Engenic Inheritance: a design pattern

Abstract

A newly emerged software design pattern called Engenic Inheritance is presented. The pattern addresses concerns of genetic programming [1] [2] in both senses of machine learning and biological systems involving inheritance. The pattern is presented complete in GoF format. The pattern has features that can be viewed as dynamic multiple inheritance. The pattern does not require built-in language support, however, for the mechanism of inheritance it describes. It is intended to be implemented in a strongly typed single-inheritance language such as the Java programming language. Two implementations are described, one giving code sample and results from experimental study.

Introduction

Creating simulation models for biological processes can sometimes help to discover computational patterns that make sense in both software language design and in the subject matter [3] [4] [5] [6] [7]. Here, the subject of interest is natural biology and bioengineering techniques. One such pattern emerged from the study of how engineered genetic material may propagate through a population. The newly understood pattern is called Engenic Inheritance as it represents a kind of "inheritance". It uses a specific form of runtime self-modification engine.

This paper defines this design pattern closely following the traditional format of GoF [8]. Its potential for use in research studies involving biological processes is the clear first area of applicability and is demonstrated here. Beyond that, it seems that the pattern has applicability to any application such as optimization using genetic programming (in the non-biological sense).

> Given a behavioral entity with the capability to create an exact or similar copy of itself (i.e. an inheritance mechanism), Engenic Inheritance is the addition, change or removal of functionality that is passed through inheritance such that the change includes an operational mechanism that overcomes limitations of the given underlying inheritance mechanism.

Definition 1. Working definition of Engenic inheritance.

Examples of real-world well-known systems that appear to meet this definition of Engenic inheritance are:

1. Stuxnet – a virus-like program created for military attack on computer-controlled systems.

2. CRISPR-Cas9 gene drive - a molecular biology system for medical (disease) and environmental (pest-control) application & simulation.

The Engenic Inheritance design pattern can be implemented without any special language syntax support. A map-reduce facility would be helpful in order to make use of massively-parallel computing infrastructures.

Figures 1, 2, and 3 illustrate the Engenic inheritance pattern in steps. First, Figure 1 illustrates the core components with emphasis on the key ideas of multiple-inheritance including a self-modifying code module, but with no context. Figure 2 illustrates an evolutionary application without Engenic inheritance, intended to set the stage for the next illustration. Figure 3 shows how the components of the Engenic inheritance pattern fit into and overlay the simpler evolutionary application structure.

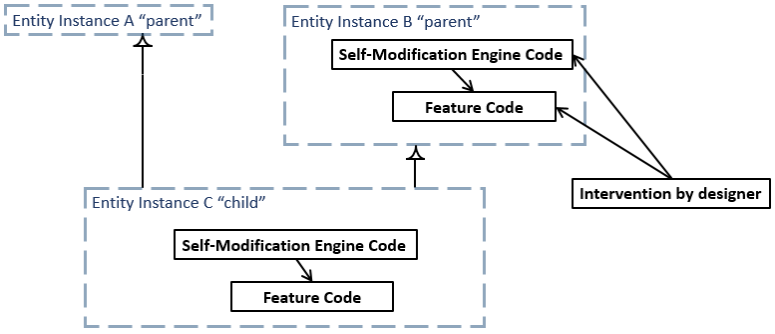

Figure 1. Core of the Engenic Inheritance Design Pattern. Behavioral Entity A and B may be one thing.

Legend.

The notation in Figure 1 borrows loosely from UML class and component diagrams.

The symbol ⍭ represents dynamic multiple inheritance. It is based on the conventional UML class diagram symbol for inheritance. Think of it as: One or several versions of matching methods are incorporated into the child class, but only one is marked Active at any one time or all may be omitted completely. This concept of holding an active and inactive version of a Method can serve to implement Mendelian traits. If the child inherits the Self-Modification component and new Feature from one parent, thus lacking the "other" version of the Method that would have conflicted, then the child class can modify itself during normal runtime operation. The result is an extra copy of the Self-Modification and new Feature components, marked inactive by default (though which copy is not important). This concept is based loosely on chromosomal recombination [9].

The boxes represent components corresponding directly to classes. The dashed boxes represent components that are installed uniquely on compute servers, one per server.

Organization of this paper.

The Introduction is followed by a long section in the traditional (GoF like) format. That section ends with two Implementation descriptions including a code sample and experimental results. Next is the discussion, future work, and conclusion.

Engenic Inheritance

name — Behavior Creational

Engenic Inheritance

Aka (may also be described as)

- Engenic Heredity
- Self-modification in genetic programming by dynamic multiple inheritance

Motivation

Distributed, massively-parallel computer networks are used for "genetic programming" applications, making it difficult to introduce a change intended to spread through the network using the replication mechanism itself.

Noise, defects, and imperfect code copying in a "genetic programming" system interferes with the propagation of a feature components. Sources of uncertainty are necessary for simulating real-world conditions. And in a real-world system are unavoidable. However, such interference can destroy new Feature components introduced by human intervention. Such introduced Features can be for research in evolutionary processes, or for security of networks, or for development of biological interventions in organisms or the environment. Thus we need means to overcome the effects of code degradation or any form of undesirable code corruption.

Genetic algorithms based on inheriting Features, often from 2 or more parent versions of a Behavioral Entity, will undergo many generations, making it impractical to use conventional inheritance. Inheritance, as in Java [10] for example, is not designed for subclassing thousands of times. Also conventional inheritance assumes that the parent(s) continue to exist (as executables, at least, and usually source is available for rebuilds). A major benefit of this traditional form of inheritance is avoidance of duplicating source or executable code for efficiency and assurance of code integrity. These benefits are not an advantage to dynamically generated versions of child classes.

The process of inheritance in the machine-learning or biological problem domain is complex and not static. Complexity arises in part from the multiple inheritance that is in the nature of the objects in that domain. It is important to avoid explicitly coding "the plumbing" of resolving duplicate methods or features, instead relying on resolution according to well-understood general principles.

Use when

Use when designing a genetic programming-based application for research or development of commercial or government machine learning systems.

Use when simulating biological evolutionary systems; to confirm or discover those mechanisms in nature.

Use when developing a biological program intended to discover or control pathogens; generate solutions in simulation, performing tests for effectiveness, reliability, and safety.

Use when a highly parallel distributed system must withstand prolonged attacks by software malware and/or extreme physical environments (such as outer space, deep sea, etc.).

Use when it is essential to avoid the "diamond problem" of multiple inheritance in a fully automated way that reflects the nature of the subject matter (objects in the problem domain).

Structure

The structure of the Engenic Inheritance design pattern needs to include the system context of the core elements from Figure 1. First the context of the system is shown in Figure 2 without the Engenic Inheritance elements. Next, Figure 3 shows the same system context but with the Engenic Inheritance core elements included and replacing a few parts of the system from Figure 2. This was done to make clear what is the core of the pattern and what is the context which, while essential, contains so many parts that it would obscure the core pattern.

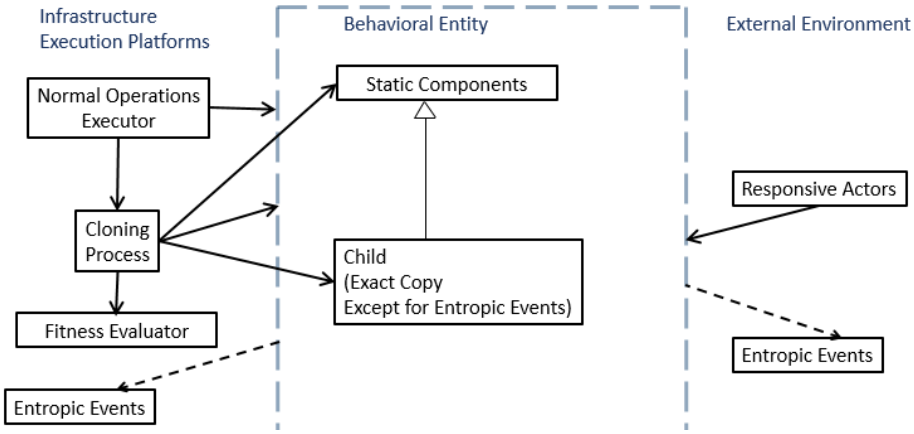

Figure 2. Evolutionary application without Engenic Inheritance

From this point forward, it is the system context with Engenic Inheritance components that will be discussed. Figure 3 has three main sections. The infrastructure, Behavioral Entity and External components. The Behavioral Entity represents the central functionality of some real-world or software system. The Behavioral Entity may be a something abstract like a recognition function. It may be factor in an optimization machine learning product, large network with security being analyzed, or a bioengineering study. Or the Behavioral Entity might be a simulation of a biological entity such as an organism, cell, or molecular chemical pathway.

