Université de Liège

INFO0027-1 Programming Techniques

Project 2
File Explorer

Guilherme Mendel de Almeida Nascimento - Jordi Hoorelbeke

Academic year 2020-2021

Contents

1 Introduction

This report is meant for explaining the strategy and design for the File Explorer project for the INFO0027-1 Programming Techniques course. It will go over the code organization and all the implemented design patterns, they're reasoning, advantages and disadvantages.

2 Program Design

2.1 Components and Nodes

Files and folders are subclasses to the main class *Component*, which implements a *ComponentInterface*. Components are responsible for storing their own individual state, and are not concerned with where they are.

The way this data is stored is in a tree structure of objects of the class *Node*. Regular nodes are always leaf nodes, but nodes of the subclass *FolderNode* may have children.

Nodes are only responsible for keeping their position in the tree. Their content is stored in an object reference that implements the *ComponentInterface*. Through this interface, nodes are able to provide a display name. Nodes actually also implement this same interface, and so act as a Decorator to it.

2.2 Aliases

Aliases are actually a subclass of *Node*. The idea is that the reference to a *ComponentInterface* of an Alias is actually a reference to another Node, and Aliases delegate all their responsibilities to this reference.

2.3 Component and Node Creation

The class *NodeFactory* is an abstract class that mocks up a method that takes a *FolderNode* and creates another node from it's reference. For each kind of component, there exists a factory subclass.

The file and folder subclasses encapsulate dealing with user input to construct their respective components and wrap them in a new node, child to the provided *FolderNode*.

There also exists an Alias factory class, which is not a subclass to the NodeFactory class, as it requires a reference to a regular node. Before constructing the new alias, it makes sure the provided node isn't another alias and that it wraps a file component.

All factories, after constructing their nodes, raise a node insertion event with the singleton class *NodeInsertionChannel*. This class is responsible for inserting the new node in the UI's JTree structure. It is necessary to insert every node in both trees since the program unfortunately does not have reading access to the former one's data.

2.4 Tying the Implementation to the Provided Interface

The abstract class *NodeHandler* is responsible for performing all the implemented processes when activated. It provides a template method *handle*, which tests all preconditions to the operation, executes it, and then refreshes the UI's JTree.

The *NodeMaker* subclass, for example, checks that the selected node is a *FolderNode* and receives a reference to a *NodeFactory*, to be able to create the requested node without having to know whether it's a folder or file.

At last, in the *GuiHandler* class, all that's left to do is tie all user events with the respective *NodeHandler* subclasses and handle any uncaught exceptions.

2.5 Text Area Management

The text area is handled by a *TextArea* class, which observer any double click events, checks if it was a file, and displays it's content if so.

3 Design Patterns

3.1 Factory

Factory was a no brainer for this project, since files and folders shared a very similar construction process. It came very in handy in conjunction with the *NodeMaker* class, because it was able to completely abstract out whether it's a file or folder being constructed. This was it's biggest advantage, and it definitely favor the *Open-Closed principle*, as it makes it very easy to add another type of component in the future.

It does increase the code complexity, though, with the addition of three new classes just for the sake of creating new nodes. Four if you count the alias factory.

3.2 Template

This design pattern worked best when it was first introduced, when there were more steps to each operation, but it still holds up to the current state of the program. It achieves a much more *DRY*ed out code, with each new operation consisting of only a minimal subclass.

Once again though, it results in a much more complex code. It's also unable to achieve it's full potential, since the first step (checking preconditions) is a very limited one, since a lot of the conditions cannot be checked before-hand.

3.3 Prototype

Copying nodes would have been a painful chore if not for this pattern. It enabled so much reusability that the *NodeReplicator* class doesn't need to care what kind of complicated structure lies underneath it's subject, all it needs to do is just call it's copy method.

This pattern might have setbacks in other projects, but in this one, it's subjects are simple enough that it stays itself also very simple.

3.4 Decorator

Wrapping components in node objects enabled responsibilities to be better distributed between the two classes. And it really shined the brightest when it turned the alias implementation in basically a class that adds (alias) to the GetName() method's return.

