EBGSD: Emergence-Based Generative Software Development

Mahdi Mostafazadeh, Mohammad Reza Besharati, Raman Ramsin
Department of Computer Engineering
Sharif University of Technology
Tehran, Iran
e-mail: {mmostafazadeh, besharati}@ce.sharif.edu, ramsin@sharif.edu

Abstract—**Generative Software Development (GSD) is an area of research that aims at increasing the level of productivity of software development processes. Despite widespread research on GSD approaches, deficiencies such as impracticability/impracticality, limited generation power, and inadequate support for complexity management have prevented them from achieving an ideal level of generativity. We propose a GSD approach based on a novel modeling paradigm called 'Ivy'. Ivy models the context domain as a set of conceptual phenomena, and depicts how these phenomena emerge from one another. Our proposed approach, Emergence-Based Generative Software Development (EBGSD), uses Ivy models for modeling how a software system (as a phenomenon) can emerge from its underlying phenomena, and can provide an effective means for managing software complexity. Developers can also elicit generative patterns from Ivy models and utilize them to increase the level of reuse and generativity, and thus improve their productivity.**

Keywords-generative software development; phenomenon; emergence; conceptual model

I. INTRODUCTION

As Mens points out, "Software systems are among the most intellectually complex artifacts ever created by humans" [1]. Managing software complexity is indeed the main impetus behind many research areas in software engineering. Generative Software Development (GSD) aims to address this issue through increasing the level of automation in software development, which also enhances productivity. Despite widespread research on GSD approaches such as Model-Driven Development (MDD), Software Product Lines (SPL), Program Development from Formal Specifications, Generative Patterns, and High-Level Programming Languages, there are certain disadvantages in each of them that have prevented researchers from achieving an ideal level of generativity in software development. For instance, in Czarnecki's GSD approach [2], two methods (Configuration and Transformation [3]) have been suggested for transition from the problem domain to the solution domain; although this approach is well-established, it has not achieved an ideal level of generativity, mainly due to deficiency in generation power, inflexibility of configuration, over-abstractness, inattention to seamlessness, and ambiguities in transformation. Furthermore, some of the approaches, such as MDD and High-Level Programming Languages, are deficient as to their support for complexity management. These shortcomings (further explained in Section II) are the main motivations behind this research.

We propose a GSD approach based on a novel modeling paradigm called *Ivy*, originally proposed by Besharati in a seminar report in 2013 [9]. *Phenomenon* and *Emergence* [13] are the two basic concepts of the Ivy paradigm. The Ivy paradigm prescribes a way for modeling the emergence of a conceptual phenomenon from its underlying phenomena. Emergence is recursive: an Ivy model takes the form of a digraph that shows how a phenomenon emerges from its underlying phenomena, which in turn emerge from other phenomena, and so on.

In the Ivy-based software development approach that we propose herein (which we have chosen to call Emergence-Based Generative Software Development, or EBGSD for short), the target software system is considered as a phenomenon that emerges from its underlying phenomena, and is therefore represented as an Ivy model. The Ivy model helps manage the inherent complexity of software systems. Furthermore, it is possible to extract generative patterns from Ivy models and utilize them to increase the level of reuse in software development processes, and thereby promote generativity. The evolutionary nature of the modeling approach makes it highly practical, and can lead to a high level of flexibility in software development. We have also proposed a methodology for applying EBGSD to real-world projects. EBGSD promotes seamlessness, and can improve software processes as to smoothness of transition among development activities.

The rest of the paper is structured as follows: Section II provides an overview of the research background through focusing on a number of prominent GSD approaches; in Section III, we introduce the Ivy modeling paradigm as the basis for our proposed approach; our EBGSD approach and its corresponding methodology are proposed in Sections IV and V, respectively; an illustrative example of the application of EBGSD is given in Section VI; finally, Section VII presents the conclusions and suggests ways for furthering this research.

II. RESEARCH BACKGROUND

Software generation is an old ideal that has been pursued and evolved over decades. The advent of programming languages and compilers can be considered as the first step towards enhanced productivity in software development. The field has evolved over decades: for instance, in the context of MDD, programming languages and compilers have been replaced by Domain-Specific Languages (DSLs) and model/code generators. Due to the vastness of the research conducted on software generativity, it is not possible to discuss all of them here; hence, we will focus on the four most prominent approaches, as listed below. Our main purpose in this section is to demonstrate the motivations for this research, and to outline the research objective.

