `release` right away because a client may still be using the old character without realizing it's released in the builder. The rest of the methods don't do much that is meaningful to build a character until we define a concrete class for `CharacterBuilder` to do that. `StandardCharacterBuilder` is a subclass of `CharacterBuilder` that defines the logic of creating real characters with different correlated traits. Its class declaration is shown in Listing 6–5 and is not much different from `CharacterBuilder`.

Listing 6–5. *StandardCharacterBuilder.h*

```
#import "CharacterBuilder.h"

@interface StandardCharacterBuilder : CharacterBuilder
{

}

// overriden methods from the abstract CharacterBuilder
- (CharacterBuilder *) buildStrength:(float) value;
- (CharacterBuilder *) buildStamina:(float) value;
- (CharacterBuilder *) buildIntelligence:(float) value;
- (CharacterBuilder *) buildAgility:(float) value;
- (CharacterBuilder *) buildAggressiveness:(float) value;

@end
```

We are re-declaring the overridden methods for clarity. `StandardCharacterBuilder` doesn't override the `buildNewCharacter` method because the default behavior in the base class is sufficient. Let's get to the implementation of `StandardCharacterBuilder` in Listing 6–6 and see how the logic can be implemented to build real characters.

Listing 6–6. *StandardCharacterBuilder.m*

```
#import "StandardCharacterBuilder.h"


@implementation StandardCharacterBuilder

- (CharacterBuilder *) buildStrength:(float) value
{
  // update the protection value of the character
  character_.protection *= value;

  // update the power value of the character
  character_.power *= value;

  // finally set the strength value and return this builder
  return [super buildStrength:value];
}

- (CharacterBuilder *) buildStamina:(float) value
{
  // update the protection value of the character
  character_.protection *= value;

  // update the power value of the character
  character_.power *= value;

  // finally set the strength value and return this builder
  return [super buildStamina:value];
}

- (CharacterBuilder *) buildIntelligence:(float) value
{
  // update the protection value of the character
  character_.protection *= value;
```

```
  // update the power value of the character
  character_.power /= value;

  // finally set the strength value and return this builder
  return [super buildIntelligence:value];
}

- (CharacterBuilder *) buildAgility:(float) value
{
  // update the protection value of the character
  character_.protection *= value;

  // update the power value of the character
  character_.power /= value;

  // finally set the strength value and return this builder
  return [super buildAgility:value];
}

- (CharacterBuilder *) buildAggressiveness:(float) value
{
  // update the protection value of the character
  character_.protection /= value;

  // update the power value of the character
  character_.power *= value;

  // finally set the strength value and return this builder
  return [super buildAggressiveness:value];
}


@end
```

Each of the foregoing methods basically sets the values of Protection and Power for the character being built based on the proportionality defined in the matrix. For example, Intelligence is directly proportional to Protection, and then a new Protection value will be obtained by doing `character_.protection *= value` where `value` is an input value for Intelligence. We are using the Objective-C dot syntax here because characters' states are simple and obvious enough to justify the convenience over any possible confusion with C structs or C++ objects (beware of when you mix any C structs and Objective-C++ in your code). Likewise, Aggressiveness is inversely proportional to Protection, and then a new value for Protection is obtained with the statement `character_.protection /= value`. And finally, it sends a message to `super` to update the target character with new values, and it then returns itself.

Let's move on to `ChasingGame` and see how it can use our `StandardCharacterBuilder` to build different characters. Its class declaration is shown in Listing 6–7.

Listing 6–7. *ChasingGame.h*

```
#import "StandardCharacterBuilder.h"

@interface ChasingGame : NSObject
{

}

- (Character *) createPlayer:(CharacterBuilder *) builder;
- (Character *) createEnemy:(CharacterBuilder *) builder;

@end
```

`ChasingGame` has two methods, `createPlayer:` and `createEnemy:`. Each of them takes an instance of `CharacterBuilder` for building a specific type of character with a predefined set of trait factors. Its implementation is shown in Listing 6–8.

Listing 6–8. *ChasingGame.m*

```
#import "ChasingGame.h"


@implementation ChasingGame

- (Character *) createPlayer:(CharacterBuilder *) builder
{
  [builder buildNewCharacter];
  [builder buildStrength:50.0];
  [builder buildStamina:25.0];
  [builder buildIntelligence:75.0];
  [builder buildAgility:65.0];
  [builder buildAggressiveness:35.0];

  return [builder character];
}

- (Character *) createEnemy:(CharacterBuilder *) builder
{
  [builder buildNewCharacter];
  [builder buildStrength:80.0];
  [builder buildStamina:65.0];
  [builder buildIntelligence:35.0];
  [builder buildAgility:25.0];
  [builder buildAggressiveness:95.0];

  return [builder character];
}

@end
```

An instance of `ChasingGame` builds a player character with Strength 50.0, Stamina 25.0, Intelligence 75.0, Agility 65.0, and Aggressiveness 35.0 in its `createPlayer:` method. In its `createEnemy:` method, an enemy is created the same as a player but with different trait factors. Apparently, an enemy is relatively stronger and more aggressive than a player, while the player is relatively more intelligent but weaker than the enemy.

Since each build* method returns an instance of the current builder, you can group the whole building process in a single statement, like in Listing 6–9.

Listing 6–9. *An Alternative Syntactic Style for Building a Character*

```
[[[[[[builder buildNewCharacter]
              buildStrength:50.0]
              buildStamina:25.0]
              buildIntelligence:75.0]
              buildAgility:65.0]
              buildAggressiveness:35.0];
```

Which one is better? It's just a matter of preferences.

Now we have a pretty good picture of building different characters. Listing 6–10 shows what happens in client code.

Listing 6–10. *Client Code*

```
CharacterBuilder *characterBuilder = [[[StandardCharacterBuilder alloc] init]
                                                                 autorelease];
ChasingGame *game = [[[ChasingGame alloc] init] autorelease];

Character *player = [game createPlayer:characterBuilder];
Character *enemy = [game createEnemy:characterBuilder];

// do something else with the
// player and enemy
```

Our client creates instances of `StandardCharacterBuilder` and `ChasingGame`. Then it sends a couple of messages to `ChasingGame` `createPlayer:` and `createEnemy:` with `characterBuilder`. After the `characterBuilder` has hammered out two characters for us, then we will take the game from there.

Summary

The Builder pattern can help us construct objects that involve different combinations of parts and representations. Without the pattern, the `Director` that knows what is needed to construct objects may end up being a monolithic "god" class, with tons of embedded algorithms for building different representations of the same class. A game that involves characters of different traits should get a good use of the pattern. Instead of defining separate `Directors` for building player and enemy, putting the generalized character building algorithm in a single, concrete `CharacterBuilder` can achieve a much better design.

In the next chapter, we are going to look at a pattern that creates and returns only a single instance of a class.