Cycle Checker

Jens Coldewey
Coldewey Consulting
Uhdestr. 12
D-81477 München
Germany
Tel: +49-89-74995702; Fax: +49-89-74995703
email: jens_coldewey@acm.org; http://www.coldewey.com

CycleChecker

Thumbnail

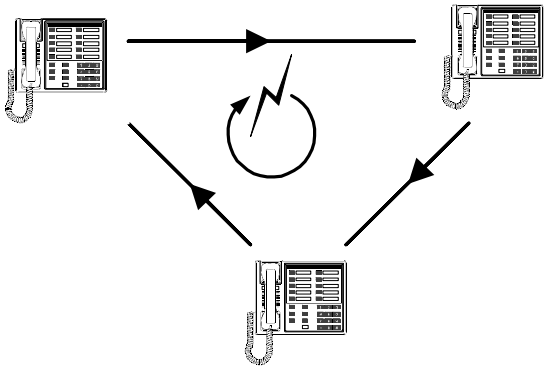
There are algorithms that do not work on object networks if they contain cycles,

therefore, implement a special Cycle Checker object to detect cyclic structures

Example

A local telephone switching system usually provides features to redirect your phone. If you are working in a testing lab, for example, you can redirect your own extension into the testing room so you don't miss any call. Many systems also allow to forward redirected calls too. If the last person leaves the testing room, she may redirect the testing room extension to the receptionist, so she gets all the calls, including yours.

Programming these exchanges, you have to take care that the users don't create redirection loops. If they could, their calls would be forwarded for ever and may even crash the complete system.



Context

Abstractly, this pattern deals with networks of objects that need to be acyclic. The objects form the nodes of the network while the references between the objects form the edges. When manipulating these edges you need to prevent that cycles evolve. Cycles may be cycles direct with only two objects involved or they may be indirect with an arbitrary number of objects involved.

Problem

So how do you check an object network for cycles?

Forces

- *Cycles may be arbitrarily complex.* In fact, trivial cycles are rare, because users tend to avoid cycles themselves. The cycles happening in realistic situation often involve several steps and are hard to recognize at all.

- *The cycle check should not change the state of the nodes.* Because the nodes are usually first class domain objects, changes of their state may trigger further actions, such as user interface updates, or database updates. What sounds like a performance problem at first sight may escalate. While superfluous interface updates "only" annoy the users, superfluous database updates set unneeded locks and thus limit scalability. If the cycle check involves a large share of the objects it may even block the complete system.

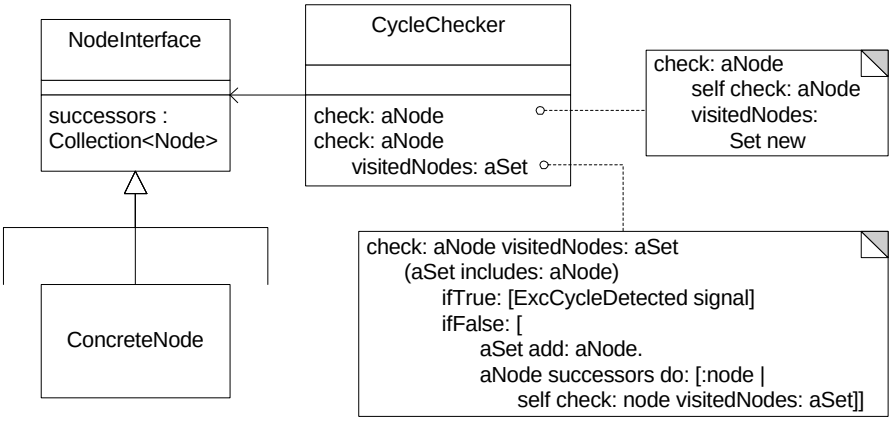
- *Several cycle checks may run concurrently.* When the application has several threads, more than one thread may check for a cycle at a given time. In this scenario, one thread may change the network while another checks for cycles. This may cause checks to fail, because the thread may insert a new circle in areas of the network that are already checked. If the cycle checker changes the checked objects you may also run into problems.

