Orphaned Linked List

An Object-Oriented approach to collapsing multiple Progress Bars

Danny Ruijters

danny.ruijters@philips.com

Abstract. Everybody knows the frustration caused by a new progress bar popping up, and starting from 0%, just after the previous one reached 100%. This annoyance is caused by several consecutive subroutines providing independent progress feedback. Collapsing these isolated instances of progress bars to one bar running from 0 to 100% can be a nasty problem. In this article an object-oriented approach is presented, called "orphaned linked list", providing an elegant and transparent solution to this problem.

1 Introduction

In complex software packages, there are often multiple lengthy tasks that provide progress feedback. A single user action could lead to a number of such tasks to be executed, typically resulting in several consecutive instances of progress, running from 0 to 100%. It is desirable to collapse the isolated instances of progress feedback to one overall instance of feedback running from 0 to 100%, in order to give the user a good notion how much of the overall task has been finished, and how much is left.

It is possible to solve this problem by hard-coding the progress range into every function, or by passing the current progress as parameters to each function. However the first solution is not very flexible (consider calling the same subroutines in a different order, or different context), and the latter poses constrains to your code. Therefore this article presents a transparent and flexible solution to the above described problem.

In this solution linked lists are used to connect the sequence of functions providing progress feedback. Linked lists, also known as chained lists, are one of the fundamental structures known to computer science for many years. However we do not want to burden the functions that give progress feedback with the control and overhead of subscribing to a linked list (after all then such a function would have to have some knowledge about its context). Therefore a variation of linked lists is introduced, dubbed "orphaned linked list".

In section 2, a brief introduction to "traditional" linked lists is given. A reader already familiar with linked lists might skip that part. All example code in this paper is presented in C++ [4], but can be easily extended to any object-oriented language. In section 3, the particularities of the orphaned linked list are introduced. Section 4 explains how an orphaned linked list can be used to collapse nested progress bars, and in section 5 we present our conclusions.

2 Linked list

Since linked lists are widely known and exhaustively described in literature, this section will be limited to a very short summary of their most important properties. An elaborate discussion of linked lists can be found in e.g. [2]. A linked list consists of a sequence of items, called nodes. Every node possesses data and a link to its successor, called the `next` pointer. For the last node the `next` pointer typically refers to `NULL`. Such a list can be regarded as a chain of nodes. Apart from the nodes there is a pointer, the `head` pointer, that refers to the first node. The linked list described above is the simplest form of a linked list, known as *singly* linked list. There are many optional variations possible, of which the most important are:

doubly linked lists Besides the `next` pointer the nodes also posses a `previous` pointer, pointing its the predecessor.
tail pointer Next to the `head` pointer, there is a `tail` pointer, referring to the last node.
cyclic lists The tail node points to head node (and vice versa in case of a doubly linked list).

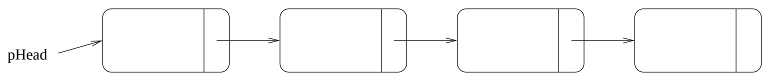

Fig. 1. A singly linked list consisting of four nodes.

Linked lists are similar to one of the most commonly used structures in programming: arrays. They both store a sequence of elements and both approaches are independent of the specific type of elements. In arrays, in contrary to linked lists, the chain of elements are sequentially stored in one continuous chunk of memory. An in-depth comparison and discussion of the pros and cons of both structures goes beyond the objectives of this paper, but summarizing it could be stated that linked lists show better dynamical behaviour (adding, removing and reordering elements) and arrays can be accessed faster (see [2]).

A straight forward object-oriented implementation of linked lists would know two types of classes. The `node` class, encapsulating the element data, including a `next` and optionally a `previous` pointer. And the `list` class, holding the `head` and `tail` pointer and implementing the list control. This class would typically offer functions for adding and removing elements, etc.

An example of an object-oriented approach can be found in the *Standard Template Library* classes `std::slist` and `std::list` [3]. In these implementations the `node` class is embedded in the `std::slist` and `std::list` class definition.

