Elemental design patterns

Elemental design patterns (EDPs) were proposed by Smith and Stotts [4]. They provide solutions to very common programming problems (we can state that these problems occur in the everyday practice of each programmer). They share the same aim with the design patterns, but they are applied to more restricted and specific issues. In fact, if design patterns propose solutions to problems which can involve a certain number of classes, EDPs address problems of much more limited dimensions, which generally do not involve more than three classes. There are 16 EDPs subdivided into three categories:

- *Object Elements*: contains three EDPs related to the creation and the referencing of objects as well as to the presence of abstract methods inside an abstract class, or interface methods inside an interface;
- *Method Call*: collects twelve EDPs which represent the various forms of possible method calls;
- *Type Relation*: contains a single EDP representing the inheritance relationship between two classes, as well as the implementation of an interface.

EDPs are defined with the same description structure used in [2] for the presentation of design patterns. For a complete description of each EDP refer to [5]. EDPs can be detected inside Java systems through the *Micro-structures detector* module, which is part of the MARPLE (Metrics and Architecture REcontruction PLugin for Eclipse) project [1]. MARPLE has been developed for design pattern detection and software architecture reconstruction purposes, and it is based on the detection of micro-structures. However, the definition of EDPs originated by considerations made on C++ source code. Smith and Stotts developed SPQR [6], an approach to design pattern detection based on the identification of EDPs inside the subject systems.

Towards a more formal definition of EDPs

In [9] we proposed to introduce a unique catalogue of micro-structures that resembles EDPs [5], clues [7, 8] and micro patterns [3] redefining them in terms of concepts that are common to all the categories of micro-structures we consider, and that will be exploited in their definitions. We call these common concepts *code atoms* (or *atoms* for brevity). Code atoms are simple code elements (more than the micro-structures) that will be used to provide a new and more formal definition of micro-structures. In the new definition of micro-structures, we will use these atoms, and eventually any micro-structure the element to be defined depends on. The new definition will provide an unambiguous meaning to each micro-structure, and will generate a unique catalogue of micro-structures based only on common concepts.

The elements and concepts that will be used in the definition of the atoms and of the micro-structures will now be defined. These concepts are strictly related with the object-oriented paradigm. As we focused in particular on Java systems, we will consider this language as our target.
Any object-oriented system is based on the key concept of type. We will use *T* to denote a type. A type can itself be either a class (denoted by *C*) or an interface (denoted by *I*). If we deal with a set of types, classes or interfaces, each of them will be specified by an index: *Ti* will be therefore the *i*-th type out of a set of n types *T1, …, Tn.* The same considerations reflect on classes and interfaces.
Given a type *T*, we can obtain information about it through the following statements:

- *name(T)*: it represents the qualified name of the type, i.e. the name of the class or interface denoting it, preceded by its package name;
- *attributes(T)*: it represents the set of attributes that have been defined by *T*;
- *methods(T)*: it represents the set of methods that have been defined by *T*;
- *inst(T)*: it represents a generic instance of *T*, that can have been created either within *T* (therefore it can be handled as an attribute of *T*) or within another type.

Given an attribute $a \in attributes(T)$, the following statements are defined:

- *name(a)*: it represents the name of attribute *a*;
- *typeOf(a)*: it represent the type of *a*, which can be either a simple type, a type *T*, or a list of n attributes *list(a1, …, an)*;

Given a method $m \in methods(T)$, the following statements are defined:

- *name(m)*: it represents the name of method *m*;
- *constructor(m)*: it represents the fact that method *m* is a constructor;
- *returnType(m)*: it represents the return type of the method *m*, that can be either a type *T*, a simple type, or void;
- *params(m)*: it represents the set of formal parameters received in input by method *m*;
- *body(m)*: it represents the body of method *m*, i.e. all the statements and operations defined by the method. The body could also be empty: this aspect is represented by the "is empty" clause. The body can itself contain instances of atoms or micro-structures, or other kinds of statements: this containment aspect is specified by the "contains" relationship;
- *returnedValue(m)*: it represents a single returned value of the method;

- *returnStatements(m)*: it represents the set of return statements or return points specified by the method implementation;
- *typeOf(m)*: it represents the type that defined method *m*. As $m \in methods(T)$, therefore *typeOf(m) = T*.