The Behavioral Entity is expected to be multiply instantiated, typically at a massive scale and run on a diversity of compute servers which is part of the Infrastructure shown. The external environment is either the simulated source of stimuli and interactive response to the Behavioral Entity (instances). Or a real-world set of responsive actors or attacks.

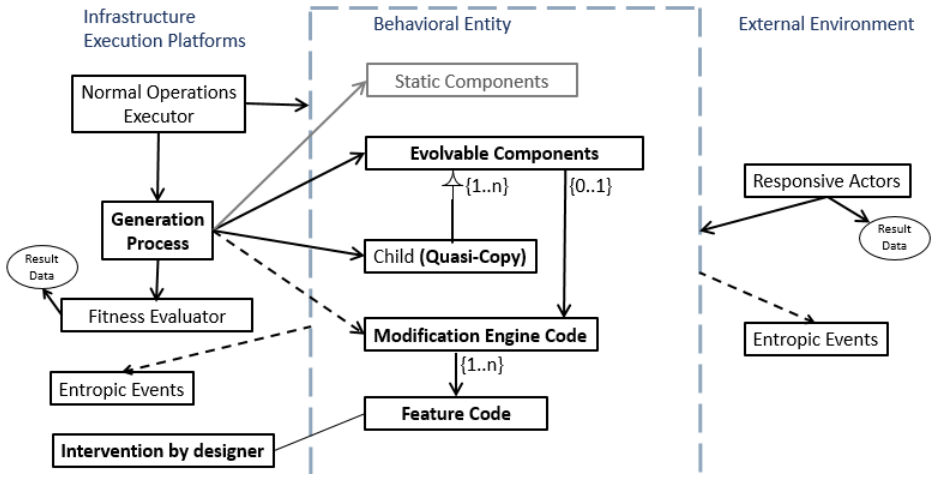

Figure 3. Evolutionary application with Engenic Inheritance

The next section of this GoF format describes each of the components summarized in Figure 3.

Participants

Infrastructure Execution Platforms

This is the massively-parallel distributed computing hardware-software infrastructure. Presumably, each compute server has a preinstalled reliable instance of the software listed (Normal Operations Executor, Generation Process, Fitness Evaluator, and appropriate interfaces for Intervention by Designer. When this system is realized as a simulation, input for inputting simulated Code Corrupting Events. In practice, these arise naturally due to causes listed under the Participants – Code Corrupting Events section.

A compute server is also called a node in this context, for short. An analysis program can expect numerous nodes to be available for scale up. In practice they may be virtual, on a small cluster, or in a data center or a large network of node on the internet. They are expected to exhibit a failure rate that is large enough to have an effect on the behavior and performance of the distributed application.

It is expected that a diversity of different equipment and versions are available and dynamically brought in and taken off-line without notice. It is expected that network connectivity among nodes will exhibit varying connectivity: rate, drop-outs, availability. Each node has enough local storage to continue running its Behavioral Entity instance stand-alone, until it attempts to generate and deploy a newly generated instance. It may also rely on the network to load software components dynamically, possible sources of failure.

For simplicity, each participating compute server runs one instance of the Behavioral Entity.

Normal Operations Executor

An application container along with the operating system and storage infrastructure. It is dedicated to run exactly one Behavioral Entity instance which is deployed to it by another node. Upon termination of the Behavioral Entity the node that it was assigned to is freed and is then available again to run another newly generated Behavioral Entity instance.

Generation Process

This component is of central importance to this design pattern.

It generates new versions of the Behavioral Entity based on this algorithm:

```
Select two or more existing Behavioral Entity instances (or one if only one is available). Make the
selection according to a given criteria based on the Fitness Evaluator (which is determined by the
researcher/developer).
```

```
For inheritable features (think of these as Method components), perform dynamic multiple inheritance: copy both
[all] implementations of one Method (having identical signatures) from each “parent” instance that would normally
cause a conflict. For each pair [group] of Method components in conflict, resolve the conflict by randomly picking
one of them and set it as “the active” Method of its kind. Thus, the completed Behavioral Entity instance will have
a combination of Method components that could often be unique in throughout the analysis run.
```

Further explanation of method substitution concepts. Unlike conventional inheritance, an *engenically* inherited Method component has independent existence from the parent class. It is in effect a clone, not merely a reference to the code resident in a base class. As such, the parent class could be wiped out of existence, source and executable, or modified, without affecting any child class that previous inherited from it. Any data elements used by the inherited Method must also be cloned in order for the Method to function. In the case where several Methods are designed to work together in both of the parent classes, the naïve Engenic inheritance algorithm could break up the set. So, for example, the set of Methods from one parent operate on an Integer while those from the other parent all do the “same” thing but on a Float. These methods can’t reliably be mixed. But even so, under the massive genetic replication scenario, many of the generated variants of the original parents would just happen to Activate the “operons”, or sets of Methods that work together by design.

Compatible method signatures. For simplicity of introducing the concept of Engenic inheritance we start by assuming exactly matching method signatures resident in two parent classes. But this constraint is not necessary and not fundamental to Engenic inheritance. Clearly, if a new syntax were to be introduce to a language such as Java to support Engenic inheritance, it would be natural to recognize identical signatures by default. Then some kind of language feature could allow programmers to explicitly mark distinct methods as intended to be engenically substituted, allowing different base name, parameters, throws, and return. That is beyond the scope of this paper.

To further clarify, upon completion of a new instance from, say, two parent instances, the new instance will have 2 versions of each dynamically inherited Method, one marked Active and the other(s) marked Inactive. Both of these are available for dynamic inheritance in the next generation (the next generation instance could inherit the Inactive Method or the Active one, chosen randomly). During normal operation of the Behavioral Entity only the Active Method code is executed. “Normal operation” here means functionality other than generating new instances.

The Generation Process finally deploys the newly generated Behavioral Entity instance to an available compute server. Presumably, the rate of generating new instances is determined by the researcher/developer.

Fitness Evaluator

This component provides success metrics. The effectiveness of the Behavioral Entity is measured by observing the reactions of the Responsive Actors during normal operations of the Behavioral Entity.

The summary of these metrics are available to the Generation Process which is expected to use these results in its selection of variations applied to the Evolvable Components.

The Generation Process also retires (uninstalls) low scoring Behavioral Entity versions, thus freeing up compute servers for new generations.

Code Corrupting Events

In the infrastructure…

Any hardware or software event that modifies code or the data of data-driven algorithms in ways that are not intended or expected by the Behavioral Entity as designed. For research using simulations, these would likely be [intentionally] artificially generated as putatively "unintended" events.

All parts of the Behavioral Entity are sensitive to, or vulnerable, to these events, not just the Evolvable Components.

Generally speaking, such changes to the Evolvable Components are of interest to the evolution of the Behavioral Entity because they have the potential to introduce behaviors that are non-fatal.

* servers, storage, and communication devices shutdown during execution.

* operating system updates in progress

* logic errors, viruses, interruptions for maintenance

* intermittent device failures due to heat, age, spikes

Intervention by designer/ researcher

The researcher or designer of the Behavioral Entity will occasionally introduce a few component, the Feature Code component.

The new code can be anything. It may or may not be related to existing code in the Behavioral Entity.

Presumably, the new Feature Code is something that could never be developed through normal evolution of the Behavioral Entity by the work of the Generation Process and related components.

Also, presumably, the new Feature is something that might evolve after deployment in the Behavioral Entity throughout the massively-parallel infrastructure of compute servers.

Finally, the new Feature Code may be introduced to one, or just a few, instances of the running Behavioral Entity rather than some kind of broadcast mechanism of the infrastructure.

Behavioral Entity

This is the component that represents a biological or conceptual entity that has interactive behaviors and can be evolved in response to its effectiveness, or "fitness". In a genetic programming system for discovering optima in a high-dimensional system (such as face recognition, say) this Behavioral Entity might involve a static set of behaviors operating on a dynamically evolving set of values. However, in a simulation of biological processes, these Behavioral Entities might evolve based on mixing different Methods (with their particular data that might vary as well). The dynamic multiple-inheritance of Methods (representing Features) is the case we are interested in for Engenic Inheritance.

Static Components

These components are designed to not be changed as the Behavioral Entities evolve.

Generally, these components are so fundamental that the Behavioral Entity will abort or loose critical functionality if they are corrupted. There is nothing about this design pattern that mandates that the Behavioral Entity should fail if static components evolve in a way that is non-fatal. But these static components would not be subject to self-modification with new Feature components.

Evolvable Components

These are the main components of interest to the research study, or commercial enterprise using machine learning in its products.

