The Telegram Problem Description

A procedure is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device. There exists a buffer of fixed size, into which portions of the input stream can be loaded on demand, and from which characters can be retrieved sequentially. If this buffer becomes empty during execution, it is refilled by transferring the next portion of the input stream into the buffer. Every telegram in the input stream is the sequence of blanks that are separated by words. Every telegram is preceded by the reserved word "ZZZZ". The telegram stream is terminated by the occurrence of an empty telegram composed of one or more blanks followed by the delimiter word "ZZZZ". The objective of the procedure is to produce a "clean" listing of each telegram accompanied by the chargeable word count and a message indicating the occurrence of an over length word. A "clean" listing of each telegram that prints in lines of 120 characters where the redundant blanks are deleted. All words in the telegram except STOP and ZZZZ are chargeable, and words of more than twelve characters are considered over length.

The documentation that follows represents the development of an object-oriented model of the above description rendered in UML-2. As with most OO models this version represents an evolving draft that would continue to be refined depending on its eventual purpose: overview, analysis, design, or implementation. The goal of modeling is to reach "a useful model." It is not possible to define a "perfect" or "correct" model. A model is an evolving understanding documented by the modeling stakeholders. The base documentation which is adapted and reformatted here was generated by Together Version 6.2™, Borland, Inc.
Class Diagrams

UML version of Telegram problem generated using Together 6.2 by Borland, Inc. This is draft of the model in progress on its way to being "useful." LJW

Class Diagrams

diagram <default>

Interaction Diagrams

diagram Print Telegram
diagram Telegram Building

Classes

class Blank
class Buffer
class Character
class Dictionary
class Digit
class Letter
class Line
class Stream
class Telegram
class TelegramStream
class Word
class WordCollection
Class Detail

Class Blank

Character

| ---Blank

public class Blank

Extends: Character

A special kind of character representing "white space" in a line.

Service Summary

| public boolean isBlank() Always returns true.

Class Buffer

public class Buffer

The is the physical area where characters from the input stream are held until they are needed for the construction of words. This is a fixed capacity resource.

Field Summary

| private int count The number of "unused" characters in the buffer.

Service Summary

| private void fill() This behavior successively requests characters from the Stream object until the maximum number of characters have been acquired.

| public Character getChar() Extracts and returns a single character from the collection.

| public boolean isEmpty() This behavior returns false if the count > 0 else it returns true.

Class Character

public class Character

Any machine representable character accessible to the telegram problem.

Field Summary

| private int value The internal machine value of the character.

Service Summary

| public boolean isBlank() Always returns false.

| public boolean isDigit() Always returns false.

| public boolean isLetter() Always returns false.

| public void printChar(int value) Print the visible version of this character to the "printing device."

Class Dictionary

WordCollection

| ---Dictionary

public class Dictionary

Extends: WordCollection

A semi-permanent collection of words with particular significance to the operation of the telegram handling in this model. A place for storing "reserved words."

Field Summary

| private Word lnkWord A dictionary contains ordered collection of words.

Service Summary

| public addReserved(Word NewWord) This behavior piggybacks on the parent addWord behavior to add a reserved word to the dictionary for later comparison.
**Class Digit**

Character

| public class Digit Extends: Character | 

A special kind of character representing numeric values.

**Service Summary**

| public boolean isDigit() | Always returns true. | 

**Class Letter**

Character

| public class Letter Extends: Character | 

A special kind of character that is used for constructing words in this problem.

**Service Summary**

| public boolean isLetter() | Always returns true. | 

**Class Line**

public class Line

The container for characters that will be printed on a single line of output ended by an end of line.

**Field Summary**

| private int length | This attribute is used to count the characters on the current output line. | 

**Service Summary**

| public boolean doesItFit(Word aWord) |
| public void lineFeed() | Generate linefeed on output device and resets the current line character count to zero. |
| public void printWord(Word aWord) | This behavior checks to see if the length of the word to be printed is <= the line limit minus the current number of characters on this line. |

**Class Stream**

public class Stream

**Field Summary**

| public character getChar() | This external device returns a single character using its native I/O capability. | 

**Service Summary**

| public void build() | This behavior creates an empty word and sends it a message to build itself. |
| public void print() | Successively send one member of the collection of words to the Line object for printing. |

**Class Telegram**

public class Telegram

WordCollection

| public class Telegram Extends: WordCollection | 

The model representation of a telegram containing and ordered collection of words.

**Field Summary**

| private int chargeableCount | The number of words to be charged for this telegram. |
| private Word lnkWord |
| private boolean overLengthWord | This flag indicates that at least one word over the prescribed letter count limit occurs in this telegram. |

©Les Waguespack, Ph.D. 2019
Class TelegramStream

public class TelegramStream
The logical aggregation of all telegrams that will pass through this model.

