Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Abstraction and Variation

Diomidis Spinellis

Master, a friend told me today that I should never use the editor's copy-paste functions when programming," said the young apprentice. "I thought the whole point of programming tools was to make our lives easier," he continued.

The Master stroked his long grey beard and pressed the busy button on his phone. This was going to be one of those long, important discussions.

"Why do you think copy-pasting is wrong?" asked the Master.

"Because I violate the DRY [don't repeat yourself] principle?" replied the apprentice hesitantly.[1]

"Exactly true, my young friend," nodded the Master.

"But it's so much easier to copy-paste code than to grapple with the alternatives," continued the apprentice.

The Master turned to his keyboard and typed "copy paste miner" in a search box. His memory was rapidly deteriorating, but luckily he could compensate by applying his formidable search skills on Google's planet-wide memory. He read aloud some key sentences from an article's abstract:[2] "CP-Miner uses data-mining techniques to efficiently identify copy-pasted code in large software suites and detects copy-paste bugs. Specifically, it takes less than 20 minutes for CP-Miner to identify 190,000 copy-pasted segments in Linux and 150,000 in FreeBSD. Moreover, CP-Miner has detected many new bugs in popular operating systems, 49 in Linux and 31 in FreeBSD."

The Master looked up from his screen, "Now that you've seen evidence of the extent and dangers of copy-pasting, let me ask you, are you sure you know all the appropriate alternatives?"

"Of course I know them," replied the disciple cockily.

"Let's see. How do you handle two identical code sequences?"

"I wrap the code up in a function or method."

"And if these functions are to execute subtly different code?"

"Then I'll factor out the differences through Boolean or enumeration parameters I pass to them. I'll set the code in the function to follow different execution paths, based on these parameters."

"And if the differences concern data embedded in code?"

"That's trivial; I'll simply pass those varying elements as parameters."

The Master was busily jotting down the apprentice's answers and his own ideas in a table (see table 1). He continued firing off questions.

"And if the differences lie in the types the functions use?"

"Then I'll employ Java's generics or the C++ template mechanism. Macros also work for this purpose, but they're ugly and error-prone."

"And if each version of the code has local data associated with it?"

"Then I'll wrap up the whole mess in a class."

"And how else can a class help you avoid copy-pasting?"

"I can abstract similarities in data by expressing them as a new type. Instead of copy-pasting a group of similar data elements, I can pack them up in a class or structure and then create many instances of it."

The Master paused. These apprentices are

Table 1

Abstraction and variation mechanisms

	Abstraction	Variation
Code	Function	■ Multistate parameter (Boolean or enumeration) ■ Value parameter ■ Function parameter (a function pointer, a functor, or an object implementing an interface) ■ Type parameter
	Decision table	■ Multiple table instances
Code/Data	Class	■ Object instance, subclass, type parameter
	Domain-specific language	■ An instance of a DSL program
	Type	■ Type parameter
Data	Database	

getting better every year, he thought. Yet he always doubted the level of understanding lying beneath the polished replies. "Let's find out," he murmured and continued in a louder voice, "But surely passing parameters around can get unwieldy when you want to express extensive code variations?"

The apprentice had an answer ready: "In that case, I can express those variations as different functions and pass one of those functions as a parameter to the code they have in common."

"But does this approach scale when you have many dimensions of variation among even more dimensions of similarity?" probed the Master.

The apprentice was now glad of the time he spent last month poring over the FreeBSD kernel source code. "It can scale if you're disciplined," he replied. "You organize the various functions together in structures representing the behavior that these functions implement. For instance, CDs and USB sticks use different layouts and data structures for storing their files. So, you have one group of functions for manipulating files on a CD and another group for manipulating files on a USB stick."[3]

The Master nodded appreciatively. "Is there a better way to express variations of code?" he asked, encouraging his friend to show off his knowledge.

The apprentice raised an eyebrow at the leading question and playfully continued the Master's game. "I'm very glad you asked me this. Indeed, we can abstract commonality and variability by organizing our code and data in a class hierarchy. Common elements go to the base classes, and classes derived from them contain the elements that vary." He paused. "Yet I sometimes indulge in copy-pasting because I feel that abstracting might not be worthwhile. Master, do you think there are limits to what we should abstract?" he asked hesitantly.

The Master smiled and sketched a rough chart (see figure 1). "Indeed my friend," he replied, "abstraction makes your code clearer by replacing concrete complexity with an abstract name, but there are costs. We can often ignore the time and space performance penalties, but the cost of comprehending any abstraction is real and slowly increases as we add abstractions in our code." He pointed at his chart: "Look here. On the left side, the gains from abstraction are so large that they shadow its cost. However, after a point, the benefits taper off. Further to the right, you see that the abstraction's benefits turn negative: your code becomes less comprehensible and maintainable."

The apprentice nodded his understanding and asked one last question. "But how can I determine when I'll gain by abstracting and when abstraction will obfuscate my code?"

The Master looked him in his eyes. "Dear friend, this is why superb programming requires a lot of experience and expert judgment. This is what makes programming an art."

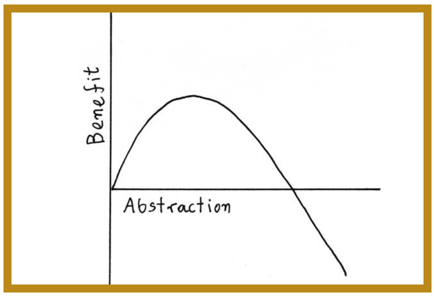

Figure 1. Abstraction and its benefits.

References

1. A. Hunt and D. Thomas, "OO in One Sentence: Keep It DRY, Shy, and Tell the Other Guy," *IEEE Software*, vol. 21, no. 3, 2004, pp. 101–103.
2. Z. Li et al., "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Trans. Software Eng*, vol. 32, no. 3, 2006, pp. 176–192.
3. D. Spinellis, "Another Level of Indirection," *Beautiful Code*, A. Oram and G. Wilson, eds., O'Reilly, 2007, pp. 279–291.

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.