Logic core of a Xanalogical Structure application through Object Oriented Programming

Francesco Cannarozzo

July 19, 2021

Contents

1 Definitions

1.1 Xanalogical structure

Xanalogical Structures take their name from Ted Nelson's own *Project Xanadu*®. from its remote beginnings, *Project Xanadu*® has been defined as an alternative hypertextual paradigm, with the purpose of providing

> "*a deep-reach electronic literary system for worldwide use and a differently-organized general system of data management. [...] the point has been to represent the world of ideas correctly and clearly, which is much*

*harder– replacing not just paper media, but conventional computer files and hierarchy, with finer-grained and wholly different families of structure. "**

a **Xanalogical Structure** is thereby described as a generalization of *Project Xanadu*®, in particular as

"*a unique symmetrical connective system for text (and other separable media elements), with **two complementary forms of connection** that achieve these functions– survivable deep linkage (**content links**) and recognizable, visible re-use (**transclusion**).*"

1.2 Entities

Within this rather essential definition, we may easily pinpoint **content links** as entities, which ought to be *permanent*. Similarly, we can be certain **transclusion** will be far more likely to correspond to an operation.
content links, however, shall not be *fundamental* entities, as their obvious function is that of linking **content**. A *unique symmetrical connective system* will necessitate of **structures** to organize and operate on **content** .

Content The word 'content' is a generic one. The same seems plausible for the corresponding entity. As stated, the system needs to be capable of handling, generically, *separable media elements*.
Thus, the **content** entity needs to be of the utmost flexibility.
As it is desirable to maintain both *persistence* and *mutability* at the same time, the **content** entity should have a way to refer, perhaps indirectly, to a **versioning system**. There should be a way for pieces of **content** to be bound to one another with bidirectional **links**.
Of course, it should be possible to perform any sort of operation on the actual content by going through the **content** entity.

Content Links According to Nelson,

"*The xanalogical content link is not embedded. It is ***applicative*** – applying from outside to content which is already in place with stable addresses. Xanalogical links are effectively overlays superimposed on contents. Any number of links, comments, etc., created by anyone anywhere, may be applied to a given body of content. By this method it is possible to have thousands of overlapping links on the same body of content, created without coordination by many users around the world. [...] profuse, overlapping, bidirectional, or diversely created [...] "*

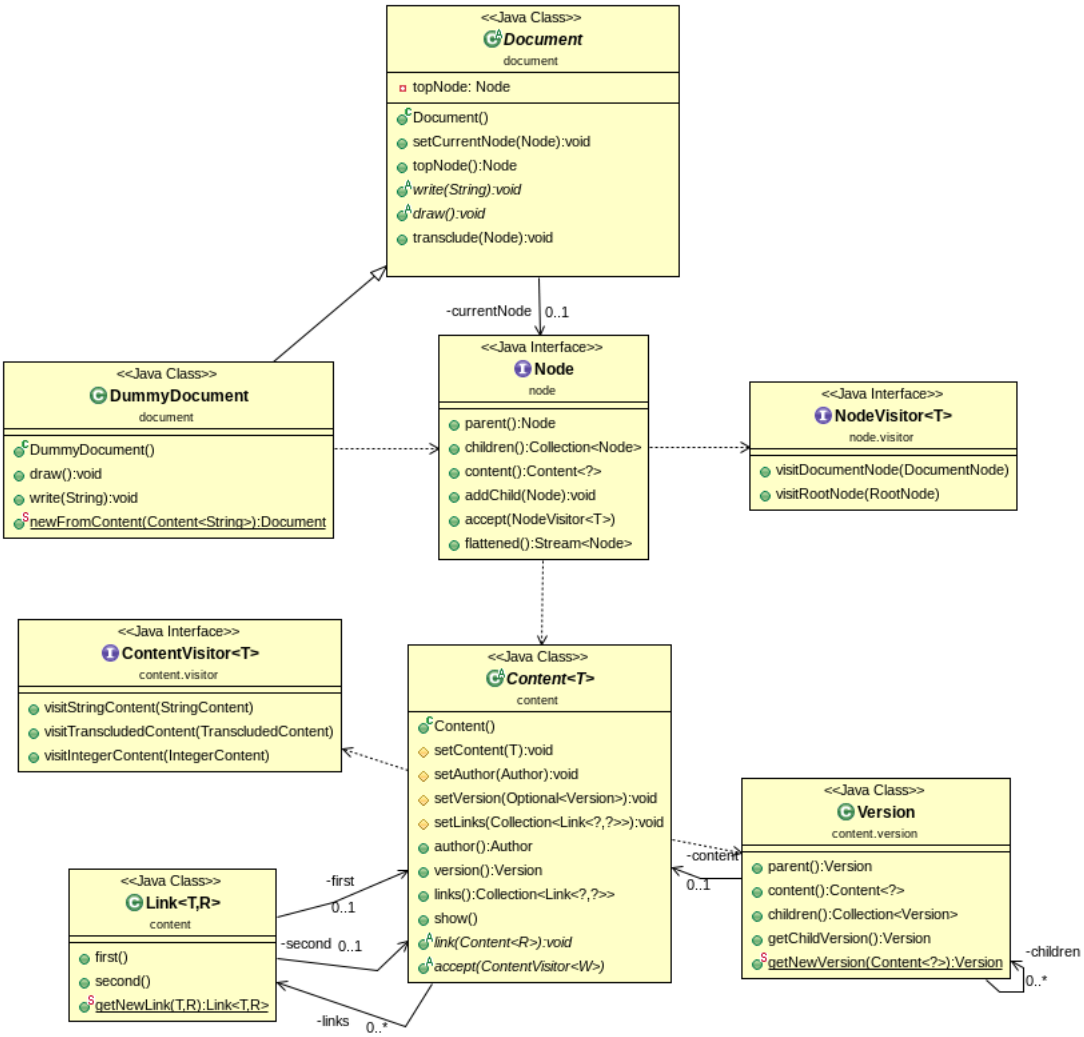
*Xanalogical Structure, Needed Now More than Ever: Parallel Documents, Deep Links to Content, Deep Versioning, and Deep Re-Use, Theodor Nelson ACM Computing Surveys 31(4), December 1999 `http://cs.brown.edu/memex/ACM_HypertextTestbed/papers/60.html`

