Elektrotehnički fakultet Istočno Sarajevo

Generički rukovalac dokumentima
Glossary

Verzija 1.1

Istorija revizija

Datum	Verzija	Opis	Autor
14.11.2016	1.0	Glossary	Srđan Kovačević
17.12.2016	1.1	Glossary	Srđan Kovačević

Sadržaj

1. Uvod 4

2. Definicije termina u okviru Java 4

3. Definicije termina u okviru UML 7

Glossary

1. Uvod

Glossary posjeduje set konzistentnih termina, a služi da se izbjegnu nesporazumi prilikom realizovanja projekta. Namjenjen je svim učesnicima projektima, a prvenstveno da razumiju termine vezane za realizaciju projekta.

1.1 Svrha

Glossary je namijenjen:

- developerima – koji će koristiti termine definisane u ovom dokumentu prilikom dizajniranja i implementiranja klasa, tabela u bazi podataka, korisnički interfejs i tako dalje.
- analitičarima – koji će koristiti ovaj dokument da precizno definišu pravila poslovanja, i osiguraju da se čitav projekat realizuje uz korištenje propisanih termina.
- edukatorima i kreatorima dokumentacije – koji će koristiti ovaj document da naprave tutorijale za obuku korisnika i pisanje dokumentacije koristeći poznatu terminologiju.

1.2 Opseg

Opseg glossary je ograničen na trajanje datog projekta. Neki njegovi dijelovi se mogu koristiti kao referenca u drugim projektima, ali kao cjelina ne bi bio odgovarajući.

1.3 Pregled

Glossary je dat u obliku rječnika, gdje su termini navedeni po abecednom redu, a ispod svakog termina nalazi se objašnjenje tog termina. Kakos u termini preuzeti iz zvaničnih dokumenata, prevođenje termina i njihovih opisa narušilo bi njihov integritet, tako da je ostavljen engleski jezik.

2. Definicije termina u okviru Java

Termini koji se koriste prilikom programiranja u Java programskom jeziku su navedeni u nastavku.Definicije su preuzete sa: https://www.cs.kent.ac.uk/people/staff/djb/oop/glossary.html#a .

2.1 Abstract class

A class with the abstract reserved word in its header. Abstract classes are distinguished by the fact that you may not directly construct objects from them using the new operator. An abstract class may have zero or more abstract methods.

2.2 Abstract method

A method with the abstract reserved word in its header. An abstract method has no *method body*. Methods defined in an *interface* are always abstract. The body of an abstract method must be defined in a *sub class* of an *abstract class*, or the body of a class implementing an interface.

2.3 Anonymous object

An object created without an *identifier*. They are usually created as array elements, *actual arguments* or method results. For instance

```
private Point[] vertices = {
   new Point(0,0),
   new Point(0,1),
   new Point(1,1),
   new Point(1,0),
};
```

2.4 Application

Often used simply as a synonym for *program*. However, in Java, the term is particularly used of programs

with a *Graphical User Interface (GUI)* that are not *applets*.

2.5 Array

A fixed-size object that can hold zero or more items of the array's declared type.
array initializer
An initializer for an array. The initializer takes the place of separate creation and initialization steps.

2.6 Boolean

One of Java's *primitive types*. The boolean type has only two values: true and false.
boolean expression
An expression whose result is of type boolean, i.e. gives a value of either true or false. Operators such as && and || take boolean operands and produce a boolean result. The relational operators take operands different types and produce boolean results.

2.7 Bytecode

Java source files are translated by a *compiler* into bytecodes - the *instruction set* of the *Java Virtual Machine (JVM)*. Bytecodes are stored in .class files.

2.8 Call-by-value

A semantics of passing an *argument* to a method in which a *copy* of the *actual argument* value is taken and placed in a separate memory location, represented by the corresponding *formal argument*. As a result, assignment to a formal argument within a method can have no effect on the value stored in the actual argument. This principle is often misunderstood in Java. It does *not* mean that an object referred to by an actual argument cannot be modified via the formal argument.

2.9 Case sensitive

A test that is sensitive to whether a character is upper-case (e.g., 'A') or lower-case (e.g., 'a').

2.10 Cast

Where Java does not permit the use of a source value of one type, it is necessary to use a cast to force the compiler to accept the use for the target type. Care should be taken with casting values of primitive types, because this often involves loss of information. Casts on object references are checked at runtime for legality. A ClassCastException *exception* will be thrown for illegal ones.

2.11 Catch clause

The part of a *try statement* responsible for handling a caught *exception*.

2.12 Class

A programming language concept that allows data and *methods* to be grouped together. The class concept is fundamental to the notion of an *object-oriented programming language*. The methods of a class define the set of permitted operations on the class's data (its *attributes*). This close tie between data and operations means that an *instance* of a class - an *object* - is responsible for responding to messages received via its defining class's methods.

2.13 Class inheritance

When a *super class* is extended by a *sub class*, a class inheritance relationship exists between them. The sub class inherits the methods and attributes of its super class. In Java, class inheritance is *single inheritance*. See *interface inheritance* for an alternative form of inheritance.