Genetic and evolutionary approaches: these approaches aim at generating complex systems through creation of a simple generative system to generate new constructions that ultimately lead to the desired complex system [4]. The main problem with these approaches is that due to their high level of inherent randomness, they are not applicable to systems with specific requirements.

MDD: MDD considers models as first-degree entities that drive the software development process and serve as the basis for generating the target software [5]. In this approach, software is developed through creation of models at a high-level of abstraction, and then transformation of these models into their lower-level counterparts (and ultimately software) based on certain mappings. Although this approach has become popular in recent years, there are major problems that prevent it from achieving an ideal level of automation. For instance, although this approach intends to reduce software complexity, it in fact just shifts the complexity [6]: development is easy and straightforward when the modeling levels and their corresponding mappings have been specified, but defining the levels and the mappings themselves is by no means straightforward.

SPL: in the software product line approach, instead of developing a single software system from scratch, the focus is on a family of systems that are developed from a set of common reusable components by applying a defined process [7]. To be more precise, a software product line is a set of software-intensive systems that share a common set of features, and that are developed from a common set of core assets [8]. As implied by this definition, SPL aims to improve the productivity of software development processes through providing a higher level of reuse; but the definition makes no hint of any automation involved in the process. Hence, SPL has not been able to achieve an ideal level of generativity. Moreover, creating reusable assets is a costly process, which might even adversely affect the productivity of software development processes.

Czarnecki's GSD approach: similar to SPL, Czarnecki's approach aims at increasing the productivity of software development processes through focusing on families of systems [2]. The main difference between this approach and the SPL approach is that it emphasizes automated composition of components, whereas manual composition is acceptable in SPL. However, just like SPL, GSD too can have an adverse affect on productivity.

As observed in the above approaches, although they have strived to increase the level of software generativity, certain deficiencies prevent them from achieving the ideal level of generativity in software development, and overcoming these deficiencies is the objective of this research. Specifically, genetic approaches enjoy a high level of automation, but are not practicable. On the other hand, MDD, SPL, and GSD are practicable, but are deficient as to complexity management, automation, and productivity; to be precise, these approaches just replace development complexity with mapping complexity.

III. IVY PARADIGM

Ivy [9] is a modeling paradigm for representing conceptual phenomena and their emergence. Conceptual phenomena are typically regarded as abstractions of real-world phenomena. Ivy is based on the notion that conceptual phenomena can be combined, and a new conceptual phenomenon thus emerges. We model this fact in the *Ivy Model*; as seen in Figure 1, an Ivy model is a directed graph in which nodes represent phenomena, and arcs represent emergences. As an example, consider the following three conceptual phenomena: *car*, *red*, and *wheel*, which are the results of abstraction from their real-world counterparts. As shown in Figure 1, from a certain point of view, the phenomena *car* and *red* can be combined, and the phenomenon *red car* thus emerges. From another point of view, the phenomena *wheel* and *red* can be combined, and the phenomenon *red wheel* emerges. The phenomena *car* and *red wheel* can be combined, and from two different points of view, two phenomena emerge: *red-wheeled car*, and *red car wheel*.

The world of software development is full of representation, combination and emergence of conceptual phenomena. Requirements engineering is concerned with conceptual phenomena directly abstracted from real-world phenomena. Some of these phenomena are combined, and other conceptual phenomena emerge as a result. For instance, the conceptual phenomena *actors*, *use cases* and their *relationships* are combined and the phenomenon *use case diagram* emerges; or in goal-oriented requirements engineering, certain phenomena (i.e., *goals)* could be combined, and a higher-level goal would emerge. Software *platforms* are themselves conceptual phenomena that emerge from other phenomena (e.g., *requirements*). Design and implementation phases are concerned with combination of *requirement* and *platform* phenomena and the emergence of *software-solution* phenomena.

Since the dependencies in an Ivy model are unidirectional, it can enhance understandability and modifiability, leading to better complexity management. The Ivy model may look very similar to other models such as goal models [10] and feature models [11], but there are fundamental differences. In those models, relationships have very specific semantics: in goal models, relationships mean *Why* and *How* [10], and in feature models, relationships show the semantics of *Has* [12]. Whereas in Ivy, the emergence relationship has a general meaning, and its concrete semantics depends on the perspective upon which it is based. The semantic generality of emergence is an important feature of Ivy, which makes it capable of tying all conceptual phenomena together. Thus, the relationships in feature models and goal models can be considered as special kinds of emergence.