content links should thus exist as an entity outside of **content**. Anyone should be able to place **content links** between any two pieces of content they have access to. The purpose of content links is established as merely that of representing a semantic relationship between pieces of content*: as such, **content links** should be represented by *small* entity.

Content Structures The characteristics of each **structure** we might conjure up to organize our **content** eventually come down to each structure's fundamental entities, and how they relate with each other and content: We shall thereby refer to such structures as made up of **Nodes**, which can organize and operate upon **content**.
Different modes of organization demand structures behaving differently; However, such structures will have to collaborate to form the *unique symmetrical connective system* Nelson speaks of.
Two modes of content organization are deemed particularly important:
that in **space**, by which pieces of content are organized towards being presented to the user, within such views as that of a **document**;
and that in **time**, by which different **versions** of the same content can be navigated through.

*E.g. an image might be linked with its description, a piece of comment with content commenting on it, and so on. All links are bidirectional.

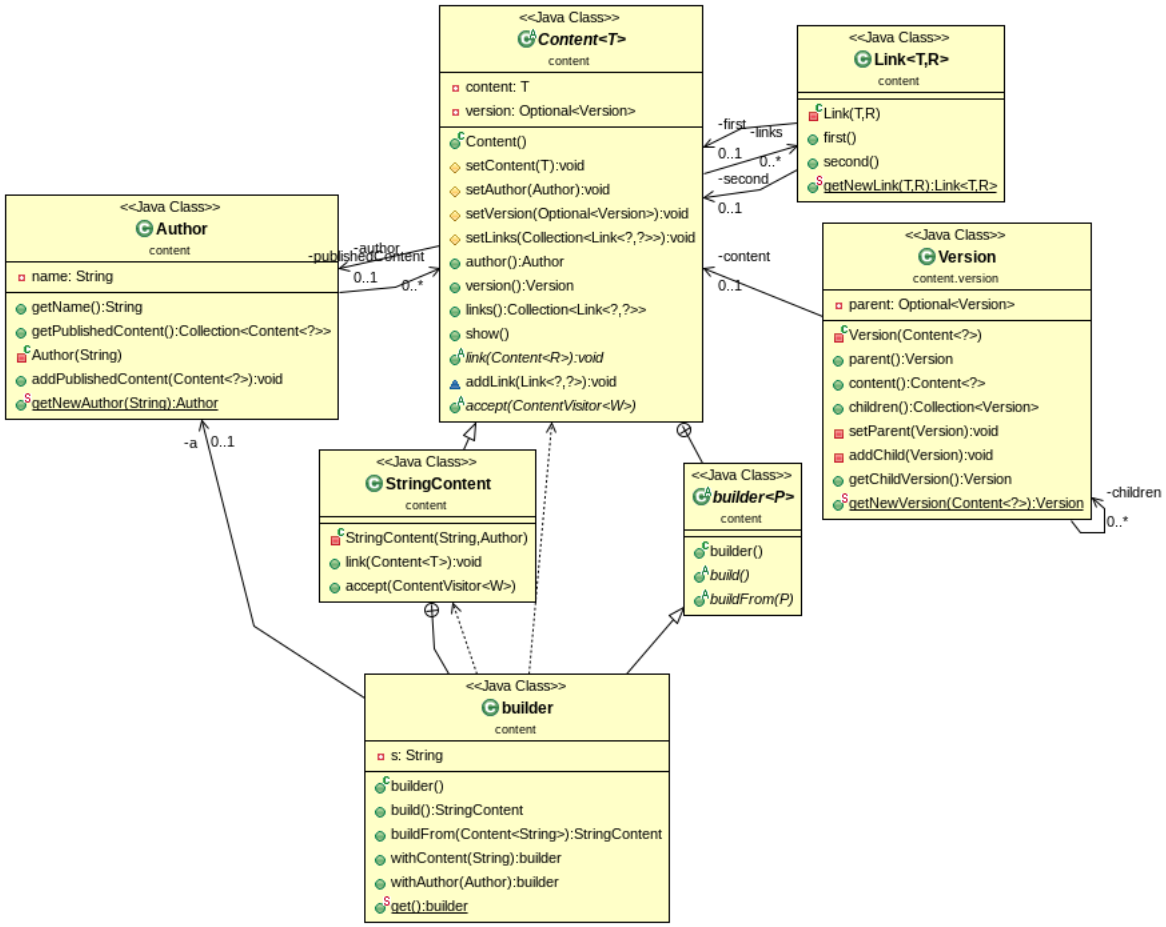
2 Implementation



2.1 General concerns

With such a system, implementation must beget two achievements: first is **elasticity**, as to allow the largest possible variety of operations upon that of content; second is **modularity**, as to allow independent development of its distinct parts, following ISP. Therefore different entities shall be represented, when reasonable, by distinct modules.

2.2 Content Module



2.2.1 Content

Guidelines The content (abstract) class and its module represent the core of our architecture. As we must be able to wrap *any* type of media in a content object, we need the basic class to be completely **independent** of underlying media type. Programming in Java, *Generics* are an easy choice: Content becomes Content<T>; it is thus possible to either subclass the generic class itself, or any specific parameterization directly, depending on the need.