- *The network may contain objects from different inheritance trees,* so it is hard to assign the responsibility to a single class. Obviously all the objects should provide a uniform protocol for network manipulation, but this protocol is not necessarily derived from a common superclass. If a common superclass exists, it may still not be responsible for cycle checking. Consider a hierarchical file system modeled with a Composite structure [GHJV95]. This structure is free of cycles by definition. The only way to introduce a cycle is to define UNIX style links modeled with a special leaf node. The only class that should know about these links inside the class hierarchy is the link class. However, a cycle may contain any number of other files. Hence, the cycle check should not be the responsibility of a node class.

Solution

Therefore, assign the responsibility to check for cycles to a separate object, the `Cycle Checker`. This objects traverses the complete object network reachable from a starting node using the common `Node Interface`. It also maintains a set of visited nodes. When visiting a node, the `Cycle Checker` tests, whether the node already is an element of the set. If it is, the object has detected a cycle, terminates and acts accordingly. If it isn't, the object registers the node in the set and carries on visiting. Because the network is acyclic in this scenario, the checker

eventually has visited all nodes reachable from its starting point and thus ends.



Consequences

- The communication between the `CycleChecker` and the `Node` consists of the single method `successor`. Because most node objects provide a method like this anyhow, the network can be heterogeneous. In typed languages, such as Java or C++, all node classes have to implement at least a common interface to manipulate the network – not a very hard restriction.

- The `CycleChecker` crawls through all reachable nodes and thus finds every cycle, regardless of the number of hops it takes. However, in large networks this may take some time. The performance penalty depends on the network structure and implementation issues, so I am going to discuss this topic in the implementation section.

- The Nodes do not know about the `CycleChecker` at all. Particularly the `CycleChecker` does not call any method of the Node that changes its external state. Therefore it triggers neither unwanted updates nor provokes any other action.

- Concurrently running checks do not interfere one another. Because the checks do not change the nodes, they are fully reentrant. However, you have to take care if the object network changes while the Checker crawls through it. In this scenario two different processes are well able to construct a cycle after running checkers and both checkers won't take any notice. There are two different approaches to prevent this, the pessimistic and the optimistic approach. The pessimistic approach locks all nodes that have been visited against change until the checker finishes. However, locking is expensive and only feasible if you can control modifications of the nodes. The optimistic approach uses the Observer pattern [GHJV95]. The

checker registers at a node when it is visited. When the successors of a node change, the node notifies all registered checkers. The checkers restart all or a part of the check. While the pessimistic approach is guaranteed to end after a certain time, the optimistic approach runs until a check finishes undisturbed. Another drawback of the optimistic approach is its runtime complexity. The Checker registers with all the nodes it visits, introducing a new runtime dependency with all of them. You have to maintain these dependencies through the whole lifetime of the Checker and you have to guarantee that the events do not start to bounce between the different threads, resulting in an indefinite burst of update messages.

Implementation

- *Improving Performance:* The time the spell checker needs to certify absence of circles is proportional to the number of paths through the network from the start node. This is ok for mainly hierarchical structures, such as file systems. However, in the worst case that means that the time grows with the square of the number of nodes – a behavior that significantly limits scalability. A small addition to the presented `CycleChecker` decreases performance complexity significantly: The `CycleChecker` object stores all nodes that have been certified during the current check in an instance variable. Therefore, if a thread leads to an area of the network that just has been tested, the `CycleChecker` can stop and carry on with the next thread. Because with this small addition every node in the network is at most visited once, the time is proportional to the size of the network instead of its square.

