Circle–ellipse problem

The **circle–ellipse problem** in software development (sometimes called the **square–rectangle problem**) illustrates several pitfalls which can arise when using subtype polymorphism in object modelling. The issues are most commonly encountered when using object-oriented programming (OOP). By definition, this problem is a violation of the Liskov substitution principle, one of the SOLID principles.

The problem concerns which subtyping or inheritance relationship should exist between classes which represent circles and ellipses (or, similarly, squares and rectangles). More generally, the problem illustrates the difficulties which can occur when a base class contains methods which mutate an object in a manner which may invalidate a (stronger) invariant found in a derived class, causing the Liskov substitution principle to be violated.

The existence of the circle–ellipse problem is sometimes used to criticize object-oriented programming. It may also imply that hierarchical taxonomies are difficult to make universal, implying that situational classification systems may be more practical.

Contents

Description

It is a central tenet of object-oriented analysis and design that subtype polymorphism, which is implemented in most object oriented languages via inheritance, should be used to model object types that are subsets of each other; this is commonly referred to as the **is-a** relationship. In the present example, the set of circles is a subset of the set of ellipses; circles can be defined as ellipses

whose major and minor axes are the same length. Thus, code written in an object oriented language that models shapes will frequently choose to make `class Circle` a subclass of `class Ellipse`, i.e. inheriting from it.

A subclass must provide support for all behaviour supported by the super-class; subclasses must implement any mutator methods defined in a base class. In the present case, the method `Ellipse.stretchX` alters the length of one of its axes in place. If `Circle` inherits from `Ellipse`, it must also have a method `stretchX`, but the result of this method would be to change a circle into something that is no longer a circle. The `Circle` class cannot simultaneously satisfy its own invariant and the behavioural requirements of the `Ellipse.stretchX` method.

A related problem with this inheritance arises when considering the implementation. An ellipse requires more states to be described than a circle, because the former needs attributes to specify the length and rotation of the major and minor axes whereas a circle needs only a radius. It may be possible to avoid this if the language (such as Eiffel) makes constant values of a class, functions without arguments, and data members interchangeable.

Some authors have suggested reversing the relationship between circle and ellipse, on the grounds that an ellipse is a circle with more abilities. Unfortunately, ellipses fail to satisfy many of the invariants of circles; if `Circle` has a method `radius`, `Ellipse` must now provide it, too.

Possible solutions

One may solve the problem by:

- changing the model
- using a different language (or an existing or custom-written extension of some existing language)
- using a different paradigm

Exactly which option is appropriate will depend on who wrote `Circle` and who wrote `Ellipse`. If the same author is designing them both from scratch, then the author will be able to define the interface to handle this situation. If the `Ellipse` object was already written, and cannot be changed, then the options are more limited.

Change the model

Return success or failure value

Allow the objects to return a "success" or "failure" value for each modifier or raise an exception on failure. This is usually done in the case of file I/O, but can also be helpful here. Now, `Ellipse.stretchX` works, and returns "true", while `Circle.stretchX` simply returns "false". This is in general good practice, but may require that the original author of `Ellipse` anticipated such a problem, and defined the mutators as returning a value. Also, it requires the client code to test the return value for support of the stretch function, which in effect is like testing if the referenced object is either a circle or an ellipse. Another way to look at this is that it is like putting in the contract that the contract may or may not be fulfilled depending on the object implementing the interface. Eventually, it is only a clever way to bypass the Liskov constraint by stating up-front that the post condition may or may not be valid.

Alternately, `Circle.stretchX` could throw an exception (but depending on the language, this may also require that the original author of `Ellipse` declare that it may throw an exception).

Return the new value of X

This is a similar solution to the above, but is slightly more powerful. `Ellipse.stretchX` now returns the new value of its X dimension. Now, `Circle.stretchX` can simply return its current radius. All modifications must be done through `Circle.stretch`, which preserves the circle invariant.

Allow for a weaker contract on Ellipse

If the interface contract for `Ellipse` states only that "stretchX modifies the X axis", and does not state "and nothing else will change", then `Circle` could simply force the X and Y dimensions to be the same. `Circle.stretchX` and `Circle.stretchY` both modify both the X and Y size.

```
Circle::stretchX(x) { xSize = ySize = x; }
Circle::stretchY(y) { xSize = ySize = y; }
```

Convert the Circle into an Ellipse

If `Circle.stretchX` is called, then `Circle` changes itself into an `Ellipse`. For example, in Common Lisp, this can be done via the `CHANGE-CLASS` method. This may be dangerous, however, if some other function is expecting it to be a `Circle`. Some languages preclude this type of change, and others impose restrictions on the `Ellipse` class to be an acceptable replacement for `Circle`. For languages that allow implicit conversion like C++, this may only be a partial solution solving the problem on call-by-copy, but not on call-by-reference.