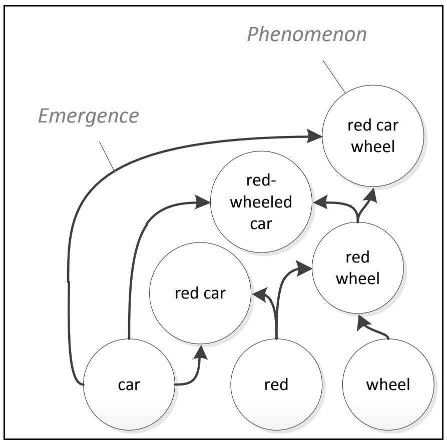


Figure 1. Example of an Ivy model.

In the next section, we will explain our proposed GSD approach based on the Ivy model.

IV. EBGSD APPROACH

In our proposed Ivy-based software development approach, EBGSD, the system for which a software solution is required is considered as a problem-domain phenomenon, which itself emerges from lower-level phenomena. We represent this fact in an Ivy model. This process is applied recursively: for each phenomenon at every level, we can represent the emergence of that phenomenon from lower-level phenomena. Although this process is theoretically endless, we continue it until the phenomena at the lowest level can be considered as basic phenomena in the problem domain; these 'leaf' phenomena are the finest-grained phenomena required for fulfilling the purposes of the developers. Similarly, we consider the software platform (solution domain) as a phenomenon, and draw an Ivy model to represent its emergence. In the next step, we combine the two Ivy models (problem-domain and solution-domain), and software-solution phenomena emerge. The combination process is initiated in a manual fashion, but it can then proceed with a certain degree of automation through producing and applying *Ivy generators*. This is done by identifying recurring and reusable patterns of combination, and capturing them as Ivy generation patterns. An Ivy generator can then be developed to automatically apply these patterns, and thereby combine the two source models (problem-domain and solution-domain) into the destination model (software-solution).

Ultimately, the code corresponding to each software-solution phenomenon is produced. To this aim, it is first determined which underlying software solution phenomena affect the generation or modification of the code corresponding to the target phenomenon; a set of *code generators* are then developed for these underlying phenomena, the outputs of which should be conveyed to the codes of higher-level phenomena. Many of these code generators are typically reusable, and therefore act as patterns. The generation logic embodied in the code generator of each phenomenon utilizes the outputs of lower-level generators (i.e., the codes of the corresponding lower-level phenomena) to generate the code of the target phenomenon. Examples of the abovementioned models and patterns are provided in Section VI.

In the next section, we propose an iterative-incremental methodology for applying EBGSD in software development projects.

V. A METHODOLOGY FOR APPLYING EBGSD

The iterative-incremental software development methodology hereby proposed for applying EBGSD consists of five iterative workflows (as shown in Figure 2): 1) Production of Problem-Domain Ivy Model, 2) Production of Software-Platform Ivy Model, 5) Emergence of Software-Solution Ivy Model, 4) Production of Ivy Generators, and 5) Production and Application of Code Generators. These workflows are iterated in order to gradually produce the models and the target system. This methodology is not a full-lifecycle process; it should be augmented with complementary activities (including umbrella activities and post-implementation activities) in order to become practicable. The workflows will be explained throughout the rest of this section. Section VI provides examples of the products of these workflows.

A. *Production of Problem-Domain Ivy Model*

We consider the system (for which we intend to develop software) as a phenomenon, and draw an Ivy model depicting the phenomena from which it emerges. Requirements and structural constituents of the problem domain are considered as important phenomena in this model. As previously mentioned, based on different points of view, different Ivy models can be produced for the same purpose. Drawing the Ivy model requires no special skills on the part of the modeler; developers can draw their own based on their particular perspectives of the system. For example, an analyst who knows how to model the requirements as use cases can regard each use case as a phenomenon emerging from its steps, and each step as a phenomenon emerging from the phenomena in the structural view of the system. It should be noted that since everything is represented as phenomena, it is necessary to add certain semantic phenomena in order to provide the readers and the generators with adequate semantics. For example, if *Add Student* is a use case (represented as a phenomenon of the same name), it is necessary to represent the emergence of this phenomenon from a phenomenon named *Use Case*.