Other genetic programming systems have described the concept of evolvable components which are varied by an algorithm for generating new versions of the Behavioral Entity.

> What differentiates the Engenic Design Pattern is the use of a Modification Engine that will execute under the as part of the normal operation of the behavioral entity.

This design feature is a way to *drive* new "gene" components through a population that might ordinarily not be selected for based on mediocre Fitness scores.

Child

Simply stated, the Child components are Behavioral Entity instances derived from the parent Behavioral Entity instance(s). The "derivation" process is recursive.

To explain the Child component more fully it is necessary to explain how the inheritance works in this design pattern.

An instance of the Behavioral Entity can be thought of as an individual.
Just as in conventional Object-Oriented languages' inheritance, a child object is
instantiated by taking some features from a parent or parent objects.
The child object inherits some combination of features from the parent individuals and becomes an individual, instance of the Behavioral Entity itself.

Unlike conventional inheritance in an object-oriented programming language,
the combination of features is determined by recombination of the features available
in the parent individuals at the time of instantiating the child object.

The particular choices for recombination may be random or some algorithm of interest.
As a software concept, all individuals (hence, child objects) are like Singletons. Although, there is no need to guarantee that their feature combination is unique.

Also unlike conventional inheritance, the features of each (possibly multiple) parent individual are
physically copied and retained in the child individual.
When there are two or more "matching" features (e.g. Methods with identical signature),
then one feature is marked as "active". That is the Method that is called when the individual is exercised.

Subsequently, when it is time for a child object to become a parent,
the recombination process includes all of the features, both active and inactive versions
of matching features into the child object.

Finally, when an individual is deleted any and all of its previously-created child individuals
continue to have the features they inherited. This too is different from conventional inheritance of OO programming languages in which the entire inheritance hierarchy must continue to exist as child objects are instantiated and destroyed.

Modification Engine Code

This component is introduced by the researcher or developer while the system is running (the system being all of the instance of versions of the Behavioral Entity). It is intended to be injected into one or a few instances of the Behavioral Entity. The Modification Engine is then executed by the Behavioral Entity as part of its normal operation. Unlike “normal” methods, however, it modifies the code base of the Behavioral Entity itself. Specifically, if it determines that it is the only copy of itself in the Behavioral Entity, it proceeds to make a clone of itself and the Feature Code component and install them as the inactive alternative (which would have been present if this Behavioral Entity instance had been generated from two parent classes both already having a copy). The implementation of this “code change” can be accomplished by a variety of mechanisms including direct code rewrite, if the language is Perl or some interpretive language, or by collections of features or runtime source or bytecode editing, or dynamic loading by JEE containers.

Feature Code

Software designed by a researcher/ software developer. It is expected to be something that will introduce a new capability to the Behavioral Entity and which would not be expected to emerge through the evolutionary process of the Behavioral Entity and interaction with its infrastructural components.

Unlike conventional rules for method overriding, Features are expected to be implemented by methods that often but not always have compatible signatures. What is more important is the type of product produced which reflects a common concern of features that should be considered matching.

In the context of an application that directly simulates biological replication, the DNA address is also a criterion of determining whether Features are matched.

External Environment

In a simulation for research purposes, the external environment provides stimuli to the network of Behavioral Entity instances. In a real-world deployment of a system based on the Engenic Design Pattern, the external environment is the physical world that the Behavioral Entity instance nodes can interact with, plus the physical infrastructure on which the nodes execute and communicate.

Responsive Actors

These are sources of input and response from the external environment to the nodes of the system.

Code Corrupting Events

In the external environment …

Any impact by responsive actors or unforeseen interventions from outside the system. For research using simulations, these would likely be [intentionally] artificially generated as putatively "unintended" events.

All parts of the Behavioral Entity are vulnerable to these events, not just the Evolvable Components.

Generally speaking, such changes to the Evolvable Components are of interest to the evolution of the Behavioral Entity because they have the potential to introduce behaviors that are non-fatal.

* computer viruses, in the case of simulations.

* actual viruses, viroids, preons, or parasites, in real-world biological implementations

* extreme physical conditions such as heat, radiation, electromagnetic phenomena, etc.

Collaboration

This section of the GoF format includes an Activity Diagram describing how Behavioral Entity instances are generated by the Generation Process. Each of the Activities is described in details.

Summary of the process

The Activity diagram, Figure 4, depicts how new generations of evolvable nodes are generated. Here, the **Instance**s of the Behavioral Entity correspond to the evolvable Behavioral Entity class instance objects shown in the previous Figures, named so for brevity here. Upon starting the system with at least one "hand-coded" **Instance** node, the **Generation Process** derives clones, each presumably slightly and uniquely varied by forces described earlier (Code Corrupting Sources), and puts them into operation on available compute servers. As described before, The **Generation Process** takes two **Instance** nodes and combines their alternative corresponding heritable components at random. Some time later, at the discretion of the researcher and software engineer, an **Engenic Feature Component** is given to the Process to inject into an **Instance** node being generated. The **Generation Process** treats the new component no different from any other heritable component. That is to say, an arbitrary **Instance** Node is copied as before except with the newly injected component added to it. Typically only one or a few **Instance** Nodes get the **Engenic Feature Component** injected as a direct result of the Inject. Afterward they propagate it by running the **Engenic Feature Component**.

At that point, the newly generated **Instance** node, now with an Engenic component, is put into operation. As part of its now-normal operation, the **Engenic Feature Component** modifies its host node. The modification inserts a copy of the **Engenic Feature Component** itself. So when the **Generation Process** next tries to randomly select the alternative Method corresponding to the **Engenic Feature Component**, it will not find one that happens to be identical. The selection will not be "aware" of the fact that it is identical; inheriting alternatives that happen to be identical will happen by chance. The result is that any further generation will inherit the new **Engenic Feature Component**. In other words, the usual random selection of dynamic inherited alternatives is thwarted. The **Generation Process** is "unaware" of this change and continues its work as normal.

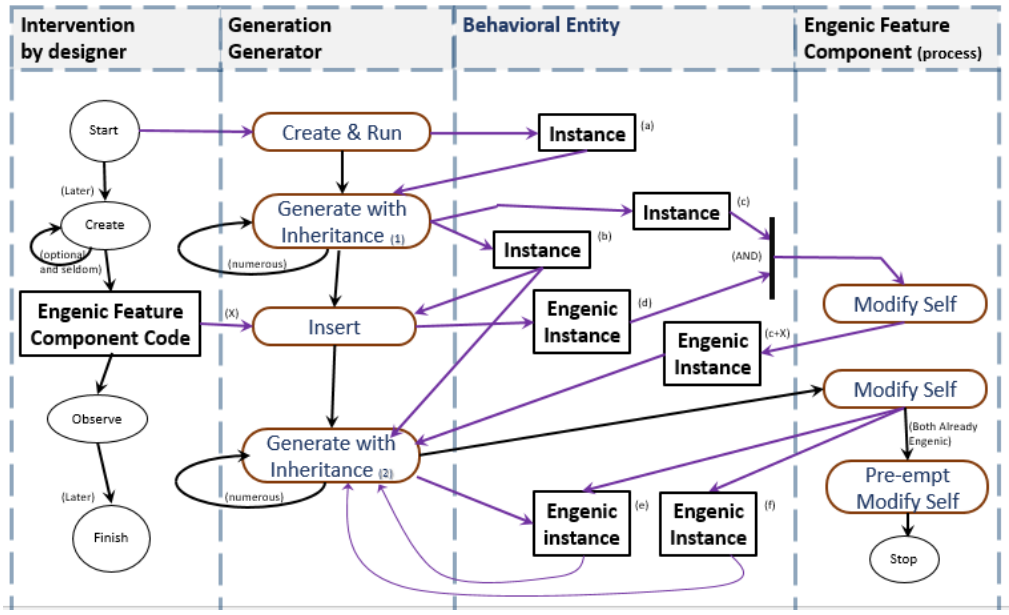

Figure 4. Engenic Inheritance initialization and normal operation – UML Activity Diagram.

The Activity diagram shows the Generate Process at two points along the way. The first, subscripted (1) generates Instances without an Engenic Feature. That is, the system has started but no Engenic component has been injected yet. Over on the right, the Modify Self activity can only begin when it has both an normal Instance AND an Engenic Instance.

The Generate Process subsequently receives an Insert command along with an Engenic Feature Component. It injects that component into an arbitrary Instance that is already running somewhere on the network. The Modify Self activity (top right) is what the Engenic Feature Component does when the Instance happens to execute it. The Modify Self concludes with its host Instance having an additional copy of the Engenic Feature Code.