Make all instances constant

One can change the model so that instances of the classes represent constant values (i.e., they are immutable). This is the implementation that is used in purely functional programming.

In this case, methods such as `stretchX` must be changed to yield a new instance, rather than modifying the instance they act on. This means that it is no longer a problem to define `Circle.stretchX`, and the inheritance reflects the mathematical relationship between circles and ellipses.

A disadvantage is that changing the value of an instance then requires an assignment, which is inconvenient and prone to programming errors, e.g.,

```
Orbit(planet[i]) := Orbit(planet[i]).stretchX
```

A second disadvantage is that such an assignment conceptually involves a temporary value, which could reduce performance and be difficult to optimise.

Factor out modifiers

One can define a new class `MutableEllipse`, and put the modifiers from `Ellipse` in it. The `Circle` only inherits queries from `Ellipse`.

This has a disadvantage of introducing an extra class where all that is desired is specify that `Circle` does not inherit modifiers from `Ellipse`.

Impose preconditions on modifiers

One can specify that `Ellipse.stretchX` is only allowed on instances satisfying `Ellipse.stretchable`, and will otherwise throw an exception. This requires anticipation of the problem when Ellipse is defined.

Factor out common functionality into an abstract base class

Create an abstract base class called `EllipseOrCircle` and put methods that work with both `Circle`s and `Ellipse`s in this class. Functions that can deal with either type of object will expect an `EllipseOrCircle`, and functions that use `Ellipse`- or `Circle`-specific requirements will use the descendant classes. However, `Circle` is then no longer an `Ellipse` subclass, leading to the "a `Circle` is not a sort of `Ellipse`" situation described above.

Drop all inheritance relationships

This solves the problem at a stroke. Any common operations desired for both a Circle and Ellipse can be abstracted out to a common interface that each class implements, or into mixins.

Also, one may provide conversion methods like `Circle.asEllipse`, which returns a mutable Ellipse object initialized using the circle's radius. From that point on, it is a separate object and can be mutated separately from the original circle without issue. Methods converting the other way need not commit to one strategy. For instance, there can be both `Ellipse.minimalEnclosingCircle` and `Ellipse.maximalEnclosedCircle`, and any other strategy desired.

Combine class Circle into class Ellipse

Then, wherever a circle was used before, use an ellipse.

A circle can already be represented by an ellipse. There is no reason to have class `Circle` unless it needs some circle-specific methods that can't be applied to an ellipse, or unless the programmer wishes to benefit from conceptual and/or performance advantages of the circle's simpler model.

Inverse inheritance

Majorinc proposed a model that divides methods on modifiers, selectors and general methods. Only selectors can be automatically inherited from superclass, while modifiers should be inherited from subclass to superclass. In general case, the methods must be explicitly inherited. The model can be emulated in languages with multiple inheritance, using abstract classes.[1]

Change the programming language

This problem has straightforward solutions in a sufficiently powerful OO programming system. Essentially, the circle–ellipse problem is one of synchronizing two representations of type: the *de facto* type based on the properties of the object, and the formal type associated with the object by the object system. If these two pieces of information, which are ultimately only bits in the machine, are kept synchronized so that they say the same thing, everything is fine. It is clear that a circle cannot satisfy the invariants required of it while its base ellipse methods allow mutation of parameters. However, the possibility exists that when a circle cannot meet the circle invariants, its type can be updated so that it becomes an ellipse. If a circle which has become a *de facto* ellipse doesn't change type, then its type is a piece of information which is now out of date, reflecting the history of the object (how it was once constructed) and not its present reality (what it has since mutated into).

Many object systems in popular use are based on a design which takes it for granted that an object carries the same type over its entire lifetime, from construction to finalization. This is not a limitation of OOP, but rather of particular implementations only.

The following example uses the Common Lisp Object System (CLOS) in which objects can change class without losing their identity. All variables or other storage locations which hold a reference to an object continue to hold a reference to that same object after it changes class.

