# Base Architectures for NLP

*Tom Mahieu, Stefan Raeymaekers et al.*

Department of Computer Science
K.U.Leuven

## Abstract

Our goal is to develop an object-oriented framework for natural language processing (NLP). With this framework it should be possible to create a variety of applications ranging from simple spelling checkers to complex translation systems, just by plugging different components (e.g. a morphological lookup component, chart parser component, etc) in the framework.

This paper compares two base architectures that can form the core of such an NLP system. The first one considers NLP from a processing point of view (a text gets processed), the second one from a text point of view (a text processes itself).

Some important considerations are made on how configurable and open the system is. Considerations to make the system concurrent and distributed to obtain better performance are also discussed.

## 1 Introduction

Natural language processing (NLP) systems encompass a large range of applications ranging from simple spelling checkers over grammar checkers to complex translation systems. As all these applications have the same NLP context, they contain several similarities. Despite these similarities, application programmers have not made any effort to build their systems modular enough in order to reuse certain parts of one system in another system. The result of this approach leads to very monolithic systems that are hard to extend and maintain. Also, the development of new systems is a very costly process since every application has to be written from scratch again.

The preferred way to build new systems nowadays is to create an open and flexible system that presents base functionality to the application programmer. This base system is created in such a way that its default behavior can be changed in a uniform way. The application programmer has to write application specific code, or in the best case reuse an already existing component, that can be plugged in the base system. In this way, the programmer can create a family of related products. Using this approach, reuse is driven to the maximum; the base system is always reused, and existing components can also be reused when desired.

Our first goal is to obtain a flexible NLP system by creating an object-oriented (OO) framework that presents the base system. The application programmer gets the necessary information on how to build components that fit in this system. The object-oriented way of doing this is to provide the application programmer the necessary interfaces to which the component must comply. Also a library of off-the-shelf components the application programmer can use to plug into the base system can be provided. We will focus on text processing systems. Our second goal is enabling the system to run in a concurrent and distributed environment. This will also be beneficial since NLP systems can perform very computational intensive tasks.

This paper will present two core architectures that can be used as a base for an NLP system. We also discuss what the possibilities of concurrency and distribution could be when applied to the two core architectures. Section 2 presents the two main ideas concerning a base system: NLP from a processing point of view (section 2.2) and NLP from a text point of view (section 2.3). Section 3 discusses the openness and flexibility of both systems. Section 4 will provide more information concerning the concurrency and distribution issues. We conclude with the future plans we make.

## 2 Base Architectures for NLP

In this section we will present two core architectures that can form the base for a NLP system, for text processing systems more specifically. Two important things in an NLP system are the representation of the text and the representation of the algorithms that will process that text. These two factors are the driving force of the two core architectures presented here. They originate from two different points of view:

- A processing point of view: the process is the driving force. A process accepts a text and processes it.
- A text point of view: the text is the driving force. A text is processed by taking a process and applying it to itself

Since the two points of view use the same basic model of a text, this text model is presented first. Section 2.2 and Section 2.3 will then explain the two approaches. Each point of view consists of two parts: first the concepts are explained, then these ideas are supported by more formal UML diagrams. Note that no knowledge of UML is required to understand the concepts explained in this paper. The UML models in this paper just show a more formal approach to the presented concepts.

### 2.1 The Text Model

The text model we present in this section is used by both points of view. First the conceptual text model we have in mind is explained, then we give an example of a UML diagram of a text model.

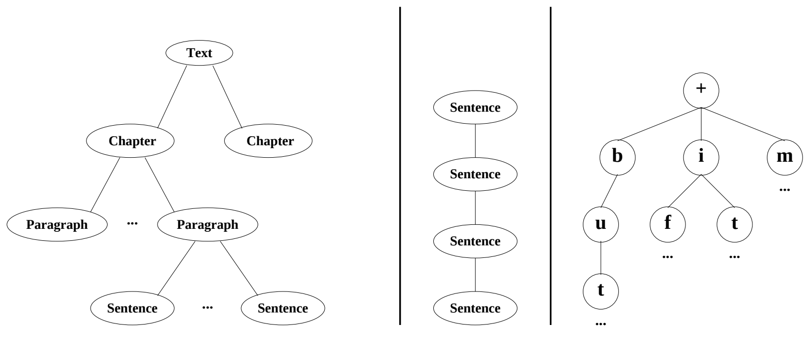


Figure 1: The text model with some TDV's

### 2.1.1 Concept

Since it is our intention to develop a framework for natural language processing, the text model should not impose any limitations to the possible applications that will be instantiated from our framework. The text model we use will thus often contain more information than we will need in one application. E.g. layout information can be included which will not be relevant for a spelling checker.

