Cybernetics and Systems Analysis, Vol. 48, No. 1, January, 2012

CREATIONAL DESIGN PATTERNS IN COMPUTATIONAL LINGUISTICS: FACTORY METHOD, PROTOTYPE, SINGLETON

A. A. Nykonenko

UDC 681.3

Abstract. *The paper analyzes the use of creational patterns in solving computational linguistics problems. The Factory Method, Prototype, and Singleton patterns are considered in detail. The basic properties of patterns and the nature and characteristics of their use are reviewed, and the comparative analysis with other creational patterns is carried out. The structure of patterns and their possible applications in software systems to solve computational linguistics problems is considered separately. For each pattern, the conditions are presented under which its use is most appropriate.*

Keywords: *design patterns, computational linguistics, natural-language text processing*

INTRODUCTION

This paper continues the study on usage of patterns in the architectural design of linguistic information systems. The paper [1] explains the basics of patterns needed to understand what is discussed here. Since the structure and specific applications, strong and weak aspects of patterns were described earlier, we will outline only three patterns: Factory Method, Prototype, and Singleton, to complete the discussion of creational patterns initiated in [1]. A general description of this category of patterns and types of problems to which they are recommended to apply can be found in the Gang of Four's catalog [2] containing five creational patterns.

Let us now go straight to patterns (some diagrams and citations from [2] will be used below).

PATTERN FACTORY METHOD

Intent. The pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. The factory method allows a class to defer instantiation to subclasses.

The essence of the pattern is simple enough: define an abstract class (Creator) that declares a method (Factory Method) returning objects of other classes (products). All these products are inherited from the abstract class Product. Next, each of the instances (ConcreteCreator) of the class Creator overrides the Factory Method to return a specific product (ConcreteProduct). The abstract class Creator may also define a default implementation of the Factory Method.

This approach allows a client to create a necessary ConcreteCreator, which will automatically produce specific classes of correct type. The pattern is schematized in Fig. 1.

Functionality. As suggested by its name, the Factory Method Pattern is very similar to Abstract Factory Pattern. The former creates just one object, while the latter creates families of related objects. Another feature of the pattern is additional operations (such as the method Creator->AnOperation() in Fig. 1, but there may be several such methods). It is agreed that these methods define the generic behavior of products of any kind; subclasses override the factory method alone.

If the Abstract Factory Pattern is an extension of the Factory Method Pattern, which creates only one product and includes additional methods not overridden in subclasses, then the question arises of why this solution is arranged as a separate template? The answer is that despite similar structure, these patterns carry out different architectural tasks.

Taras Shevchenko National University of Kyiv, Kyiv, Ukraine, *andrey.nikonenko@gmail.com*. Translated from Kibernetika i Sistemnyi Analiz, No. 1, pp. 163–174, January–February 2012. Original article submitted October 14, 2010.

 1060-0396/12/4801-0138

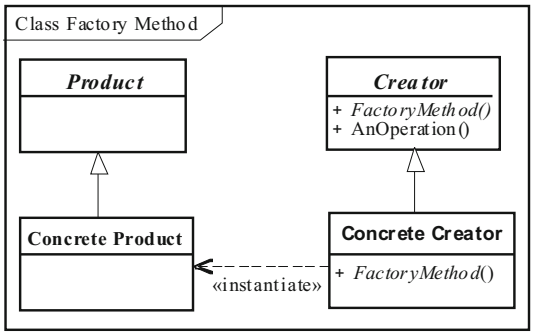

Fig. 1. Structure of Factory Method Pattern.

Patterns in which one class has several factory methods are most similar. Unlike the Abstract Factory, it is not required to override all factory methods in a subclass. Differently, if the Abstract Factory has two methods CreateProductA and CreateProductB, then one subclass of factory (Factory1) will create ProductA1 and ProductB1, and the other subclass (Factory2) will create ProductA2 and ProductB2 because the factory always produces families of related products.

If the Factory Method Pattern is used instead, then with the factory methods CreateProductA and CreateProductB in the Creator class, it is possible to create subclasses of the Creator that would override only one of the factory methods (provided that the Creator has a default method). It is also possible to create the Creator's subclass that would produce ProductA1 and ProductB2.

Thus, the Factory Method is one of the methods to implement the Abstract Factory Pattern.