B. *Production of Software-Platform Ivy Model*

We consider the software platform (solution domain) as a phenomenon, and draw an Ivy model depicting the phenomena from which it emerges. For example, in the object-oriented platform, the phenomenon *Class* emerges from the phenomena *Attribute* and *Method*, the phenomenon *Attribute* itself emerges from its *Type*, and so on. In some cases, it is enough to just model the solution-domain phenomena; emergences are left out in such cases.

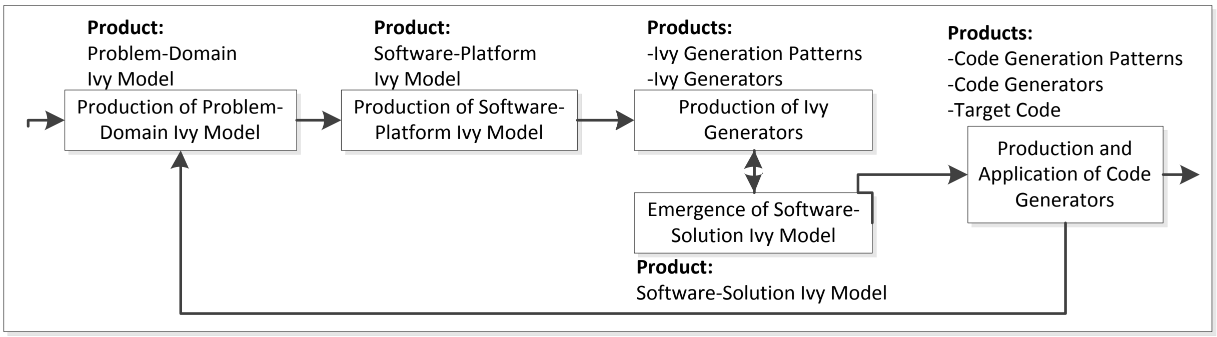

Product:
Problem-Domain Ivy Model
Production of Problem-Domain Ivy Model
Product:
Software-Platform Ivy Model
Production of Software-Platform Ivy Model
Products:
-Ivy Generation Patterns
-Ivy Generators
Production of Ivy Generators
Emergence of Software-Solution Ivy Model
Product:
Software-Solution Ivy Model
Products:
-Code Generation Patterns
-Code Generators
-Target Code
Production and Application of Code Generators

Figure 2. A methodology for applying EBGSD (workflows and products).

C. Emergence of Software-Solution Ivy Model

The problem-domain and software-platform Ivy models are combined (partly manually, and mostly by applying the Ivy generators produced in the next workflow), and the software-solution Ivy model emerges; as this workflow is dependent on Ivy generators, it generally overlaps with the next workflow. It should be noted that three types of phenomena may affect the emergence of each software-solution phenomenon: problem-domain phenomena, solution-domain phenomena, and other software-solution phenomena. For instance, the software-solution phenomenon *Student* emerges from other software-solution phenomena (*Name* and *Age*), as well as a solution-domain phenomenon (*Class*) and its problem-domain counterpart (*Student*).

D. Production of Ivy Generators

Based on the problem-domain and software-platform Ivy models, a set of Ivy generation patterns are elicited and their corresponding Ivy generators are developed (to be updated iteratively). This workflow starts after combination patterns have been identified through manual combination of the source Ivy models, and its results are in turn used for producing the software-solution Ivy model; it therefore overlaps with the previous workflow.

E. Production and Application of Code Generators

Based on the software-solution Ivy model, a set of code generators are developed (as explained in Section IV). These generators are in fact responsible for realizing what the literature on emergence calls *Radical Novelty* [13]: unpredicted and rich features that cannot be anticipated until they actually surface.

