AGILE

Automatic Generation of Instructions in Languages of Eastern Europe

---

Title ***Preliminary model of the CAD/CAM domain***

Authors Richard Power

Deliverable *MODL1*
Status *Final*
Availability *Public*
Date *June 1998*

INCO COPERNICUS PL961104

Abstract:

A domain model is a set of concepts for representing knowledge in a specific domain — in this case, the domain of CAD/CAM applications. This set of concepts is sometimes called a terminology, or "T-box". Using the concepts in the T-box, a model of the content of a specific text can be constructed: such a model is called an "A-box". To generate texts with the Agile system, the author builds an A-box, using the interface developed in WP1, then calls the generator, which consults the A-box as it plans the general structure and the detailed wording of the text. To allow the interface and the generator to edit and consult the A-box, we have developed a domain model API (Application Programmer's Interface). The API also allows the definition of the T-box, which includes a set of concepts specific to the CAD/CAM domain; these concepts are linked to an "Upper Model"' (also part of the T-box), which represents the abstract semantic concepts that are expressed by the syntactic features of natural languages.

More information on AGILE is available on the project web page and from the project coordinators:

URL:	http://www.itri.brighton.ac.uk/projects/agile
email:	agile-coord@itri.bton.ac.uk
telephone:	+44-1273-642900
fax:	+44-1273-642908

Contents

1. Introduction

The distinctive feature of language generation, as compared with machine translation, is that the author encodes the meaning in a formal language rather than a natural language. This formal language should be precise, so that no problems of ambiguity arise. For instance, a word like "button", which can have several meanings (a mouse button, a button in a dialogue box, not to mention buttons on clothing) will be associated with several distinct concepts in the formal language, one for each of its senses.

The purpose of WP2 is to provide a formal language for modelling instructions for using CAD/CAM applications, along with some programs that allow models to be constructed, edited, and consulted. The core of the domain model is the "T-box", which defines the concepts of the formal language, and thus determines the range of meanings that can be expressed. The T-box has two parts, one general and one specific. The general part is called the "Upper Model'', and includes abstract notions like "quality" and "process" which are linked to syntactic features of natural languages (roughly, adjective and verb). Since these concepts are abstract and universal, they would be applicable to any language and any subject-matter. The specific part comprises a set of concepts for modelling CAD/CAM applications (e.g. "multiline'', "justification''); by linking these domain-specific concepts to the upper model, we encode constraints on how they may be expressed linguistically.

Most of the Agile T-box has been drawn from two existing resources: the Penman Upper Model (Bateman et al, 1990), and the Drafter concepts for modelling instructions (Paris et al, 1995). To these we have added some concepts specifically relevant to the CAD/CAM domain. Our initial aim has been to cover the content of two target texts, which are reproduced in the Appendix.

For reasons of simplicity and efficiency, we have implemented the domain model in the Common Lisp Object System CLOS (Keene, 1989), instead of LOOM (MacGregor et al, 1987), which was used in Drafter. The following code is now available:

- An Application Programmer's Interface (API), which provides a set of LISP functions for defining concepts in the T-box, building an A-box, and consulting both the T-box and the A-box.
- A version of the Upper Model, defined through the API. This is the first part of the T-box.
- Definitions of concepts in the CAD/CAM domain. This is the second part of the T-box; it includes some more general concepts for defining instructions, which are adapted from the Drafter domain model.
- Some utilities for printing out T-boxes and A-boxes in a readable form (either as indented trees or as feature-structure matrices).

2. Domain Model API

In the Drafter project the domain model was implemented in LOOM. Since it turned out that Drafter required only a fraction of LOOM's facilities, we decided that for Agile it would be simpler to implement the domain model in CLOS (the Common Lisp Object System).

2.1 Summary of main features

- As in LOOM, we distinguish "T-box" from "A-box". The T-box provides the concepts with which knowledge is defined; the A-box is a set of assertions, employing these concepts, which represents the meaning of the generated text.
- As in LOOM, the T-box is a set of concepts organized in a hierarchy by the "superconcept" relation. We allow a concept to have several superconcepts, thus supporting multiple inheritance.
- On each concept, slots may be defined. An instance of a concept will inherit the slot specifications of its superconcepts. Slots may be filled either by instances of concepts, or by standard LISP data objects such as integers or strings.
- Our slot specifications differ from those in LOOM in three ways:
    1. We omit quantifiers (e.g. ":the" or ":all").
    2. We disallow multi-valued slots. To achieve the same result, the value of the slot may be a single list object.
    3. We classify each slot either as obligatory or optional. This allows a precise definition of whether an A-box is potentially complete: a complete model is one in which all obligatory slots are filled.
- A slot specification on a concept may be over-ridden by a specification with the same slot name on a subconcept. The two specifications may differ with regard to both type and optionality.
- As in LOOM, a concept can contain a statement that some of its subconcepts are disjoint (i.e. mutually exclusive). However, we do not check that if A is a subconcept of B, B is among the direct superconcepts of A.