Given two methods *m1* and *m2*, *m1 = m2* will indicate that the two methods have the same signature.
Within the body of a method we can find two special elements, that we call *containers*: they are *controlStatement*, which represents all the control structures that are available in the reference programming language (e.g. in Java, if and switch blocks), and *loop*, which represents all kind of loops available in the reference programming language (e.g. for, while, do-while, enhanced for). Both *controlStatement* and *loop* may operate on a set of parameters:

- *param(controlStatement), param(loop)*: it represents the set of attributes handled by the control statement or loop structure.

Another kind of statement that needs to be considered in order to correctly define the sets of atoms and micro-structures is the method invocation between two methods:

- *methodInvocation(m1, m2)*: it represents the invocation of method *m2* occurring within the body of method *m1*;

Given a method invocation, the following properties can be obtained:

- *source(methodInvocation)*: the actual object invoking *m2*, that is an instance of *typeOf(m1)*;
- *target(methodInvocation)*: the actual object on which *m2* is invoked, that is an instance of *typeOf(m2)*;
- *params(methodInvocation)*: it represents the set of actual parameters passed to the method invocation.

On both types, attributes and methods we can use the logical operators $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$, $\exists!$, according to their usual meaning. Moreover, we use the cardinality operator $||$ to obtain the number of elements composing a specific set (e.g. $|methods(T)|$ will return the number of methods defined in *T*). Finally the operator "is" will be used to declare that a type, an attribute or a method must satisfy a particular modifier (e.g. "*a* is private" means that the attribute *a* must be defined private).

Now that we have introduced the notions and concepts that will guide us in the definition and specification of micro-structures, each of them can be defined (according to its definition) on a type, on an attribute, or on a method:

- *micro_structure_name(T)*: the micro-structure is defined on type *T;*
- *micro_structure_name(a)*: the micro-structure is defined on attribute *a;*
- *micro_structure_name(m)*: the micro-structure is defined on method *m.*

However, the largest part of micro-structures represents information relating two entities; in this case, both the source of the micro-structure (i.e. the entity that actually represents it) and its destination (i.e. the entity the micro-structure depends on) must be specified (for example, *micro_structure_name(T1, T2)* represents a micro-structure that is implemented in *T1*, but whose existence is strictly related to *T2*).
In this document, we provide the definitions for each elemental design patterns according to the concepts presented so far. For a complete overview of code atoms, clues and micro patterns, please refer to [9].

Elemental design patterns definitions

EDP category	**EDP name**	**Definition**	**Explanation [5]**
Object Elements	Abstract interface	*AbstractInterface(m)* iff *interface(typeOf(m))* ∨ (*abstract class(typeOf(m))* ∧ *abstract method(m)*)	It provides a common interface for operating on an object type family, but delaying definition of the actual operations to a later time.
	Retrieve	*Retrieve(o)* iff *o* ∈ *attributes(C)* ∧ ∃ *assignment(o, value)*: *value = returnedValue(m)* ∨ *value = o2* ∈ *attributes(C2)* ∧ *typeOf(o) = typeOf(o2)*	To use an object from another non-local source in the local scope, thereby creating a relationship and tie between the local object and the remote one.
Type Relation	Inheritance	*Inheritance(T1, T2)* iff *interface inherited(T1, T2)* ∨ *class inherited(T1, T2)*	To use all of another classes' interface, and all or some of its implementation.