Finally, the Generation Process takes any combination of Instance and Engenic Instance and produces Engenic Instances, thus propagating the modified version of the Behavioral Entity. Note that the entire population of nodes still has a lot of unmodified Instances. They will continue to operate until any scheduled or detrimental event causes them to terminate.

Details of key components

Generation Process

The Generation Process generates new versions of the Behavioral Entity based on the effectiveness and some random effects of each instance of the Behavioral Entity, and starts an instance of that unique version on a compute server.

The new versions will contain components from one or two (or a few) Behavioral Entities currently operating on the infrastructure.

As new generations are generated, they are deployed to newly available compute servers.

All are subject to new generations of the Behavioral Entity.

The Generation Process is responsible for making a copy of the current Behavioral Entity code base, and its dependent library components.

This "copy" includes evolvable components that the Generation Process intentionally varies.

The variations are influenced by the results from the Fitness Evaluator.

Some versions of the Behavioral Entity may be copied without deliberate modification, such as when the Fitness Evaluator return a high score for it.

The evolvable components will often include a copy of the Modification Engine Code component which in turn brings along a copy of the new Feature Code component.

Initialization: The system can be started with one instance of the Behavioral Entity (running on one compute server). The Generation Process will recruit available servers as it generates new copies of the Behavioral Entity.

Single parent/progenitor generation:

The Generation Process may be designed to create the Child Behavioral Entity based only on one Behavioral Entity. Thus, no mixing of components from different versions of the Behavioral Entity's components.

Using the same propagation mechanism any new Feature Code will naturally spread through the population of Behavioral Entity instances, provided that associates with a sufficiently high Fitness score.

Dual or small number of parent Behavioral Entities:

The Generation Process may be designed to always generate new Child Behavioral Entities by using a mix of components from more than one parent.

Presumably, the number of such parents will be fixed at 2 throughout the experiment or commercial run. For simplicity, the rest of this paper assumes 2 because the interesting use of the Engenic Design Pattern involves more than one parent, while 2 is also enough for discussion.

Consequences

The Engenic Inheritance design pattern defines a form of inheritance that departs from other mainstream software and bioengineering forms of inheritances.

As a software construct …

The design pattern for Engenic Inheritance resembles multiple inheritance and uses runtime self-modifying code. However, the specific conflict-resolution formula is distinct from that of C++ or other static (compile-time) multiple inheritance approaches. The decision among same-signature Method objects is not final at instantiation time. Both versions are copied into the new instance object. One of the matching Methods is marked as active. That is the Method invoked when the analysis or simulation calls for it. This structure makes it possible for parent objects to be destroyed while allowing descendants to continue to exist. This structure also allows the inheritance process to see both the active and inactive versions of Methods. It is possible that a

child object will inherit an inactive Method from a parent object and then that Method might be marked active in the child object.

This inheritance infrastructure supports the biological mechanism of gene recombination and inheritance. However, it is designed here to also support implementation of mechanisms other than Mendelian and non-Mendelian heredity [11]. As such, analysis and simulation of systems other than biological systems can be implemented on the same infrastructure.

The self-modification capability is the defining characteristic of Engenic Inheritance. It is a mechanism not found in Nature.

As a bioengineering construct …

Biological Engenic Inheritance resembles the recently-developed technology called Endonuclease Gene Drive [ref]. This was the intention of this paper from its inception. The terminology used in this paper is geared toward the software community. Some of the software ideas can be recognizable to the Biology (biotech, molecular biology, etc.) community in their vocabulary.

Feature components, or "methods" in object-oriented terminology, and their necessary data correspond to genes.

The states called "Active" (and "inactive") correspond how genes are silenced or not.

The generation process corresponds to how sexual reproduction takes place at the level of chromatin.

The special self-modification component with a Feature payload corresponds to endonuclease gene drive technology. Incidentally, this design pattern is not limited to the particular approach to "gene drive" which is based on endonuclease. It is sufficiently general to cover other approaches that may soon come about, approaches that might be based on mitochondrial genes which naturally drive to all offspring, or retrotransposon-based technology, or by persistent plasmids, mini-chromosomes, or simply highly selected-for phenotype-driven genes. Note that "*the **terms dominance and recessivity characterize phenotypes, not genes**. Genes do not have the dominance and recessivity character*." [12] [13] It seems likely that additional techniques will be discovered/invented in the future.

Known Uses

This design pattern is find a commonality between naturally-occurring inheritance, as well as the kind of inheritance developed in the world of bioengineering, and in machine learning .e.g. in biology, we need to know: 1. what biological system, 2. what perturbation/treatment, 3. what measurements.

Question: is it possible that we might create a doomsday organism? one that just won't stop and will kill other species such as humans?

Related Patterns

The Abstract Factory GoF design pattern seems to be somewhat related in the sense that different sets components may replace a baseline version of the system during runtime. However, that set is generally fixed and complete with methods or classes with a predetermined place in the abstract design. The infrastructure of Engenic Inheritance does not predetermine any components that make up the growing list of generated individuals.

The Engenic Inheritance infrastructure might read on the Decorator design pattern for the purpose of inserting Features into a new instance of an application component [8]. *Design Patterns. Reading, MA: Addison-Wesley Publishing Co, Inc. pp. 175ff. ISBN 0-201-63361-2*.) It differs in two ways. The Decorator pattern is not concerned with how and why Features are selected. Features are not automatically inserted at instantiation. Features and not thought of as mere decorations but as definitive of the instantiated individual. In contrast, Engenic features are concerned with modifying the individual's set of features. This functionality is not specifically supported by Decorator.

Implementations

Two of the known implementations of Engenic Inheritance are briefly described. The first is a simple example based on the Singleton design pattern as a framework. The second demonstrates a rather complex functionality based on biological processes.

Computer Science example implementation

An implementation of the Engenic Inheritance design pattern was developed and excerpted for this paper. It exercises each of the components at a "minimal" functional level. To motivate the design of this prototype an experimental design was posed. The experiment seeks to exhibit the different rise and fall of a hypothetical population of organisms comparing ordinary "Mendellian" heredity with Engenic heredity.

The java stream facility is an appropriate infrastructure for designing applications where a set of behavioral entities can execute in parallel if the underlying hardware supports it. Additional layer of stream functionality can be useful to support the exponential propagation of new and interacting entities. For this purpose, an implementation of Delta Streams [14] is currently

being explored. To show where this facility might play a role in and implementation of the Engenic Inheritance design pattern, the demonstration code included in the supplementary section of this paper shows a stub-out of the DataStream interface.

Implementation Example 1. Singleton-based Behavioral Entity

The Engenic design pattern is illustrated in this section. This example is based around the Singleton design pattern. The BehavioralEntity itself is implemented as a Singleton that (1) does some pertinent work, (2) propagates itself with some variations. To illustrate the Engenic concept and how it might be useful, a detailed sequence diagram is presented, Figure 6. But first, the structure of the Singleton embodying a BehavioralEntity is given in Figure 5.

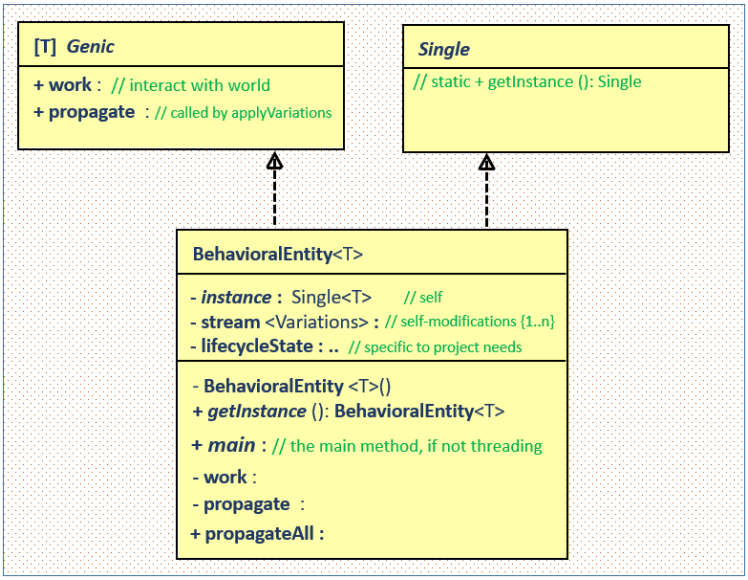

Figure 5. UML class diagram of a Singleton-based design of BehavioralEntity. .

