# Option #1: Common Personality Traits

Scott Miner

Colorado State University – Global Campus

## Abstract

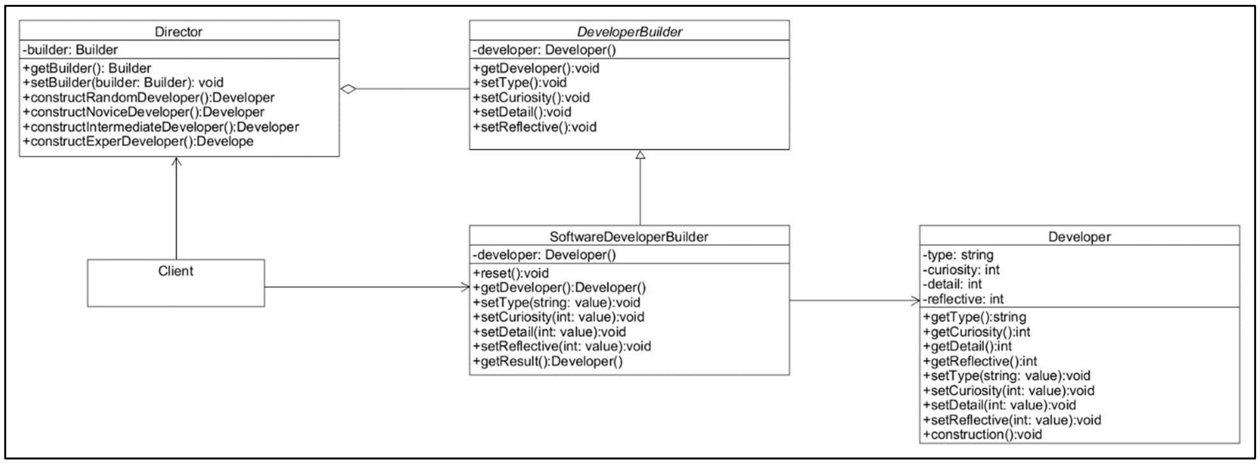

Director
-builder: Builder
+getBuilder(): Builder
+setBuilder(builder: Builder): void
+constructRandomDeveloper():Developer
+constructNoviceDeveloper():Developer
+constructIntermediateDeveloper():Developer
+constructExperDeveloper():Develope
DeveloperBuilder
-developer: Developer()
+getDeveloper():void
+setType():void
+setCuriosity():void
+setDetail():void
+setReflective():void
Client
SoftwareDeveloperBuilder
-developer: Developer()
+reset():void
+getDeveloper():Developer()
+setType(string: value):void
+setCuriosity(int: value):void
+setDetail(int: value):void
+setReflective(int: value):void
+getResult():Developer()
Developer
-type: string
-curiosity: int
-detail: int
-reflective: int
+getType():string
+getCuriosity():int
+getDetail():int
+getReflective():int
+setType(string: value):void
+setCuriosity(int: value):void
+setDetail(int: value):void
+setReflective(int: value):void
+construction():void

