Self-configurator

Pietu Pohjalainen
Department of Computer Science
University of Helsinki, Finland
Pietu.Pohjalainen@cs.helsinki.fi

Abstract. Traditional object-oriented design patterns often focus to improve flexibility of software. A usual pattern in traditional design patterns is to introduce a new layer of dynamic abstraction that gives an opportunity to parameterize behavior at runtime. A trade-off for this flexibility is often increased complexity and reduced performance. We present a pattern, the *self-configurator*, for resolving symptoms of unnecessary indirections introduced by many standard object-oriented patterns.

Keywords: Software maintenance, automated software engineering, self-configuration.

Intent

A famous quote states, "*All problems in computer science can be solved by another level of indirection*". This statement is a common driver for many traditional object-oriented design patterns. The less often cited continuing to the phrase states: "*but that usually will create another problem*" [1]. The self-configurator pattern's intent is to document and automatize the rules for resolving internal dependencies that are introduced by the use of other patterns.

Motivation

Software is full of hidden dependencies, where a seemingly innocent change can cause malfunction or crashing. Programmers have learned to defend themselves and their colleagues and customers from excessive rework by designing their software in a way that is resilient to future changes. As an illustrating example, we will use a dynamic data structure for implementing a simple processor for the Command design pattern [2].

Let’s consider the code in Figure 1. It first registers three objects for handling different commands, and then repeatedly reads in a command and dispatches following arguments to the given command.

```
class CommandProcessor {
    static Map<String, Cmd> funcs =
        new HashMap<String, Cmd>() {{
            put(“print”, new PrintCmd());
            put(“noop“,  new NoopCmd());
            put(“quit”,  new QuitCmd());
        }};

   private static Scanner scanner =
           new Scanner(System.in);

   public static void main(String a[]) {
        while(true) {
            String cmd = scanner.next(“\\w+”);
            String args = scanner.nextLine();
            funcs.get(cmd).Execute(args);
        }
    }
}
```

Figure 1: Code for a command line processor

For our discussion, the interesting property in this code lies in how the processor uses a dynamic data structure as the storage for the registered commands. Using a dynamic structure makes it easy to add new commands at later time. In contrast to implementing the same functionality by using e.g. a switch-case construct and hard coding the possible commands into the structure of the command processor, this dynamic solution makes the program easier to modify.

This flexibility is gained with the minor runtime cost of using a dynamically allocated data structure with every command fetching being routed through the object’s hashing function. Although the runtime cost is small, it still adds some memory and runtime overhead, since the generic hashing implementation cannot be optimized for this specific use case. For example, the standard Java implementation for HashMap allocates the default value of 16 entries for the map implementation. In this case, only three of the entries are used, as shown Table 1. Also, when fetching the command object for a given command, a generic hashing function is used, which also gives room for optimization.

0	*null*
1	*null*
2	*null*
3	*null*
4	*null*
5	*null*
6	*NoopOb*
7	*null*
8	*null*
9	*null*
10	*null*
11	*null*
12	*null*
13	*QuitOb*
14	*PrintOb*
15	*null*

Table 1: HashMap default layout

With this discussion, we can see characteristics of accidental maintainability in our example. With accidental maintainability we mean that in this case the solution uses a dynamic data structure for handling a case that does not require a dynamic solution. Namely, the set of available commands is a property that is bound at design time, but a structure that allows runtime binding is used. There are a number of reasons for implementing the command processor in this way. The map implementation is available in the standard class library, its use is well known and understood among programmers and for many cases, and the induced overhead is negligible. Yet another reason can be the lack of viable alternatives in current pattern knowledge. In cases where any overhead should be minimized, introducing this dynamic structure purely due to implementer's comfort would not be good use of scarce resources.

Solution

An alternative solution for this example is to create a specific implementation of the map interface that is statically populated to contain all the required elements. This would make it possible to use context-specific knowledge of the structure in implementing the command fetching system: instead of using a fully generic hashing table, more memory and runtime efficient, specific hash table and hashing functions for these three commands could be implemented.