We hierarchically decompose a text in chapters, paragraphs, sentences and so long. Since it is not likely that this hierarchical model will always be used, we provide the possibility to create user defined text data views. It is possible to create a text data view (TDV) that contains only the information we need for a certain text processing routine. Figure 1 shows our conceptual text model with some possible TDV's. In the figure, the text is also seen as a list of sentences and a letter tree containing all the words in our text.

The way we intend to work with TDV's is the following. A TDV gives us access to a particular subset of our text. During the processing this subset can be changed. This change will be propagated to the actual complete text when it is desired. This is necessary because some processing routines will use the normal, hierarchical text model, and others will access the text model through a TDV. It should be possible to propagate changes made using a TDV to the hierarchical text model. In this way the latter routines can always cooperate with the former.

### 2.1.2 UML

A UML model of a text model for a translation system can be found in figure 2. The Text class contains one (and only one) obligatory hierarchical text model. This hierarchical text model is a Composite pattern (Gamma, Helm, Johnson and Vlissides 1995). In this model the basic components are sentences. The other components such as chapters, sections, paragraphs etc. are implemented as a Hier-

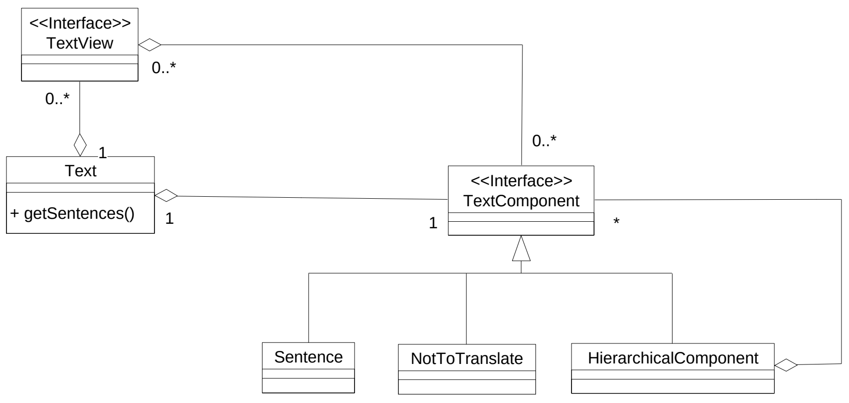


Figure 2: The UML text model

archical Components. Components that are not interesting for translations such as pictures are encapsulated in NotToTranslate objects. The basic components don't have to be sentences. They can also be words, depending on the problem domain you are dealing with.

A text can have several TDV's (TextView class) which can be related to the hierarchical text model. The TextView class is an abstract class because the behavior of every single TDV will be different from another one.

## 2.2 The Processing Point of View

### 2.2.1 Concept

**A Pipeline and Dynamic Behavior** This point of view originates from the traditional way of procedural thinking. A text undergoes several processing steps to obtain the desired result. This way of working can be seen as sending a text through a pipeline of processing steps. A pipeline is modeled using different pipeline processes that can be connected to each other according to a specific configuration. An example of a pipeline is depicted in Figure 3. The example represents a sequence of possible processing steps in a translation system. A text is submitted to the first processing step. When the step is finished the text is sent to the next processing step. When the last step exits, the result is returned to the caller of the pipeline.

This way of working always leaves us with fixed pipelines: all processing steps will be taken when a text is submitted to the pipeline. For a framework this model is not flexible enough. Often another pipeline setup will be needed when certain conditions are met. These conditions can be static and can sometimes even change

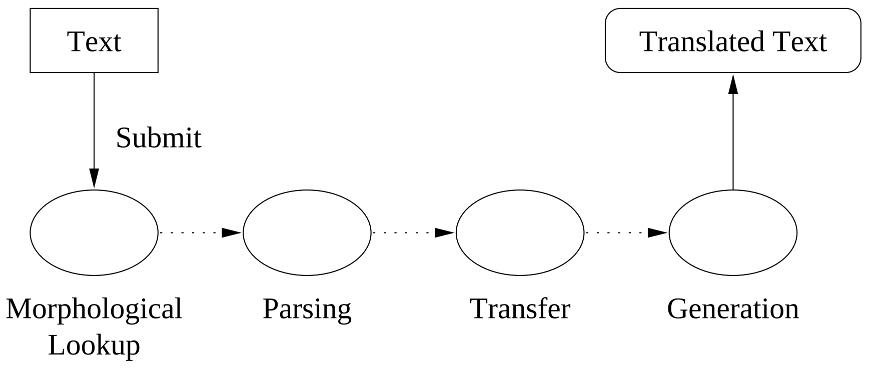


Figure 3: A simplified translation pipeline

dynamically (during processing). To be able to change the pipeline, we introduce a pipeline monitor. When a text needs to be processed it is submitted to the pipeline through the pipeline monitor.