VI. EXAMPLE

A partial problem-domain Ivy model for an education system is illustrated in Figure 3. The Ivy model has been drawn based on two different points of view. The left part of the model is drawn based on a structural view of the system. *School* and *Student* are two pivotal entities in the system, so we have represented them as two phenomena that have both emerged from the *Entity* phenomenon. The **Bold Tags** shown on some of the phenomena indicate the emergence of those phenomena from a phenomenon with the same name as the tag; hence, there is no need to explicitly show the corresponding emergence arcs, and excessive complexity is thereby avoided. For instance, the **Entity** tag on the *Student* phenomenon is equivalent to an emergence arc from the *Entity* phenomenon to the *Student* phenomenon. The PD prefix means that the phenomenon belongs to the problem domain. One important relation in the system is the relation between a school and its students; hence, we have represented it as a phenomenon that has emerged from three phenomena: *School*, *Students*, and *Relation*.

The right half of the Ivy model is drawn based on a functional view of the system. One of the system's use cases (*Add Student*) has been represented as a phenomenon, emerging from its *Steps* and the *Use Case* phenomenon. The use case steps themselves have emerged from structural-view phenomena and certain semantic phenomena such as *Add* and *Command;* these semantic phenomena are essential for developing code generators. It should be noted that these points of view are chosen from among many possible alternatives; developers draw the Ivy model based on their own perspectives (e.g., a feature-driven point of view). Each phenomenon and emergence itself may possess implicit semantics, which can be represented separately as an Ivy model; however, due to practicality considerations, the process of Ivy modeling should be brought to an end before the complexity becomes pointlessly overwhelming.

A partial solution-domain Ivy model for the object-oriented platform is shown in Figure 4. This Ivy model has been produced based on well-established object-oriented notions, e.g., a class is an encapsulation of certain attributes and methods. It should be noted that solution domains are typically application-independent.

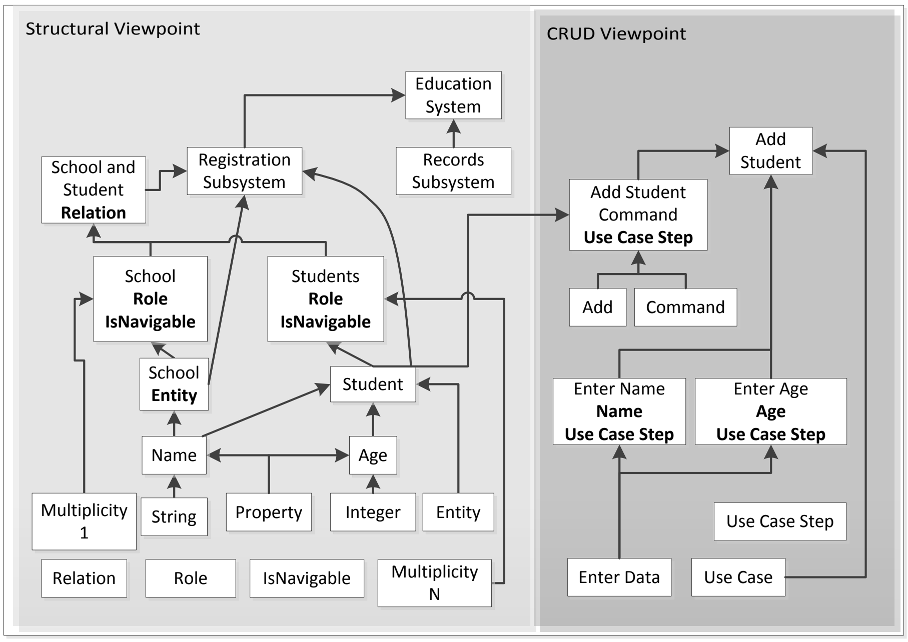


Figure 3. Partial problem-domain Ivy model for an education system.

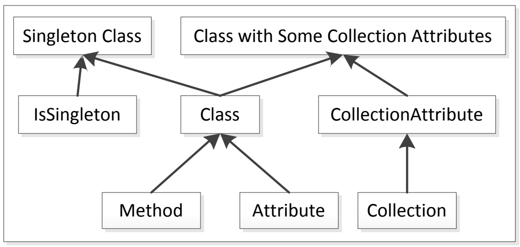


Figure 4. Partial solution-domain Ivy model for the object-oriented platform.

A partial software-solution Ivy model for the education system is illustrated in Figure 5. This model is obtained through extracting and applying a set of Ivy generation patterns. These patterns are illustrated in Table I. Each pattern has a *Name*, a *Before* state, and an *After* state (which is basically an extension of the Before state). The phenomena whose names are shown in braces are placeholders for any phenomenon that conforms to the topology of the pattern. As seen in Table I, these placeholders are used for naming new phenomena in the After state.