- *Stack Consumption:* The `CycleChecker` works itself recursively through all the possible threads in the network. The nesting depth is as deep as the longest thread in the network. If the network is strongly forked, you can estimate pretty safely that this is proportional to some logarithm of the number of nodes. The best example of a network like that is a tree. However, if there are long runs nearly without forks the nesting depth may be in the magnitude of the number of nodes resulting in an enormous greed for stack memory. Luckily, the case with the worst stack behavior also is the case with the least need for the stack. The `CycleChecker` uses the stack to manage the forks in the network. Therefore, you can iterate over unforked nodes using a loop, while you use recursion only if the network forks. With this improvement, the stack behavior is worst, if every fork only has two branches, leading to a maximum depth of $\log_2$ of the number of nodes in the network – even for extremely large networks a number significantly below hundred.

- *Networks stored in a database:* Be careful when you use this pattern with a network stored in a database – or even worse distributed across the world. In this scenario every access of a node needs significant time. A few milliseconds for a database access or even a second for an Internet access. This may take up to minutes or even hours for large networks. Hence, you cannot use the CycleChecker to ensure consistence in advance. Still you may use one of the variants, described below.

Sample Code

The following Smalltalk code shows a complete Cycle Checker combined with an Iterator (see below):

```
recursiveDoWithCycleCheck: aBlock
  ^self recursiveDoWithCycleCheck: aNode
    checkedNodes: Set new
    visitedNodes: Set new.

recursiveDoWithCycleCheck: aBlock
  checkedNodes: checkedNodesSet
  visitedNodes: visitedNodesSet

  | result |
  (checkedNodesSet includes: self)
    ifTrue: [^self].
  (visitedNodesSet includes: self)
    ifTrue: [ExcCycleDetected signalWith: self].
  visitedNodesSet add: self.

  "Evaluate the Block"
  result := aBlock value: self.

  self successors do: [:node |
       node recursiveDoWithCycleCheck: aBlock
       checkedNodes: checkedNodesSet
       visitedNodes: visitedNodesSet copy].

  checkedNodesSet add: aNode.
  ^result
```

Please observe, that the nodes visited in the current path are passed by copy while the checked nodes are passed by reference. This ensures that reunions of two different threads are not interpreted as cycles, because the visited nodes are local to the current thread.

Variants

Iterating Cycle Checker: If you cannot prevent cycles for performance reasons, you may at least want to protect actions on the network from crashing uncontrolled if they run into a cycle. Rather you want to fail them in a defined manner, for example by throwing an exception.

The easiest way to do that is to combine the CycleChecker with an Iterator pattern [GHJV95]. While the Iterator crawls through the network to perform its task, it checks whether it has run into a cycle. The

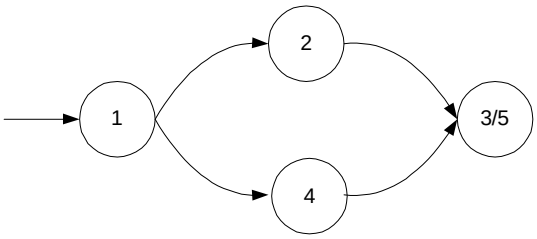
following sample code shows the implementation of an internal Iterator[1] using an Enumeration Method [Bec97, pp. 99]:

```
Collection>>doWithCycleCheck: aBlock
  | visitedNodes |
  visitedNodes := Set new.
  ^self do: [ :aNode |
    (visitedNodes includes: aNode) ifTrue: [
      ExcCycleDetected signalWith: aNode].
    visitedNodes add: aNode.
    aBlock value: aNode
    "In Java or C++ you would call a processItem
     method of the node instead of the value:, see
     [GHJV95, pp 267]"
  ].
```

The method augments the standard Smalltalk Iterator `do:` defined in the `Collection` class in a class extension. Therefore the new feature applies to all types of collections.

The main benefit of this variant is that you only check those nodes you actually need to accomplish your task. For example if Iterator only iterates one thread, it is sufficient to check this thread for cycles. This may spare a lot of node accesses.

