Granule-Oriented Programming (Extended Abstract)

Yinliang Zhao

Department of Computer Science and Technology
Xi'an Jiaotong University, Xi'an 710049, P.R.China
zhaoy@mail.xjtu.edu.cn

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features –*Control structures, frameworks.*

General Terms

Languages, Design.

Keywords

Reflection, granule-oriented programming, program grinding, code granulation space, object-oriented programming.

1. INTRODUCTION

A program will become obsolete or less effective in solving domain problems due to many reasons. One of the main reasons can be the fact that the program becomes unfit to its context. A program's context can be the descriptions of the program's runtime environment, the meta-strategies in its domain, and/or the architectural features of the machines it runs on, etc. The "unfitness" phenomena exist in many complex systems, cause them terminate the life cycles prematurely, or decrease the performance and accuracy in problem solving. In existing programming systems, from the perspective of language expressivity, little attention has been paid to this unfitness problem.

One "unfitness" phenomenon most frequently may happen while a program is forced to solve a new problem in the domain. A program is normally designed for solving a particular domain problem. When it is required to solve another problem in the domain, it may result the program to work improperly. In this case, the system becomes obsolete, that is, it has reached to the end of its life cycle. Normally, a new program will be applied to solve current domain problem. Note that the program is viewed as an implementation of a system in this paper.

Another unfitness phenomenon happens when the program faces new runtime support systems such as improved memory management, selected communication means, or even an adopted new machine, etc. This kind of unfitness is often related to architecture features. So the invariable agreement between program and its context will make premature end of the program's life cycle, or lead an improperly continuous execution with inaccuracy or low efficiency.

OOPSLA'04, Oct. 24–28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

One of the explanations for which the unfitness phenomena happen is that the existing programming languages have their expressive abilities being limited in describing the agreement of a program and its context explicitly. We should add some facilities to the programming language so that the unfit parts can be localized. In other words, the program should be able to access to, interact with its context to deal with unfitness.

This conclusion has led us to develop a concept we call granule-oriented programming, GOP in short. GOP is an evolvement metaphor, in which the programs can be "ground" into code ingredients in order to localize unfit parts of a program as explicitly as possible. The code ingredients can be "compounded" into program components, called code granules. The family of code granules can be organized as a granulation space in which we can control the unfitness from multi-level abstraction means, such as zooming-in or zooming-out. In GOP, we pay attention to code granules, their evolution from one program to another. Granules can be layered through one or more lower level granules being compounded into high-level granules. We believe that programming on the zooming-in and zooming-out along with granulation layers in the granulation space is important to localize unfitness.

2. UNFITNESS

The basic limitation of a programming language is that the context of the program cannot be easily programmed as domain problem solving does. In other words, the context cannot be seen at the programming stage. Here the term, *the context of the program*, is employed to indicate all the things that are related to how the program is being processed in programming phrase and how the program runs in execution phrase. This context is defined as a collection of all functionalities that support the program to solve the domain problems. In classical programming, programmers are forced to use language facilities to express how to solve domain problems at the language abstraction level. There is an invariable, no doubtful and static agreement between the program and its context. In other words, the programmer may say that it is not his/her responsibility when he/she faces the above difficulties. For instance, a logic programmer may say that keeping the memory reference locality in logic programming should be the compiler's work or the operating systems' responsibility. But in reality, it is hard to control the memory reference locality from outside of the program, especially for the requirement of some knowledge representation, such as *frames*, although some garbage collection algorithms tend to localize object representation in memory.

The agreement between a program and its context will be broken down in many situations, for instance, new problem solving in the domain is considered, or new supporting system is adopted, etc. A concrete example about the invariable agreement between a program and its context is remote object invocation. There is normally an invariable agreement between remote object and the client program, keeping the consistence of name binding, parameter passing. This agreement is beyond the program's known. Now we assume the remote object is modified for some reason, for instance, an extra parameter is added to the interface of a method of the object; therefore, the agreement cannot be kept any more. This may cause an exception to the client program in most cases, although the program still does not know the change. In this example, the remote object invocation is viewed as the context of the client program. It is lack of such facilities in the programming languages that the context of the program (coordination with the remote object) is programmable during the life cycle of the program. Obviously, the program will be in the state of unfitness while the agreement is broken down.

Some special cases of unfitness can be fixed by program online upgrade mechanism [1] to a limited extent. In above example, the agreement should be extended to cover the concern of remote object modification. Then client program can be designed in a polymorphism style, which can adapt to the notification of remote object update.

