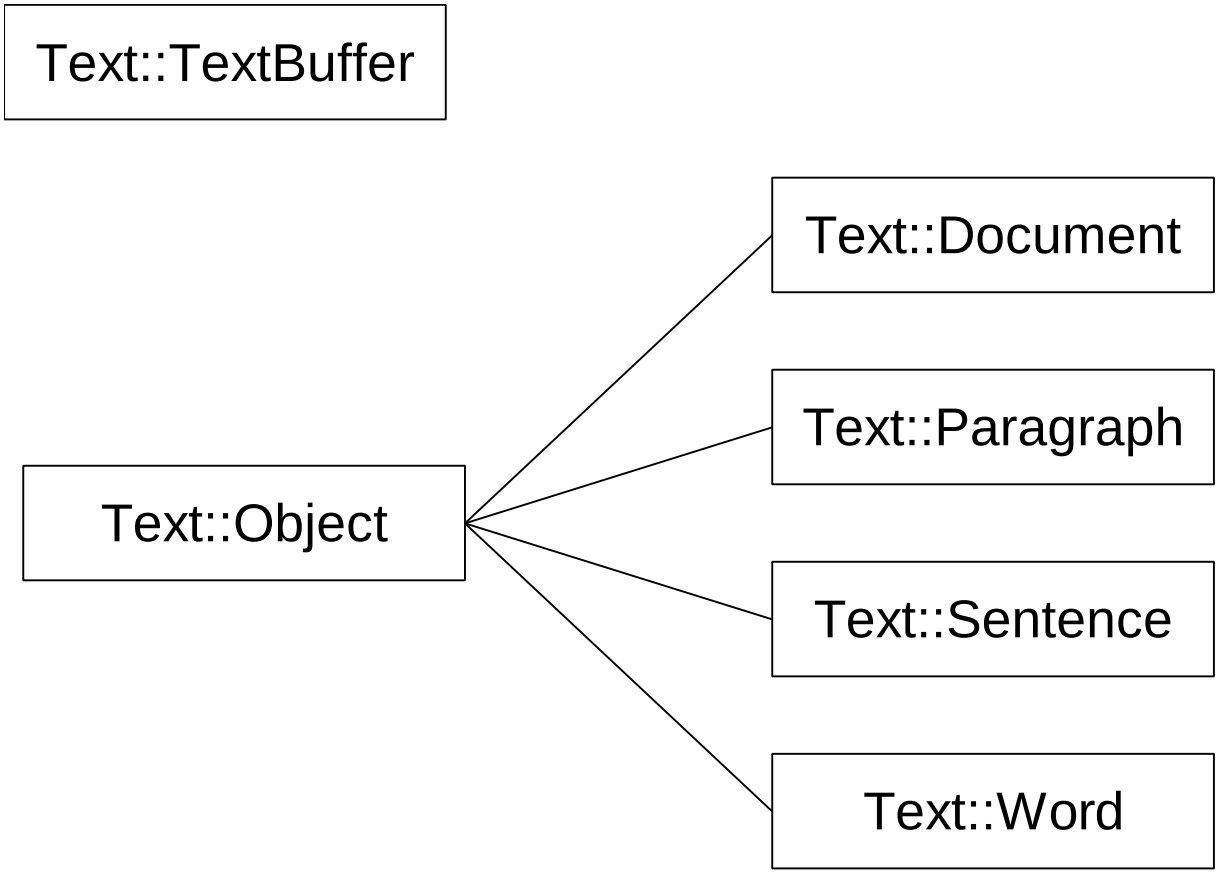
1.Beispiel 8 - Textviewer

1.1. Lösungsansatz

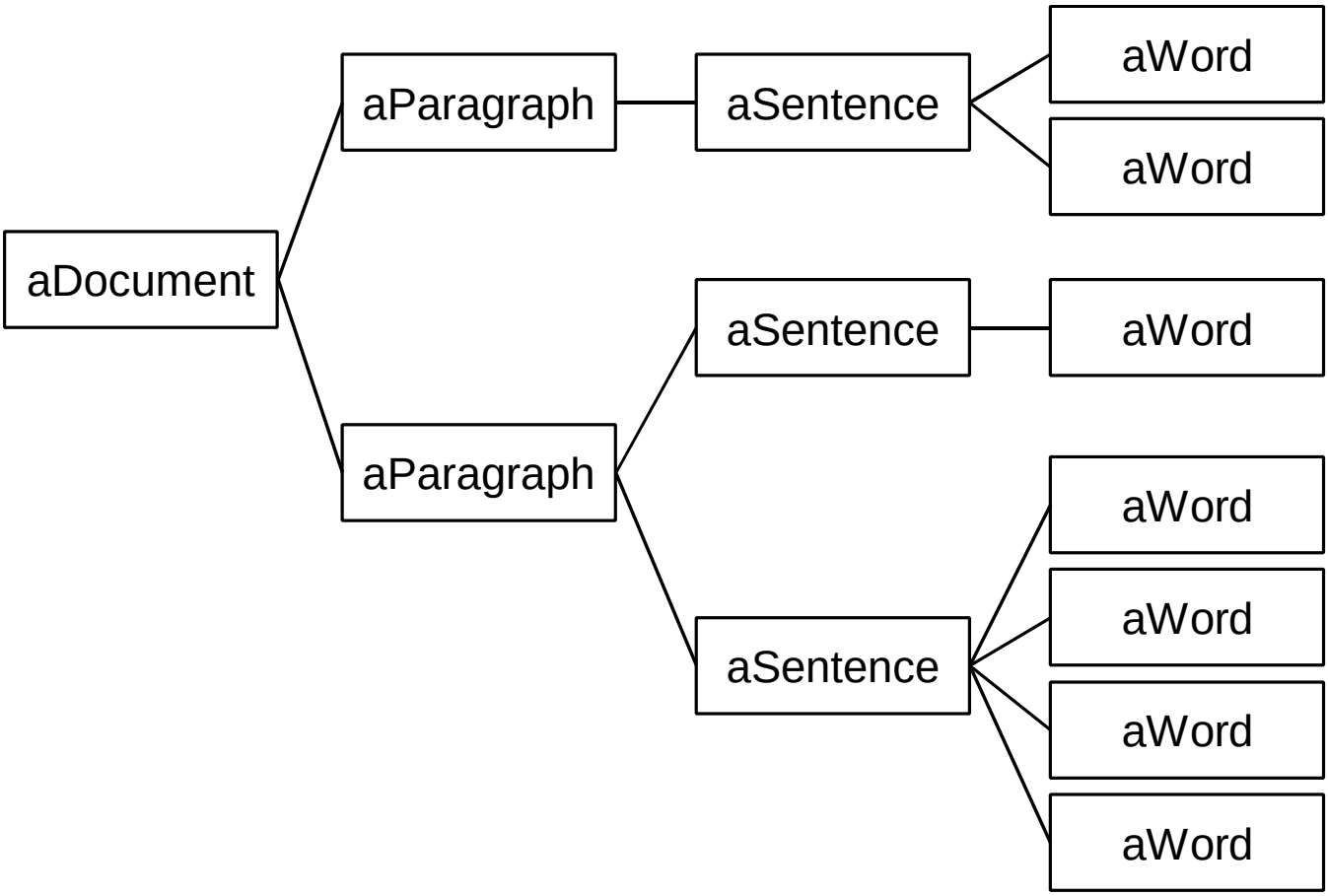
1. Analyse der Kommandozeilenparameter
2. Einlesen der Datei und gleichzeitig Aufbau der internen Datenstruktur
3. Generieren eines aus Zeilen bestehenden Textpuffers entsprechend Kommandozeilenparameter (z.B. kein spezieller Parameter für normale Ausgabe und „S“ für Satzweise Ausgabe) aus der Datenstruktur
4. Seitenweise Ausgabe des Textpuffers.
   (Es ist nicht notwendig den Bildschirm zu löschen, wenn einfach 25 Zeilen ausgegeben werden. Die letzte Zeile kann statt Text einen Hinweis enthalten, welche Tasten zum Blättern und welche zum Verlassen des Programms dienen. Die Eingabe kann einfach über `cin` eingelesen werden.)

1.2. Klassenhierarchie

Die Klassen sind im Modul „Text“ in einem gleichnamigen Namespace deklariert.



1.3. Datenstruktur zur Laufzeit



1.4. Implementierungshinweise

Einlesen *aller* Zeichen aus einer Datei mittels ifstream, z.B.:

```
ifstream file(filename,ios::in);
char ch;

if (file) { // Nur wenn im Konstruktor die
            // Datei geöffnet werden kann,
            // wird ein Objekt erzeugt.

  file.read(&ch,1); // alle Zeichen
  file >> ch;       // Leer- und Steuer-
                    zeichen werden überlesen
```

Die Methoden zum Lesen aus dem Stream der Eingabedatei und zum Schreiben in den Textpuffer müssen in der abstrakten Basisklasse Text::Object als `virtual` deklariert werden, damit diese Methoden beim Arbeiten mit Listen von Objekten richtig funktionieren, wenn die Objekte in der Liste den statischen Typ „Zeiger auf Basisklasse“ haben, tatsächlich aber auf ein Objekt einer abgeleiteten Klasse zeigen.

Aus dem gleichen Grund muß in der Basisklasse ein `virtual` Destruktor deklariert werden, sogar wenn er leer bleibt.

