# GranuleJ: A Granule-Oriented Programming Language

Yinliang Zhao

Department of Computer science and technology, Xian Jiaotong University, Xian, Shaanxi, China
{zhaoy}@@mail.xjtu.edu.cn

## 1 Design Principle

Granule-oriented programming (GOP) has been proposed as the idea for capturing and maintaining the fitness of computable program for their execution contexts. In most cases, the execution environments have played significant impacts on the program behavior at run time. Although current program language mechanisms are extremely useful to write almost perfect programs, they are unable to consider the intrinsic contextual characteristics the running programs depend on. For this reason, the program might lead to boundary violation or malfunction, even halt. As a result, we believe the opinion that in the construction of any programs, there must be unpredictable execution contexts that inherently prevent those programs from running in perfect state at runtime. In other words, the program behavior must match its environment contexts. The matching is called the program's fitness.

GOP addresses the fitness of programs, and provides core linguistic mechanisms to explicitly catch fitness-related structure, which makes it possible to conduct fitness concerns in a straightforward way and achieves the usual benefits of improved robustness and extensibility. Thus, program's fitness can be detected dynamically and fixed with more attached code during the lifecycle of the program, which has great potential for software evolution in a natural way. Here, the well-encapsulated and well-modularized fitness concern is specialized as a granule.

GranuleJ is a simple and practical granule-oriented paradigm extension to Java language. It provides integrated support for developing and executing programs written in GranuleJ, appending a number of core language features, basic compilation strategy and granulej virtual machine (GVM) simply extended from Java virtual machine (JVM) and so forth. With GranuleJ, we can code fitness concerns easily, reducing development costs and combating increasingly software or hardware complexity and heterogeneity.

## 2 Fitness Model

The fitness model is a critical element in the design of granule-oriented language mechanism. This model builds the foundation of program code grinding and

compounding, which makes it possible for programs to localize unfitness and improve them during their entire lifecycles. The proposed fitness model covers how to express the programs fitness, how to evolve the program according to fitness and how to create the programs individuals. Furthermore, the model gives a specific solution for single inheritance of object-oriented language operations, including granule and granule-tree construction, similar granule substitution.

### 2.1 Context-based Fitness

Program execution implies that there are execution environments and a program that is running in them. Since the unfitness phenomenon always occurs during program execution is that the program is not fit for the execution environment where it runs, the context constraint condition that prescribes the program fails to fit the runtime environment is able to capture and measure. Hence, a program in the design time can reflect the relationship between its execution and runtime environment, making program become evolution possible. We adopt "fitness" here to reflect the proper relationship between an executing program and its execution environment, which means that if the fitness of the program is satisfied, the program behaves properly at runtime. We adopt "context" as the abstraction of the execution environment where the programs execute. Therefore the fitness of programs reflects whether these programs are fit for the current context or not.

Contexts are not measurable by Turing Machine because some of them are continuous functions, real number intervals, etc. One solution for this problem is to express the used contexts in design time and runtime of programs respectively with innovative mechanisms. Consequently, the fitness is modeled as whether the expected contexts match the real contexts. In the fitness model, context variables are designed as a kind of language entities to express contexts, containing both expected and real ones.

### 2.2 Population-based Evolution

The fitness model adopts population-based evolution which is similar as natural evolution to describe the program changes to meet the fitness requirement. Any program is always static in design time, but at runtime we consider it as an individual which lives in the execution environment. So the program is a part of individual, describing the behaviors of individual. Individuals have their lifetimes and always correspond to the real contexts. In brief, programs usually constitute individuals and have expected contexts. An individual could be changed to another individual due to unfitness. As such, it is possible to realize a natural evolution of the program as long as we express the evolution of individuals properly and record evolution history about individuals adequately, and all these work can be reflected by the proposed fitness model, which makes the model meaningful.

In the fitness model, we organize the individuals in accordance with their population. Population is the universal set of individuals originated from the

same code base, which also called seed program. All individuals collected together in a population indicate what contexts the seed is based on to improve and what the results are. As time goes by, the history of the population shows the times that the successful automatic and manual evolution and the failed evolution for unfitness phenomenon. The GOP paradigm can increase the ratio of the successful automatic evolution.

The evolution built by the fitness model has several important characteristics. Firstly, the aim of evolution is to reduce occurrence of unfitness phenomenon for the population and to keep fitness for a single individual. Secondly, the evolution procedure is consistent with the enhanced level that people understand things. At last, the evolution is based on population and evolution history accumulation. Thus, our program evolution for fitness is actually a simulation of natural evolution. We adopt horizontal evolution in which individuals of same population can change dynamically, while the seed code of the population is unchanged. The significance of horizontal evolution is to reflect how much a given seed program would be improved.