It is most appropriate to use the Factory Method when the complex behavior of a class should be changed in subclasses. Not to override the entire code implementing this behavior (the source code of a class is not always available), its changeable part is defined as a separate product class. The complex behavior to be changed in subclasses is defined by the AnOperation method of the Creator class.

The AnOperation method uses an object of the ConcreteProduct class, which implements the interface of the Product class. In other words, the variable part of the complex behavior of Creator->AnOperation is now belongs to the Product class. It remains to answer the last question: How to create necessary subclasses of the Product class without the need to override the method AnOperation in descendants of the Creator class? The answer is to use the Factory Method Pattern. AnOperation does not create objects of the Product class, but calls the special operation FactoryMethod to do that. For ConcreteCreator in AnOperation to use an instance of the class ConcreteProduct2 instead of an instance of the class ConcreteProduct1, it is sufficient to override ConcreteCreator->FactoryMethod so that it would return an instance of the class ConcreteProduct2.

Thus, the Factory Method Pattern makes it possible to change the complex behavior of a class by overriding just a part of the code implementing this behavior.

Let a class have several methods defining complex behavior that needs to be configured. One way is to create in this class so many factory methods as there are behavioral methods. In other words, for the Creator class having three such methods (AnOperation1, AnOperation2, AnOperation3), it is necessary to create three factory methods (FactoryMethod1, FactoryMethod2, FactoryMethod3).

To decrease the number of factory methods, it is possible to apply the parametrized factory method. The parameter specifies the type of the created product. Such a design includes only one factory method (FactoryMethod), and its parameter indicates the operation for which the product is created. Such an approach makes the overriding of FactoryMethod in subclasses somewhat specific in the situation where in one of the subclasses it is necessary to change only the products created for, for example, AnOperation1 and to leave the products for other operations unchanged. Since it is only possible to override the whole method, making changes in the way new products are created for AnOperation1 in FactoryMethod, we lose the old behavior responsible for the creation of products for AnOperation2 and AnOperation3.

A branching is a reasonable way out. A branching condition is the parameter value obtained with the factory method. If the parameter indicates the need to create an object for AnOperation1, then a new code is executed; otherwise, FactoryMethod of the parent class is called.

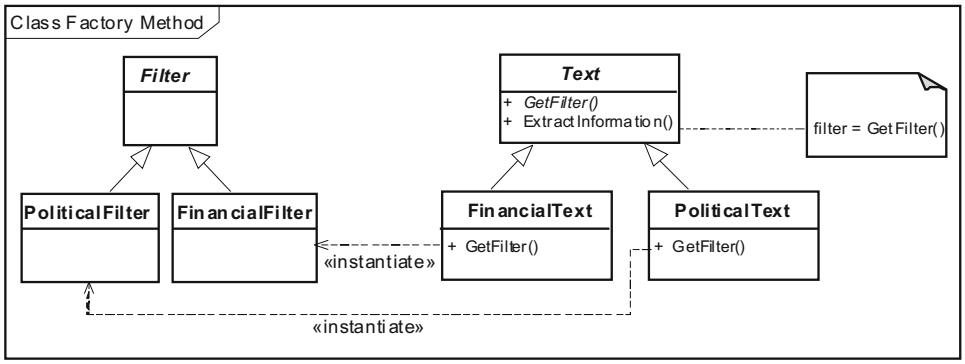

Fig. 2. An example of using the Factory Method Pattern in a text data retrieval system.

Factory Method Pattern in Computer Linguistics. As mentioned above, the pattern is used when different subclasses of one parent class require different methods for the same complex operation on an object. In linguistic applications, a typical object is a text. Therefore, we will consider, as an example, the architecture of a text data retrieval system (Fig. 2).

This system is intended to retrieve significant information from texts. Significant data in financial texts are amount of capital invested, stock quotes, change in interest rates, cross rates, etc. Significant data in political texts are actions of political leaders, dates and places of meetings, statements, contracts, etc.

The architecture of the system is such that there is a single class Text containing a default data retrieval method (ExtractInformation). Texts of specific subjects will be represented by the subclasses FinancialText and PoliticalText of the class Text. The standard method ExtractInformation is insufficient to extract information from texts. It should be modified so as to take into account the subject of the text in identifying significant information.