The application programmer encapsulates all the pipeline knowledge into this pipeline monitor(PM): what steps are needed to obtain the desired result (static conditions) and what alternatives are possible that can only be determined at run-time (dynamic conditions), such as characteristics of the text, whether some TDV is present, whether the text has already been processed etc.

Figure 4 shows the way the PM interacts with the actual pipeline. The PM first selects a first step in the pipeline and sends the text to this step. The particular processing step accepts the text, handles it and then returns the resulting text back to the PM. The PM then selects the following processing step, submits the text, etc. When certain conditions are met the PM can change the pipeline by inserting processing steps or taking alternative processing steps as can be seen on the figure.

**Nested Pipelines** Often the entire detailed text structure is not needed when we want to process a text. We do not want to write a spelling checker algorithm that has to know the entire text structure. We just want it to accept a word, check if it is written correctly, and report when a misspelled word is encountered.

We solve this problem by making it possible to nest pipelines. The pipelines can be nested in any way, but they will mostly be nested according to the text structure. First, we make a distinction between iteration processes and implemented processes. An iteration pipeline process knows only about (a part of) the text structure and *nested pipelines*. An implemented pipeline process also knows how to *process* (a part of) the text structure. An iteration process can only structurally decompose the text and delegate the substructures to a nested pipeline where an implemented process also implements a part of the actual text processing.

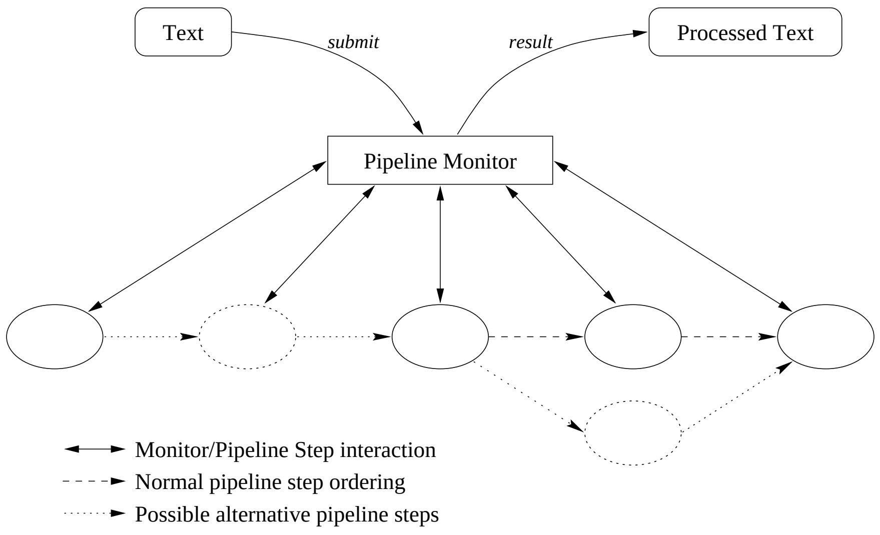

Text
submit
result
Processed Text
Pipeline Monitor
Monitor/Pipeline Step interaction
Normal pipeline step ordering
Possible alternative pipeline steps

Figure 4: The pipeline monitor enables dynamic pipelines

Iteration processes are always related to a nested pipeline. Figure 5 is an example of a translation system that is implemented using a nested pipeline. First the text is submitted to a text pipeline. First a morphological lookup is done (ML), after which the actual processing is started. However, if we only translate on a sentence by sentence basis, our translation step (TL) can be modeled as an iteration process. The process decomposes our text in sentences (by using a TDV or by really decomposing the text) and submits those sentences one by one to a sentence translation pipeline. The Sentence Pipeline Monitor then does the actual translation of each sentence (Parsing (PA) and Transfer (TR)). When the sentence is translated, the result is passed to the caller of the pipeline, which is the TL process. After TL iterated over all the sentences, the text is translated and the next text processing step in our text pipeline can be called (Generation (GE)).

This nesting can go as deep as desired. E.g. The sentence pipeline can consist of an iteration process that splits the sentence up in words, which can then be sent to a word pipeline.

#### 2.2.2 UML

A UML class model of the process point of view is depicted in figure 6. Every pipeline monitor is aware of several processes that can process the data structure the monitor is responsible for. These processes are divided between Implemented Processes and Iteration Processes. Iteration Processes can delegate substructures to pipelines that can process the substructures.