3 Orphaned linked list

An orphaned linked list knows only one type of class. This class encapsulates the element data and a `next` and a `previous` pointer (it is also possible to use a singly connected list, which consumes less memory, instead of a doubly connected list). But additionally it has a static data member: the `head` pointer. Since static data members in a class are shared between all instances of that class [4], there is only one `head` pointer for all nodes. By default the `head` pointer will point to `NULL`. When an instance of the class is created, first the constructor will be called. In the constructor the particular instance will add itself to the linked list. The insert algorithm, deciding where a particular will be inserted, can be chosen randomly. The simplest strategy is to insert new nodes at the beginning of the list, and that is what we will do in our examples. In that case the code for the constructor could look as follows:

```
COrphanedNode::COrphanedNode() {
    //Insert this node in the list
    this->pNext = pHead;
    this->pPrevious = NULL;
    if (pHead) pHead->pPrevious = this;
    pHead = this;
}
```

The destructor should unsubscribe to the list. The code might look like this:

```
COrphanedNode::~COrphanedNode() {
    //Remove this node from the list
    if (this->pPrevious) this->pPrevious->pNext = this->pNext;
    if (this->pNext) this->pNext->pPrevious = this->pPrevious;
    if (pHead == this) pHead = this->pNext;
}
```

It should be noted that this code only works correctly if all nodes are valid members of the list at all times. If there are nodes with `next` or `previous` pointers pointing to list members, but which are not member of that particular list themselves, the code would break. Therefore the `next` or `previous` pointers should be private members of the class, to prevent outside code from altering them.

An orphaned linked list node class might have many public accessible functions for various purposes, e.g. sorting the nodes or iterating over the nodes, etc.

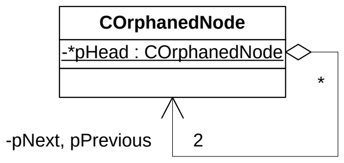

Fig. 2. UML diagram of an orphaned linked list node.

What makes a orphaned linked list special? Why should you consider using it? Its most important feature is the fact that the code that instantiates a node, does not need to know where to find the parent of the linked list. As a matter of fact, from the outside it might seem as if a simple instance of a particular class is created, without any relationships with other entities. The actual context is hidden, which might simplify code, and programming, as the following section will demonstrate.

4 Collapsing multiple progress bars

Remark: where "progress bars" is written any form of progress feedback can be used, e.g. textual feedback in percentages.

Consider a simple class for progress feedback, `CProgress`, which could be used in the following way:

```
void A() {
    CProgress myProgress;
    myProgress.Set(0);
    B(); //call function B
    myProgress.Set(60);
    C(); //call function C
    myProgress.Set(100);
}
```

It is very simple to imagine how the code of `CProgress` could look like. The `Set()` function simply sets the progress bar to the number that is passed. Now suppose that function `B()` and `C()` have their own progress bars and suppose that we want to capture that progress also in the progress bar of function `A()`. In other words, we want to collapse the progress feedback of function `B()` and `C()` in the progress bar of function `A()`.

Now it would be possible to pass the current progress as parameters to the functions `B()` and `C()`, or a pointer to the current progress bar, but that would make our functions very context aware, and is not a very flexible solution. To solve this dilemma we extent the `CProgress` class to be an orphaned linked list node. Further, one small adjustment to our code concerns replacing the `Set()` member function with the `SetRange()` function. Rather than telling how much percent of the progress already has been finished, this function indicates the fraction of the total time the next block of code is expected to demand, as is demonstrated in the following piece of code:

```
void A() {
    CProgress myProgress;
    myProgress.SetRange(0.60);
    B(); //call function B
    myProgress.SetRange(0.40);
    C(); //call function C
}
```

In essence the `SetRange()` function still sets the displayed progress bar to the progress that has been finished at that point. However, since a `CProgress` instance is part of a list, it is aware of its context. It can lookup which progress bars have preceded it, and what the range is it should be mapped to. For instance, a progress bar in function `C()` can look up that it should be mapped from 60 to 100%. Figure 3 shows how a progress bar used in a certain function could be collapsed in another one, which in its turn is part of yet another one. The beauty of this approach lies in the fact that a function using a progress bar does not need to know whether its caller already has a progress bar, or whether its callees posses progress feedback. It does not need to be aware of its context, and simply provides its own progress feedback, running from 0 to 100%.