Assume that a text analysis algorithm is standard for texts of any subject. Only information filtration methods will depend on the subject of the text. Therefore, we will create a special class Filter responsible for the identification of significant information. Its subclasses FinancialFilter and PoliticalFilter will extract financial and political information from texts.

The text analysis algorithm in Text->ExtractInformation uses these filters. However, it does not create objects of the Filter class by itself, but uses the factory method GetFilter for this purpose. The subclasses FinancialText and PoliticalText override the method GetFilter so that it returns instances of the classes FinancialFilter and PoliticalFilter, respectively. If it is needed to extend the system to include a sports text analyzer, it will be sufficient to create a class SportFilter and to add a subclass SportText of the class Text. The method GetFilter returning filters of different classes is the only difference between the class SportText and, for example, the class FinancialText.

Applicability. The factory method is used when:

- a class cannot anticipate the class of objects it must create;
- a class wants its subclasses to specify the objects it creates;
- A class delegates responsibilities to one of several helper subclasses, and it is necessary to localize the information on which of the subclasses is the delegate.

PROTOTYPE PATTERN

Intent. The pattern specifies the kind of objects to create using a prototypical instance and creates new objects by cloning this prototype.

The essence of the pattern is as follows. If Client wants to create objects of different subclasses of the Prototype class, it is parametrized with a prototype of the subclass desired. In other words, the parameter is an object of one of the subclasses of the Prototype class. If Client wants a new instance, it calls the operation Prototype->Clone which returns a new object of the same class as the parameter. The pattern is schematized in Fig. 3.

Functionality. A significant shortcoming of the Factory Method Pattern is a high hierarchy of classes. The last example for the Factory Method Pattern demonstrates that adding a new filter requires adding a new subclass of the Text class with the only task to create a correct filter. Differently, adding a new product subclass leads to adding a new factory subclass. In the case described above, such an extension of the system is not critical. However, if the number of filters (the Filter class in the example is a product) in the system increases to 20, then the number of subclasses of the Text class (which is a factory) will also increase to 20. A system with more than 40 classes is very difficult to maintain.

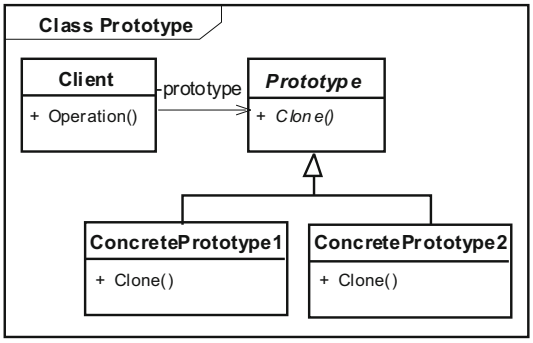

Fig. 3. Structure of Prototype pattern.

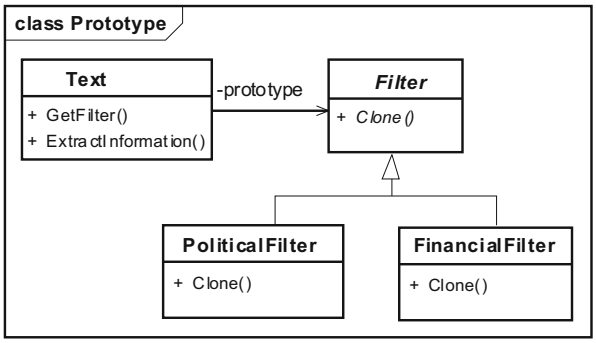

Fig. 4. Prototype Pattern in text data retrieval system.

The Prototype Pattern offers a solution avoiding high interrelated hierarchies of classes. The solution of the above problem with the help of the Prototype Pattern is shown in Fig. 4.

In this case, there is no need for the hierarchy of the subclasses of the Text class, which should be regularly extended to include a new filter. In the previous example, the client must create a correct subclass of the Text class that would correctly override the operation GetFilter, whereas in this case, the client must initialize the Text class with a correct filter prototype, which will then be cloned and used in the application.

The above example demonstrates the reduction in the hierarchy of classes provided by the Prototype pattern; however, it is not ideal for solving the problem. The solution based on the Factory Method Pattern is more suitable because the Prototype Pattern is intended to solve somewhat different problems. If it is applied in the given context, the client should create prototype classes only to store in the Text class in the form of templates. To use this object, the Text class should call the cloning method. Thus, we have two copies of the same class one of which is obviously redundant. In that case, it is more natural to change of the method GetFilter so that it returns a class rather than its clone.