We ought to remember that this class has it's purpose and **responsibility** in providing methods to access content, as well as related entities like **links** and **versions**:
as most of those operations do not depend on media type, and thus on runtime type, we shall take care of providing default implementation for as many of these methods as possible. This will allows for anyone extending the class to concen-

trate on what really *must* be type-dependent: that is, **object construction**.

Referenced Structures Any Content<T>object will reference four different ones: one of type T,called *content*, that is, the *actual* content, one of type *Author*, a collection of Links, and a Version object.

Methods The **Content** class includes package-private *setters*, plus a method to add a single link, for all of its referenced entities, as to enforce proper modification of the object's internal state and to discourage arbitrary action on it:

```
    protected void setContent(T content) {
       this.content = content;
    }

    protected void setAuthor(Author author) {
       this.author = author;
    }

    protected void setVersion(Optional<Version> version) {
       this.version = version;
    }

    protected void setLinks(Collection<Link<?, ?>> links) {
       this.links = links;
    }

    void addLink(Link<?, ?> l) {
    this.links = Stream.concat(links.stream(), Stream.of(l))
    .distinct()
    .collect(Collectors.toList());
 }
```

public methods allow access to stored objects:

```
    public Author author() {
       return this.author;
    };

    public Version version() {
       if (!version.isPresent()){
          this.version = Optional.of(Version.getNewVersion(this));
       }

       return this.version.get();
    }

    public Collection<Link<?, ?>> links(){
```

```
    return this.links;
  }

  public T show() {
    return this.content;
  }
```