However, due to the lack of adequate means in the language expressivity, there are still some cases in that programming on the context of a program is beyond the program's ability to process. For example, for distributed computing there is a local data processing system, which uses remote data access, and obtains data from some data centers. Along with different performance variance of the networks and the servers that the system is using, data processing algorithms easily become unfit to data access because, as a theoretical strategy, data processing should match the data access rate precisely for low overhead. However, in reality, it is hard to know either how data access rate varies, or all available alternatives that balance the data processing and data access.

A natural thinking about a solution to the unfitness problem is to localize unfit code ingredients, and provide alternatives for them in programming. Suppose a program P_0 fits context T_0. After T_0 is transited, P_0 does not fit the new context T_1. One needs to design another program that fits either T_1 or T_1-T_0. One can use P_1 in place of P_0 for context T_1, or compound P_0 and P_1 such that P_0 is for context $T_0 \cap T_1$ and P_1 is for context T_1-T_0. In this paper, the compound of P_0 and P_1 is the program one pursues for the context T_1. Moreover, we treat the programs P_0 and P_1 as two families of granules, so they can be also denoted as G_0 and G_1, respectively. All the granules of G_0 and G_1 have the fitness/unfitness problem as what programs P_0 and P_1 have.

We believe this unfitness phenomenon hooks up the complexity of most existing complex software systems. The concern, that how granules fit their contexts in the execution of the program, is at the heart of much of the complexity in those software systems. The objective of granule-oriented programming is to provide an operational way to deal with unfitness in complex systems. More specifically, GOP allows the programmers first capture the primary problem solving cases in the domain, express each of them as code granules, and then compound them into granular output code.

2.1 A Methodology

We propose a methodology of granule-oriented programming as shown in Figure 1. The concept of primary problem solving, PPS in short, is employed to express an incremental effort in programming. In other words, programmers capture a context snapshot of a domain, and write out a program that has an invariable agreement with this determined context. On the other

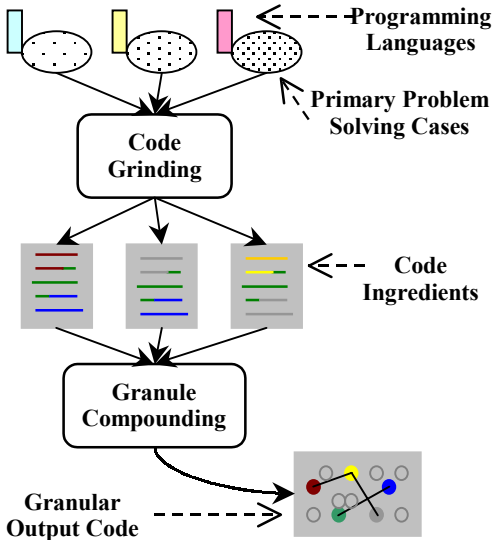

Figure 1. The basic elements of a GOP system.

hand, programmers can also derive code from some basic PPS cases. We suppose that all PPS cases are coded in separate programs or program segments, and they may be written in different languages, respectively.

When not only a PPS case is programmed originally, but also a new PPS program is derived from some basic PPS cases, a fundamental observation is that programs can be ground into pieces. By program grinding, programmers are able to find code ingredients from every PPS cases and to do code similarity analysis between them.

It is intuitive to think an ingredient as a piece of code, such as a function, a procedure, a class, or a method. The objective of program grinding is to locate unfit things by code similarity analysis. We hope that the similar code ingredients can be compounded into granules. The granules that are directly derived from code ingredients are called base granules. Base granules can be compounded to form high-level granules. For example, the granules derived from PPS cases can be compounded into a high-level granule, which can be the solution of a more complex PPS case. We can achieve the compounding by some predefined granulation facilities such as add class or add method.

An example in parallel programming shows that program grinding is reasonable in this situation. In parallel programming, a problem can be decomposed into a group of tasks using Ian Foster's PCAM (partition, communication, aggregation, and mapping) methodology. The produced tasks can be viewed as granules with concurrency coherence. Mapping the aggregated granules onto

processing elements in a parallel system can be viewed as a kind of granule compounding, by which the group of granules that are mapped onto one processing element can be viewed as a high-level granule. The goal of such granulation of the program is to gain load balance.