A specific property of the pattern is also that the configuration of cloned objects coincides with the configuration of the parent object, i.e., both objects have identical values of parameters by default. This circumstance can also be used to reduce the hierarchy of classes. Suppose we are developing a text cataloging system that places arrived texts in folders of given subjects. The list of subjects must be easily extendable and configurable. In other words, it is necessary to create a parent class Thematic whose subclasses are specific for each subjects, such as FinancialThematic, PoliticalThematic, SportThematic, EcologicalThematic, etc. This involves two difficulties.

- A good cataloging system requires creating a great number of subjects. Hence, there will be a great number of classes, and it will be difficult to maintain and change such a system.
- If the system allows the user to create new and delete old subjects, then it should be possible to dynamically add and remove classes in the system.

The Prototype pattern can accomplish both these tasks. In the former case, it is necessary to change the structure of the system's classes. Instead of creating subclasses of the Thematic class, we will configure it with different parameters to represent different subjects. Let a subject be simply specified by a set of keywords. To represent different subjects, it is necessary to create instances of the Thematic class initialized with various key sets. Doing so gives objects Thematic1, Thematic2, and Thematic3, each representing a specific subject. If there is a need for one more instance, such as object Thematic1, to create a subsubject, the method Thematic1->Clone can be used.

In the latter case, a static set of classes cannot be used because the structure of the system must be changed dynamically. However, it is possible to use the above model in which all classes are created dynamically. In this model, it is not difficult to add new and remove old classes because instances of objects play the role of classes (so-called class objects). Moreover, such a model allows easy storing of the structure of classes in external memory, from which it can then be easily read and used to create necessary objects. Thus, the system can be configured dynamically at runtime, which is what is required. If the system is intended to dynamically load a great number of various prototypes, then it is recommended to use a prototype manager, which is a special storage that is responsible for rolling-in, rolling-out, and controlling prototypes.

The benefits of the Prototype pattern can be briefly summarized as follows:

1. Adding and removing classes at runtime, i.e., dynamic creation of new classes.

2. Dynamic configuration of the application with classes, i.e., loading of new classes at runtime.

3. Creation of new class objects by configuring parameters, i.e., cloning the parent class in the necessary state rather than creating subclasses.

4. Reduction of the number of subclasses, i.e., one hierarchy of products instead of parallel hierarchies of classes (example of an alternative architecture for a text data retrieval system).

5. Creation of new classes by grouping the existing ones, i.e., storing of a set of objects as one class and using the Clone method to create copies of this class. If the subjects of texts in the above system are specified by a set of texts of similar subjects rather than by keywords, then it is possible to use the Prototype pattern to create new subjects. In this case, the user could select a set of objects of Text type and store it as an aggregate object. This object could be stored in external memory or cloned.

The main difficulty of implementing this pattern is the Clone method. First, if the system employs classes whose source codes are unavailable, it is very difficult to add this method. Moreover, the Clone method involves the issue of deep/shallow copy related to processing of references to other objects stored by a copied (cloned) object. Shallow copying means that the original object and its clone share references, i.e., both of them refer to the same object. Shallow copying is often implemented as default in modern programming languages (such as Java). However, shallow copying is frequently insufficient because the clone shares the components with the original and, thus, depends on it.

The class implementing the Clone method must have a copy constructor that uses an object of its class as an input parameter. It is this constructor that will be used in the Clone method to clone a prototype. Classes that provide cloning sometimes include the Initialize method to change the values of some parameters of the clone after cloning.

Prototype Pattern in Computer Linguistics. The benefits of the Prototype pattern have been discussed above. Despite a wide set of useful properties, the pattern is most frequently used in systems that allow the user to create objects from components. The ampler opportunities are provided by the system to configure the components and to construct objects from them, the more useful the Prototype pattern is.

Let us consider, as an example, the architecture of a system for finding texts with given information. Such a system allows the user to construct special filters for further use in logical expressions. All input texts are tested by the system against the expression. The user receives only those texts for which the expression is true.