The circle and ellipse models are deliberately simplified to avoid distracting details which are not relevant to the circle–ellipse problem. An ellipse has two semi-axes called `h-axis` and `v-axis` in the code. Being an ellipse, a circle inherits these, and also has a `radius` property, which value is equal to that of the axes (which must, of course, be equal).

```
(defclass ellipse ()
  ((h-axis :type real :accessor h-axis :initarg :h-axis)
   (v-axis :type real :accessor v-axis :initarg :v-axis)))

(defclass circle (ellipse)
  ((radius :type real :accessor radius :initarg :radius)))

;;;
;;; A circle has a radius, but also a h-axis and v-axis that
;;; it inherits from an ellipse. These must be kept in sync
;;; with the radius when the object is initialized and
;;; when those values change.
;;;
(defmethod initialize-instance ((c circle) &key radius)
  (setf (radius c) radius)) ;; via the setf method below

(defmethod (setf radius) :after ((new-value real) (c circle))
  (setf (slot-value c 'h-axis) new-value
        (slot-value c 'v-axis) new-value))

;;;
;;; After an assignment is made to the circle's
;;; h-axis or v-axis, a change of type is necessary,
;;; unless the new value is the same as the radius.
;;;
(defmethod (setf h-axis) :after ((new-value real) (c circle))
  (unless (= (radius c) new-value)
    (change-class c 'ellipse)))

(defmethod (setf v-axis) :after ((new-value real) (c circle))
  (unless (= (radius c) new-value)
    (change-class c 'ellipse)))

;;;
;;; Ellipse changes to a circle if accessors
;;; mutate it such that the axes are equal,
;;; or if an attempt is made to construct it that way.
;;;
;;; EQL equality is used, under which 0 /= 0.0.
;;;
;;;
(defmethod initialize-instance :after ((e ellipse) &key h-axis v-axis)
  (if (= h-axis v-axis)
    (change-class e 'circle)))

(defmethod (setf h-axis) :after ((new-value real) (e ellipse))
  (unless (typep e 'circle)
    (if (= (h-axis e) (v-axis e))
      (change-class e 'circle))))

(defmethod (setf v-axis) :after ((new-value real) (e ellipse))
  (unless (typep e 'circle)
    (if (= (h-axis e) (v-axis e))
      (change-class e 'circle))))

;;;
;;; Method for an ellipse becoming a circle. In this metamorphosis,
;;; the object acquires a radius, which must be initialized.
;;; There is a "sanity check" here to signal an error if an attempt
;;; is made to convert an ellipse which axes are unequal
;;; with an explicit change-class call.
```

```lisp
;;; The handling strategy here is to base the radius off the
;;; h-axis and signal an error.
;;; This doesn't prevent the class change; the damage is already done.
;;;
(defmethod update-instance-for-different-class :after ((old-e ellipse)
                                                       (new-c circle) &key)
  (setf (radius new-c) (h-axis old-e))
  (unless (= (h-axis old-e) (v-axis old-e))
    (error "ellipse ~s can't change into a circle because it's not one!"
           old-e)))
```

This code can be demonstrated with an interactive session, using the CLISP implementation of Common Lisp.

```
$ clisp -q -i circle-ellipse.lisp
[1]> (make-instance 'ellipse :v-axis 3 :h-axis 3)
#<CIRCLE #x218AB566>
[2]> (make-instance 'ellipse :v-axis 3 :h-axis 4)
#<ELLIPSE #x218BF56E>
[3]> (defvar obj (make-instance 'ellipse :v-axis 3 :h-axis 4))
OBJ
[4]> (class-of obj)
#<STANDARD-CLASS ELLIPSE>
[5]> (radius obj)

*** - NO-APPLICABLE-METHOD: When calling #<STANDARD-GENERIC-FUNCTION RADIUS>
      with arguments (#<ELLIPSE #x2188C5F6>), no method is applicable.
The following restarts are available:
RETRY           :R1      try calling RADIUS again
RETURN          :R2      specify return values
ABORT           :R3      Abort main loop
Break 1 [6]> :a
[7]> (setf (v-axis obj) 4)
4
[8]> (radius obj)
4
[9]> (class-of obj)
#<STANDARD-CLASS CIRCLE>
[10]> (setf (radius obj) 9)
9
[11]> (v-axis obj)
9
[12]> (h-axis obj)
9
[13]> (setf (h-axis obj) 8)
8
[14]> (class-of obj)
#<STANDARD-CLASS ELLIPSE>
[15]> (radius obj)

*** - NO-APPLICABLE-METHOD: When calling #<STANDARD-GENERIC-FUNCTION RADIUS>
      with arguments (#<ELLIPSE #x2188C5F6>), no method is applicable.
The following restarts are available:
RETRY           :R1      try calling RADIUS again
RETURN          :R2      specify return values
ABORT           :R3      Abort main loop
Break 1 [16]> :a
[17]>
```