*Figure 1.* UML class diagram portraying the developer builder application

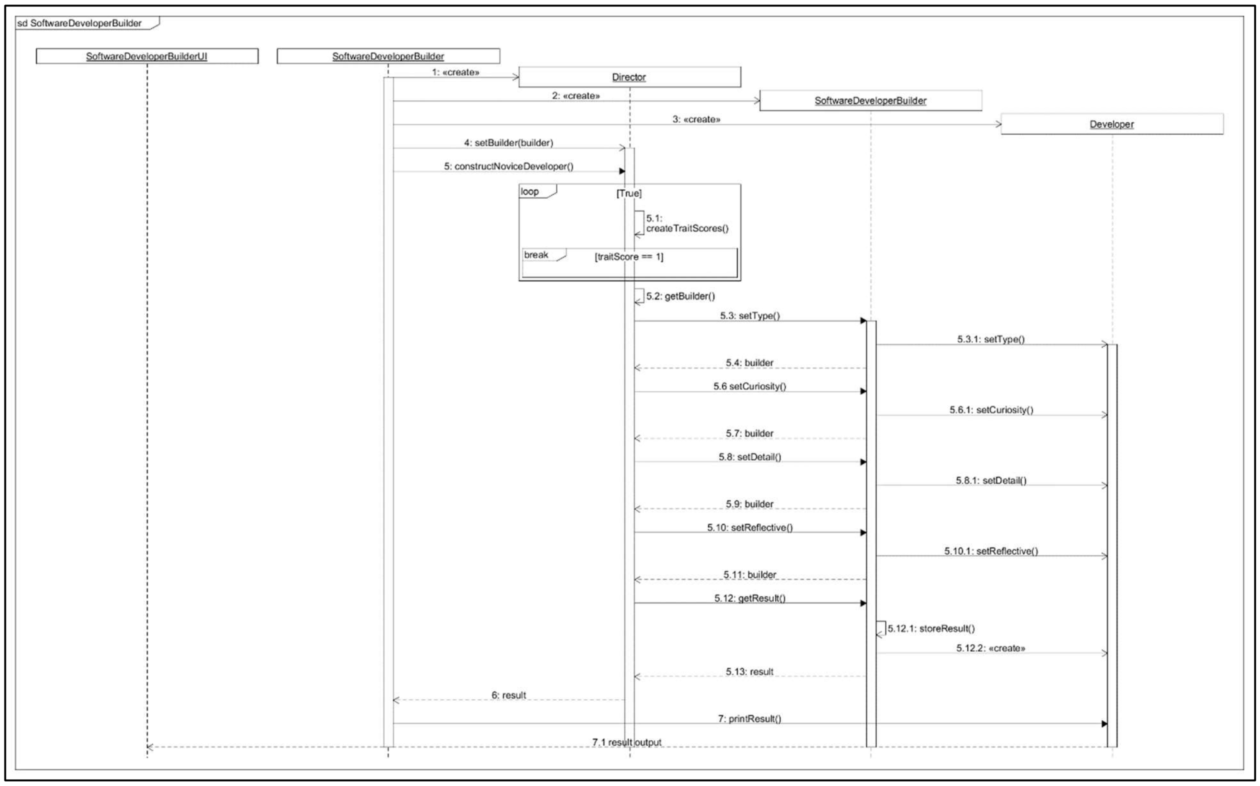
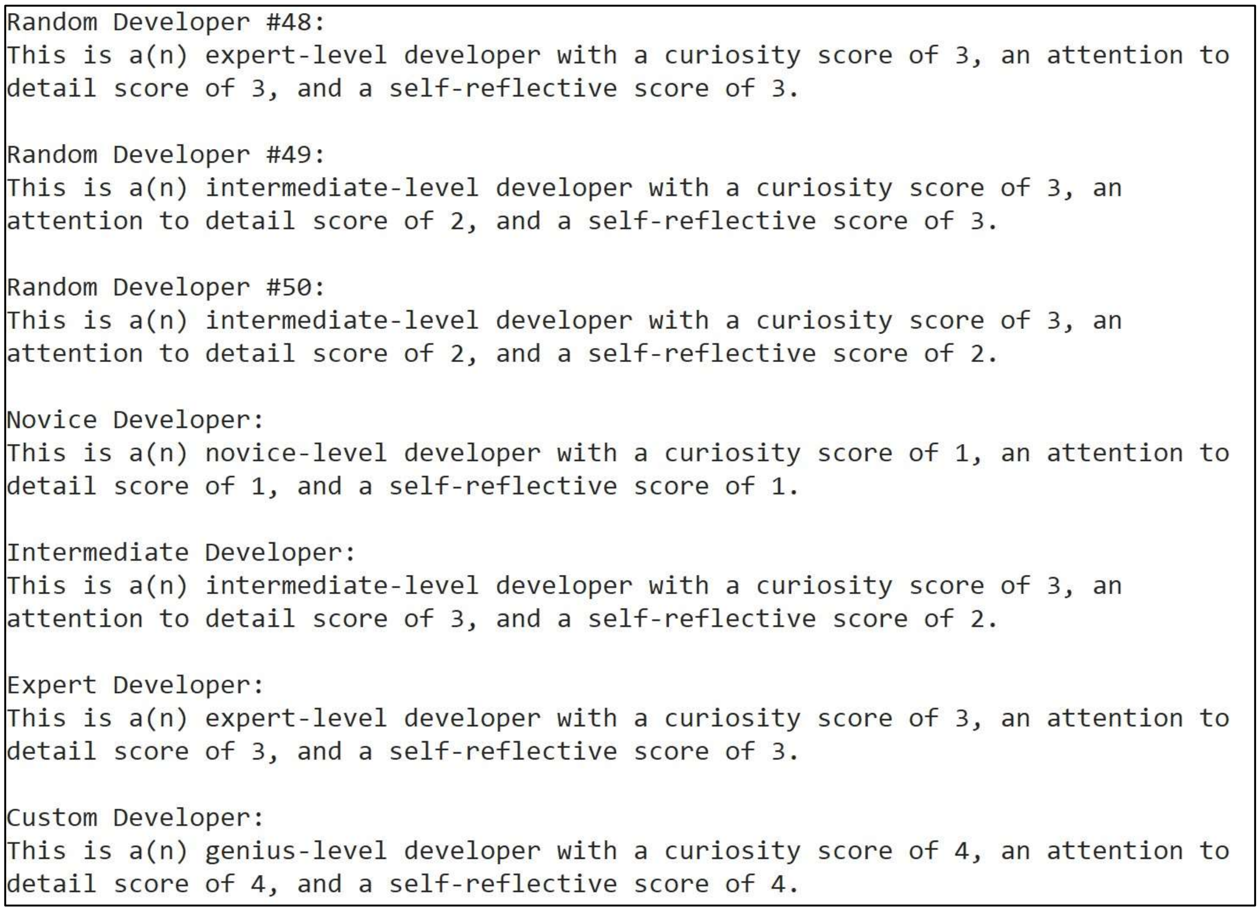