Clearly, the simple class diagram does not capture any of the self-modification and self-propagation functionality of Engenic inheritance. Figure 6 builds on the concept diagram at the beginning of this paper, Figure 3, by overlaying the graphical elements of a UML class diagram. The diagram attempts to show how each of the conceptual components (Figure 3) are turned into software components by placing corresponding concept and UML class components in close proximity, with the concept "behind" the design element. It should be noticed that none of the conceptual components of Engenic inheritances shown earlier are missing from the Singleton-based implementation suggested here.

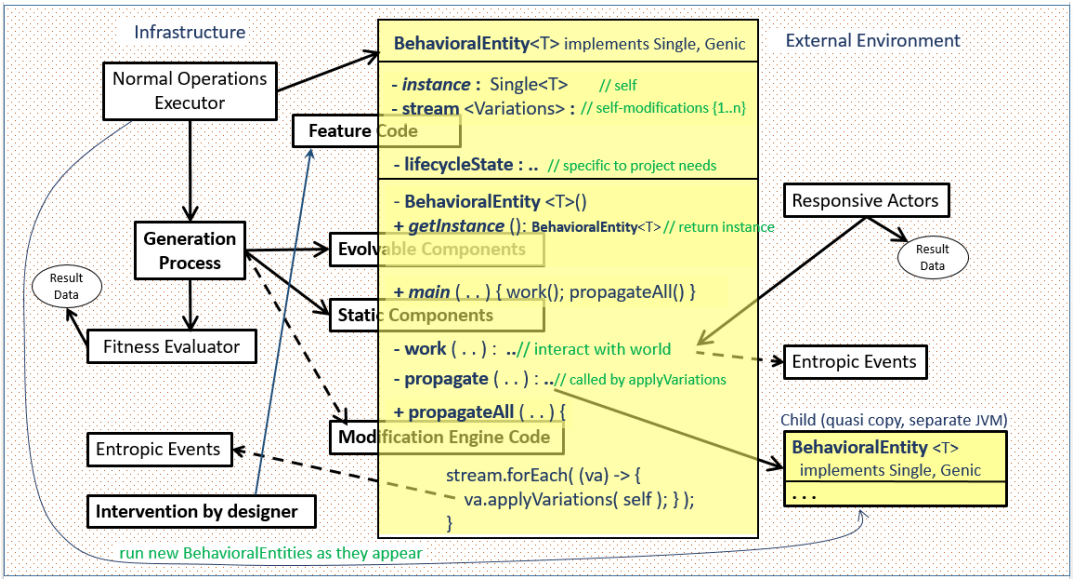

Figure 6. Mixed concept and UML Class-like diagram.

In order to keep the portrayal of Engenic inheritance general and not tied to any particular implementation approach, some liberties are taken with the UML Class diagram graphics.

The basic flow of control is shown by the UML Sequence diagram in figure 7.

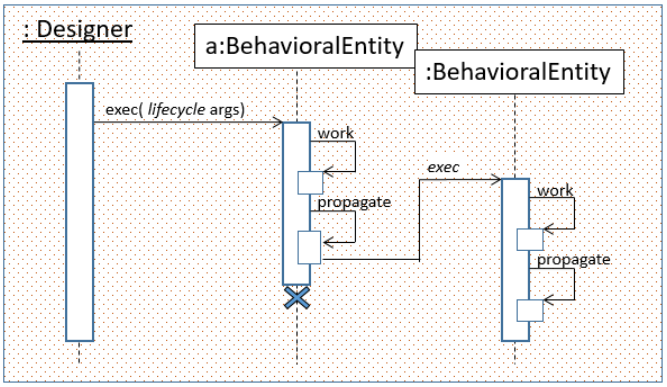

Figure 7. sequence of basic inheritance of behavioral entities.

Although this sequence diagram is technically a proper sequence diagram, the static nature of UML fails to bring out the dynamic self-propagation feature-modification picture. The UML sequence diagram notation is also weak in depicting a system in which the population of instances grows, perhaps exponentially.

A UML Sequence-like diagram, Figure 8, builds on the given inheritance mechanism and illustrates all of the functionality of Engenic inheritance.

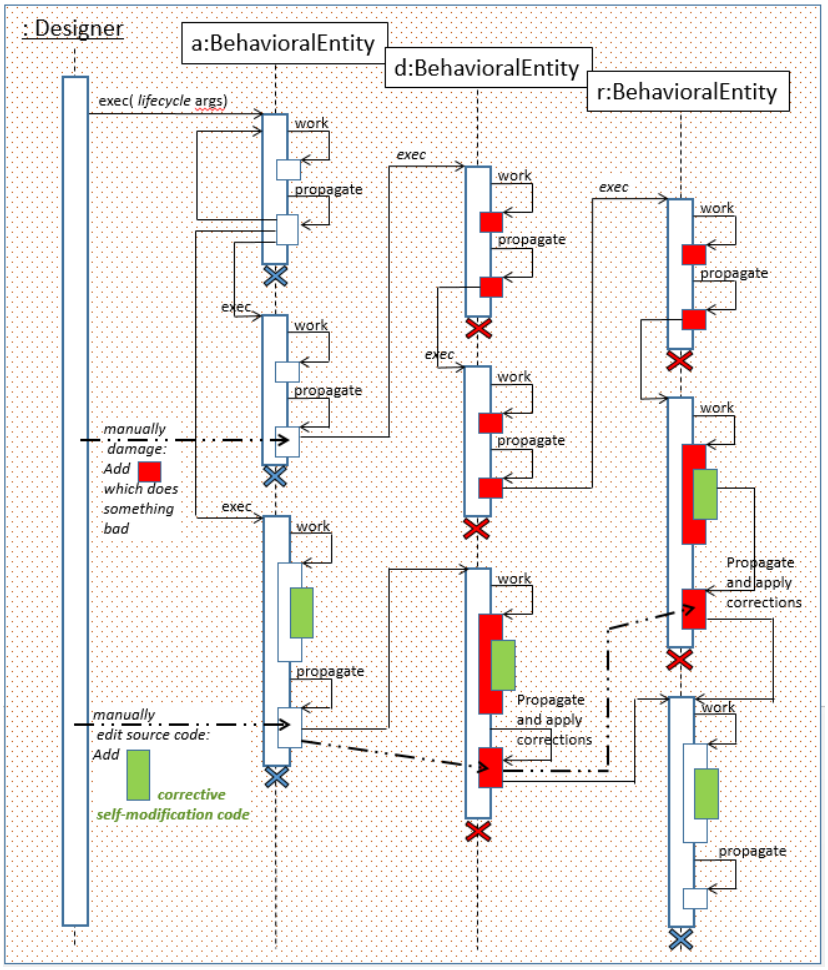

Figure 8. Singleton-based Engenic Inheritance, UML Sequence-*like* diagram.

Legend. diagram elements that diverge from standard Sequence diagrams are:

(1) source code modification by manual or self-modifying code is indicated by dashed arrow. ·· — ··→ This arrow does not imply flow of control or invocation.
(2) "exec" – each "exec" implies typically multiple invocations, yielding multiple instances.
(3) Types a, d, and r BehavioralEntity mean a: initial "normal" entity, d: "damaged", r: "repaired"
(4) red indicates that normal function is defective or deleted. green means self-modifying corrective code.
(5) X at the bottom of a lifeline means instance exits. if red, ability to exit is defective or missing.

As was pointed out earlier, the concept of Engenic inheritance does not use "extends" or subclassing as in conventional procedural programming languages. Instead, the inheritance is given by transmitting state and functionality to new copies of existing instances of an entity.

In keeping with the understanding, the Sequence diagram shows a number of lifecycles on each of the three BehavioralEntity lifelines. Here is a high-level walk-through.

The Designer is a person or team performing some kind of research or production development. In the diagram, the Designer is seen first creating one or more "normal" BehavioralEntity" compilation units. Each runs in a separate environment, here it is assumed to be a separate JVM although some users could just use threading.

At some later time, the Designer manually edits the source code of an instance of the BehavioralEntity, chosen arbitrarily. This trick is accomplished by the prototype implementation in java based on Figures B-2 and B-3. (This is a source-code level procedure; no byte-code knowledge required.) Listing B-1 shows top-level excerpt of code included in the supplementary material.

The damage does to the victim entity can be anything including changing the work it does, how it propagates itself, and the self-termination behavior. For example, to model a cancer cell line, we might interfere with the normal apoptotic behavior of an entity so that it keeps propagating and not going away.