3. CODE GRANULATION SPACE

The code granulation space is dedicated to primary problem solving in a domain, which is an expression of the program in multiple-abstraction framework. The goal of building code granulation space for PPS cases is to localize unfitness in a well-formed and multi-layered framework. We believe that for a given domain, the code granulation space of every PPS can be merged into a complete code granulation space of the domain. From this viewpoint, the code granulation space of every PPS is a partial space of the complete one the domain has. A programmers' responsibility is to make a step forward to the complete space from the current PPS cases.

3.1 An Example

we present a granular output code for three cases of primary problem solving in a simple and classical problem domain, sorting using `quicksort`. As a GOP example, this sorting domain is just used for demonstrating how unfitness phenomenon occurs and the significance the granules are formed.

Though the problem domain is simple, the combination of matching sorting algorithms to their contexts is large. As an example, we just choose three PPS cases as follows.

PPS0. Data are read from an input file and sorted in main memory using `quicksort`. The results are written to an output file.

PPS1. Data are read from a serial port (or a stream) and sorted in main memory using `quicksort`. The results are written to an output file. In this problem, we suppose the data access is significantly slow, so it may be better to sort currently arrived data while waiting for the rest data. A final merge needs to be carried out for all the intermediate sorting results.

PPS2. Data are read from an input file and sorted in bounded physical memory (no virtual memory is allowed). Partition of that data file may be required because of the limited memory size. As a result, an external merge of all the intermediate sorting results is required to generate a final result.

The contexts of these three PPS cases are different. Context0, initially corresponding to PPS0, is defined as an infinitive memory support and a fast data input/output access. Context1, initially corresponding to PPS1, is defined as an infinitive memory support and a slow data input. Context2, initially corresponding to PPS2, is defined as a limited memory support and a fast data access. The contingency Table 2 shows the combinations of PPS cases and contexts, which shows that the unfitness phenomenon can easily happen in the system.

Each PPS program is assumed that it fits its own context. PPS0, PPS1 and PPS2 fit context0, context1 and context2, respectively. Moreover, PPS2 fits context0, because PPS2 program must be able to run in the context of PPS0 case. This can be explained in detail as the fact that if a program can run in a bounded memory then it must be run in an infinitive memory. However, PPS0 does not fit context1 because the PPS0 program reads data in high speed from an input file, but not from a slow port. Data come from the port may not have a steady transmit speed, and thus decreases the PPS0 performance. A PPS0 program will run successfully in the context2 if the memory size of context2 happens to be not less than the requirement of the execution of the program. However, the PPS0 program generally does not fit context2. All the other situations are analyzed in Table 1.

Table 1. Fitness of PPS cases to contexts

	Context0 **• Inf. Mem.** **• File I/O**	**Context1** **• Inf. Mem.** **• Slow Access**	**Context2** **• Bounded Mem.** **• File I/O**
PPS0	Fits	Data source, Performance	Fits special cases; Unfits general cases
PPS1	Data source, Performance	Fits	Fits special cases; Ufits general cases
PPS2	Fits	Data source, Performance	Fits

3.2 Some Issues

Similarity detection. The main objective of program grinding is to localize unfitness by similarity analysis between primary problem solving cases for a domain. Therefore, similar code ingredients are more general than dissimilar ones.

Context distribution. As described above, the goal of context distribution is to determine which PPS granule that locates inside the whole granulation space is associated with which part of the context of the PPS. A general idea about context distribution is that each granule is only responsible for nice fitness to its own context.

Zooming-in/zooming-out. Zooming-in and zooming-out are basic mechanisms to describe how the low-level granules are compounded into the high-level granules in code granulation space. In the example, the granule `<producer:>` is compounded by the granule `<read_port:>` and `<mutual_write:>`. The zooming-in/out between lower level {`<read_port:>`, `<mutual_write:>`} and upper level {`<producer:>`} can be defined as following code:

```
{ int tmp;
  while(!(<read_port:data,port>)){
     <mutual_write:lock,data,buffer>;
     notify();
  }
}
```

4. RELATED WORK

Lots of existing work appears to be based on intuitions similar to those underlying granule-oriented programming.

Aspect-oriented programming. AOP makes it possible to define additional implementation to run at certain well-defined points in the execution of the program, namely dynamic crosscutting mechanism, which is based on a small but powerful set of constructs [3]. AOP stresses the separation of concerns in programming, by means of *advice* and *pointcut*, a method-like construct, which is comparable to the CLOS method combination

framework. The goal of AOP is to make it possible to deal with crosscutting aspects of a system's behavior as separately as possible. AOP provides a means of program grinding by which the code ingredients can be classified according to the aspects they belong. As the result, these code ingredients can be compounded and therefore form a granulation space where each aspect will be a granule at some level.