2.14 Comment

A piece of text intended for the human reader of a program. Compilers ignore their contents.

2.15 Compiler

A program which performs a process of *compilation* on a program written in a *high level programming language*.

2.16 Exception

An object representing the occurrence of an exceptional circumstance - typically, something that has gone wrong in the smooth running of a program. Exception objects are created from *classes* that extend the Throwable class. See *checked exception* and *unchecked exception*.

2.17 Garbage collector

A *daemon thread* that recycles objects to which there are no extant references within a program.

2.18 Graphical User Interface

A Graphical User Interface (GUI) is part of a program that allows user interaction via graphical components, such as menus, buttons, text areas, etc. Interaction often involves use of a mouse.

2.19 Instance

A synonym for *object*. Objects of a class are *instantiated* when a class *constructor* is invoked via the *new operator*.

2.20 Look-and-feel

The visual impression and interaction style provided by a user interface. This is predominantly the responsibility of the *window manager* (in collaboration with the underlying *operating system*) running on a particular computer. It refers to style of such things as window title bars, how windows are moved and resized, how different operations are performed via a mouse, and so on. It is preferable to have a consistent look and feel within a single user environment. However, some window managers do allow individual programs to present a different look and feel from the predominant style of the host environment. Java's Swing components support this idea by allowing an application to select a `pluggable look and feel' from those provided by a user interface manager. An application running in a Microsoft Windows environment could be made to look like one that normally runs in an X Windows environment, for instance. This allows an application to look similar on different platforms, but it can also lead to confusion for users.

2.21 Main method

The starting point for program execution
public static void main(String[] args)

2.22 Memory leak

A situation in which memory that is no longer being used has not been returned to the pool of free memory. A *garbage collector* is designed to return unreferenced objects to the free memory pool in order to avoid memory leaks.

2.23 Object

An *instance* of a particular *class*. In general, any number of objects may be constructed from a class definition (see *singleton*, however). The class to which an object belongs defines the general characteristics of all instances of that class. Within those characteristics, an object will behave according to the current state of its *attributes* and environment.

3. Definicije termina u okviru UML

Termini koji se koriste prilikom projektovanja softvera u Power Designer-u su navedeni u nastavku. Definicije su preuzete sa: https://en.wikipedia.org/wiki/Glossary_of_Unified_Modeling_Language_terms

3.1 Abstraction
The process of picking out common features and deriving essential characteristics from objects and procedure entities that distinguish it from other kinds of entities.

3.2 Activity diagram
Diagram that describes procedural logic, business process or work flow. An activity diagram contains a number of Activities and connected by Control Flows and Object Flows.

3.3 Actor
Role that a user takes when invoking a use case. Also see actor modeling.

3.4 Aggregation
Special type of association used to represent a stronger relationship between two classes than a regular association; typically read as "owns a", as in, "Class A owns a Class B". A hierarchy of classes where the child object may or may not continue to exist if the parent object is destroyed; see 'composition'. 2. An aggregation is a structural relationship that specifies that one class represents a large thing which constitute of smaller things and represents "has-a" relationship.

3.5 Association
Relationship with 2 or more ends, where each end is on a class (or other classifier). Each end is called a Role, and may have a role name, Multiplicity, and may be Navigable. 2. An association is a structural relationship that specifies that the objects of one thing are connected with the objects of another.

3.6 Cardinality
The current number of occurrences of a Property. The cardinality must be a value that is allowed by the multiplicity

3.7 Class diagram
Type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes and the relationships between the classes.

3.8 Composition
Specific type of relationship describing how one Object is composed of another Object; a form of Aggregation where the child object is destroyed if the parent object is destroyed.

3.9 Dependency

Dependency exists between two defined elements if a change to the definition of one would result in a change to the other. In UML this is indicated by a line pointing from the dependent to the independent element.

3.10 Generalization

Relationship between a specific classifier (typically a class) to a more general classifier asserting that the general classifer contains common features among both the specific classifier and the general classifier. Features include, for example, properties, and constraints. The use of generalization is often logically restricted to cases where the specific classifier is a "kind-of" or "sort-of" the general classifier: for example, a Boxer is a "kind-of" Dog. When the classifiers involved are software engineering classes, generalization usually involves reusing code; it is often implemented using inheritance, where the more specific code reuses the more general code.

3.11 Realization

Realization shows the relationship between an Interface and the class that provides the implementation for the interface.

3.12 Scenario

Narrative describing foreseeable interactions.

3.13 Sequence diagram

Describes the Messages sent between a number of participating Objects in a Scenario.

3.14 State

Object exists at one of the States described in a State machine diagram. A state encompasses all the properties of the object along with the values of each of these properties.

3.15 State diagram

Synonym for State machine diagram.

3.16 State machine diagram

Describes the lifetime behaviour of a single Object in terms of in which State it exists and the Transition between those States.

3.17 Stereotype

Notation allowing the extension of UML symbols. Some are defined within Profiles. Examples of predefined UML stereotypes are Actor, Exception, Powertypeand Utility.

3.18 Use case

A use case can be defined as a sequence of actions, including variations, that the system can execute and that produce an observable result which has some value for an actor that interacts with the system.