Der gesamte Text kann in der Implementierung der Methode zum Lesen aus dem Stream der Eingabedatei in der Klasse Text::Word gelesen werden. Die Methode muß dem Aufrufer nur zurückliefern, warum das Lesen eines Wortes beendet wurde (z.B. EOF, Satzende, Absatzende,...). Zeichen, die z.B. das Ende eines Wortes oder Satzes anzeigen, können als Ende eines Wortes gespeichert werden. Zum Beispiel kann ein Exemplar der Klasse Text::Word so den Text „fertig!“ enthalten. Nur CR LF Folgen dürfen nicht gespeichert werden, da sie später an anderer Stelle im Text neu generiert werden sollen.

2.Container-Klassen

2.1. Allgemein

Dienen der generischen Verwaltung von Listen von Objekten.

Objekte, die mittels der Standard-Containerklassen verwaltet werden sollen, müssen folgende Bedingungen erfüllen:

1. Sie müssen „copy-constructible“ sein.

2. Sie müssen „assignable“ sein.

Klassen für Listen: deque, list, queue, stack, vector

Klassen für assoziative Mengen: map, set

Klasse für Bitfolgen: bitset

2.2. Template-Klasse „list“

Schnittstellenspezifikation (gekürzt):

```
template<class T, class A = allocator<T> >
  class list {
    public:
      typedef A allocator_type;
      typedef A::size_type size_type;
      typedef A::difference_type difference_type;
      typedef A::reference reference;
      typedef A::const_reference const_reference;
      typedef A::value_type value_type;
      typedef T0 iterator;
      typedef T1 const_iterator;
      iterator begin();
      iterator end();
      reverse_iterator rbegin();
      reverse_iterator rend();
      void resize(size_type n, T x = T());
      size_type size() const;
      size_type max_size() const;
      bool empty() const;
      A get_allocator() const;
      reference front();
      const_reference front() const;
      reference back();
      const_reference back() const;
      void push_front(const T& x);
      void pop_front();
      void push_back(const T& x);
      void pop_back();
      void assign(const_iterator first, const_iterator last);
      void assign(size_type n, const T& x = T());
      iterator insert(iterator it, const T& x = T());
      void insert(iterator it, size_type n, const T& x);
      void insert(iterator it, const_iterator first,
                  const_iterator last);
```

```
void insert(iterator it, const T *first, const T *last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
void clear();
void swap(list x);
void splice(iterator it, list& x);
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x,
            iterator first, iterator last);
void remove(const T& x);
void remove_if(binder2nd<not_equal_to<T> > pr);
void unique();
void unique(not_equal_to<T> pr);
void merge(list& x);
void merge(list& x, greater<T> pr);
void sort();
template<class Pred> void sort(greater<T> pr);
void reverse(); protected: A allocator; };
```

2.3. Wichtige Methoden

2.3.1. push_back

```
void push_back(const T& x);
```

Ein Element am Ende der Liste anhängen.

2.3.2. begin

```
iterator begin();
```

Diese Funktion gibt einen Iterator zurück, der auf das erste Listenelement verweist.

2.3.3. end

```
iterator end();
```

Diese Funktion gibt einen Iterator zurück, der nach dem letzten Listenelement verweist. Deshalb darf dieser Iterator nicht dereferenziert werden (außer man verwendet „--“, um auf das letzte Element zurückzukommen). Diese Funktion wird häufig verwendet, um ein Abbruchkriterium zu erhalten.

2.4. Iteratoren

Ermöglichen den Zugriff auf Listenelemente entsprechend der Listenreihenfolge.
Die passende Iteratorklasse zu einem Exemplar einer Listenklasse ist als Teil der Listenklasse verfügbar. Zum Beispiel:

```
typedef std::list<char*>StringList;

StringList myList;
StringList::iterator myIterator;
```

Der Zugriff auf das aktuelle Element erfolgt über den Dereferenzierungs-operator, z.B.:

```
free (*myIterator);
```

Das Wechseln auf das nächste oder vorhergehende Element erfolgt über den Inkrement- bzw. Dekrementoperator, z.B.:

```
myIterator++;
```

2.5. Beispiel

```
typedef SomeClass* PObject;
typedef std::list<PObject> ObjectList;


ObjectList aList;
ObjectList::iterator iter;

...
```

Die folgende Schleife ruft für alle Elemente der Liste die Methode „AnObjMethod“ auf. Beachten Sie: die Dereferenzierung in der Klammer dient dazu, um vom Iterator auf das Listenelement zu kommen. Das Listenelement selbst ist ein Zeiger auf eine Klasse, die Methode muß deshalb mit „->“ statt mit „.“ aufgerufen werden. Die Klammer ist notwendig, um die korrekte Reihenfolge der Anwendung der Operatoren herbeizuführen.

```
for (iter = paragraphList.begin();
     iter!=paragraphList.end();
     iter++)
{
  (*iter)->AnObjMethod();
}
```