*Figure 2.* UML Sequence diagram showing the construction of a novice software developer object



```
Random Developer #48:
This is a(n) expert-level developer with a curiosity score of 3, an attention to
detail score of 3, and a self-reflective score of 3.

Random Developer #49:
This is a(n) intermediate-level developer with a curiosity score of 3, an
attention to detail score of 2, and a self-reflective score of 3.

Random Developer #50:
This is a(n) intermediate-level developer with a curiosity score of 3, an
attention to detail score of 2, and a self-reflective score of 2.

Novice Developer:
This is a(n) novice-level developer with a curiosity score of 1, an attention to
detail score of 1, and a self-reflective score of 1.

Intermediate Developer:
This is a(n) intermediate-level developer with a curiosity score of 3, an
attention to detail score of 3, and a self-reflective score of 2.

Expert Developer:
This is a(n) expert-level developer with a curiosity score of 3, an attention to
detail score of 3, and a self-reflective score of 3.

Custom Developer:
This is a(n) genius-level developer with a curiosity score of 4, an attention to
detail score of 4, and a self-reflective score of 4.
```

*Figure 3.* Successful program execution, showing developers of different types returned to the UI

## Option #1: Common Personality Traits

For his fourth *Critical Thinking* assignment in *CSC505: Principles of Software Development,* the student uses his personal experience and observation of expert software developers to create UML class and sequence diagrams depicting three personality traits shared among expert software developers. Figures 1 and 2 display these UML diagrams. The student also implements his solution as a Python script, referencing the *builder pattern* for inspiration.

Baltes and Diehl (2018) write that expert software developers possess a particular set of knowledge, skills, and experience, presenting the first theory of software development expertise based on surveys of 335 developers and relevant literature. One of the main questions guiding the authors' research was, "What characteristics do developers assign to novices and experts?" Participants in the study named expert developers' shared personality traits, including open-mindedness, curiosity, attentiveness to detail, patience, and self-reflectiveness.

Based on his personal experience and observations, this writer chose three of these characteristics to model in his UML diagrams: (a) curiosity, (b) attentiveness to detail, and (c) self-reflectiveness. Self-reflectiveness shares a critical connection with the concept of *deliberate practice* and includes recognizing one's strengths and weaknesses, as well as the ability to learn from one's past mistakes (Baltes & Diehl, 2018). Figure 1 shows these characteristics documented as methods in the *DeveloperBuilder*, *SoftwareDeveloperBuilder*, and *Developer* classes. Figure 2 shows these methods being called during the construction of a developer object. Though this figure presents the interactions pertinent to constructing a single developer object, the interactions to create all developer objects are nearly identical. As shown in Figure 1, the *SoftwareDeveloperBuilder* class inherits the methods of the *DeveloperBuilder* interface, which declares construction steps common to all builder types (*Builder,* n.d.).