The filter is a set of templates for the arrangement of words in a sentence. For example, the class TimeFilter may be set by the following templates:

— *[0−9]{2}:[0−9]{2}:[0−9]{2}[0−9]{2}:[0−9]{2}.[0−9]4* to represent time as hours:minutes:seconds day.month.year (for example 11:30:22 12.09.2010);

— *[0−9]{2}:[0−9]{2}:[0−9]{2}* to represent time as hours:minutes:seconds (for example 11:30:22);

— *[0−9]{2}:[0−9]{2}* to represent time as hours:minutes (for example 11:30);

— *[0−9]{2}−[0−9]{2}* to represent time as hours-minutes (for example 11-30);

— *[0−9]{2}−[0−9]{2}. (am, pm) to represent time as hours-minutes am/pm (for example, 11-30 am or 11-30 pm), and other templates.

The creation of the class PlaceFilter requires wider functionality, combination of regular expressions (as in the templates above) with syntactic/morphological templates, such as:

— *"City"*, where “City” is a morphological template for all cities present in the morphological database;

— *"Area"*;

— *"Street"*;

— *(near, nearby, next to, close to...) “Noun”*

and other templates.

The class PersonFilter must recognize capitalized words, apply morphological filters to them, and analyze syntactic expressions.

The class ActionFilter must find word combinations “verb∗noun” (earned∗money, bought∗house) and verbs (won, was∗elected) in a text.

The basic version of the system is assumed to include a certain number of initial preconfigured filters, such as those listed above. However, the user must have an option to add new filters, such as top-level filters (for example, a disaster filter) or subfilters (for example, a country filter as a subtype of PlaceFilter).

Next, the user can derive specific value classes from each filter. Each such class is a filter configured by a list of exact values, such as the class Berlin, a subclass of PlaceFilter, in which the list of admissible values consists of “Berlin” and “berlin.”

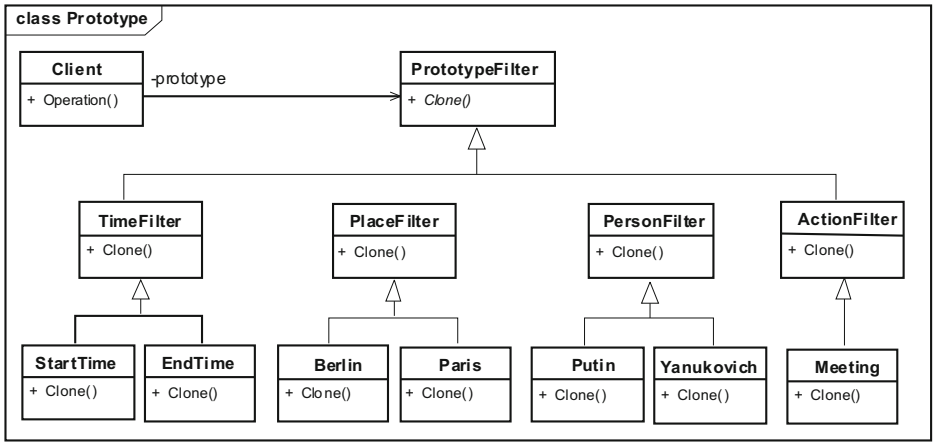

Fig. 5. Example of using Prototype pattern in data filtering system.

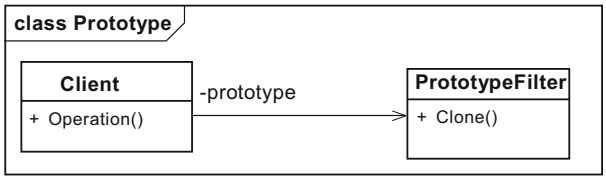

Fig. 6. Architecture of data filtering system

All the above information will be used to search for documents in which the user is interested. Suppose the user wants to find documents on the meeting between Putin and Yanukovych took place in Berlin in September 2010. He can use the classes described above to create classes StartTime configured to September 1, 2010, and EndTime configured to September 3, 2010, and to create a class Berlin as a subclass of the class PlaceFilter, classes Putin and Yanukovich as subclasses of the class PersonFilter, and a class Meeting as a subclass of ActionFilter.

Next, it is necessary to use these classes to specify a logical expression. For this example, we have:

(PersonFilter=Putin and PersonFilter=Yanukovich) and (TimeFilter>=StartTime and TimeFilter<=EndTime) and (PlaceFilter=Berlin) and (ActionFilter=Meeting).

Now the system can find all texts with information of interest.