A set of code generation patterns, which can be used in order to generate the code of our target system, are illustrated in Table II. Each generation pattern has a Name, an Ivy pattern that corresponds to the generator, and a generation logic that generates or modifies the code of some phenomena using the code of lower-level phenomena. Table III shows a more complex code generation pattern corresponding to class reification (extraction of a class from an association relationship), and Table IV shows an example of its application to a concrete Ivy model.

VII. CONCLUSION AND FUTURE WORK

We have proposed EBGSD as a GSD approach, and have proposed a methodology for applying it to real-world projects. Achieving high levels of reuse, maintainability, and complexity management are some important potential benefits of our proposed approach. Since Ivy generation patterns and code generation patterns are fine-grained patterns expressed at a high level of abstraction, EBGSD can achieve high levels of reuse. EBGSD can increase maintainability and manage software complexity from two aspects: increasing software understandability, and improving modifiability. The Ivy model can be seen as a software construction map, which can be easily reviewed through a graph traversal algorithm. On the other hand, since the couplings among Ivy elements are simple and unidirectional, it is easy to apply the necessary changes, as the changes do not propagate in an unmanageable fashion. Furthermore, because this approach performs all the steps of software development and produces the target artifacts in a smooth and seamless manner, it can be a potential solution to the conflict between modeling and agile development; model-phobic agile methodologies might find it worthwhile to invest in Ivy modeling, as Ivy models are simple and straightforward, and can be used in such a way that agility is not adversely affected.

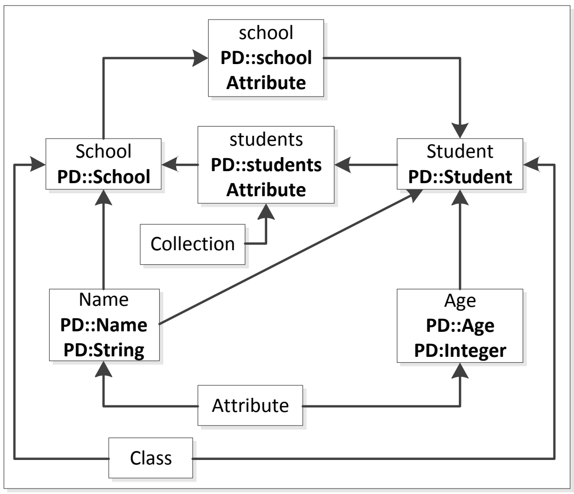


Figure 5. Partial software-solution Ivy model for the education system.

At present, we are evaluating the approach through a case study. Future research can focus on exploring existing opportunities for using the approach for enhancing automation through *construction*, rather than just *reuse*.

REFERENCES

[1] T. Mens, “On the Complexity of Software Systems,” Computer, vol. 45, Aug. 2012, pp. 79–81, doi: 10.1109/MC.2012.273.

[2] K. Czarnecki, “Generative Programming,” PhD thesis, Technical University of Ilmenau, 1999.

[3] K. Czarnecki, “Overview of Generative Software Development,” Proc. European Commision and US National Science Foundation Strategic Research Workshop on Unconventional Programming Paradigms, 2005, pp. 326–341, doi: 10.1007/11527800_25.

[4] R. Poli, W. B. Langdon, and N. F. McPhee, A Field Guide to Genetic Programming, Lulu Enterprises, 2008.

[5] M. Volter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, Model-Driven Software Development: Technology, Engineering, Management, Wiley, 2013.

[6] B. Hailpern and P. Tarr, “Model-driven development: The good, the bad, and the ugly,” IBM Systems Journal, vol. 45, July. 2006, pp. 451–461, doi: 10.1147/sj.453.0451.

[7] S. Apel, D. Batory, C. Kastner, and G. Saake, Feature-Oriented Software Product Lines: Concepts and Implementation, Springer, 2013.

[8] J. Royer and H. Arboleda, Model-Driven and Software Product Line Engineering, Wiley, 2012.

[9] M. Besharati, “Generativity in Software Development: Survey and Analysis,” M.Sc. Seminar Report, Sharif University of Technology, 2013 (In Persian).

[10] A. Van Lamsweerde, Requirements Engineering: From System Goals to UML Models to Software Specifications, Wiley, 2009.