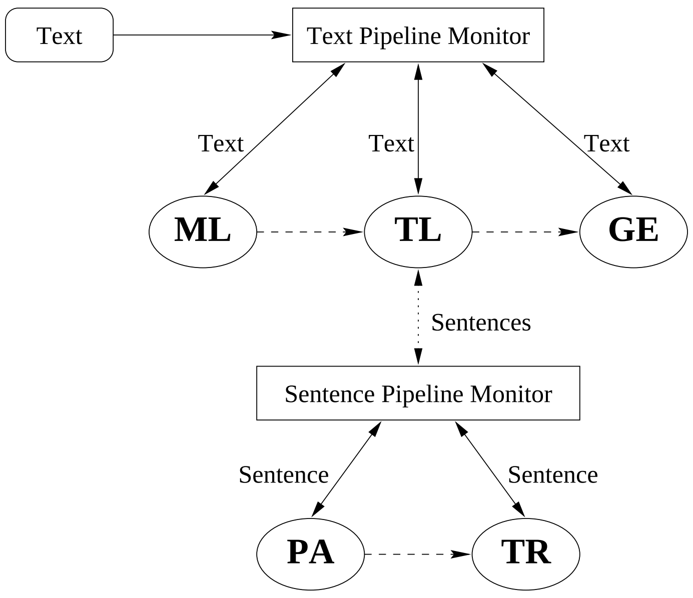


Figure 5: A nested translation pipeline

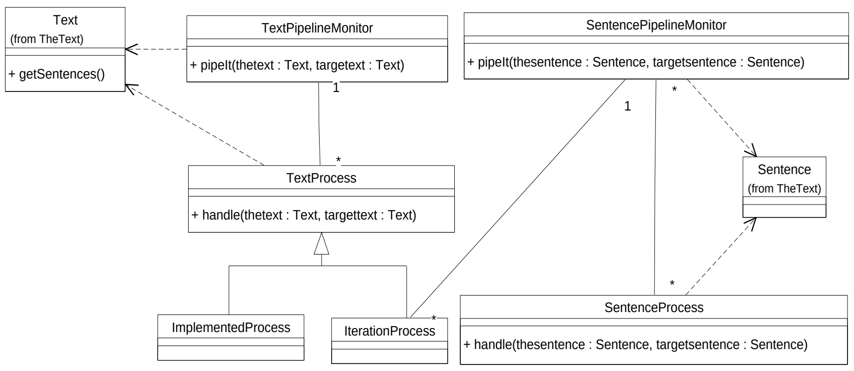

Text
(from TheText)
+ getSentences()
TextPipelineMonitor
+ pipeIt(thetext : Text, targetext : Text)
1
*
TextProcess
+ handle(thetext : Text, targettext : Text)
ImplementedProcess
IterationProcess
*
SentencePipelineMonitor
+ pipeIt(thesentence : Sentence, targetsentence : Sentence)
1
*
Sentence
(from TheText)
*
SentenceProcess
+ handle(thesentence : Sentence, targetsentence : Sentence)

Figure 6: A UML model of a nested pipeline

The figure is not complete and is only a part of a complete pipeline. Iteration processes can be related to every type of pipeline as long as it is a pipeline that can handle a substructure of the structure the iteration process can handle. The example in the figure shows us only a subset of the entire situation: an iteration process from a text pipeline related with a sentence pipeline monitor.

Pipeline monitors have nothing in common with each other. Indeed, they accept text (sub)structures. We could have made one general pipeline monitor that handles text component classes. The pipeline monitor could then be connected to a general text component process. However, we would have to insert reflection code, which enables us to obtain precise type information on the classes we work with, to check if we make correct pipelines. We could easily build pipelines where a sentence pipeline has a text pipeline nested in it, if we don't do the necessary checks.

In our model everything is made specific. Which means we have a lot more classes. Instead of one general monitor with one general text component it can handle, we have made pipeline monitors and related processes for every type of text component. Disadvantage is that we have an explosion of classes when we have a very fine grained text model. In that case, working with a general pipeline monitor and processes would be more feasible.

## 2.3 The Text Point of View

### 2.3.1 Concept

The text point of view originates from a more object oriented background. Instead of having an algorithm that processes a text, the text will process itself. This means that next to the data representation, the text encapsulates information on how to process itself.

The biggest problem that arises is the reusability of the system. Since all the information will be placed in the text data structure, the data structure will need to be changed when we want to write another text processing application. Since this is not the desired situation for a text processing framework, we decouple the text and the processing algorithm from each other. The text model we use is the same as presented in section 2.1. The processing components are encapsulated in *text visitors*.