Figure 5 shows not the architecture of the system, but rather the classes that are created in the example. Since the system is designed using the Prototype pattern, its real architecture consists of only two elements indicated in Fig. 6. The other subclasses of the class PrototypeFilter are created as class objects. The user can store the created filters in external memory, to add filters created by other users to the system, to delete unnecessary filters, etc., i.e., he can enjoy all the advantages of the flexible design of class hierarchy.

Applicability. The prototype is used when the system does not depend on the creation methods, configuration and representation of products. Moreover:

- the created classes are defined at runtime (for example through dynamic loading);
- it is necessary to avoid constructing class hierarchies or factories that are parallel to the hierarchy of product classes;
- copies of a class can be in one of a small number of states; it is sometimes reasonable to define the appropriate number of prototypes and to clone them, instead of creating a class in the required state manually on a regular basis.

SINGLETON PATTERN

Intent. Ensure a class has only one instance, and provide a global point of access to it.

The essence of the Singleton pattern is as follows. If there can be no more than one instance of a class, then a special static method (Instance) is declared in that class to provide access to its instance. The Instance method has access to a variable storing a reference to an instance of the class. If the reference is empty when the instance is accessed, the Instance method creates an instance and returns a reference to it; otherwise, it returns a reference to the already existing instance. The class constructor is protected to make the use of other instantiation methods impossible. Thus, the instance can only be accessed through the Instance method.

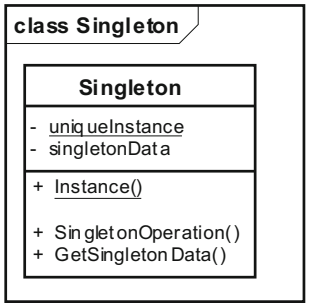

Fig. 7. Structure of the Singleton pattern

Functionality. Singleton implements a simple idea: one class, one instance, i.e., it is applied to systems that restrict the number of instances of some classes. Such classes may be various managers controlling all other elements of the system. For example, a window manager may be such a class for an operating system. Note that Singleton that creates only one its instance is a special case. Actually, the pattern can be configured so as to create any number of instances.

The Singleton pattern is to some extent similar to the Prototype pattern because it also creates its own instances. Moreover, the Singleton pattern, as well as the Prototype pattern, is based on object–class structure, though created in a different way. Despite formal resemblance, the implementations of the patterns are substantially different. The Singleton pattern is a class with hidden constructor; therefore, this class can be instantiated only through the Instance operation. Accordingly, this method is declared static, i.e., as a class method. Calling this method returns a reference to the instance of the class, this reference being stored in the static variable uniqueInstance. If the reference appears empty (the class has not yet been instantiated), the constructor is called to create an instance of the class. The reference to the new instance is stored in the variable uniqueInstance.

This approach allows the client to ignore the time the class Singleton was instantiated and to use it as if a necessary instance exists at any time while the system operates. The class itself is responsible for creating its instances. An additional advantage of this approach (postponed initialization) is that the system is not overloaded with nonused objects because the class Singleton is instantiated at the time of access, i.e., when the system needs it indeed.

Another advantage of the pattern is the global accessibility of an object without the use of global variables. This means that an object created with Singleton has a single instances in the system; therefore, a global variable is the most reasonable storage for the reference to such an object because it provides the simplest access to the object from any place of the code. The Singleton pattern saves the programmer the trouble of creating global variables to access its instances. Any instance of the Singleton class can be directly accessed through the class name. This pattern allows easy change of the number of allowed instances. For two and more instances, it is necessary to change only the access operation, i.e., the Instance method, and to add static variables for storing the references to the additional instances, for example, to add a parameter specifying the instance number to the Instance method and to implement a branch returning a reference to the instance indicated by this parameter.

Subclassing the Singleton Class is a More Difficult Task. The difficulty lies not in the definition of a subclass, but in the instantiation method because the Instance method creates instances of the Singleton class instead of instances of a subclass. There are three methods to solve this problem.

1. The Instance operation in subclasses is redefined so that it creates instances of the class needed.

2. A branch responsible for creating the correct class is implemented in the Singleton's Instance operation. The branch is defined using an environment variable (it is assumed that Singleton is used to represent a single global object of the system; therefore, such an approach is valid).