[11] C. Kastner and S. Apel, “Feature-Oriented Software Development: A Short Tutorial on Feature-Oriented Programming, Virtual Separation of Concerns, and Variability-Aware Analysis,” in Generative and Transformational Techniques in Software Engineering IV, J.M. Fernandes, R. Lammel, J. Visser, J. Saraiva, Eds. Springer, 2013, pp. 346–382, doi: 10.1007/978-3-642-18023-1.

[12] P. Schobbens, P. Heymans, and J. Trigaux “Feature Diagrams: A Survey and a Formal Semantics,” Proc. International Conference on Requirements Engineering, 2006, pp. 139–148, doi: 10.1109/RE.2006.23.

[13] J. Goldstein , “Emergence in complex systems,” in The SAGE Handbook of Complexity and Management, P. Allen, S. Maguire, B. Mckelvey, Eds. SAGE, 2011, pp. 65–78.

TABLE I. A SET OF IVY GENERATION PATTERNS ELICITED FROM THE EXAMPLE

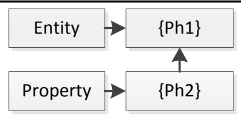
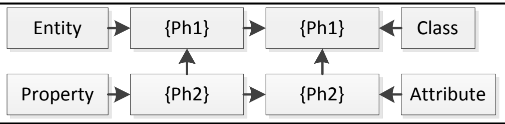
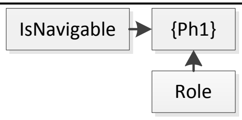
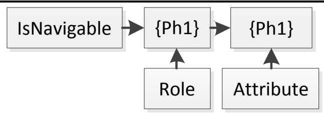
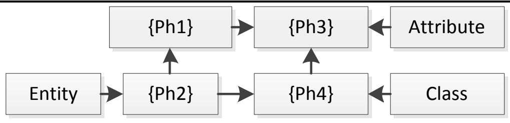
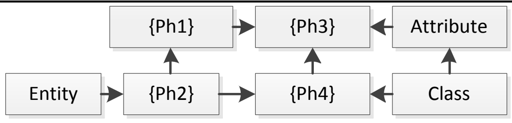
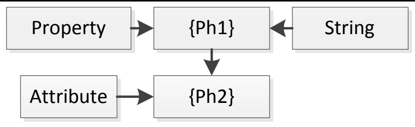
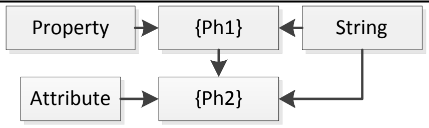
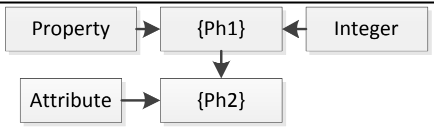
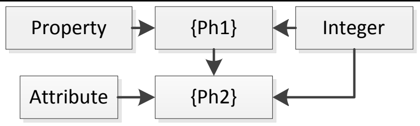
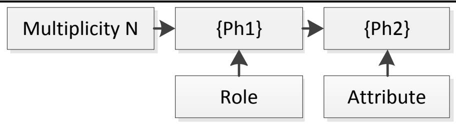
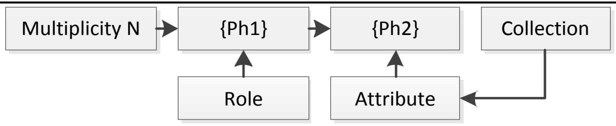
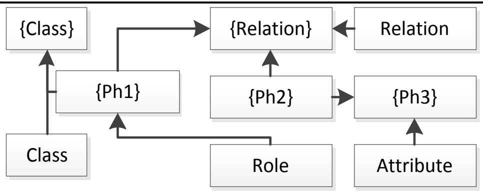
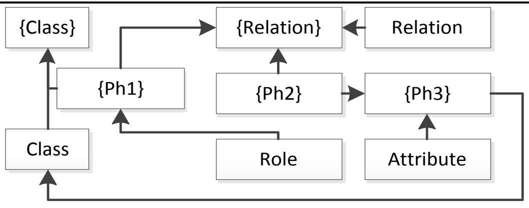
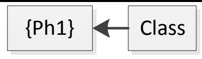
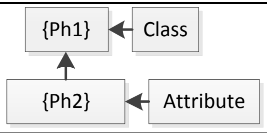
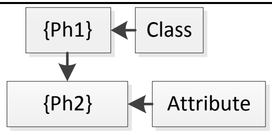
NAME	BEFORE	AFTER
Emergence of Classes and Attributes		
Emergence of Relational Attribute		
Specifying Relational Attribute Type		
Injecting String Type		
Injecting Integer Type		
Emergence of Collections		
Injecting Relational Attribute into Class	 {Class} {Relation} Relation {Ph1} {Ph2} {Ph3} Class Role Attribute	 {Class} {Relation} Relation {Ph1} {Ph2} {Ph3} Class Role Attribute