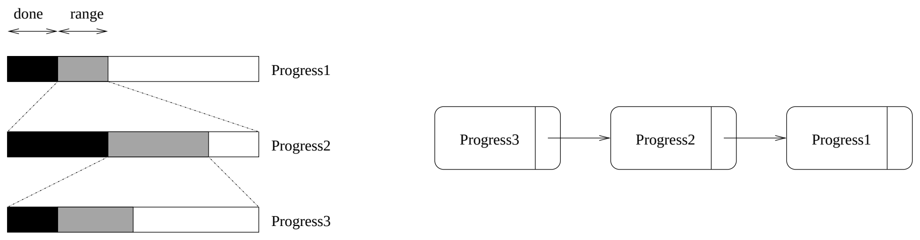

Fig. 3. Three progress bars are collapsed into each other. In black the progress that is already finished, in grey the progress range for the current function. At the right the linked list is shown.

Following the orphaned linked list approach, `CProgress` class registers itself to the anonymous linked list in the constructor. The progress status of an embedded progress bar at any given moment can be expressed by the following recursive formulas:

$$\begin{aligned} R_n &= r_n \cdot R_{n-1} && n > 0 \\ P_n &= P_{n-1} + p_n \cdot R_{n-1} && n > 0 \end{aligned} \tag{1}$$

Whereby p_n indicates the local progress, that has been finished, within the context of subroutine n. The overall finished progress, of the complete tree of routines is indicated by capital P. The progress range of the next block of code is denoted by r. The overall range R is the same range, but relative to all the preceding stack of progress feedback. Obviously, for the top `CProgress` node R_0 equals r_0 and P_0 equals p_0.

The hierarchy of progress bars, collapsed in earlier ones, can be as deep and complex as desirable. For instance three progress bars could be mapped to a range within their predecessors, the deepest one could finish, another function with a progress bar is created and maps to the next range, in its turn it maps yet another progress bar to a range within its range, etc. etc.

Even though strictly speaking orphaned linked lists can be regarded as a structural design pattern [1, 5, 6] (after all a linked list is clearly a data structure), its use can, and will typically be, highly dynamic. As is illustrated for the example of collapsing progress bars in figure 4, nodes are added and removed dynamically during runtime. Only one branch of the tree in figure 4 can exist at any given time, but following all branches that ever exist while executing a certain set of functions could result in such a tree.

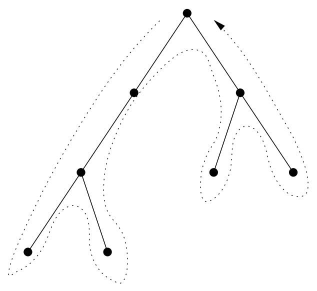

Fig. 4. Example of a tree representing the linked lists that could exist while a set of functions are executed. Nodes are added when new functions are called and removed once a function has finished.

5 Conclusions

In this article an object-oriented approach has been introduced, dubbed "orphaned linked list", to solve the problem of collapsing multiple instances of progress feedback.

The presented approach is transparent in the sense that a particular routine does not need to know by which routine it was called (its caller), and whether the subroutines it calls (its callees) present progress feedback. In other words it does not need to be aware of its context. The approach is flexible in the sense that the hierarchy of functions may be as deep and complex as necessary, and may change every time a set of functions is called.

Orphaned linked lists can be used anywhere where traditional linked lists are used. But their real strength lies in the fact that they can decouple the instantiater of a node and the context of the linked list.

References

1. Gamma (Erich), Helm (Richard), Johson (Ralph) and Vlissides (John). – *Design Patterns Elements of reusable object-oriented software.* – Addison-Wesley Publishing Company, 1995.
2. Parlante (Nick). – Linked list basics. – http://cslibrary.stanford.edu/103/.
3. Plauger (P.J.), Stepanov (Alexander A.), Lee (Meng) and Musser (David R.). – *The C++ Standard Template Library.* – Prentice Hall PTR, 2000, 1st edition.
4. Soustrup (Bjarne). – *The C++ Programming Language.* – Addison Wesley Longman, 1997, 3rd edition.
5. Shalloway (Alan) and Trott (James R.). – *Design Patterns Explained: A New Perspective on Object-Oriented Design.* – Addison-Wesley Publishing Company, 2001.
6. Tichy (Walter F.). – A catalogue of general-purpose design patterns. *In: Proc. Technology of Object-Oriented Languages and Systems (TOOLS 23), IEEE Computer Society.*