Florijn *et al.* (1997) write that design patterns describe general solutions for recurring design problems and offer several benefits for developing object-oriented software, including minimizing design work and allowing developers to focus on critical decisions. Similarly, Zimmer (1994) writes that design patterns support the reuse of design information and allow developers to communicate more effectively. Lee *et al.* (2008) describe the *builder pattern* as separating the construction of complex objects from their representation. Pressman and Maxim (2020) expand upon this definition, writing that the builder pattern is a creational pattern allowing the same construction process to create different object representations.

Believing that most individuals possess some level of the personality traits described previously, this author implemented the builder pattern to construct different representations of software developers (e.g., novices, intermediates, and experts) who possess varying amounts of said characteristics, denoted by levels ranging from 1 to 3. For instance, if a developer achieves a score of "1" for any of the three personality traits, he is ranked a novice. Intermediate-level developers must score a "2" or higher in all characteristics, while expert-level developers must achieve a score of "3" for all personality traits. The application uses random numbers to generate these hypothetical trait scores, though one could attribute such scores to those achieved through questionnaire scales designed to measure personality constructs (McCrae & John, 1992). Step 5.1 in Figure 2 shows the creation of these trait scores in the *Director* class.

The program's main function demonstrates the client code used to build the developer representations. Lines 6 – 8 in the file *client.py* create new director and builder objects, associating the builder object with the director. The *Director* class defines the order to call construction steps, whereas concrete builders provide different implementations of the steps (*Builder*, n.d.). Lines 10 – 13 of the *client.py* file call the *Director* class's method to construct 50 random developer types using a *for loop*. The methods in the *Director* class return either novice, intermediate, or expert-level developers by chaining the methods of the *SoftwareDeveloperBuilder* class together (e.g., builder.setType("expert").setCuriosity(3).setDetail(3).setReflective(3).getResult()), a technique often used when implementing builder patterns (*Design Patterns*, n.d.). Steps 5.3 – 5.12 in Figure 2 display these interactions.

Upon returning the result to the client code, Lines 14 – 17 of the file *software_developer_builder.py* call the class's reset method so that the builder instance is ready to start creating another developer object, as shown in steps 5.12.1 and 5.12.2 of Figure 2. Line 13 of the client code calls the *Developer* class's construction method, which outputs the developer's type and personality trait scores. Lines 15 – 25 of the client code demonstrate the creation of novice, intermediate, and expert-level developers. Finally, as shown in Figure 1, the client code has access to both the *Director* and *SoftwareDeveloperBuilder* classes. If the client code needs to assemble a custom developer, such as the genius-level developer shown in lines 27 – 30 of the *client.py* file, it can access the builder directly. Otherwise, the *Director* class handles the assembly of the most common developer types (*Builder,* n.d.). Figure 2 shows how the *Director* class hides most of the details of developer construction from the client code, simplifying the creation of developer representations. Figure 3 shows screenshots of successful program execution.

In conclusion, this paper uses UML class and sequence diagrams to demonstrate three personality traits of expert developers: curiosity, attentiveness to detail, and self-reflectiveness. The student implemented his solution in a Python script using the builder pattern as inspiration, simplifying object creation by using director and builder classes.

## References

Baltes, S., & Diehl, S. (2018). Towards a theory of software development expertise. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 187–200. https://doi.org/10.1145/3236024.3236061

*Builder*. (n.d.). Retrieved March 7, 2021, from https://refactoring.guru/design-patterns/builder

*Design Patterns: Builder in Python*. (n.d.). Retrieved March 7, 2021, from https://refactoring.guru/design-patterns/builder/python/example

Florijn, G., Meijers, M., & Van Winsen, P. (1997). Tool support for object-oriented patterns. *European Conference on Object-Oriented Programming*, 472–495.

Lee, H., Youn, H., & Lee, E. (2008). A design pattern detection technique that aids reverse engineering. *International Journal of Security and Its Applications*, *2*(1), 1–12.

McCrae, R. R., & John, O. P. (1992). An Introduction to the Five-Factor Model and Its Applications. *Journal of Personality*, *60*(2), 175–215. https://doi.org/10.1111/j.1467-6494.1992.tb00970.x

Pressman, R. S., & Maxim, B. R. (2020). *Software engineering: A practitioner's approach* (Ninth edition). McGraw-Hill Education.

Zimmer, W. (1994). Relationships between Design Patterns. *Pattern Languages of Program Design*, 345–364.