TABLE II. A SET OF CODE GENERATION PATTERNS ELICITED FROM THE EXAMPLE

NAME	PATTERN	GENERATION LOGIC

Class Code Generation		class {Ph1} { ClassTemplate c;
		Code() { if(c == null) c = Class::Code();
		c.SetName({Ph1});
		return c; } }

Injecting Attribute Code into Class Code		class {Ph1} { ClassTemplate c;
		Code() { if(c == null) c = Class::Code();
		AttributeTemplate a = {Ph2}::Code();
		c.AddAttribute(a) } }

Injecting Type into Attribute Code		class {Ph2} { AttributeTemplate a;
		Code() { if(a == null) a = Attribute::Code();
		a.SetType({Ph1});
		return a; } }

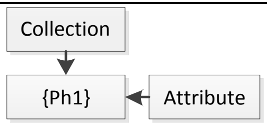
Injecting Collection Semantic into Attribute Code		class {Ph1} { AttribueTemplate a;
		Code() { if(a == null) a = Attribute::Code();
		a.SetAsCollection();
		return a; } }

TABLE III. A MORE COMPLEX CODE GENERATION PATTERN, CORRESPONDING TO CLASS REIFICATION

NAME	PATTERN	GENERATION LOGIC

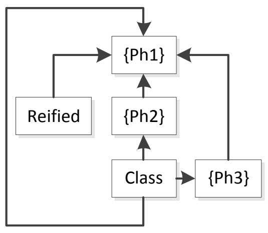
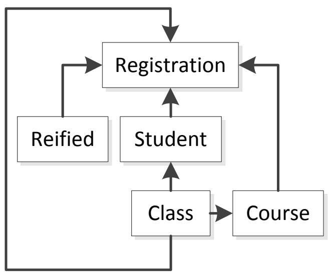
Reified-Class Code Generation		class {Ph1} {
		ClassTemplate c;
		Code() {
		c.SetName({Ph1});
		AttributeTemplate a1 = new AttributeTemplate();
		a1.SetType({Ph2});
		a2.SetName(lowercase({Ph2}));
		c.AddAttribute(a1);
		AttributeTemplate a2 = new AttributeTemplate();
		a2.SetType({Ph3});
		a2.SetName(lowercase({Ph3}));
		c.AddAttribute(a2);
		AttributeTemplate a = new AttributeTemplate();
		a.SetType(c);
		a.SetName(plural(lowercase({Ph1})));
		a.SetAsCollection();
		a.SetAsStatic();
		c.AddAtribute(a);
		MethodTemplate m = new MethodTemplate();
		m.SetName("Add"+{Ph1});
		m.AddParameter(1, {Ph2}, lowercase({Ph3}));
		m.AddParameter(2, {Ph3}, lowercase({Ph3}));
		CreateTemplate cr =
		new CreateTemplate({Ph1}, m.GetParameter(1), m.GetParameter(2));
		AddToCollectionTemplate ad =
		new AddToCollectionTemplate(cr.GetResult(), a);
		m.AddStatement(1, (StatementTemplate)cr);
		m.AddStatement(2, (StatementTemplate)ad);
		c.AddMethod(m);
		return c;
		}
		}

TABLE IV. EXAMPLE OF APPLYING THE CODE GENERATION PATTERN SHOWN IN TABLE III (CLASS REIFICATION)

CONCRETE IVY	TARGET CODE

	class Registration {
	Student student;
	Course course;
	Registration(Student student, Course course) {
	this.student = student;
	this.course = course;
	}
	static Collection<Registration> registrations;
	static AddRegistration(Student student, Course course) {
	Registration registration = Registration(student, Course);
	registrations.Add(registration);
	}
	}