7. Unlike LOOM, we do not require that each instance has a name. If the domain requires that an instance is named, then a "name" slot should be defined on the relevant concept.

2.2 List of API functions

2.2.1 Defining a concept

```
(DEFINE-CONCEPT name (superconcept1 superconcept2 ...)
  ((slotname1 :TYPE concept1 :OPTIONAL T)
   (slotname2 :TYPE concept2 :OPTIONAL NIL)
   ...)
  :DISJOINT-SUBCONCEPTS ((subconcept11 subconcept12 ...)
                         (subconcept21 subconcept22 ...)
                         ...))
```

Comments:

- All concepts mentioned in the definition (e.g. “superconcept1”) must be defined elsewhere.
- The :TYPE feature indicates the class of possible slot fillers, and must be specified. It can be a concept, or a LISP data type (e.g. integer, string).
- The :OPTIONAL feature may be omitted; it defaults to NIL.
- The :DISJOINT-SUBCONCEPTS feature holds a list of lists of concept names. The concepts within each sublist are mutually exclusive subconcepts of the concept under definition. The feature may be omitted, in which case it defaults to NIL.
- Except for the root of the upper model, all concepts should have at least one superconcept.

Examples of concept definitions:

```
(define-concept PROCEDURE (INSTRUCTION-SCHEME)
    ((GOAL :type USER-ACTION)
     (METHOD* :type METHOD* :optional T)))

  (define-concept LABELLED-OBJECT (CADCAM-OBJECT)
    ((LABEL :type STRING)))

  (define-concept SCREEN-OBJECT (LABELLED-OBJECT DISPLAY-OBJECT) NIL)

  (define-concept BUTTON (SCREEN-OBJECT SIMPLE-OBJECT) NIL)
```

Comments:

- A “procedure” has a goal and a method. An instance of this concept could be the root of the A-box. The “method” slot is optional because to avoid an infinite regress we must eventually arrive at sub-procedures for which no method is defined.
- The concept “labelled-object” covers things like the Save button, the File menu, the AutoCAD program.
- “Screen-object’’ is a labelled object that appears on the screen; hence it does not cover (for example) “the AutoCAD program’’. Since “button” is a screen object, which is in turn a labelled object, the slot “label” is inherited and need not be defined on “button.
- All the other concepts mentioned in these definitions (e.g. “user-action” would have to defined elsewhere, except for “string” which is a LISP data type.

2.2.2 Creating an instance

```
(CREATE-INSTANCE concept-name) --> instance
```

Creates and returns an instance of the concept with the specified name, which should be a quoted symbol. Typical usage:

```
(setq button-instance (create-instance 'button))
```

2.2.3 Deleting an instance

```
(DELETE-INSTANCE instance) --> T
```

Removes the instance from the A-box. All slots previously filled by the instance become unbound. Example (deleting the instance created above):

```
(delete-instance button-instance)
```

2.2.4 Checking whether something is an instance

```
(INSTANCE-P item) --> T/NIL
```

T if "item" is an instance of a concept, otherwise NIL.

Implementation note: The root of the T-box descends from a class called "agile-object" which has housekeeping slots "uid" (unique identifier) and "contexts". When an instance is created, a new symbol is generated as its "uid" value; this is used when saving the A-box. The "contexts" value is initially nil, but every time the instance becomes the filler of a slot, a sublist of the form (instance slot-name) is added to this list. When an instance is deleted, the list of contexts is used in order to reset to unbound all slots which the instance previously filled.

2.2.5 Assigning a slot value

```
(SET-INSTANCE-SLOT-VALUE instance slot-name value) --> T
```

Assigns a value to the named slot of "instance". We do not check whether the value is of the appropriate type. Mistakes in the first two arguments (instance, slot-name) will result in CLOS errors. Examples:

```
(set-instance-slot-value save-plan-instance 'substeps
  (list open-window-instance enter-filename-instance click-button-
instance))

(set-instance-slot-value open-window-instance 'actee window-instance)

(set-instance-slot-value window-instance 'name "Save As")
```

2.2.6 Reading a slot value

```
(GET-INSTANCE-SLOT-VALUE instance slot-name) --> value
```

Returns the value of the named slot. A CLOS error results if the slot is unbound. Typical usage:

```
(setq actee-instance (get-instance-slot-value open-instance 'actee))
```

2.2.7 Testing whether a slot is filled

```
(SLOT-FILLED-P instance slot-name)
```

T if the slot has a value, NIL if it is unbound.