Two abstract methods are present, plus a subclass:

```
  /* Return a new Link<T,R> Object, add it to c.links and this.links
     */
public abstract <R> void link(Content<R> c);

  public abstract <W> W accept(ContentVisitor<W> visitor);

    public static abstract class builder<P extends Content<?>> {
    /* Return a new Content<T> Object> */
    public abstract P build();
    /* Return a new version of given Content<T> Object */
    public abstract P buildFrom(P content);
  }
```

the *link(Content <R> c)* methods needs to be abstract, as it will create a *Link<T,R>* object, depending on both linked object's runtime type.

the *accept* methods allows **Content<T>** object to take a visitor of an especially written subclass, allowing for operative flexibility.

the blueprint for a static *builder* class is also provided, outlining methods to allow creation of content objects either *ex novo*, or as new versions of previously existing objects.

Derived classes Several derived classes are provided to illustrate implementation of undefined methods. in particular, a *StringContent* class has been defined as extending Content<String>. Its private constructor will be used by an extension of *Content<T>.builder*:

```
  private StringContent(String s, Author a) {
  super.setContent(s);
  super.setAuthor(a);
  super.setLinks(List.of());
  super.setVersion(Optional.empty());
  }

  public static class builder extends
      Content.builder<Content<String>> {

  private Author a;
```

```
    private String s;

    public StringContent build() {
       StringContent newContent = new StringContent(s, a);
       newContent.version();
       a.addPublishedContent(newContent);
       return newContent;

    }
    @Override
    public StringContent buildFrom(Content<String> content) {
       StringContent newContent = new StringContent(content.show(),
            content.author());
       newContent.setVersion(Optional.of(content.version().getChildVersion()));
       return newContent;
    }

    public builder withContent(String s) {
       this.s = s;
       return this;
    }

    public builder withAuthor(Author a) {
       this.a = a;
       return this;
    }

    public static builder get() {
       return new builder();
    }

  }
```

We can see how *buildFrom(Content<T>) content* method in the *builder* inner class does nothing more than instancing a new object with the same *author* and *content* as the parameter, and setting its version to a *childVersion* of the first. Abstract methods are implemented as follows:

```
   @Override
public <T> void link(Content<T> c) {

   Link<StringContent, Content<T>> newLink = Link.getNewLink(this, c);

   this.addLink(newLink);

   c.addLink(newLink);

}
```

```
@Override
public <W> W accept(ContentVisitor<W> contentVisitor) {
   return contentVisitor.visitStringContent(this);
}
```

The *link(Content<T> c)* method instantiates a *Link* object with the appropriate typing, then uses the *addLink(link<?, ?> l)* method on both **Content** objects, effectively updating their *links* list, while the *accept(ContentVisitor<W>cv)* uses the appropriate method of the Visitor interface.

2.2.2 Link

Guidelines Linking is a capability that is fundamental to our system. The requirement of persistence binds this behavior to a reified form; thus, *link* is represented as an object of its own, and not merely a method of the *Content* class. This is helpful, as it allows for object composition, adding to extensibility of the class itself: in effect, however, if we want clients of the module to be able to govern such behavior we ought to *separate* it from its concrete part. That is, we will act upon **Link** objects through *Content* objects. The link class should be elastic, lightweight, as links may become numerous, extensible but without much necessity to do so, as it is, after all, fundamental, and should not be reimplemented with varying *Content* subtypes.

Referenced Structures The requirement of type-independence is crucial, as the class ought not to be extended often. *Generics programming* is once again useful, and we may declare our class as

```
Link<T extends Content<?>, R extends Content<?>>
```

Thus, the private fields are

```
private T first;
private R second;
```

Methods Instantiation is provided by a static method calling upon a private constructor:

```
private Link(T t, R r) {
   this.first = t;
   this.second = r;
}

public static <T extends Content<?>, R extends Content<?>> Link<T, R>
    getNewLink(T t, R r) {
   return new Link<T, R>(t, r);
}
```

Finally, methods are provided to access linked elements:

```
public T first() {
   return this.first;
}

public T second(){
   return this.second;
}
```

2.2.3 Version

Guidelines Persistence and ease of modification are both useful features, and we would like to have both. A *versioning system* is the structure we need: a navigable version tree, which allows preservation of different versions of the same *Content* and their relations, providing us the capability to store and update references. a **Version** object shall represent the *atom* of such a structure. Its responsibility is that of allowing access to corresponding *content*, as well as *parent* and *children* versions. It should allow for creation of new **Version** objects in relation to previously existing one, as to grow our structure. Like other structures connecting different content, we should be able to concretize it without mind of runtime *content* type.