The fitness model adopts such a program unit granule to represent the improvement for the program. Granule indicates which parts of the program have been improved, how to improve and what the expected context is. The individuals of the same population can live in same or varied environments, and one environment allows individuals from different populations run in. Each execution environment provides monitoring mechanism to get the unfitness appearing and make the individuals respond it. The model follows the can-be-monitored principle, that is, all the occurrence of unfitness phenomenon can be monitored theoretically.

In fact, it is impossible to monitor all occurrences of the unfitness phenomenon. However, it does not violate the can-be-monitored principle, meaning that the occurrence of an unfitness phenomena is not monitored. It is equivalent to that the execution environment has been monitored the unfitness but chooses the third response-abandon improvements, and takes these consequences directly.

## 3 Language Constructs

### 3.1 Context Variables

A context variable is an attribute used in our fitness model to characterize the environment. By determining whether the corresponding context variables are of identical values, it is easier to figure out whether the program is fit for the current environment or not. A context variable takes context information as its value, and varies over the environment.

Context variable introduced by GranuleJ is a kind of language entity different from the conventional one. It is global variable residing in multiple address space. Each GranuleJ program runs in its own address space, while context variable can be defined by multiple individual programs, even though these individuals are not

necessarily belonging to the same population. The role of context variable is to express the correlation between individuals and contexts. Accroding to program location, context variable can be divided into Read Only Context Variable (ROC Variable) and Definition-Only Context Variable (DOC Variable). If an individual declares a variable as a ROC variable, then it shows that the individual is affected by the context variable, which can cause unfitness once its value changes in the execution of the individual. GranuleJ uses ROC variables to express the expected context of individual programs. Accordingly, if an individual declares a variable as a DOC variable, then it shows that the individual publishes and maintains the value of the context variable. From the point of view of GOP, it is an expression of the real context in execution environment. From the point of view of relationship between individuals, the DOC variable of current individual is a way of impact on the fitness of other individuals.

**Example:**

```
import java.awt.BorderLayout;
import javax.swing.JFrame;
external public String envType;
public class DrawTest extends JFrame{
 ... ...
}
```

or

```
granule gLCD(DrawGraph){
external public String envType;
{
StringTokenizer st = new StringTokenizer(envType, ":");
... ...
}
```

### 3.2 Granules

A granule in GranuleJ can be declared using granule construct which is a means to express fitness explicitly. The evolution of the individuals is based on the behaviors of the granules and the relationships between them. The behaviors of a single granule can trigger unfitness, but the relationships between granules helps for localizing the unfitness in the individual.

*Granule declaration.* A granule declaration specifies the identifier and the root class of this granule, and a granule body is defined as well. The granule identifier is global, the root class is defined in seed class to specify which class the granule is attached to, and granule body includes expected context and declaration of shadow class.

The matching degree between the expected context and the real context of the granule reflects and determines the fitness of program. The context constraint

of granules is enclosed by a Java anonymous block with a boolean return type. The ROC variable defined in the current granule is used, so the matching degree can be got through the returned value from the context constraints, where *true* means match while *false* means mismatch. The fields and methods of the seed class can also be referenced in the context constraints. If an ancestor granule improves the base class, then these references will be mapped to the compound class. The reference to the fields is meaningful for the granule with single ROC context currently, which ensures the structure completeness of the granule tree.
**Example:**

```
class A{
... ...
}
class B extends A{
... ...
}
granule g(A){
external int type;
 {
   if(type > 2)
   return true;
   return false;
 }
}
```

During the individual running, the context constraints of the granule are executed only when the control flow enters this granule. The context constraints are statically constructed when coding and execution of this constraints is to detect the fitness of the granule. If false is returned, it indicates that unfitness occurs in this granule and the individual that contains this granule incurs the unfitness.

*Similar granule.* A similar granule is determined by the structural comparison of the current granule tree within the same population, and it is allowed to substitute the current granule without conflicting to the main execution flow.
**Example:**

```
granule g(A){
external int type;
 {
   if(0<type<2)
   return true;
   return false;
 }
}
```

*Granule substitution.* Granule is the basic unit that can be substituted. If the unfitness phenomenon occurs during program execution, it leads to the evolution of the individuals

The individual program constitutes with seed program and granules. Loading and substituting the granules only changes granule hierarchy but not class tree. As a result, the class tree is same for all the individuals in the same population, while the granule tree is different due to granules loading and substitution. The granule substitution in GranuleJ only occurs between similar granules, called similar granule substitution. If the substitution is arbitrary, it will greatly reduce the robustness of the program and increase the probability of evolution failure. The similar granule substitution solely supports a limited evolution, which is automatically executed by GranuleJ execution environment.