3. The most flexible approach is to use a registry of Singletons, a list of Singletons' names and their classes. To create a class, the Instance method identifies its type by consulting the registry using its name. As in 2, the Singleton name is stored in an environment variable.

Singleton Pattern in Computer Linguistics. As described above, it is reasonable to use the Singleton pattern in a class representing a sole large control element. It can also be used in combination with other patterns. Let us consider, as an example, the application of the Singleton Pattern together with the Abstract Factory Pattern. In [1], the AbstractFactory pattern is schematized as an example of a multifunctional linguistic system. First, a needed subclass of HandlersFactory is created to produce concrete products of "correct" classes. Hence, the Singleton pattern should be applied to the class HandlersFactory. Next, it is necessary to create subclasses of the Singleton class. Three possible methods to subclass the Singleton class was discussed above: creating a branch in the Instance operation, overriding this operation in subclasses, and using a registry of Singletons. Let us apply branching as the simplest approach: in the class HandlersFactory, the Instance

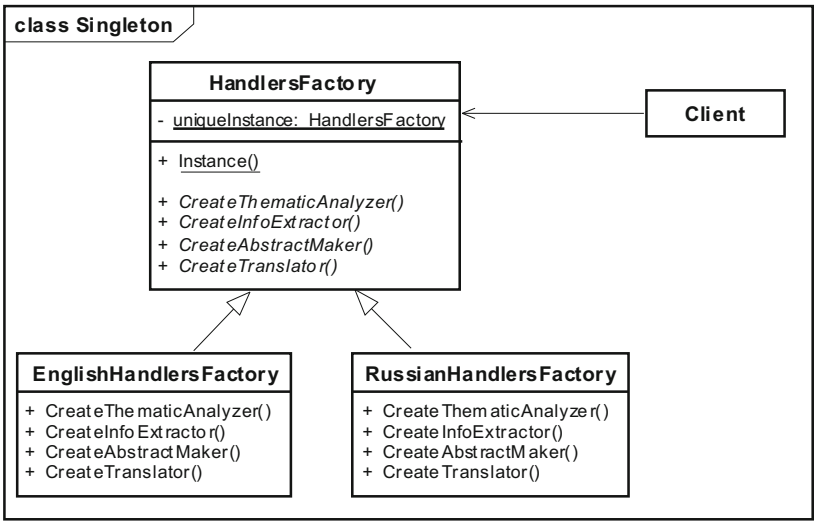

Fig. 8. Example of the architecture of a multifunctional application with the use of the Singleton and AbstractFactory patterns.

method is implemented so that it returns either an instance of the class RussianHandlersFactory or an instance of the class EnglishHandlersFactory, depending on the value of the environment variable lang. If it is planned to change the subclass HandlersFactory at runtime, then after change in the value of the variable lang, the Instance method must be capable of deleting the old subclass HandlersFactory (in Java, the garbage collector performs these functions) and initializing a new subclass.

Figure 8 shows the classes after making corrections. The hierarchy of product classes remains without changes. Note that the class HandlersFactory is no longer abstract, and the Instance method is static. Adding the Singleton pattern to the application, we obtain the following results:

— any time, there can be only one instant of the factory accessible through the method HandlersFactory->Instance;

— the client does not need to create a factory; it is sufficient to set the appropriate value of the variable lang; the needed instance will be created automatically upon first call of the factory.

Applicability. It is reasonable to use the Singleton pattern when:

- there must only one instance of some class easily accessible by all clients;
- the sole instance must be extendable by creating subclasses, and the clients must have an option to use the extended instance without modification of the code.

CONCLUSIONS

Three creational patterns have been discussed: Factory Method, Prototype, Singleton. Their features and application methods have been detailed. The information presented above can be as a manual with recommendations for designing applied systems. The examples of applying the patterns in computer linguistics are complete architectural solutions. They can be used to develop relevant program systems.

The application of each pattern in solving a specific applied problem should be considered individually, taking into account the specific features of the system being developed. Though the majority of the above-mentioned patterns are interchangeable, they are capable of solving a problem with different methods. The above information is intended to help choosing the correct pattern, taking into account unobvious details.

REFERENCES

1. A. A. Nykonenko, “Using design patterns in computer linguistics: Creational patterns. Part I: Abstract Factory and Builder,” Iskusstv. Intellekt, No. 4, 278–286 (2010).
2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Boston (1995).