Referenced Structures the class, simply declared as *public class Version*, has three private fields:

```
private Optional<Version> parent;
private Collection<Version> children;
private Content<?> content;
```

use of *Optional* in the *parent* fields aids in the handling of parentless objects. as for referenced content parameter, it is established at runtime as the wildcard indicates.

Methods Instantiation of **Version** objects is handled both statically and dynamically:

```
private Version(Content<?> content) {
   this.content = content;
   this.children = List.of();
   this.parent = Optional.empty();
}

public static Version getNewVersion(Content<?> content) {
   return new Version(content);
}
```

```
public Version getChildVersion() {
   Version childVersion = getNewVersion(this.content());
   this.addChild(childVersion);
   childVersion.setParent(this);
   return childVersion;
}
```

While static method *getNewVersion* merely calls upon the private constructor, returning a **Version** object referencing the passed *content*, instance method *getChildVersion* provides a new version object, sets it as a child of the one calling that method, and returns the first; content creation methods can then use the returned **Version** and attach it to corresponding *Content*. Simple methods are provided for access and manipulation of *parent* and *children*:

```
public Version parent() {
   return this.parent.orElseGet(() -> this) ;
}
public Content<?> content(){
   return this.content;
}

private void setParent(Version v) {
   this.parent = Optional.of(v);
}
private void addChild(Version v) {
   this.children = Stream.concat(this.children.stream(), Stream.of(v))
   .collect(Collectors.toList());
}
```

2.2.4 ContentVisitor

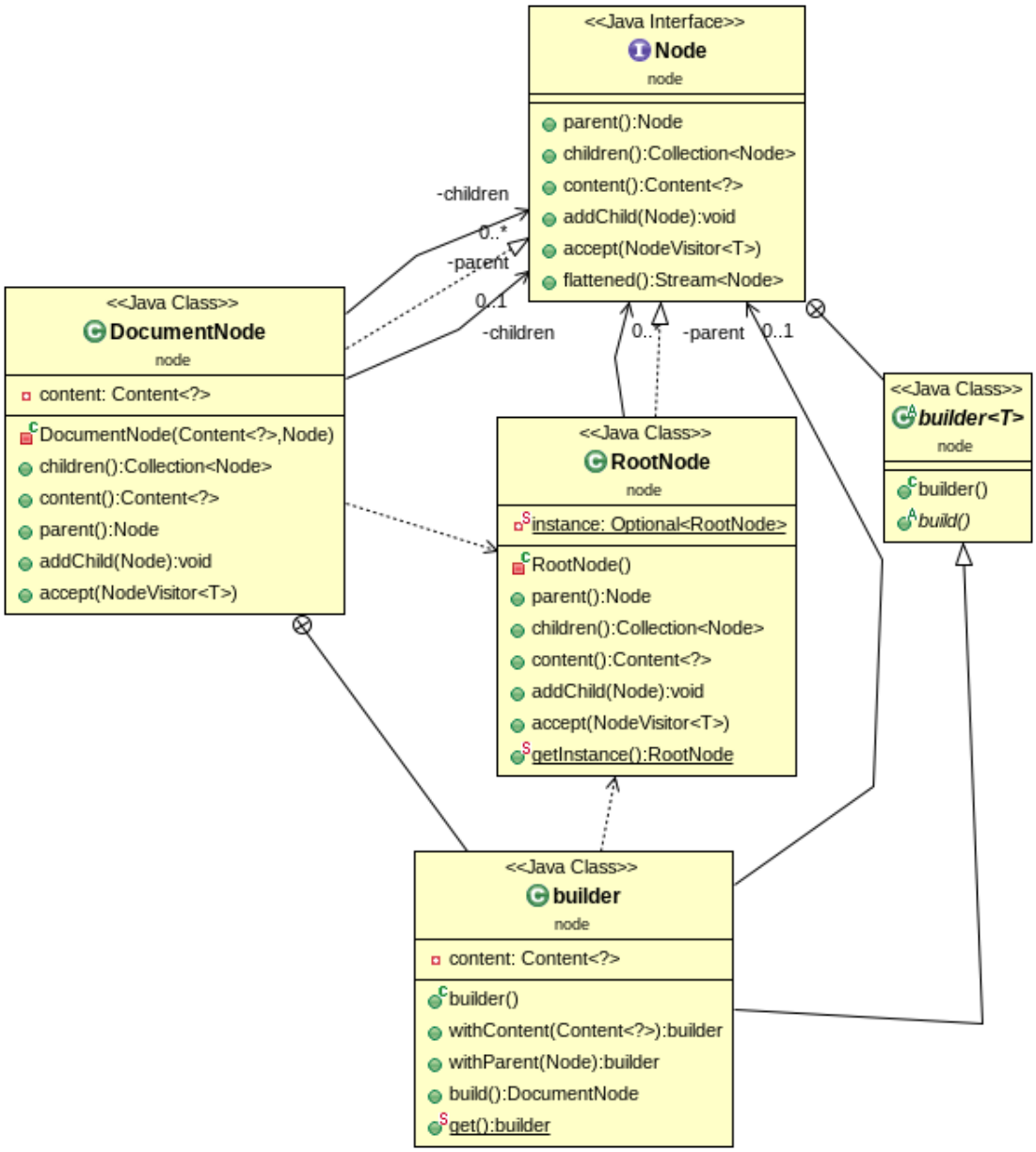
Guidelines This should represent nothing more than a typical Visitor interface, allowing for different operations on different *Content* subtypes. One method is provided for each given implementation of *Content*.

```
public interface ContentVisitor<T> {
   T visitStringContent(StringContent c);
   T visitNodeContent(NodeContent c);
   T visitIntegerContent(IntegerContent c);
   }
```