### 3.3 Shadow Classes

Shadow class is a language construct that appends code to base classes in a modular way. The essence of individual evolution is compounding and grinding shadow classes and base classes. In order to ensure the correct and successful running of individuals before and after compounding and grinding, shadow classes are required to be restricted and comply with reasonable rules. These rules ensure that shadow classes are executed correctly as an effective part of individual programs.

Shadow class specifies class and granule associated with it, and provides a shadow class body which can only contain the declaration of fields and methods. Note that these declarations should correspond to Java language specification.
**Example:**

```
class A within g{
public void toString(){
System.out.println(``shadow class A in g, and value of
type is'' + type);
}
}
class B within g{
private String type = ``person'';
public void toString(){
System.out.println(``shadow class A in g, and value of
type is'' + type);
  }
}
```

*Composite class.* A composite class is the result of compounding a base class with its shadow classes in sequence. The shadow class sequence in any composite class of an individual is in accord with the individual granule tree. The structure and behavior of a granulej program are represented by composite classes.

## 4 Implementation Framework

The language framework includes GranuleJ compiler, Context manager and Individual manager. They are integrated into GranuleJ IDE.

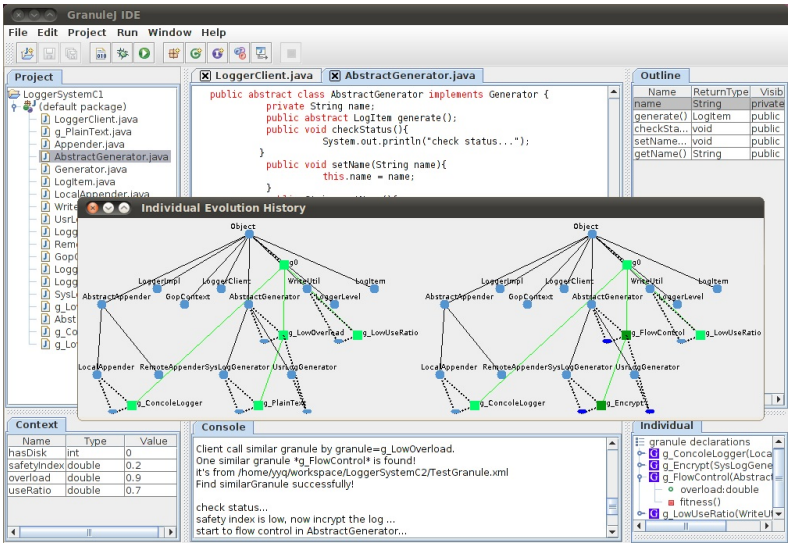


**Fig. 1.** GranuleJ IDE

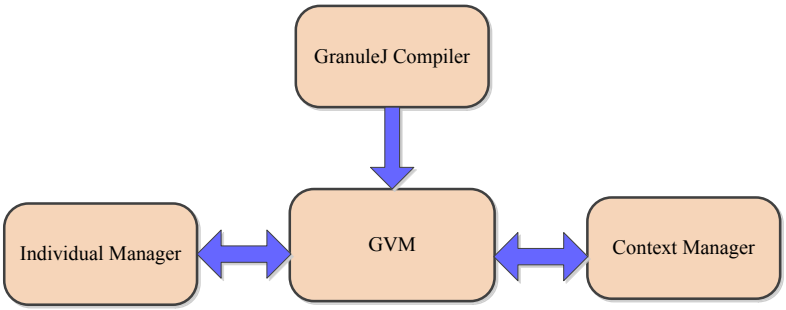


**Fig. 2.** The GranuleJ Language Framework

The compiler generates bytecode that is used by Java standard VM, context manager is responsible for publishing/subscribing and maintaining desired context information, and individual manager stores the evolution history of individuals and provides code fragments for similar granules. Note that, we simply

take fully advantage of the property of hot-swap built on of VM, without modifying it. As granule evolution involves changes to class schema, consisting of field or method additions or removals, it is required to perform dynamic class update. In addition to that, our implementation is based on lazy update strategy, which can significantly improve the response time unlike what global update does.