2.2.8 Getting all instances of a concept

```
(GET-INSTANCES-FROM-CONCEPT concept-name) --> list-of-instances
```

The list includes instances of subconcepts of the named concept.

2.2.9 Getting the concept name of an instance

```
(GET-CONCEPT-FROM-INSTANCE instance) --> concept-name
```

The concept name is the symbol given as argument to CREATE-INSTANCE when the instance was created. Example:

```
(setq button-instance (create-instance 'button))

(setq concept-name (get-concept-from-instance button-instance))
```

The value of "concept-name" will now be the symbol 'button.

2.2.10 Checking whether something is a concept

```
(CONCEPT-P item) --> T/NIL
```

T if "item" is the name of a concept in the T-box, otherwise NIL. Examples:

```
(define-concept button (named-object nondecomposable-object) ())

(concept-p 'button) ;;; evaluates to T
```

2.2.11 Recovering the slot descriptions of a concept

```
(GET-CONCEPT-SLOT-DESCRIPTIONS concept-name) --> list-of-slot-
descriptions
```

Returns descriptions of all slots on the named concept, except for the housekeeping slots "uid" and "contexts". The list includes descriptions of inherited slots. If a slot on a superconcept is overridden on a more specific concept, only the slot on the most specific concept is described.

Each slot description is a sublist containing three items:

- the slot name (a symbol)
- the value restriction (a concept name, also a symbol)
- the optionality status (either :optional or :obligatory).

Example:

```
(define-concept dispositive-material-action (directed-action)
  ((actee :type object)))

(define-concept enter (dispositive-material-action)
  ((location :type spatial :optional T)))

(get-concept-slot-descriptions 'enter)

;;; Evaluates to ((location spatial :optional)(actee object
:obligatory))
```

2.2.12 Getting the subconcepts of a concept

```
(GET-CONCEPT-SUBCONCEPTS concept-name &key direct) --> list-of-
subconcepts
```

If ":direct T" is specified, returns a list of the names of immediate subconcepts; otherwise, returns a list of the names of all subconcepts. Example:

```
(get-concept-subconcepts 'directed-action)

;;; Might evaluate to (dispositive-material-action save enter)
```

```
(get-concept-subconcepts 'directed-action :direct T)

;;; Would evaluate to (save enter), assuming these descend from
;;; dispositive-material-action
```

2.2.13 Getting the superconcepts of a concept

```
(GET-CONCEPT-SUPERCONCEPTS concept-name &key direct) --> list-of-
superconcepts
```

If ":direct T" is specified, returns a list of the names of immediate superconcepts; otherwise, returns a list of the names of all superconcepts. Example:

```
(get-concept-superconcepts 'enter)

;;; Might evaluate to (dispositive-material-action directed-action
process thing)

(get-concept-superconcepts 'enter :direct T)

;;; Would evaluate to (dispositive-material-action)
```

2.2.14 Getting the disjoint subconcepts of a concept

```
 (GET-CONCEPT-DISJOINT-SUBCONCEPTS concept-name) --> list-of-lists-of-
subconcepts
```

Returns the list given as the value of the :disjoint-subconcepts key in the definition of the named concept, or NIL if this key was omitted. Example:

```
(define-concept object (thing)
  ()
  :disjoint-subconcepts ((non-conscious-thing conscious-being)
                         (nondecomposable-object decomposable-object)))

(get-concept-disjoint-subconcepts 'object)

;;; Evaluates to ((non-conscious-thing conscious-being)
;;;               (nondecomposable-object decomposable-object))
```

2.2.15 Getting all instances in the current A-box

```
(GET-INSTANCES) --> list-of-instances
```

2.2.16 Clearing the current A-box

```
(CLEAR-INSTANCES)  --> T
```

2.2.17 Saving the current A-box

```
(SAVE-INSTANCES pathname) --> T
```

The instances in the current A-box are saved in the named file. Example:

```
(save-instances "/research/cl/agile/text1.lisp")
```

2.2.18 Loading a saved A-box

```
(LOAD-INSTANCES pathname) --> T
```

An A-box is loaded from the named file. Example:

```
(load-instances "/research/cl/agile/text1.lisp")
```