The simple procedure in Listing 1 is one way to implement the inheritance mechanism that can support Engenic inheritance. This code snippet does not include how the dynamic intervention would be performed. Presumably, in this simple example, the series of Command objects defined in the variations stream would be interfered with manually or by another behavioral entity that has already been given the capability to modify the workings of its neighbors.

```
try {
    final BehavioralEntity self = getInstance();

    self.variationsStream.forEachOrdered((variations) -> {

        self.work(); // optionally, interleave work of project with propagations

        variations.setContext( self);
        Iterator<Command> it = variations.iterator();
        while (it.hasNext()) { // propagateAll() may be used
            it.next().run(); // run each command comprising propagation
        } });

} catch (Throwable t) {
    // react to exception halting all work and propagation as required
    ...
} finally {
    . . . // for example:  Population.delete( self);
}
```

Listing 1. Singletonic Engenic code excerpt.

Finally, after a number of damaged entities proliferate, the Designer can perform another edit on either an affected or normal entity. This edit introduces a new piece of code and/or state into a running entity. That is a piece of code that is capable of modifying the entity it is then part of, and presumably make corrections of some kind. When that entity propagates, it includes the new code as if nothing unusual has happened. All of its descendants run the repair and become healthy once again. The self-modification feature should be designed so that if it finds itself in an entity that has already inherited it and its repairs, it will not attempt to do the repair again.

Biological perspective.

For readers interested in biology, we might notices that some of this concept of new code inserted to make corrections is reminiscent of CRISPR-Cas9 [15] or the recent Cpf protocol [16]. While that is true, these gene engineering technologies are based on delivery mechanisms that use a large quantity of a viral or similar "vector" to travel to cells throughout the body. That approach has several drawbacks, dangers, and costs.

While it is quite possible that researchers using gene engineering have already thought of using the approach suggested here, conceptually illustrated in Figure B-3, no references to that effect have been found in a recent literature search or discussions.

Applying Engenic inheritance as in the present Singletonic example would depend on the ability of affected cells to transfect their immediate neighbors with the self-modifying pathology-correction package. That would eliminate the need to grow a large quantity of engineered foreign vectors for interventions intended to cure individuals. Like these biomedical approaches, the Singletonic Engenic approach is, by design, applicable to individuals rather than populations. Another Engenic approach in the next section that is applicable to populations.

Another important difference is that conventional techniques based on CRISPR-Cas9 and Cpf target the genome of the host. That introduces dangers because it is irreversible (unless it is also engineered to be epigenetically switchable, as with TET for example). Also, the genome-wide modification increases the number of opportunities for a pathogenic adaptation, rendering a loss of efficacy in individuals and population.

In contrast, the Singletonic Engenic inheritance approach is not limited to editing the genes. As mentioned in the walk-through, a modification of state can be performed (instead or in addition to code changes). The biological analogy to this approach is the heritable epigenetic modification such as is done with heritable methylation, chromosomal imprinting during early embryonic development, and other things affecting transcription factors.

Conclusion of Singleton-based example of Engenic inheritance.

A very simple example of implementation of Engenic inheritance has been shown. It shows how a single entity might propagate itself into a larger population, presumably each with variations of some core functionality. This is one approach in which the use of Singleton was considered convenient and meaningful to the interests of some application development project. Other design patterns can be useful for other purposes. One can imagine that patterns like Composition, Decorator, Abstract Factory, say, would be the best framework for the purposes of other projects.

The simple example, while representing all of the conceptual components of Engenic inheritance, does not by itself compel us to see how this pattern might play out when the core functionality is much more complex. For that purpose, the next example simulates recombinant replication based on living systems.

Implementation example 2: Bioengineering-related

The preeminent work that has achieved what we call Engenic Inheritance is that of the Andrew James lab at UC Irvine this year. It is called Endonuclease Gene Drive.

At the time of this writing, no software enterprise has come to public attention that could be considered to include a form of implementation of Gene Drive or Engenic Inheritance.

Notwithstanding the problem of being on the leading edge of a technology, the following code is offered here as a minimum sufficient example of a system based around Engenic Inheritance.

The term "Engenic Heredity" would be more descriptive of the design pattern when talking with members of the biological science community, simply to emphasize that the "inheritance" function is what they normally call heredity.

This concludes the GoF formatted presentation of the Engenic Inheritance design pattern.

Nomenclature

The mechanism and consequent effect of Engenic Inheritance is neither Mendellian nor non-Mendellian. The term "non-Mendellian" does not mean "anything that isn't Mendellian". The terms Mendellian and non-Mendellian address what kind of effect that a hereditary trait exhibits, not how the underlying mechanism of replication and recombination work. (Effects are seen as polygenic and multifunctional traits.) The term Engenic heredity, in contrast, addresses the underlying mechanism of replication, which is distinct from that of Mendellian and non-Mendellian inheritance. Consequently, Engenic Inheritance is not accurately described as non-Mendellian.

There is precedence for defining "Engenic Inheritance" as a form of inheritance in software and biology. It is independent of Mendellian (and non-Mendellian) inheritance, consider genetic imprinting. The term Genomic imprinting was introduced to describe the phenomenon of genes being expressed in a parent-of-origin specific way. The allele imprinting from parent A is silenced and that of parent B is expressed. About 75 imprinting genes were known in H. *sapiens* as of 2014. To summarize, "*Genomic Imprinting is an inheritance process independent of the classical Mendelian Inheritance*" [*https://en.wikipedia.org/wiki/Genomic_imprinting*]. Similarly, "Engenic Inheritance" is distinct from that of classical inheritance involving nesting, delegation, and extension in computer programming languages." See also [17]

Finally, consider abnormalities in any inheritance process. We might reasonably ask, in passing, whether Engenic Inheritance itself is a form of abnormality. Abnormalities include mutation (change, gain, loss of DNA sequences), extra or missing chromosomes. While EI could introduce such errors the process itself is not deemed an abnormality. An example of abnormal inheritance is a condition known as uniparental disomy (UPD). In UPD, the child receives two copies of a particular chromosome from one parent and no copy of that chromosome from the other parent, usually with no noticeable effect [18].

Sample Code

Code Listing 2 illustrates some of the main steps in an analysis using the design pattern.

```
// bootstrap behavioral entities
        String[] tagNames = {"A/a", "B/b"};
        for (int i = 0; i < tagNames.length; i++) {
            analysis.createNormalType(i, tagNames[i]);
        }

// bootstrap initial population: two normal no-action types, one normal terminator (no engenic types)
...
        List<BehavioralEntity> newGenerationChildren = new ArrayList<>();

// run all four swim-lane activities in pseudo-parallel under clock control
        do {
// bring new children into infrastructure that were created in previous timepoint
            newGenerationChildren.forEach((c) -> Infrastructure().add(c));
            newGenerationChildren.clear();
// propagate new children
...
            for (int j = 0; j < nParents; j++) {
                BehavioralEntity parentA = removeRandomIndividual(availableParentEntities);
                BehavioralEntity parentB = removeRandomIndividual(availableParentEntities);
                List<BehavioralEntity> children = replication.propagate(parentA, parentB);
...
            }
// do normal living activity
            /** regulatory decisions are made for all individuals and saved in
             * expressFeatures. next these features are expressed */
...
                List<FeatureGene> allFeaturesOfIndividualRegulatedOn //
                        = individual.regulateAll();
                putAll(allFeaturesOfIndividualRegulatedOn, individual, expressFeatures);

// the infrastructure now runs the express() of the Features that were regulated.
            // this is where individuals might be terminated
            /** tag messages for monitoring from Features that emit them */
            expressFeatures.entrySet().forEach( // may use stream
                    (Map.Entry<FeatureGene, BehavioralEntity> feature) -> {
...
                        tagsAllExpressedFeaturesDetected.add(expressedFeatureTypeTagsDetected);
                    });

            removeIndividualsTerminated();

            UtilMonitorSensorReading.collectTagsDetectedAtTimepoint(tagsAllExpressedFeaturesDetected);

            if (Infrastructure().timePointForEngenicModificationInjection.get()
                    >= Infrastructure().getTimePoint()) {
// create Engenic Feature Component Code
                BehavioralEntity targetZero = getRandomIndividualAlive();
                ChromosomePair ch = targetZero.getChromosomePairById(targetChromosomeId);
                FeatureGeneEngenic engenicFeature = this.createEngenicFeature(ch);
            }
        } while (!scenario.exitAnalysis());
```

Listing 2. Excerpt of analysis program using Engenic inheritance.

Experimental results

The prototype sample code was used to generate time course data based on *hypothetical population parameters*. *This analysis and the associated charts do not represent real-world measurements, organisms, or natural behaviors. This experiment only illustrates the potential use of the programming feature of Engenic Inheritance as it might be applied to computational biology projects. Data was chosen to dramatize the effects visually.*

In this experiment the Behavioral Entity represents a mock mosquito with a certain fecundity level and rate of predation. There are no Code Corrupting Events, however, the environment deletes individuals from the population according to a mock predator which also has a rate of reproduction.

