

Object Life Cycle for the File System Application

Now that we have had a look at the issues around object life cycle, let us return to our [file system application](#) and add life cycle operations to it, so clients can create and destroy files and directories.

To destroy a file or directory, the obvious choice is to add a `destroy` operation to the `Node` interface:

Slice

```
module Filesystem {

    exception GenericError {
        string reason;
    };
    exception PermissionDenied extends GenericError {};
    exception NameInUse extends GenericError {};
    exception NoSuchName extends GenericError {};

    interface Node {
        idempotent string name();
        void destroy() throws PermissionDenied;
    };

    // ...
};
```

Note that `destroy` can throw a `PermissionDenied` exception. This is necessary because we must prevent attempts to destroy the root directory.

The `File` interface is unchanged:

Slice

```
module Filesystem {
    // ...

    sequence<string> Lines;

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    // ...
};
```

Note that, because `File` derives from `Node`, it inherits the `destroy` operation we defined for `Node`.

The `Directory` interface now looks somewhat different from the previous version:

- The `list` operation returns a sequence of structures instead of a list of proxies: for each entry in a directory, the `NodeDesc` structure provides the name, type, and proxy of the corresponding file or directory.
- Directories provide a `find` operation that returns the description of the nominated node. If the nominated node does not exist, the operation throws a `NoSuchName` exception.
- The `createFile` and `createDirectory` operations create a file and directory, respectively. If a file or directory already exists, the operations throw a `NameInUse` exception.

Here are the corresponding definitions:

Slice

```

module Filesystem {
    // ...

    enum NodeType { DirType, FileType };

    struct NodeDesc {
        string name;
        NodeType type;
        Node* proxy;
    };

    sequence<NodeDesc> NodeDescSeq;

    interface Directory extends Node {
        idempotent NodeDescSeq list();
        idempotent NodeDesc find(string name) throws NoSuchName;
        File* createFile(string name) throws NameInUse;
        Directory* createDirectory(string name) throws NameInUse;
    };
};

```

Note that this design is somewhat different from the factory we designed for the [phone book application](#). In particular, we do not have a single object factory; instead, we have as many factories as there are directories, that is, each directory creates files and directories only in that directory.

The motivation for this design is twofold:

- Because all files and directories that can be created are immediate descendants of their parent directory, we avoid the complexities of parsing path names for a separator such as "/". This keeps our example code to manageable size. (A real-world implementation of a distributed file system would, of course, be able to deal with path names.)
- Having more than one object factory presents interesting implementation issues that we will explore in the following discussion.

Let's move on to the implementation of this design in C++ and Java. You can find the full code of the implementation (including languages other than C++ and Java) in the `demo/book/lifecycle` directory of your Ice distribution.

Topics

- [Implementing Object Life Cycle in C++](#)
- [Implementing Object Life Cycle in Java](#)

See Also

- [Slice for a Simple File System](#)
- [Object Creation](#)