To enable easy configuration, we again have a supervising unit, called the processor, that implements a text processing system using several text visitors and controls the entire processing statically as well as dynamically (by taking runtime decisions). The processor selects several text visitors in some order, hands them to the text and asks the text to process itself using these text visitors. Bottom line is that the processor knows *what* needs to be done, step by step. The text visitors are the steps that know *how* to do it, as long as the processor sends them to the text in the right order.

The text visitor (TV) reflects the hierarchical structure of the text. For every

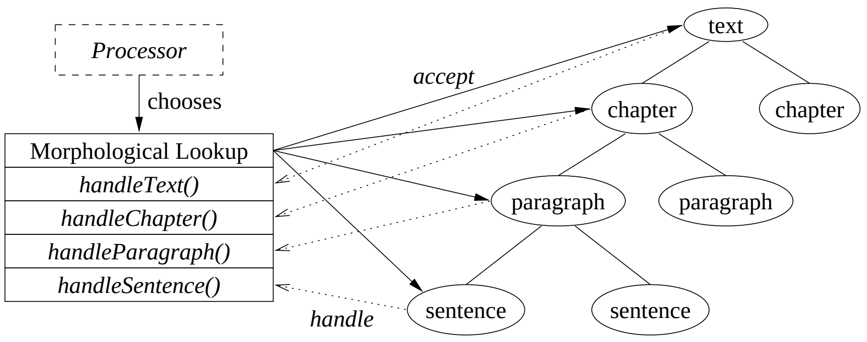


Figure 7: The text point of view

subcomponent of the text, the TV contains a handler that tells the visitor how to process that type of subcomponent. For every possible TDV there should be a separate text visitor. The processor can instantiate the right TV when a TDV is present and send it to the text.

Figure 7 clarifies this approach with an example. We are implementing the translation process we already presented in section 2.2.1. First we want to look up word information in our dictionary. Our processor chooses a morphological lookup (ML) component and asks the text to accept the component. Upon acceptance, the text starts processing itself by calling the *handleText()* method. *handleText()* takes the title of the text, which can be retrieved from the root text component of the text structure, and starts looking up the words from the title. When *handleText()* finishes looking up the words, *handleText()* starts iterating over the subcomponents of the Text object (chapters) by sending the ML component to these subcomponents. The chapter objects also accept, but this time *handleChapter()* is called. The words in the title of the chapter are looked up, and then *handleChapter()* iterates again over the paragraphs in the text. This continues until the leaves in our text structure are looked up. The ML component finishes and the processor can select the following component, e.g. a chart parser (PA). The chart parser is sent over the text structure in the same way, parsing the text title, chapter title and so long[1]. Subsequently a Transfer (TR) TV and a Generation (GE) TV (in that order) complete the processing of the text.

[1]Note that because of the structural processing of the text, it is possible to plug in different grammars for different sentence types. A chapter title will probably not be a grammatically correct sentence, but can be an incomplete sentence. To obtain a correct parsing of these constructions, a modified grammar can be used for different structural text components, such as a chapter or a sentence.

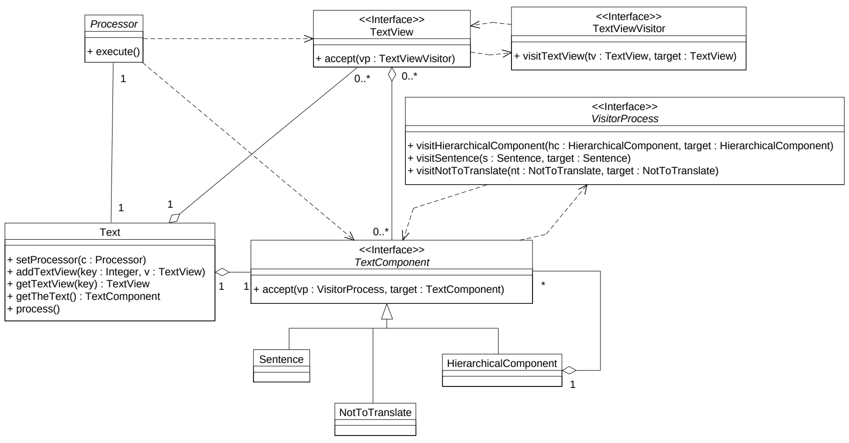

Processor
+ execute()
<<Interface>>
TextView
+ accept(vp : TextViewVisitor)
<<Interface>>
TextViewVisitor
+ visitTextView(tv : TextView, target : TextView)
<<Interface>>
VisitorProcess
+ visitHierarchicalComponent(hc : HierarchicalComponent, target : HierarchicalComponent)
+ visitSentence(s : Sentence, target : Sentence)
+ visitNotToTranslate(nt : NotToTranslate, target : NotToTranslate)
Text
+ setProcessor(c : Processor)
+ addTextView(key : Integer, v : TextView)
+ getTextView(key) : TextView
+ getTheText() : TextComponent
+ process()
<<Interface>>
TextComponent
+ accept(vp : VisitorProcess, target : TextComponent)
Sentence
HierarchicalComponent
NotToTranslate
1
1
1
0..*
0..*
0..*
1
1
*
1

Figure 8: A UML model of the text point of view

### 2.3.2 UML

The implementation of this system can be reduced to two design patterns(Gamma et al. 1995): A Composite pattern for the text representation and a Visitor pattern to implement the Text Visitors. A model of the system can be found in figure 8.