The experiment starts with 2 gray-eye individuals. Eye color is the Feature of interest. After a few generations, 2 red-eye individuals are introduced, and after a few more generations 2 predators are introduced.

In the control case, the red-eye individuals are transgenic Ae. *aegypti* artificially given the Red gene and allowed to reproduce through normal Mendellian heredity. This feature is not Engenic; just ordinary gene editing. In the experimental condition, an individual is introduced to the population that is given the self-modification as well as the Red feature code in its germ line. This is the Engenic inheritance component. In this computational experiment, that individual propagates its Red feature "gene" to all of its offspring.

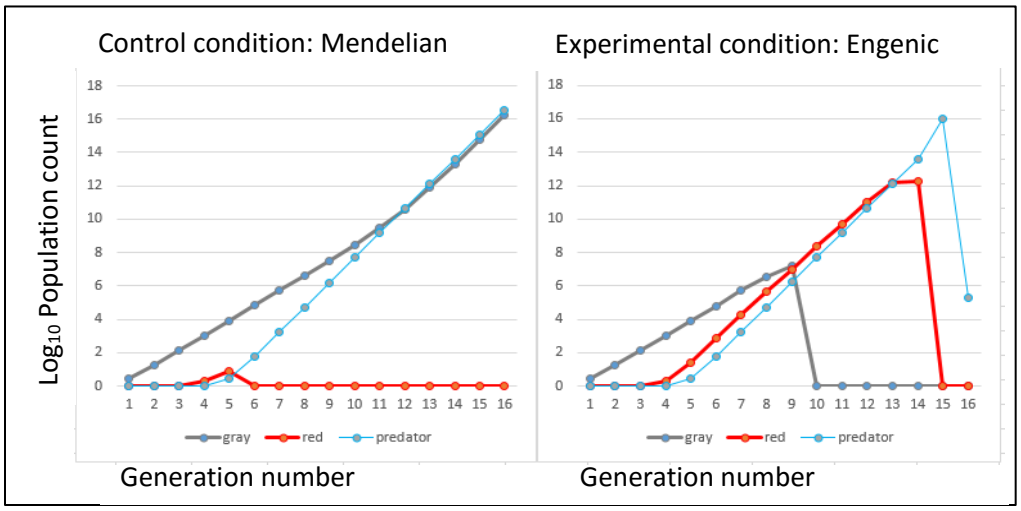

Figure 9. Survival of Red-eye Ae. *aegypti* with and without Engenic Heredity

The time course of the experiment is summarized in Figure 9. The gray-eye population is shown by the gray trace, the red-eye population count is represented by the red trace, and the number of predators is represented by the thin blue trace. The experiment was run for 16 generations beginning with 2 gray-eye individuals at generation 1.

The control condition shows that the red-eye type failed to survive past 3 generations after its introduction to the experiment at point 3. The Red phenotype is recessive while the Gray is dominant. After an Engenic Red type is introduced, if the Red population overwhelms the Gray, that effect can be attributed to the effect of Engenic Inheritance.

In the experimental condition, the rate of increase of the red type was nearly 100% of the offspring. This completely offset the effect of its still-high rate of fatality due to predation. The chart (right) shows that the population of the gray type tapered around generation point 7 and collapsed at point 10 while the population of red continued until it was overcome by explosive predation at generation 15.

The gray type was subject to normal predation levels, but was continually replaced by interbreeding with the Engenic Red-eye Ae. aegypti type. Finally, in this experiment, the red type did not possess a superior selective evolutionary advantage. This is consistent with the idea that it does not exhibit Mendelian heredity and instead is demonstrating the entirely new kind of inheritance called Engenic heredity here. This was fully demonstrated by the ultimate collapse of the red (Engenic) type under growing predatory pressure, as might be expected in the wild and could be considered in planning for the eradication of pathogens such as flaviviruses and Malaria (by Ae. *stephensi*).

The charts in Figure 5 were configured by hand in Excel from exploratory output as a quick convenience in writing this initial draft of this paper. For the review and final drafts, I expect to plan to produce a similar result using the R programming language directly and reproducibly taking the output of the Java code sample with the actual parameters defining the experiment.

Discussion

About inheritance, heredity

The field of biological science, in its broadest sense, has long recognized that living organisms operate on a system of molecular structures and processes which are based on what looks like computer code.

It seems to be the case that we, as hardware/software designers and practitioners are about to face the challenge of programming living organisms, or programmable molecular systems derived in some way from life.

Traditionally in computer programming languages, particularly in object-oriented design, a key feature of many design patterns is that of inheritance. The same term appears in biology, however, with a quite different meaning. In biology we have Mendelian inheritance and non-Mendelian inheritance.

Non-Mendellian inheritance is further subdivided into **polygenic inheritance** and **multifactorial inheritance** by the biological science establishment. Engenic Inheritance is not a "non-Mendellian" form of inheritance as it is based on a fundamentally different mechanism which is not a presence or absence of Mendel's mechanisms.

However, the invention of "**gene drive**" techniques [19], the first successful implementation being based on endonuclease with CRISP-cas9 nicase, creates a new kind of reproductive inheritance. This is not Mendelian inheritance in that gene dominance/recessive mode is irrelevant. The fact that gene drive technology is a profound departure from natural reproductive processes has been recognized. Several practitioners in the field have articulated this recognition in their talks and papers with titles such as "Life Code" [20], "Active Genetics and MCR – Mutagenic Chain Reaction [21], and "Gene Drives" [22]. Life Code is unacceptable as a term before of the now-infamous company LIFE CODE labs https://csidds.com/2016/06/10/unfounded-calculation-assumptions-in-lifecode-dna-testimony-2014-sc-ruling-in-simmons/

About viruses

Is Engenic Inheritance, as defined here, just what makes a virus a virus? In nature, a virus has one built-in program for making a cell make copies of itself (copies of the virus). The virus does not have any kind of computational capability to "figure out" how to overcome an unexpected variation. Some viruses have multiple paths to success. But those are all hard-coded as a result of long evolution. That evolution was the result of very large numbers of random mutations throughout a virus population, not a computational capability now resident in the virus.

In contrast, the "figuring out" capability is part of the design of an Engenic inheritance application.

Aside from viruses, Engenic Inheritance can be incorporated into a cell, or program simulating living cells, without a virus. The modification is part of the cell's replication mechanism. You could argue that this is like what a retrovirus does. But retroviruses still do not work to counteract Mendelian recombination. They haven't thought of that. People did.

About new terminology

None of these proposals are satisfactory to concisely and powerfully place the new kind of inheritance alongside Mendelian. "Gene Drive" is a description rather than a compelling word or phrase that "rolls off the tongue". It does not speak to the concept of "inheritance" as its distinguishing quality. **This paper proposes the term "Engenic Inheritance" as the new general name for all approaches that drive genes or software modifications robustly through a population, and "Engenic heredity" when it is discussed specifically in the context of biological science.**

The etymological reasoning behind "Engenic" is twofold. First, "engen" derives from "engender", suggesting how a genetic modification engenders propagation of a feature. The term "Engenic" also suggests the concept of an "engine." The self-modifying component that performs during normal operation is an engine, in a sense, as compared to the normal static function of naturally occurring genes. It is important that any new term be based on established patterns in the field.

Finally, on the question why give a specific name to the design pattern discussed in this paper, we may ponder this comment from Eric Lander [23]:

> "What do you do in science when you have no idea what something is? … you give it a name …."

Contribution

This paper introduced the design pattern, Engenic Inheritance. It shows that a unique pattern formalism can describe a bioengineering concept and a software design concept as aspects of a single underlying phenomenon. Because of this commonality, it was demonstrated that a software simulation based on a close analogy between software components and the biological elements it is intended to represent can be written using the Engenic inheritance (software) feature.

Other Work

Other types of inheritance in programming languages have been explored.

The concept "Implementation Inheritance" described by Karie Lucas [24] is not what the name seems: it means that class operations delegate to the base class in a static type system.

The previous efforts to solve the "diamond problem" of multiple inheritance [25], mentioned in the introduction, yielded no complete solutions.

Future Work

Biological interests:

The prototype software is being used to study the actual gene code involved in endonuclease gene drive. Results will be reported as significant finding come about.

Current research concentrates on introducing a single or narrowly focused transgenic feature using gene drive. However, there is reason to believe that multiple, and perhaps very large numbers of new features can be introduced into an organism. When

this kind of work becomes established, we might consider a new terminology in the same tradition of neologism leading to the terms “proteomics”, “genomics”, “omics” and so forth. Acceptable terms for genome-wide, high-dimensional Engenic Inheritance would likely be “**engenomics**” [or some may prefer “engeniomics” which is a little more distinct from “genomics”, but seems unnecessarily long].