Challenge the premise of the problem

While at first glance it may seem obvious that a Circle **is-an** Ellipse, consider the following analogous code.

```java
class Person
{
    void walkNorth(int meters) {...}
    void walkEast(int meters) {...}
}
```

Now, a prisoner is obviously a person. So logically, a sub-class can be created:

```
class Prisoner extends Person
{
    void walkNorth(int meters) {...}
    void walkEast(int meters) {...}
}
```

Also obviously, this leads to trouble, since a prisoner is *not* free to move an arbitrary distance in any direction, yet the contract of the `Person` class states that a Person can.

Thus, the class `Person` could better be named `FreePerson`. If that were the case, then the idea that `class Prisoner extends FreePerson` is clearly wrong.

By analogy, then, a Circle is *not* an Ellipse, because it lacks the same degrees of freedom as an Ellipse.

Applying better naming, then, a Circle could instead be named `OneDiameterFigure` and an ellipse could be named `TwoDiameterFigure`. With such names it is now more obvious that `TwoDiameterFigure` should extend `OneDiameterFigure`, since it adds another property to it; whereas `OneDiameterFigure` has a single diameter property, `TwoDiameterFigure` has two such properties (i.e., a major and a minor axis length).

This strongly suggests that inheritance should never be used when the sub-class restricts the freedom implicit in the base class, but should only be used when the sub-class adds extra detail to the concept represented by the base class as in 'Monkey' is-an 'Animal'.

However, stating that a prisoner can not move an arbitrary distance in any direction and a person can is a wrong premise once again. Any object which is moving to any direction can encounter obstacles. The right way to model this problem would be to have a `WalkAttemptResult walkToDirection(int meters, Direction direction)` contract. Now, when implementing walkToDirection for the subclass Prisoner, you can check the boundaries and return proper walk results.

References

- Robert C. Martin, The Liskov Substitution Principle (https://web.archive.org/web/20150905081111/http://www.objectmentor.com/resources/articles/lsp.pdf), C++ Report, March 1996.

1. Kazimir Majorinc, Ellipse-Circle Dilemma and Inverse Inheritance, ITI 98, Proceedings of the 20th International Conference of Information Technology Interfaces, Pula, 1998 (http://iti.srce.hr/ITI_PAST/98proc.html)

External links

- https://web.archive.org/web/20150409211739/http://www.parashift.com/c++-faq-lite/proper-inheritance.html#faq-21.6 A popular C++ FAQ site by Marshall Cline. States and explains the problem.
- *Constructive Deconstruction of Subtyping* (http://alistair.cockburn.us/index.php/Constructive_deconstruction_of_subtyping) by Alistair Cockburn on his own web-site. Technical/mathematical discussion of typing and subtyping, with applications to this problem.
- Henney, Kevlin (2003-04-15). "From Mechanism to Method: Total Ellipse" (http://www.ddj.com/dept/cpp/184403771). *Dr. Dobb's*.
- http://orafaq.com/usenet/comp.databases.theory/2001/10/01/0001.htm Beginning of a long thread (follow the *Maybe reply:* links) on Oracle FAQ discussing the issue. Refers to writings of

C.J. Date. Some bias towards Smalltalk.

- LiskovSubstitutionPrinciple (http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple) at WikiWikiWeb
- *Subtyping, Subclassing, and Trouble with OOP* (http://okmij.org/ftp/Computation/Subtyping/), an essay discussing a related problem: should sets inherit from bags?
- *Subtyping by Constraints in Object-Oriented Databases* (https://web.archive.org/web/20110719101727/http://www.informatik.uni-augsburg.de/de/lehrstuehle/dbis/db/publications/all_db_publications/1996_koe_kie_isotas96/1996_kow_koe_kie_isotas96.pdf), an essay discussing an extended version of the circle–ellipse problem in the environment of object-oriented databases.