The text model used here is exactly the same as the one in Section 2.1.2. The model is enriched with a Visitor that can handle the possible concrete Composite classes Sentence, NotToTranslate and HierarchicalComponent. For each concrete class in the text model, an operation should be present in the visitor. For TextViews, another Visitor class is present, since it will not use the standard text model, but the specific TDV structure.

The usage of this architecture is the following. First the text class is configured with a Processor through the *setProcessor()* member function. Then the *process()* function is called which starts the processing of the text using the Processor. There is also the possibility to add text views and requesting those text views to the Text class.

The TextComponent class contains an *accept()* operation which is the entry point for the Visitor. The Visitor class is accepted through this accept call. Every concrete text component then implements this method in order to call the correct visitor method, which will then do the actual processing. Also note that each text component will also have to provide an interface so that component specific information can be retrieved. This is not an easy task, since the interfaces should provide information for all the possible text processing applications.

### 2.4 Discussion

The two approaches can be seen as opposite point of views. The process point of view boils down to sending a text through several processes to obtain a result. The text point of view is sending several processes through a text.

Although the two models can be seen as opposites, they have a lot in common too. They share the same text model. They both offer the possibility to take run-time processing decisions based on text characteristics. This is obtained by using pipeline monitors in the process point of view and processors in the text point of view.

Also note the following: since a text visitor has an operation to handle every text substructure, the application programmer is forced to obey the text structure. A handling routine needs to be written for every substructure. There is no other way to visit the entire text structure otherwise. In the process point of view this is not the case. You can handle the entire text in one routine if you want that.

In the process point of view, pipelines get nested according to text structure. Since in the text point of view, text processing always happens according to text structure, there is no nesting of text visitors necessary.

## 3 Some Non-Functional Considerations

The previous section discussed two ways of how to process text. In this section, some considerations are made on how feasible these two ways of working are to form the base of a framework. We discuss how the base system can be configured into different text processing applications. We also make some reusability consideration concerning the systems. Also important is how flexible the systems are in creating applications.

### 3.1 Configurability

A framework should offer some base functionality that can be configured into a complete, working application. Both our systems introduce a pattern the application programmer must work with. In the process point of view, we need to build a pipeline to create a text processing system. In the text point of view we have to send a sequence of text visitors to the text to create a text processing system.

In the processing point of view, an application can be made by making a pipeline monitor, provided that all the pipeline steps are available. The situation is similar for the text point of view, where a processor encapsulates all of the application's logic. In both approaches, the actual application logic is centralized in one object and with both approaches, creating a new system boils down to choosing the right pipeline processing steps or text visitors and putting them in the correct order.

## 3.2 Reusability

The goal of a framework is to drive reusability to the maximum. When we want to create a new application, we should be able to do this with a minimal effort. The ideal situation is to build component based systems: a new application is built by taking several relatively independent components and bringing them together. The cooperative behavior of the components (morphological lookup system, chart parser, text formatter, word counter, etc) results in a very specific application (translations system, spelling checker, text categorization system, etc.).

The two systems encourage you to work component oriented, but none of them really enforces you to build relatively independent components. In the pipeline model, every pipeline step can be programmed to be an independent processing step. However, there is nothing to stop you from passing other information, other than the information stored in the text, to other pipeline processes. By doing this, you create perhaps unwanted dependencies between pipeline processes.

The same problem occurs in the text point of view. Text visitors are the reusable components, but they can be related to each other by passing data structures to each other, instead of just relying on the information present in the text structure.

This may sound bad, but usually there is always some kind of dependency between several components. A grammar checker will not work if there is no grammatical information about the words present in the text. In that way the grammar checker will depend on a word lookup system. But the word lookup system can for instance also be used in a translation system.

The dependencies we want to avoid are the following. E.g. Some pipeline process can pass information, other than information that is directly related to the text, by means of a TDV to another process, which is further down the pipeline. This creates a dependency between the two pipeline processes. The latter process will never be able to work if the first process did not create the TDV.