Computer science interests:

This paper does not discuss the impact of our “dynamic multiple inheritance” on an otherwise statically typed programming language in which Engenic Inheritance might be embedded. Although it is not necessary to extend the code programming language, it would be interesting to see if it can be done in a way that does not break the type system of the language. In either case, Engenic Inheritance need not replace or complicate dynamic binding and polymorphism in a language such as Java.

This paper does not elaborate on how the data elements in a class are to be “inherited” into generated child classes. It also does not elaborate on what happens when a heritable method is resident several levels up the class extension chain or at different levels. These issues should be fully explored.

Engenic Inheritance provides resolution of “conflicts” different from all of the major languages. In the list of methods of mitigating conflicts in the Wikipedia article, https://en.wikipedia.org/wiki/Multiple_inheritance, [cit.] none preserve the “not selected” alternative and make it available dynamically for subsequent inheritance or reactivation. This article could be updated if a Java syntax is created to “natively” support Engenic inheritance.

The relation of Engenic Inheritance and its infrastructural inheritance scheme should be examine with respect to other foundational features of software inheritance. The application of this form of inheritance should comply with the Liskov Substitution Principle [cit.].

Technical problems

One problem with the implementation of replication caused a serious delay in development of the implementation code. At several point we need a randomization algorithm for choosing individuals to propagate, to express features, and for choosing among chromosomes and feature genes to mix. As the number of individuals and features increase, the simple random-without-replacement code became too expensive in time and memory resources. The work-around in the sample code falls back to a linear search for unused items, which introduces a non-random bias that would be unacceptable for real-world computational experiments. We are not aware of a pseudo-random number generator that satisfies our requirement:

The random sequence stateless seeded operator problem [26]: Given a range 0 through m inclusive for any m>1, and given an integer seed, generate a series of repeatably random integers in the range 0..m without repeats.

1. The sequences should score well on at least one test of randomness, e.g. Z. Kalogiratou and T. Monovasilis (eds.), AIP Conf. Proc. 1618, 531–534 implemented in an R package [27] .
2. The algorithm must be stateless (except perhaps for knowing that it is at step ***i*** in the sequence), i.e. …
    a. Do not use an explicit list of used values.
    b. Do not use a shuffle list or look-up table.
3. Generate different pseudo-random sequences for different seeds.
    a. more than 1 seed should be available for each m.
    b. Provide a practical way to choose seeds that meet the algorithm’s criteria appropriately.

Even the most specialized languages designed for problems requiring a random-value function do not satisfy these criteria. For example, the `sample( 0:m, n, replace=F)` function in the R programming language [28] is not repeatable (no seed specification) and appears to run slow for larger m; n specifies the vector size (how many elements) to return in the same call. The reader is invited to alert the author of a better solution.

Programming language

As mentioned earlier, a new syntax could be introduce to a language such as Java to support Engenic inheritance. It would recognize identical signatures by default and allow programmers to explicitly mark distinct methods as intended to be engenically substituted, allowing different base name, parameters, throws, and return.

Conclusion

The Engenic Inheritance design pattern has been defined in this paper along with its relationship to computer language implementation and the bioengineering or biological constructs it can represent. The full GoF format design pattern has been completed. Implementations from computer science and from biology perspectives have been given. An experiment proving the utility of Engenic Inheritance for research in biology and development of bioengineering was presented. Readers are invited to contact the author with questions and to potentially look into collaborations using this technology.

References

[1] R. Poli, W. B. Langdon, N. F. McPhee and J. R. Koza, "A Field Guide to Genetic Programming," 2008.. Available: http://cswww.essex.ac.uk/staff/poli/gp-field-guide/index.html.

[2] L. (. Spector, Genetic Programming and Evolvable Machines, Springer.

[3] Novak and Tyson, "Numerical analysis of a comprehensive model of M-phase control in Xenopus oocyte extracts and intact embryos," *Journal of CellScience,* pp. 106:1153-1168, 1993.

[4] M. Guevara-Souza and E. Vallejo, "A computer simulation model of Wolbachia invasion for disease vector population modification," *BMC Bioinformatics,* vol. 16, no. 317, 2015.

[5] A. Fog, "Computer simulation of biological evolution in structured populations,". Available: http://www.agner.org/evolution/.

[6] A. Csikász-Nagy and et_al., "Analysis of a generic model of eukaryotic cell-cycle regulation," *Biophysics Journal,* vol. 90(12), pp. 4361-79, 2006.

[7] C. Chen, L. Calzone, A. Csikász-Nagy, F. Cross, B. Novak and J. Tyson, "Integrative analysis of cell cycle control in budding yeast," *Molecular Biology of the Cell,* vol. 15, no. 8, pp. 3841-62, 2004.

[8] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, p. 175.

[9] B. Alberts and et_al., Molecular Biology of the Cell, Garland Science.

[10] Oracle, "Inheritance,". Available: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html.

[11] E. Lander, "Mendel's Laws, excerpt 1 | MIT 7.01SC Fundamentals of Biology," 2012.. Available: https://www.youtube.com/watch?v=9dHBTckFvME.

[12] Wikipedia, "Allosome," 2016.. Available: https://en.wikipedia.org/wiki/Allosome#Medical_applications. [Accessed Jun 2016].

[13] E. Lander, "MITx: 7.00x Introduction to Biology - The Secret of Life[1],". Available: https://courses.edx.org/courses/course-v1:MITx+7.00x_3+2T2015/courseware/Week_3Global/Genetics_1/. [Accessed 2016].

[14] J. Reza, "Delta streams with application to sequencing," in *IEEE SoutheastCon*, Fort Lauderdale, FL, USA, 2015.

[15] M. Jinek, K. Chylinski, I. Fonfara, M. Hauer, J. A. Doudna and E. Charpentier, "A programmable dual-RNA-guided DNA endonuclease in adaptive bacterial immunity," *Science,* pp. 816-21, 17 August 2012.

[16] B. Zetsche, J. Gootenberg, O. O. Abudayyeh, I. M. Slaymaker, K. S. Makarova, P. Essletzbichler, S. E. Volz, J. Joung, J. van der Oost, A. Regev, E. V. Koonin and F. Zhang, "Cpf1 Is a Single RNA-Guided Endonuclease of a Class 2 CRISPR-Cas System," *Cell,* vol. 163, no. 3, p. 759–771, 22 October 2015.

[17] R. Feil and F. Berger, "Convergent evolution of genomic imprinting in plants and mammals," *Trends in Genetics,* vol. 23, pp. 192-9, Apr 2007.

[18] W. community, "disomy,". Available: https://en.wikipedia.org/wiki/Uniparental_disomy .

[19] J. A. Doudna and et_al., "Compositions and methods of nucleic acid-targeting nucleic acids". US Patent 9,260,752, 16 Feb 2016.

[20] J. Enriquez, "The life code that will reshape the future," Feb 2003.. Available: https://www.ted.com/talks/juan_enriquez_on_genomics_and_our_future?language=en.

[21] V. M. Gantz and E. Bier, "The dawn of active genetics," *Bioessays,* vol. 38, no. 1, pp. 50-63, Jan 2016.

[22] Wyss Institute at Harvard, "FAQs: Gene drives,". Available: http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0ahUKEwj67NKCo83NAhUG8CYKHVVJDcoQFggeMAA&url=http%3A%2F%2Fwyss.harvard.edu%2Fstaticfiles%2Fnewsroom%2Fpressreleases%2FGene%2520drives%2520FAQ%2520FINAL.pdf&usg=AFQjCNFCAMj-C7ykHC0rOfInylsk. [Accessed 2 Jun 2016].

[23] E. Lander, "Eric Lander (700.x, lecture 4 sec 6," 2015..

[24] K. Lucas, "Implementation Inheritance,". Available: http://c2.com/cgi/wiki?ImplementationInheritance.

[25] E. Truyen, W. Joosen, B. Nørregaard and P. Verbaeten, "A Generalization and Solution to the Common Ancestor Dilemma Problem in Delegation-Based Object Systems," in *Proceedings of the 2004 Dynamic Aspects Workshop*, 2004.

[26] J. R. Reza, "The random sequence stateless seeded operator problem," In-prep.

[27] F. Caeiro, "bartels.rank.test randtests-package. Testing randomness in R," CRAN, 17 Nov 2014.. Available: https://cran.r-project.org/web/packages/randtests/randtests.pdf.

[28] D. Smith, "How to choose a random number in R," 11 Feb 2009.. Available: http://blog.revolutionanalytics.com/2009/02/how-to-choose-a-random-number-in-r.html.