The self-configurator pattern resolves this problem by introducing a configurator component to this structure. It is a pattern for implementing self-organizational components.

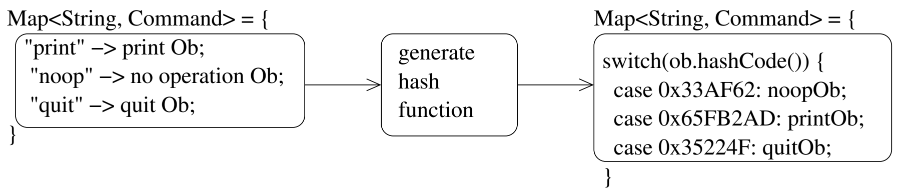


Figure 2: Self-configuring function for the command processor

Figure 2 illustrates the configuration process. The self-configuring component reads the static list of commands and generates a specific hashing function for this set of commands to be used. Now the runtime and memory overhead of generic hashing is avoided. The hash generation function is bound (i.e. executed) at the same time as all other parts are compiled. This way, the runtime overhead can be minimized. However, the design-time allocation of command names and associated functions still enjoys the flexibility of defining the command mapping as a well-understood, standard Map interface.

There is a degree of freedom in placing this generative part in the binding time continuum. The hash generating function and associated hash map generation can happen as part of the normal compilation process, or it can be delayed up until first use of the command processor object. As usual, earlier binding time gives opportunities for optimizing for that special case, while delaying binding gives more flexibility and possibilities to use contextual information to determine behavior.

Applicability

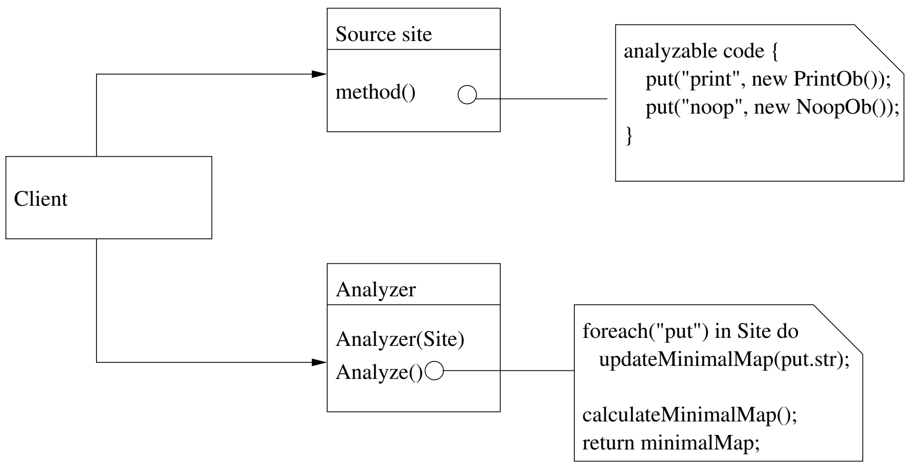
There are many situations when you can apply this pattern. First of all, the pattern is applicable when you are using dynamic structures to guard against changes that a future developer might be performing. In the example in the previous section, the dynamic mapping structure gives defines a clear place for implementing additional commands.

However, this flexibility is gained by introducing additional runtime cost.

Another scenario where you can find this pattern useful is if you need to provide characteristics of one code site to parameterize another routine. An example of this case can be e.g. a dependency between a set of different algorithms performing a computation upon data that is held in the database. Each algorithm requests certain set of data, but you want to separate the database fetching code from the algorithm's processing code. In this case, you can introduce a self-configuring component to analyze each specific algorithm and to automatically produce optimized queries for each algorithm without introducing a dependency between the query site and the algorithm.

Optionally, the pattern can also expose details of the processed dependency via a dependency interface, which allows programmatic access to characteristics of this dependency. In the previous section's example, this kind of dependency lies between the statically allocated list of commands and the command-line processing loop.