## 3.3 Flexibility

Now that we have a, to some extent, configurable and reusable system, it is also interesting to know how easy it is to create a new text processing system. Since we try to work component oriented, it will be pretty easy to create a new applications when all the necessary components are available. We only need to create a new Pipeline monitor (process p.o.v.) or Processor (text p.o.v.) which instantiates the correct components.

It will not always be easy though. Since it can not always be anticipated what application we will have to write using the framework, it is possible that changes need to be made to the framework. Some changes can have a large impact on our system.

Writing components will not impose any problems. We can easily create different implementations for a component. The real problems occur when we change the text model. If we decide to use a modified text model, or change the interfaces to retrieve information from the text components, chances exist that none of our components will still work, because all components rely on the structure. An example: suppose we introduce an extra structural component in our text: a text can first be split up in parts before it gets split up in chapters. None of our text visitors will work, because there is no *handlePart()* method available to do the right thing. Also, the *handleText()* method will need to be changed too, because *handleText()* might iterate over its subcomponents, which are now parts instead of chapters. The same problem can happen in the pipeline model.

We can conclude that both systems are flexible with a fixed text model. When the text model changes, our entire repository of off-the-shelf components breaks. This is obvious because the text will always be the center of our applications. A careful design of the text model is therefore a must.

## 4 Concurrent and Distributed Systems

### 4.1 Concurrency and Distribution possibilities in the models

Text processing systems perform very computational intensive tasks. Hence they would benefit from a concurrent and distributed environment. Concurrency introduces the notion of simultaneity. A concurrent system can e.g. run on a multi-processor computer system, spreading its tasks over the available processors, which then execute these tasks simultaneously. A distributed system runs on multiple computers in a network. These computers can share tasks in order to distribute the workload over several computers in the network. Above all, network environments are usually a lot cheaper than multi-processor systems. Distributed and concurrent systems usually benefit from a performance increase because of the increased computing power that is available, but also introduce network failure and synchronization problems.

The two points of view presented in this paper can easily be mapped on such environments. The processing point of view can for instance be mapped on several computers in a network, by putting every pipeline step as well as the pipeline monitor on a different computer. Coordination between the steps and the monitor happens by using the network. One pipeline step can even be distributed over several computers when necessary.

Concurrent behavior can occur within one pipeline step by making a chart parser parse several sentences at the same time. It can also occur between several pipeline steps. In a translation system, when the words for the first sentence are looked up, the sentence can already be submitted to the parsing step, while the lookup component continues looking up words for the second sentence. When the lookup component is a lot faster than the parse component, we can distribute our sentences to several parse components that are scattered over our network and let our pipeline monitor distribute the load over the different parse components. The number of possibilities is endless.

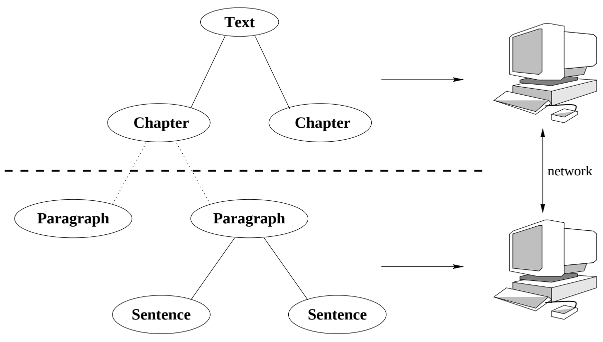


Figure 9: A text distributed over several computers

Mapping the text point of view to a distributed system can be done in a similar way by sending the text to several hosts, each presenting a different text visitor to the text. The processor decides the order in which the text is sent to the several computers. Mapping the text point of view can also be done in another way. We can spread the text structure over different computers. A simple example of this approach can be seen in figure 9. Note that this division of the text over several computers is out of balance, but is only meant as an example. The text is then scattered in several parts over our network. On each computer, text visitors can then handle the text, eventually in a concurrent way. While, for example, a text visitor is doing a morphological lookup on the text and chapter subcomponents, another morphological lookup text visitor can do the same on the paragraph and sentence components that are located on another host.

This concurrent processing can be pretty easy by just instantiating several visitor that are independent of each other. In other cases the visitors will have to be synchronized with each other. E.g. in a text formatting system, it will be difficult to format a sentence when the paragraph has not been formatted yet.

### 4.2 Orthogonal Implementation

Writing NLP systems is already a difficult task and it does not get easier when the system should offer possibilities for concurrent and distributed development. Techniques like multithreading, network programming through sockets, remote method invocation (RMI) and object request brokers (ORB's) make it possible to think about such systems at a higher level. We will not discuss these techniques in detail here. A good introduction to multithreading (in Java) can be found in (Lea 1996). (Eckel 1998) is also good reading when already acquainted with Java. Information on ORB's seen from a component point of view can be found in (Szyperski 1998).