Object-oriented programming. OOP provides powerful language constructs for organizing a program as a group of communicating objects. For example, classes, methods, and inheritance hierarchy are helpful to describe the system's building blocks and relationship between them; the message-passing mechanism is useful to realize the behavioral relationship between objects. Classes and methods can be special cases of code ingredients and granules in GOP. And sub-classing and message passing can be special cases of granulation.

Reflection. Reflection is a powerful mechanism of some of the programming languages, which supports the program to deal with its own facilities in the course of domain problem solving. Reflective programming languages provide language constructs or facilities to deal with the program's context in more explicitly than non-reflective languages. Great efforts have been done on reflective language design, such as 3-Lisp [4], the CLOS metaobject protocol, and some work on prolog, Java, Smalltalk, and C++ reflective mechanism extensions, etc. For example, the sub-classing mechanism and the method combination framework, provided by a reflective object-oriented language, could be processed by the program that is written in exactly the same language. In this case, we can say the unfitness phenomenon has been explored in a certain extent with reflective facilities in these languages.

Some other languages. Component-based design [5] is a methodology that tries to find the system's common behavior and then generalizes them into the reusable components. Reusable components provide similar functionality as granules to localize special behavior and separate them with the other part of the system.

Generative Programming [2] provides a means for developing programs that synthesize other programs. The goal of generative programming is to replace manual search, adaptation, and assembly of components with the automatic generation and configuration of components on demand. This idea is similar to GOP that a program can be partially derived from the existing programs.

5. FURTHER DISCUSSION

We observe that the unfitness problems occur in many complex systems. The unfitness may cause the program work improperly. To analyze this phenomenon, a concept, the context of the program, is employed to describe all functionalities that support the program solving domain problems. The unfitness phenomenon is then explained as that the program does not fit its context dynamically. In normal programming, there is an invariable agreement between the program and its context, so the programmer is forced to obey it unconsciously. This static agreement is the source of unfitness phenomenon, and may cause the program terminate prematurely or run improperly. These contextual requirements imply that a program should be programmed in an innovative way in which not only the program itself but also its context can be programmed.

The expressivity of granule-oriented programming is that programs can be ground into code ingredients for localizing unfitness, and some of these ingredients can be compounded into a new program. GOP assumes that domain problems can be solved using a gradually generated program. This means the domain can be described partially by some primary problem solving cases. The generated program by grinding-compounding can be viewed as a new PPS of the domain. Therefore, GOP is an evolvement metaphor.

A code granulation space is an expression of a program in multiple-abstraction framework. It is used to localize unfitness in a well-formed and multi-layered framework. Zooming-in and zooming-out between multiple layers in the granulation space are helpful to explore the unfitness phenomenon.

Some code granules can evolve from one primary problem solving case to another, which are more generic, that is, they have been reused by more primary problem solving cases in the domain. Case studies on granule-oriented programming show that the fitness of a program to its context can be expressed with the fitness of the program granules to their contexts, respectively.

Some of future directions of granule-oriented programming are: practice and application of granule-oriented programming; foundations of program grinding, code granulation space, and granule compounding; development of granule-oriented programming toolkit; technologies of granule-oriented software development; etc.

6. ACKNOWLEDGMENTS

The author would like to thank Ms. Yan Zhao for English improvement of this paper. Also thank to anonymous referees for their comments and suggestions on this paper.

7. REFERENCES

[1] Chaudron, M. R. V. and Laar, F.V. An upgrade mechanism based on publish/subscribe interaction. In Workshop on Dependable On-line Upgrading of Dist. Systems, COMPSAC 2002, (Oxford, England), Aug. 2002.

[2] Czarnecki, K. and Eisenecker, U.W. Generative programming – methods, tools, and applications, Addision-Wesley, 2000

[3] Kiczales, G. Hilsdale, E., Hugunin, J., Kerstn, M., Palm, J. and Griswold, W. An Overview of AspectJ. In proc. European Conference on Object-Oriented Programming, 2001, Springer-Verlag LNCS 2072.

[4] Smith, B.C. Reflection and Semantics in LISP. In Proceedings of the Symposium on Principles of Programming Languages (POPL). ACM. (1984) 23-35

[5] Szyperski, C. Component Software – Beyond Object-Oriented Programming, 2nd Ed., Addison-Wesley, ACM Press, 2002