It doesn't fit as well as the other patterns, though. The amount of complexity it saved from the alias implementation was vastly undermined by all the complexity brought by having two separate hierarchies of nodes and components.

3.5 Observer

The user double click event was tied with the file display functionality at first. It just so happened that this tie wasn't intuitive, and shouldn't be embedded in the *GuiHandler* class at all. The observer pattern helped isolate the file display behavior from the double click event, making it much easier to add new behavior to the event in the future, if needed.

It also adds to the code complexity, though not as much as many of the other projects. And if in the future more observers were introduced, there would be no way to control the order in which they would be notified.

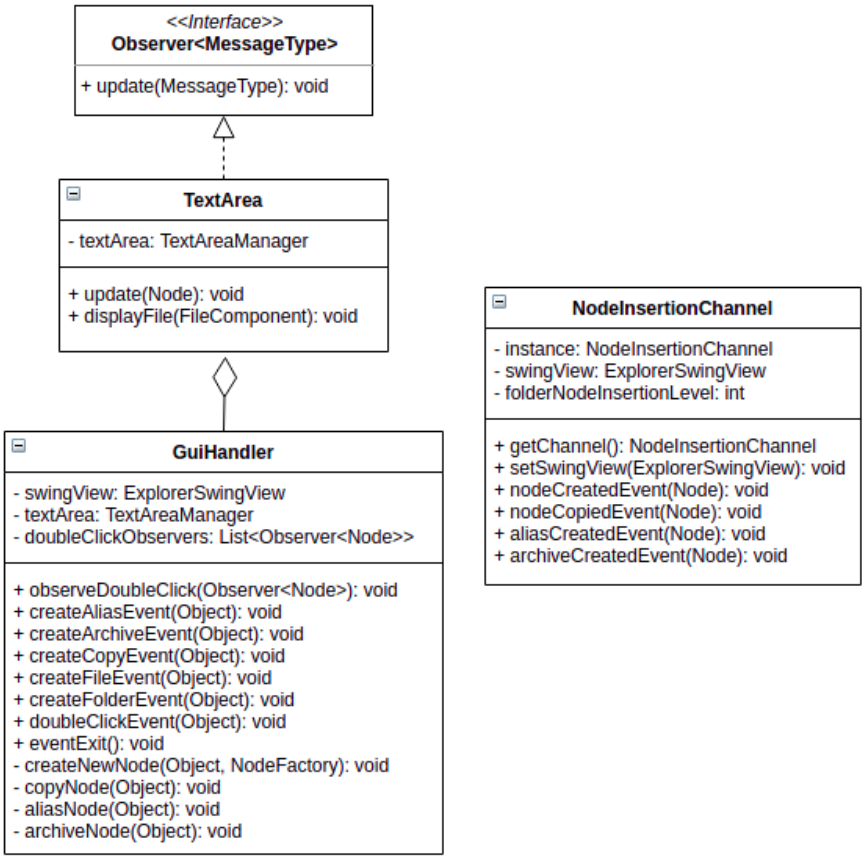
3.6 Singleton

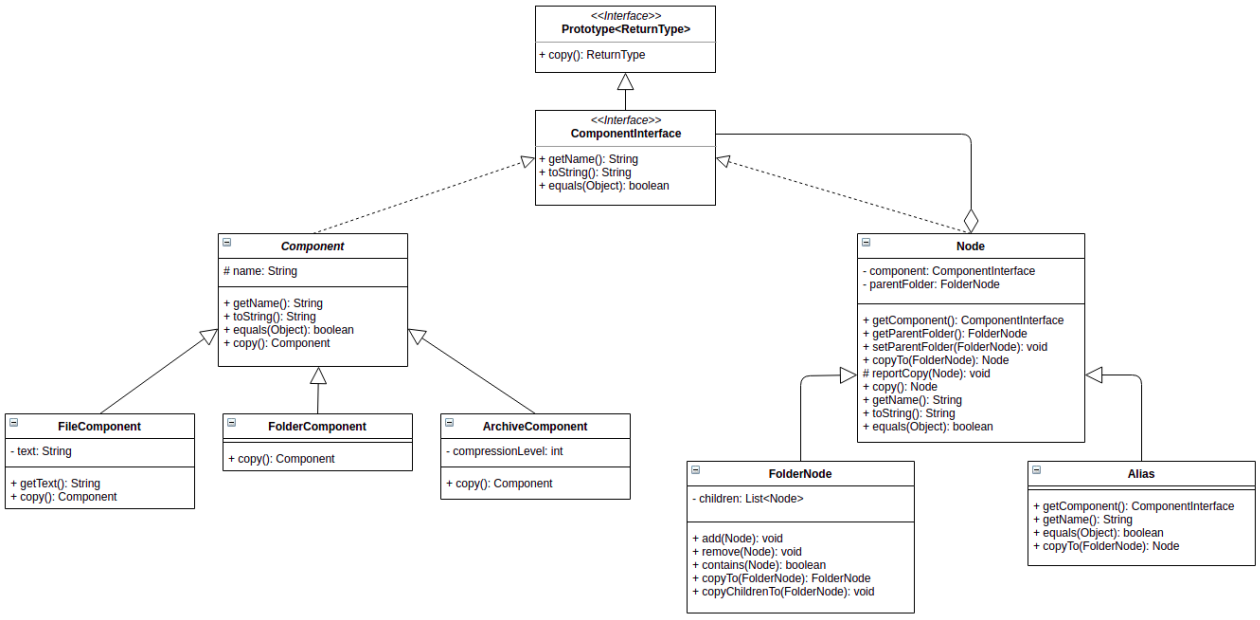
Bringing all the UI's JTree interaction to a single and public object instance was a big advantage, as it vastly reduces the chance for inconsistency and error. Best of all, this object is seen by other objects only as a channel who should be notified of any node insertions, and thus keeps all the JTree complexity to himself.

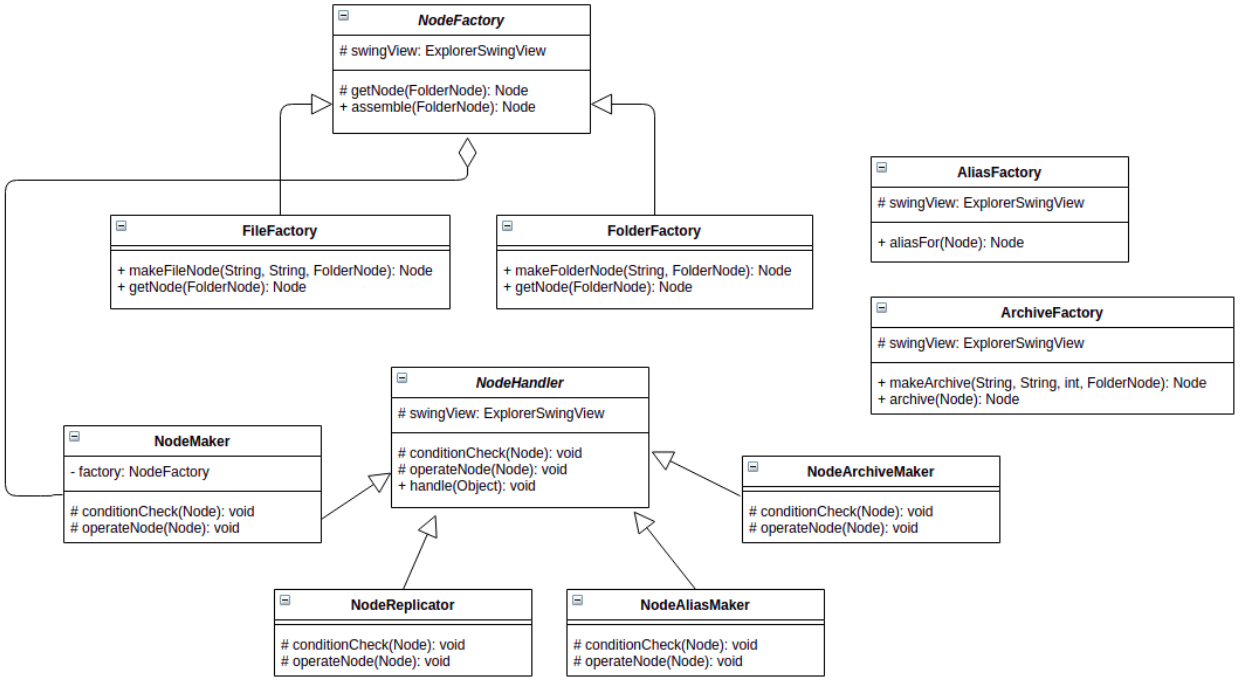
It sure violates the *Single Responsability principle* though, and would make unit testing a nightmare.