Please note that this solution defines no Cycle Checker class but stores the information in a local variable of the method. This implies that the method may be overzealous. It detects every double visit of an object as cycle. This is correct if the iteration follows only one path through the network – an assumption that is true most of the time. However, if you traverse several paths you may reach the same node several times without getting into a cycle. The figure shows an example. To manage situations like this the Iterator has to have a notion of the path and adjust the `visitedNodes` set if it switches to a new path.



Hop Counter: There are two preconditions for the pattern to work. It must be possible to identify the nodes uniquely and the CycleChecker must be able to transport the information from one node to the next. Both preconditions are no limitation inside an object-oriented system. However, when it comes to distributed networks, the preconditions may

[1] [GHJV95] defines an internal Iterator as an Iterator that controls the iteration in contrast to an external Iterator that leaves the control to the client, that "must advance the traversal and request the next element explicitly from the iterator" (p. 260). Internal Iterators are more suitable to traverse complex object structures, "because they define the iteration logic for you" (p. 261)

break. In this case you can use a minimized variant of the pattern. Instead of storing a reference to every node, the CycleChecker just counts the number of nodes it visits. If this counter exceeds a reasonable number, the CycleChecker assumes that it has run into a circle. While this technique detects a circle reliably, it may signal a circle, even if no circle is present.

Example Resolved

In the initial example of the telephone exchange you may check for cycles at two different times. At first you may apply the pattern when the user enters a redirection. If this leads to a cycle the command fails.

You may also accept every redirection and apply the Iterating variant of the pattern: When a calls arrives you memorize every phone you redirect the call to in a set. When the redirection graph meets a phone already in the list you take some fallback action. For example you may cancel the redirection completely or forward the call to the receptionist.

Known Uses

The `CycleChecker` is a well-known algorithm in Computer Science and is described in many books about algorithms. For example Aho, Hopcroft, and Ullman present it as “Test for Acyclicity” in [AHU83, p. 221].

Unidraw, a graphical editor framework, uses this pattern while finding a data flow through a graph to prevent running into a cycle [Vli90, pp. 86]

Most Internet email servers use the Hop Counter variant to detect forwarding loops.

Knuth describes an algorithm for topological sort that outputs elements found to be acyclic [Knu97, pp. 265]. When all output is done he checks whether there are still nodes left and thus detects a cycle.

Acknowledgements

Neil Harrison and Bob Hanmer provided known uses, Christa Schwanninger helped me to find the Aho, Hopcroft Ullman reference, and John Vlissides provided is Ph.D. thesis as known use. He also shepherded this paper for EuroPLoP and helped to improve the paper significantly. Alwine Brem of Generali in Munich suggested the Example Resolved section. Andreas Rüping suggested several structural improvements and Kevlin Henney pointed me to Kent Beck’s work. Finally, the participants of the demo writer’s workshop of EuroPLoP’99 gave several important hints. Thanks to you all.

References

[ABW98] Sherman Alpert, Kyle Brown, Bobby Woolf: *The Design Patterns Smalltalk Companion;* Addison-Wesley, Reading, Massachusetts, 1998; ISBN 0-201-18462-1

[AHU83] Alfred Aho, John Hopcroft, Jeffrey Ullman: *Data Structures and Algorithms;* Addison-Wesley, Reading, Massachusetts, 1983; ISBN 0-201-00023-7

[Bec97] Kent Beck: *Smalltalk Best Practice Patterns;* Prentice Hall, Eaglewood Cliffs, New Jersey, 1997; ISBN 0-13-476904-X

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software;* Addison-Wesley, Reading, Massachusetts, 1995; ISBN 0-201-63361-2

[Knu97] Donald E. Knuth: *The Art of Computer Programming - Volume 1: Fundamental Algorithms - Third Edition;* Addison-Wesley, Reading, Massachusetts, 1997; 0-201-89683-4

[Vli90] John Vlissides: *Generalized Graphical Object Editing;* Ph.D. thesis, Stanford University, June 1990. (Available as Stanford University Computer Systems Laboratory Technical Report CSL-TR-90-427.)