Typing **T** ought not to correspond to content's, but rather to desired return type.

2.3 Node module

2.3.1 Node



Guidelines Content alone, without the possibility of logical organization, which is by nature more arbitrary than either contextual linkage or versioning, is not very useful. We ought to be able to compose content in any way we

deem useful, and organize data according to the varying requests of clients.

Having abstracted away content, we abstract its organization at a slightly higher level, introducing abstract coupling on the **Node** side.

Node objects will thus pass around **Content**, organizing it logically.

Although it is useful to arrange **Nodes** in some kind of tree structure, it would be an error to employ the *Composite* pattern as it is: **Content** may indeed need to be recursive and granular, but that would in any case be the responsibility of Content itself. Instead, in the **Node** package, the role of nodes is inverted: inner nodes will store Content, while empty leaves will all point to a single instance of the concrete subclass *RootNode*.

Node There is no need for the **Node** interface to explicitly embed type variance. Having a method to return stored content shall give us all the variance we need.

```
public interface Node {

   Node parent();

   Collection<Node> children();

   Content<?> content();

   void addChild(Node newChild);

   <T> T accept(NodeVisitor<T> nodeVisitor);

   default Stream<Node> flattened() {
      return Stream.concat(
      Stream.of(this),
      children().stream().filter(i ->
          !(i.equals(RootNode.getInstance()))).flatMap(Node::flattened));
   }

   public static abstract class builder<T extends Node> {

      public abstract T build();

   }

}
```

The interface is very simple, to the basic methods for accessing the contents of a node and to those allowing navigation of the structure, we only ought to add a method to add children, one to accept those visitors which will actually navigate the structure, and the blueprint for a *builder* inner static class. A default method is given to provide a flattened representation of the node tree through a Stream.

Derived Classes The fundamental concrete subclass is **RootNode**. Such class allows for easier maintainability of the structure: we are guaranteed that elements at the extremes of such structure will be the *exact same instance* of the RootNode class.

```
public class RootNode implements Node {

   private static Optional<RootNode> instance = Optional.empty();
   private Collection<Node> children;

   private RootNode() {
      this.children = List.of(this);
   }

   @Override
   public Node parent() {
      return this;
   }

   public static RootNode getInstance() {
   if (instance.isEmpty()) {
      RootNode r = new RootNode();
      instance = Optional.of(r);
      }
      return instance.get();
   }
```

RootNode will return itself when asked for its parent. By default, its list of children will only contain itself: however, since RootNode is supposed to be the root of all trees, this changes if when we add the first new node:

```
   @Override
public void addChild(Node newChild) {
   this.children = Stream.concat(children.stream(),
       Stream.of(newChild))
   .distinct()
   .filter(i -> !i.equals(RootNode.getInstance()))
   .collect(Collectors.toList());
}
```