Table 1 *– Object Elements and Type Relation EDPs definitions*

EDP category	EDP name	Definition	Explanation [5]
Method Call	Recursion	*Recursion(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) = target(method invocation(m1, m2)* ∧ *typeOf(m1) = typeOf(m2)* ∧ *signature(m1) = signature(m2)*	To accomplish a larger task by performing many smaller similar tasks, using the same object state.
	Conglomeration	*Conglomeration(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) = target(method invocation(m1, m2)* ∧ *typeOf(m1) = typeOf(m2)* ∧ *signature(m1) ≠ signature(m2)*	To bring together, or conglomerate, diverse operations and behaviours to complete a more complex task within a single object.
	Extend method	*Extend method(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) = target(method invocation(m1, m2)* ∧ *Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) = signature(m2)*	Add to, not replace, behaviour in a method of a superclass while reusing existing code.
	Revert method	*Revert method(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) = target(method invocation(m1, m2)* ∧ *Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) ≠ signature(m2)*	Bypass the current class' implementation of a method, and instead use the superclass' implementation, reverting to an 'earlier' method body.
	Redirect	*Redirect(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *typeOf(m1) ≠ typeOf(m2)* ∧ *signature(m1) = signature(m2)*	To request that another object perform a tightly related subtask to the task at hand, perhaps performing the basic work.
	Delegate	*Delegate(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *typeOf(m1) ≠ typeOf(m2)* ∧ *signature(m1) ≠ signature(m2)*	To parcel out, or delegate, a portion of the current work to another method in another object.

Table 2 *– The first six Method Call EDPs definitions*

EDP category	EDP name	Definition	Explanation [5]
Method Call	Redirected recursion	*Redirected recursion(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *typeOf(m1) = typeOf(m2)* ∧ *signature(m1) = signature(m2)*	To perform a recursive method, but one that requires interacting with multiple objects of the same type.
	Delegated conglomeration	*Delegated conglomeration(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *typeOf(m1) = typeOf(m2)* ∧ *signature(m1) ≠ signature(m2)*	A Conglomeration pattern is appropriate, but we need to work with a distinct instance of our object type, resulting in a need for the Delegate pattern to be used.
	Redirect in family	*Redirect in family(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) = signature(m2)*	Redirect some portion of a method's implementation to a possible cluster of classes, of which the current class is a member.
	Delegate in family	*Delegate in family(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) ≠ signature(m2)*	Related classes are often defined as such to perform tasks collectively. In such cases, multiple objects of related types can interact in generalized ways to delegate tasks to one another.
	Redirect in limited family	*Redirect in limited family(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ ∃ *T*: *ancestor(typeOf(m1), T)* ∧ *ancestor(typeOf(m2), T)* ∧ ¬*Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) = signature(m2)*	When Redirect in family is too generalized, and it is necessary to pre-select a sub-tree of the class hierarchy for polymorphism.
	Delegate in limited family	*Delegate in limited family(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ ∃ *T*: *ancestor(typeOf(m1), T)* ∧ *ancestor(typeOf(m2), T)* ∧ ¬*Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) ≠ signature(m2)*	When Delegate in family is too generalized, and it is necessary to pre-select a sub-tree of the class hierarchy for polymorphism.

Table 3 *– The second six Method Call EDPs definitions*

References

[1] F. Arcelli et al., MARPLE: Metrics and Architecture Recovery Plug-in for Eclipse, Technical Report, Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo)-12-02-06, University of Milano-Bicocca, 2006.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994.

[3] Y. Gil, I. Maman, Micro Patterns in Java Code, in Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA '05), October 2005, pp. 97-116.

[4] J. McC. Smith, D. Stotts, Elemental Design Patterns: A Formal Semantics for Composition of OO Software Architecture, in Proceedings of the 27th Annual IEEE/NASA Software Engineering Laboratory Workshop, Greenbelt, MD, 2002, pp. 183-190.

[5] J. McC. Smith, An Elemental Design Patterns Catalog, Tech. Rep. 02-040, Computer Science Department, University of North Carolina at Chapel Hill, December 2002.

[6] J. McC. Smith, D. Stotts, SPQR: Flexible Automated Design Pattern Extraction From Source Code, in Proceedings of the 2003 IEEE International Conference on Automated Software Engineering, Montreal QC, Canada, October, 2003, pp. 215-224.

[7] S. Maggioni, Design Pattern Clues for Creational Design Patterns, Proceedings of the 1st International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE 2006), Benevento, Italy, October 2006.

[8] S. Maggioni, F. Arcelli, C. Tosi, M. Zanoni, Refining Design Pattern Detection through Design Pattern Clues, submitted to the Journal of Systems and Software, July 2009.

[9] S. Maggioni, Design Pattern Detection and Software Architecture Reconstruction: an Integrated Approach based on Software Micro-structures, Ph.D. Thesis, Milano-Bicocca University, 2009.