Structure



```
analyzable code {
   put("print", new PrintOb());
   put("noop", new NoopOb());
}
```

```
foreach("put") in Site do
   updateMinimalMap(put.str);

calculateMinimalMap();
return minimalMap;
```

Participants

In general, the pattern deals about computationally resolvable dependencies between code artifacts. The resolver component for configuring the dependencies can be implemented in various ways, depending on the contextual needs. Typical variations for the resolving process can be e.g.:

- Compilation time configuration
- Instantiation time configuration
- Runtime configuration

Usually, the later this configuration is done, the more information is available for the configurator. However, earlier resolving usually offers opportunities for better performance and more options for further optimization.

The participants can collaborate in runtime configuration as follows:

Client instantiates the analyzer, with a parameter that defines the source site to be analyzed

Analyzer reads in the source site definition, and resolves the wanted properties of the source.

Collaboration

The results of the self-configuration can be characterized as intrinsic or extrinsic. In intrinsic mode, the pattern implementation represents a substitute for the analyzed dependency site; e.g. the implementation for the command processor would represent itself as a map from Strings to Commands.

In extrinsic mode the self-configurator analyzes a dependency site and drives another object's configuration based on the results.

Implementation

In order to analyze a code site for configuring its dependents, there needs to be a way to access the source data. When using compilation-time configuration, all the source code is available for analysis. For instantiation time and runtime configurations the analysis interface is defined by the execution environment characteristics: some environments, such as the LISP language expose the full structure of the program for further analysis; but many current environments do not. Popular alternatives range from byte-code analysis, such as the BCEL library [3] in the Java environment, to standardized API access to program definition, as implemented in .NET's Expression trees, available in C# since its third version [4].

Regardless of the used access method, the configurator component analyzes the dependent's source. Based on this analysis, the dependent is configured to adhere to the form that is required by the source site. In the previous section's example, a possible configuration could be a generation of a minimal perfect hashing table for the different registered commands.

Often the required target configuration varies from one context to another. What is common in different variations is the built-in ability for the architecture to adapt to changes between architectural elements, which help both in maintenance and in gaining understanding of the overall system.

Known uses

To illustrate the idea of self-configuring software components, we present example cases from our previous work and from the industry.

a) Interpreters and compilers
Tim Barners-Lee is quoted of saying: "*Any good software engineer will tell you that a compiler and an interpreter are interchangeable*". The idea behind this quote is that since the interpreter executes code in the interpreted language, it necessarily has the required knowledge for producing the equivalent lower level code. Also the other way applies: the compilation routines for a given language can also be harnessed to build an equivalent interpreter.

This interchanging process can be seen as the self-configuration component. This has been applied e.g. to build compilers for embedded domain-specific languages [5] and to produce portable execution environments for legacy binaries [6].

The self-configurator in this case can build a compiler from an interpreter by analyzing each opcode definition of the interpreter and by emitting each opcode's corresponding code as the code generation step.

b) Self-configuring database queries

Many useful information systems can be characterized as typical database applications: they read data from a database to the main memory, perform an algorithm on the data, and then write the result back into the database.

These types of applications have a dependency between the data that is read from the database, and the algorithm performing the calculations. Within the object-oriented style of programming, an additional object layer is built on top of a typical relational database, creating an additional problem of object/relational mismatch. An approach of building object-to-relational mapping frameworks, such as Hibernate [7] proved to be popular as a bridge between object-oriented application code and relational persistence structures. In order to provide a fluent programming experience for the object-oriented design, *transparent persistence* is one of the key phrases. The promise of transparent persistence means that objects can be programmed as objects, without thinking the underlying relational database.

One of the tools for achieving transparent persistence is the usage of the proxy design pattern [2] to hide if an object's internal state is stored in the database, or whether it is already loaded to the main memory.

However, in many cases this delayed fetching hides symptoms of bad design: the program relies on the slow, runtime safety net implemented with the proxy. A better design would be to explicitly define, which object should be fetched. If the objects to be processed within certain algorithm can be known beforehand, the usage of the proxy pattern can be classified as a design fault.