What do we make of RootNode's Content? We can again make use of the *singleton* design pattern, with an inner implementation of **Content**:

```
   @Override
public Content<?> content() {
   return EmptyContent.getInstance();
}
```

```
   static private class EmptyContent extends Content<String> {

   static Optional<EmptyContent> instance = Optional.empty();

   private EmptyContent(){
      super.setContent("It's considered rude to look inside the root
          node.");
      super.setLinks(List.of());
      super.setVersion(Optional.empty());
   }


   @Override
   public <R> void link(Content<R> c) {
   }

   @Override
   public <W> W accept(ContentVisitor<W> visitor) {
      return null;
   }

   static EmptyContent getInstance() {
      if (instance.isEmpty()) {
         return new EmptyContent();
      }
      else return instance.get();
   }

}
```

since **Content** should be accessed through **Nodes**, it is much safer to have a null reference in the accept method of the inner private class than in the content() method of **RootNode**: Visitors of the Node interface should first visit the node appropriately, then, if necessary, call on Content Visitors. Looking for contents in nodes designed to be functionally empty is of course inappropriate.

Non-empty **Content** shall be instead accessed through inner nodes, denominated **DocumentNode**:

```
public class DocumentNode implements Node {

   private Collection<Node> children;
   private Content<?> content;
   private Node parent;

   private DocumentNode(Content<?> content, Node parent) {
      this.content = content;
```

```
        this.parent = parent;
        this.children = List.of(RootNode.getInstance());
    }

    @Override
    public Collection<Node> children() {
        return this.children;
    }

    @Override
    public Content<?> content() {
        return this.content;
    }


    @Override
    public Node parent() {
        return this.parent;
    }

    @Override
    public void addChild(Node newChild) {
        this.children = Stream.concat(children.stream(),
            Stream.of(newChild))
        .distinct()
        .dropWhile(i -> i.equals(RootNode.getInstance()))
        .collect(Collectors.toList());

    }

    @Override
    public <T> T accept(NodeVisitor<T> nodeVisitor) {
        return nodeVisitor.visitDocumentNode(this);
    }
```

an implementation of the abstract builder inner class is also provided:

```
public static class builder extends
    Node.builder<DocumentNode>{

private Content<?> content;
private Node parent = RootNode.getInstance();

public builder withContent(Content<?> content) {
    this.content = content;
    return this;
}

public builder withParent(Node parent) {
    this.parent = parent;
```

```
            return this;
        }


        @Override
        public DocumentNode build() {
            DocumentNode newNode = new DocumentNode(content,
                parent);
            parent.addChild(newNode);
            this.content = null;
            this.parent = RootNode.getInstance();
            return newNode;

        }

        public static builder get() {
            return new builder();
        }

    }
}
```

The build() method resets the builder after instantiating the new object, allowing for the builder to be reused.

2.3.2 NodeVisitor

A light interface, providing methods to visit both internal and external nodes:

```
public abstract interface NodeVisitor<T> {

    public abstract T visitDocumentNode(DocumentNode n);
    public abstract T visitRootNode(RootNode n);


}
```

Various implementations are provided. **MatchingVisitor** allows us to find the first Node, either the target node or one of its descendants, matching a given predicate:

```
public class MatchingVisitor implements NodeVisitor<Node> {

    Predicate<Node> predicate;

    private MatchingVisitor(Predicate<Node> p) {
        this.predicate = p;
    }
```

```
    @Override
    public Node visitDocumentNode(DocumentNode n) {
       return n.accept(Flattener.get())
       .filter(i -> predicate.test(i))
       .findFirst()
       .orElseGet(() -> RootNode.getInstance());
    }

    @Override
    public Node visitRootNode(RootNode n) {
       return RootNode.getInstance();
    }

    public static MatchingVisitor matching(Predicate<Node> p) {
       return new MatchingVisitor(p);
    }


}
```

2.4 Transclusion

Nelson defines transclusion as

> *"[...]transclusion, or reuse with original context available,* ***through embedded shared instancing (rather than duplicate bytes)****. Thus the user may intercompare contexts of what is re-used, both for personal work (keeping track of reuse) and publication (for deep comprehension and study)"*[†]