Field Summary

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telegram</td>
<td>private</td>
</tr>
<tr>
<td>lnkTelegram</td>
<td></td>
</tr>
</tbody>
</table>

Service Summary

<table>
<thead>
<tr>
<th>Service</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buildAll()</td>
<td>public void</td>
<td>This behavior creates &quot;empty&quot; telegrams and sends them the message to &quot;build&quot; themselves.</td>
</tr>
<tr>
<td>printAll()</td>
<td>public void</td>
<td>This iterates through each of the previous &quot;built&quot; telegrams and sends them to the printing device.</td>
</tr>
</tbody>
</table>

Class Word

public class Word
An ordered collection of characters that make up a "countable" word in this model.

Field Summary

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>private</td>
</tr>
<tr>
<td>lnkCharacter</td>
<td></td>
</tr>
</tbody>
</table>

Service Summary

<table>
<thead>
<tr>
<th>Service</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addWord()</td>
<td>public void</td>
<td>Add a word object to the ordered collection whole.</td>
</tr>
<tr>
<td>findWord()</td>
<td>public boolean</td>
<td>Searches through the collection of words looking for a match to the target word object sent as a parameter.</td>
</tr>
<tr>
<td>sortWords()</td>
<td>public void</td>
<td>Orders the collection of word objects according to their alphabetic sorting sequence (ascending/descending based on the &quot;ordering&quot; parameter).</td>
</tr>
</tbody>
</table>

Method Detail

addWord

public void addWord(Word target)
Add a word object to the ordered collection whole.

findWord

public boolean findWord(Word atarget)
Searches through the collection of words looking for a match to the target word object sent as a parameter. Returns true if found.

sortWords

public void sortWords(Boolean ordering)
Orders the collection of word objects according to their alphabetic sorting sequence (ascending/descending based on the "ordering" parameter).
Interaction Diagrams

Sequence Diagram *Print Telegram*

This sequence diagram indicates the interaction of actions to print a "clean" telegram on the output device.

**Object Detail**

**Object aCharacter**
A character knows how to print itself, perhaps on another "device."

**Instantiates:**
Character

**Object aLine**
A line is responsible for maintaining the "clean" listing. It checks whether each word sent by the telegram will fit on the current line. If not a linefeed is generated and the word is output to the "next" line. A blank is added after the word if there is room.

**Instantiates:**
Line

**Message Detail**

to **Object aWord**

**Documentation:**
Line asks the word how many characters it has.

**Service:**
Word.printWord(aWord)

to **Object aLine**

**Synchronization:**
call
Object aTelegram
A telegram prints itself by sending words to the line.
Instantiates:
Telegram

Object aLine
Service:
Line.lineFeed()

Service:
Line.doesItFit(word)
to Object aLine

Message Detail
to Object aLine
Documentation:
Telegram sends words to the line one at a time.

Service:
Line.printWord(Word)
Iteration:
for each word

Object aWord
The word "knows" its length and responds to the line's inquiry.

Instantiates:
Word
Message Detail
to Object aCharacter
Documentation:
The word instructs each of its characters to print itself.

Service:
Character.printChar(int)
Iteration:
for each character

USE Case Diagrams
UseCase Name of Use Case
Prose description of Use Case Purpose

system name

use case

use case

Relationship(s)

Actor 1
Actor 2

System Boundary

Use case: Name of Use Case
actors: Actor 1, Actor 2, . . .

Preconditions: . . . Conditions that must exist in order for the Use Case's actions to be meaningful

Primary Scenario: normalFlow: sequence of steps including iteration or selection that order the actions in the Use Case
1. Actions in the "real world" that occurs 1st
2. Actions in the "real world" that occurs next.
3. Actions in the "real world" that occurs next.
   . . .

Secondary Scenarios: alternateFlow: sequence of steps including iteration or selection that order the actions in the Use Case that would account for conditions considered outside the "normal" flow
5a. The basket is empty
6a. The customer id is not found
   . . .

Postconditions: . . . Conditions that must exist when the actions of the Use Case have completed (either successfully or not) that has changed the attributes or relationships in the involved objects
**REQUIRED FORMAT SUMMARY:**

All class, use case, and sequence diagrams should be “computer drawn” (Visio™, SmartDraw™, PowerPoint, draw.io, etc.)

Descriptions should be word processed.

You must submit your modeling package as a single PDF file formatted for legibility and efficient analysis.

Organize your package as follows:

a) Class diagram, class diagram descriptions ordered alphabetically by class name with respective data attributes, services, and relationships under each class description,

b) Sequence Diagrams, with descriptions of each numbered message including purpose and outcomes, and

c) Use Case diagram and prose.

You may submit drafts of the phase packages early and often for preliminary feedback as faculty availability allows. Page headers with name, section #, page #, date!