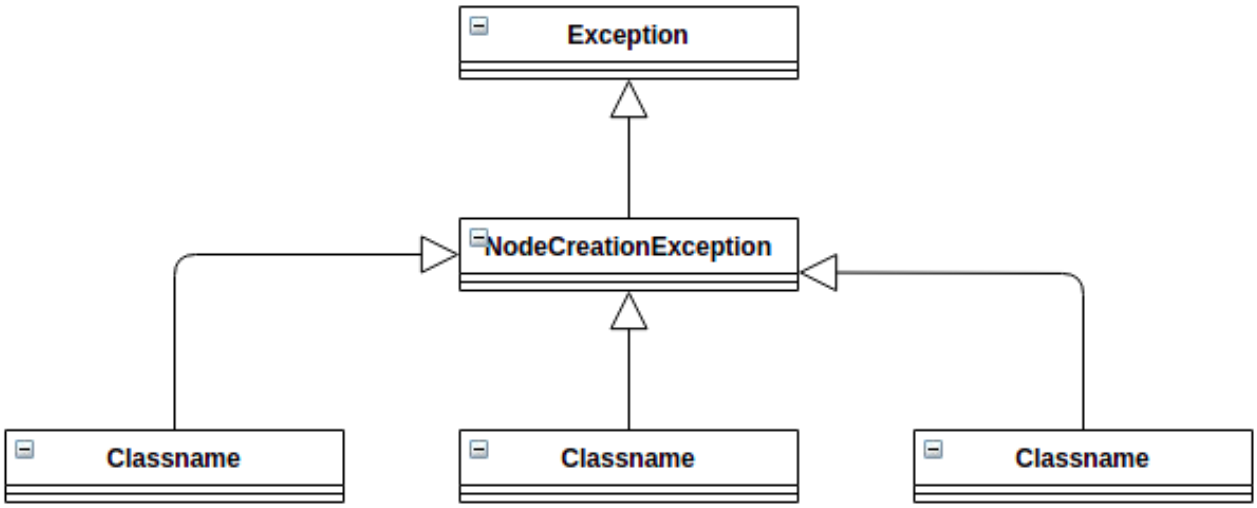
4 Static diagrams

Interaction between the different classes represented with static diagrams:






NodeFactory
swingView: ExplorerSwingView
getNode(FolderNode): Node
+ assemble(FolderNode): Node
FileFactory
+ makeFileNode(String, String, FolderNode): Node
+ getNode(FolderNode): Node
FolderFactory
+ makeFolderNode(String, FolderNode): Node
+ getNode(FolderNode): Node
AliasFactory
swingView: ExplorerSwingView
+ aliasFor(Node): Node
ArchiveFactory
swingView: ExplorerSwingView
+ makeArchive(String, String, int, FolderNode): Node
+ archive(Node): Node
NodeHandler
swingView: ExplorerSwingView
conditionCheck(Node): void
operateNode(Node): void
+ handle(Object): void
NodeMaker
- factory: NodeFactory
conditionCheck(Node): void
operateNode(Node): void
NodeArchiveMaker
conditionCheck(Node): void
operateNode(Node): void
NodeReplicator
conditionCheck(Node): void
operateNode(Node): void
NodeAliasMaker
conditionCheck(Node): void
operateNode(Node): void


Exception
NodeCreationException
Classname
Classname
Classname