A requirement when creating distributed and concurrent systems is the orthogonality between the functional aspects of the application, being the processing of text, and the non-functional aspects of the application, being the concurrency and distribution. The application programmer of the text processing system should not (or as little as possible) be bothered with the distribution aspects of the framework. For example, When you are building a pipeline architecture, you don't want to be bothered with the deployment characteristics of this architecture: you are not interested in whether every pipeline step will be on one computer or not, whether you will use two parser pipeline steps to concurrently parse your sentences or just one parser step, etc. State of the art techniques to decouple functional aspects from non functional aspects this are Meta level programming or Aspect Oriented programming. (Introductions in (Zimmerman 1996), (Kickzales, Lamping, Mendhekar, Maeda, Lopes, Loingtier and Irwin 1997), (Mens, Lopes, Tekinerdogan and Kiczales 1997))

## 5 Conclusion and Future Work

In this paper we present two base architectures for text processing systems. A requirement for these architectures is that they are flexible in usage and improve component reusability. The first architecture processes text from a process point of view and the second from the text point of view. The process point of view is a pipeline architecture which allows nesting of pipelines to allow more structural processing of texts. The text point of view is based on the Visitor design pattern (Gamma et al. 1995) and therefore inherently enforces structural text processing.

Computational intensive applications can benefit from a concurrent and distributed environment. This has been illustrated and some techniques to enable the two core architectures for such environments have been mentioned.

We are currently designing a framework for an NLP translation system, which is being developed in a project in cooperation with LANT nv. We intend to use one of the two core architectures presented in this paper. Before the actual design of the framework, we will first evaluate the two systems. The approach we take is as follows: a small application (a word by word translator) will be programmed in Java using both base systems. The text point of view has already been programmed and works properly. The pipeline approach still needs to be programmed.

After both approaches have been evaluated, the actual framework using one of the approaches will be designed by modeling a more realistic text model and integrating this text structure in the system. Next step is integrating a morphological lookup component (Stefan Raeymaekers) in the system. After successfully completing this step, a translation system will be incrementally (i.e. componentwise) be designed and implemented to the framework architecture.

Next to the creation of the framework, research will be done on the orthogonal mapping of the framework on a concurrent and distributed system, using the techniques mentioned in section 4.2.

## References

Eckel, B.(1998), *Thinking in Java*, Prentice Hall, chapter 14–15, pp. 519–716.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J.(1995), *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley.

Kickzales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.(1997), Aspect-oriented programming, *in* M. Aksit and S. Matsuoka (eds), *Ecoop'97 Proceedings - Object-Oriented Programming*, Springer-Verlag, pp. 220–242.

Lea, D.(1996), *Concurrent Programming in Java*, The Java Series, Addison-Wesley.

Mens, K., Lopes, C., Tekinerdogan, B. and Kiczales, G.(1997), Aspect-oriented programming workshop report, *in* J. Bosch and S. Mitchell (eds), *Ecoop'97 Proceedings - Object-Oriented Technology, Workhop Reader*, Springer-Verlag, pp. 483–496.

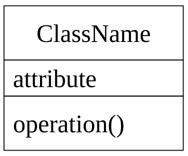
Quatrani, T.(1998), *Visual Modeling with Rational Rose and UML*, Object Technology Series, Addison-Wesley.

Szyperski, C.(1998), *Component Software*, Addison-Wesley, chapter 12–19.

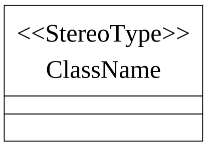
Zimmerman, C. (ed.)(1996), *Advances in Object-Oriented Metalevel Architectures and Reflection*, CRC Press.

## A UML

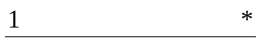
This appendix contains a short reference to the UML notation. For a more complete but short explanation of the UML we refer to (Quatrani 1998).



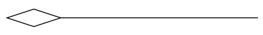
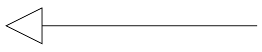
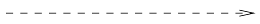
UML Class



Stereotyped Class



An association relationship that has left multiplicity *one* and right multiplicity *many*. An object of the left hand side class can be associated with *many* object of the right hand side class. An object of the right hand side class can be associated with *one* object of the left hand side class.

| | |
|---|---|
|  | An aggregation relationship. An object of the left hand side class of the relationship contains a number of objects of the right hand side class |
|  | A generalization, specialization relationship. An object of the left hand side class is a generalization of an object of the right hand side class |
|  | A dependency or instantiates relationship. An object of the left hand side class depends on or instantiates an object of the right hand side class |