We have documented the usage of the self-configurational database queries as a tool to improve runtime properties of this case at [8]. In this design, a code analyzer reads in the byte-code of given algorithm and deducts the required queries for prefetching the needed data from the database. This design helps maintenance properties: should the algorithm change for some reason, the fetching code is automatically updated to reflect the change. Another benefit is that on architectural level, the number of database-accessing components is reduced, since this one component can configure itself for multiple cases.

c) Self-configuring user interface components

Component-based software engineering is widely employed in the area of user interface composition. User interface widgets can be developed as stand-alone components, and a new application's interface can be built by composing from a palette of these ready-made components. Pioneered in Visual Basic, the approach has been adapted to numerous architectures.

One of the drawbacks in component-based user interface composing is the need for duplicated binding expressions when programmatically defining multiple properties of user interface components. For example, when defining whether a user interface component is active or not, a corresponding tooltip should be placed. Without sufficient support for cross-referencing to other binding expressions, providing this kind of conceptual coherence in the user interface requires cloning of the behavior defining expressions.

We built a prototype for analyzing these binding expressions in the standard Java environment for building web interfaces, the Java Server Faces [9]. By exposing the structure of the binding expressions to backend code, we were able to reduce the amount of cloned binding expressions by a factor of 3 in a demo application [10].

d) Generating languages for domain-specific queries

The previous examples work in the expression-level abstraction. The approach of self-configuring components can be scaled to component-level. For example, the QueryDSL framework [11] generates internal domain-specific languages in the spirit of fluent interfaces and interface chaining.

For the problem of querying data in a domain model, the QueryDSL framework generates a class structure that reflects the domain model, augmented with a set of querying functions. These querying functions can be used to formulate aggregation, filtering and sorting queries in the standard Java environment. The generative nature of the framework is exploited to build type-safe queries, which is in contrast to the previous model of using generic objects to bring the domain model concepts to the program's structure.

The difference between the QueryDSL approach and the previous examples is the applicability scale. While the previous example work on intra-component level, there is no reason why the approach could not be scaled to component and systems level.

Related patterns

Many traditional object-oriented design patterns can be analyzed and optimized via this pattern.

The self-configurator pattern can be seen as a formalized variation of maintenance patterns [12]. In maintenance patterns, the idea is to document the required tasks to perform feature adding maintenance tasks. In the self-configurator pattern, these tasks are documented in executable code (reconfiguration rules), so that the software can adapt itself to the new situation.

The pattern uses the idea of introspection and reflection from the CLOS meta-object protocol [13] to build the maintenance instructions.

References

1. Diomidis Spinellis. Another level of indirection. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 17, pages 279–291. O'Reilly and Associates, Sebastopol, CA, 2007.
2. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
3. Markus Dahm, Byte Code Engineering with the BCEL API. Technical Report B-17-98. Freie Universität Berlin, 2001.
4. C# Language Specification, version 3.0. Microsoft Corporation, 2007.
5. Conal Elliott, Sigbjørn Finne and Oege De Moor, Compiling Embedded Languages. Journal of Functional Programming, Volume 13 Issue 3, May 2003
6. Alexander Yermolovich, Andreas Gal and Michael Franz. Portable execution of legacy binaries on the Java virtual machine, PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, ACM: New York, NY, 2008; pp. 63-72.
7. Christian Bauer and Gavin King. Hibernate in Action. Manning Publications, 2004.
8. Pietu Pohjalainen and Juha Taina. Self-configuring object-to-relational mapping queries, PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, ACM: New York, NY, 2008; pp. 53–59.
9. JSR 252: JavaServer Faces 1.2. Technical report, Sun Microsystems, 2006.
10. Pietu Pohjalainen. Self-configuring user interface components. In A. Dix, T. Hussein, S. Lukosch, J. Ziegler (Eds.), Proceedings of the First Workshop on Semantic Models for Adaptive Interactive Systems, 2010.
11. Timo Westkämper, Samppa Saarela, Vesa Marttila and Lassi Immonen. QueryDSL reference documentation [online].
12. Imed Hammouda and Maarit Harsu. Documenting Maintenance Tasks Using Maintenance Patterns. In proceedings of CSMR 2004, IEEE Computer Society, pp. 37-47, 2004.
13. Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow, The Art of the Metaobject Protocol. MIT Press, 1991.