We need to be able to reference content contextually, as to reuse without duplication and maintain access to surrounding content. the **Content** class being generic aides us one more time. we can conform to the same interface by embedding the very node targeted by transclusion inside a content object. Thereby, we need to instantiate the type **Content<Node>**, e.g. trough a concrete implementation like *TranscludedContent*, which uses a simple static method *from(Node n)* returning the appropriate object.

```
public class TranscludedContent extends Content<Node> {

   private TranscludedContent(Node n) {
      super.setContent(n);
      super.setAuthor(n.content().author());
```

[†]The Heart of Connection: Hypermedia Unifiedby Transclusion, Theodor Nelson, Communications of the ACM August 1995/Vol. 38, No. 8 https://www.ics.uci.edu/~redmiles/ics227-SQ04/papers/Hypertext/Primary/p31-nelson.pdf

```
        super.setLinks(List.of());
        super.setVersion(Optional.empty());
    }

    public <T> void link(Content<T> c) {
        Link<TranscludedContent, Content<T>> newLink =
            Link.getNewLink(this, c);
        this.addLink(newLink);
        c.addLink(newLink);
    }

    @Override
    public <W> W accept(ContentVisitor<W> contentVisitor) {
        return contentVisitor.visitNodeContent(this);
    }

    public static Content<Node> from(Node n) {
        return new TranscludedContent(n);
    }

}
```

an appropriate visitor may return a Node representing a transclusion of target node, instantiating such objects and putting them into a node that is then returned:

```
public class Transcluder implements NodeVisitor<Node> {

    @Override
    public Node visitDocumentNode(DocumentNode n) {
        DocumentNode.builder nodeBuilder = DocumentNode.builder.get();
        Content<Node> newNodeContent = TranscludedContent.from(n);
        return nodeBuilder.withContent(newNodeContent).build();
    }

    @Override
    public Node visitRootNode(RootNode n) {
        return RootNode.getInstance();
    }

    public static Transcluder get() {
        return new Transcluder();
    }

}
```

proper operation on such content will depend on visitors of the Content class: for example in the case of class *ToStringVisitor*

```
    @Override
  public String visitTranscludedContent(TranscludedContent c) {
    return c.show().content().accept(this);
  }
```

2.5 Document

In designing such a system, we ought to take care in building *segregated* interfaces for those who intend to implement classes declared by us; however, a *facade* hiding information from clients may be useful as to ease use, as well as to simplify GUI implementation. We thus provide the **Document** abstract class.

```
public abstract class Document {

  private Node topNode;
  private Node currentNode;

  void setTopNode(Node n) {
    this.topNode = n;
    this.currentNode = n;
  }

  public void setCurrentNode(Node node) {
    this.currentNode = node;
  }

  public Node topNode() {
    return topNode;
  }

  public abstract void write(String s);

  public abstract void draw();


  Node getCurrentNode() {
    return currentNode;
  }

  public void transclude(Node n) {
    NodeVisitor<Node> transcluder = Transcluder.get();
    Node newNode = n.accept(transcluder);
    currentNode.addChild(newNode);
    this.currentNode = newNode;
  }

}
```

An example implementation is listed as **DummyDocument**

```
public class DummyDocument extends Document{

   @Override
   public void draw() {
      topNode().accept(ContentListVisitor.get())
      .stream()
      .map(i -> i.accept(ToStringVisitor.get()))
      .forEach(i -> System.out.println(i));
   }

   @Override
   public void write(String s) {
      StringContent.builder contentBuilder =
          StringContent.builder.get();
      DocumentNode.builder nodeBuilder = DocumentNode.builder.get();

      Content<String> newContent = contentBuilder
      .withAuthor(topNode().content().author())
      .withContent(s)
      .build();

      Node newNode = nodeBuilder
      .withContent(newContent)
      .withParent(getCurrentNode())
      .build();

      setCurrentNode(newNode);
   }


   public static Document newFromContent(Content<String> c) {
      Document newDocument = new DummyDocument();

      Node newNode =
          DocumentNode.builder.get().withContent(c).build();

      newDocument.setTopNode(newNode);

      return newDocument;

   }
```

implementing void methods either to write a string to the document, or to display the document itself. *Documents* only have only two private fields: *currentNode* refers to the node being currently selected, while *topNode* represent the document's head. While the abstract **Document** class does not depend on the content class, its implementation will be inevitably tasked with instancing

both Nodes and Contents, and packaging one into the other.

Thus far, we have applied the visitor, singleton, builder, and facade pattern, plus a slight variation of the Composite pattern.