3. CADCAM Domain Model

3.1 Modelling objects

Our starting point was the following hierarchy based on Drafter. Sub-concepts are indented, and attributes are shown in brackets.

```
CADCAM-OBJECT
  DATA-OBJECT
    DRAWING
    FILE
    DOCUMENT
  COMPONENT-OBJECT (OWNER)
    DATA-NAME (OWNER)
  LABELLED-OBJECT (LABEL)
    SCREEN-OBJECT (LABEL)
      WINDOW (LABEL)
      SCREEN-LIST (LABEL)
      MENU (LABEL)
      FIELD (LABEL)
      DIALOGUE-BOX (LABEL)
      OPTION (LABEL)
      ICON (LABEL)
      BUTTON (LABEL)
    SOFTWARE-TOOL (LABEL)
      PROGRAM (LABEL)
      SOFTWARE-COMMAND (LABEL)
```

This hierarchy has three basic patterns:

- The *data object*, with no attribute, yields noun phrases like “the document” or “the drawing”.
- The *component object*, with the OWNER attribute, yields noun phrases like “the name of the document”: here the name belongs to the document, so the document is the owner of the name.
- The *labelled object*, with the LABEL attribute, yields noun phrases like “the Save button” or “the PLINE command”, where the strings “Save” and “PLINE” are labels.

Some further concepts have been added in order to cover the material in the target texts..

3.1.1 Objects in Text 1

the PLINE command

An instance of SOFTWARE-COMMAND with the label “PLINE'.

Polyline

The context is “choosing Polyline from the Polyline flier”, so this is an instance of OPTION with the label “Polyline”.

the Polyline flyout on the Draw toolbar

We need to introduce a new concept FLYOUT, a sub-concept of SCREEN-OBJECT (inheriting the LABEL attribute). However, this concept requires a second attribute, its

location (which in this case is the Draw toolbar). For now we will simply add this attribute to FLYOUT; eventually we will probably want LOCATION to be an optional attribute on a wider class of screen objects.

the Draw toolbar

It suffices to add the concept TOOLBAR as a sub-concept of SCREEN-OBJECT.

the first point of the polyline

This new pattern requires a concept POINT with attributes OWNER and NUMBER. We will classify POINT as a sub-type of COMPONENT-OBJECT so that it inherits the OWNER attribute.

the polyline

With no attributes, this could be classified under DATA-OBJECT. However, we need to distinguish data complexes which can be opened and saved (documents, drawings, etc.) from data elements which can be saved only as part of a data complex. Note that we must distinguish polyline instances from the string instance “Polyline”. The former cannot be the value of a LABEL attribute.

the endpoint of the polyline

Elsewhere we have “the first point”, “the second point”, etc. so the NUMBER attribute must cover various indicators of sequential position. We might add SEQUENTIAL-POSITION as a subconcept of the Upper Model concept SENSE-AND-MEASURE-QUALITY; in its turn, SEQUENTIAL-POSITION would have subconcepts including FIRST, SECOND, LAST.

Return

In the context “Press Return” this obviously denotes a key, so we need to introduce KEY as a subconcept of LABELLED-OBJECT.

Here are two A-Box fragments underlying the phrases “the Polyline flyout on the Draw toolbar” and “the endpoint of the polyline”. They are shown as feature-structure matrices with concepts in lower case and attributes in capital letters.

```
flyout
LABEL: "Polyline'
LOCATION: toolbar
          LABEL: "Draw'

point
NUMBER: last
OWNER: polyline
```

3.1.2 Objects in text 2

Since there is much overlap with Text 1, we only mentioned noun phrases that introduce new patterns.

st

In the context “Enter st” this denotes a command typed in by the user. Since we already

have SOFTWARE-COMMAND, we need a new concept COMMAND-LINE classified under SCREEN-OBJECT (inheriting the attribute LABEL). So the ACTEE of ENTER is an instance of COMMAND-LINE having the string “st” as the value of its LABEL attribute.

the prompt

Introduces PROMPT, which must be classified as a screen object without attributes. Perhaps POLYLINE and MULTILINE could be placed in the same category.

a style

In the context “to select a style” this denotes a style specification; later we encounter specifications of justification and scale.

the style list

This means something like “the list of style specifications”. To avoid problems of modelling notions like “all the current style specifications”, we could simply introduce STYLE-LIST as a screen object without attributes.

a justification

Along with “a style” and “a scale” this denotes a specification. So we could have SPECIFICATION (no attributes) with subconcepts STYLE-SPECIFICATION, JUSTIFICATION-SPECIFICATION etc.

the scale of the multiline

This denotes the actual scale, which must be distinguished from the scale specification. SCALE can be classified under COMPONENT-OBJECT so that it inherits the OWNER attribute. MULTILINE is a sister of POLYLINE.

3.1.3 Revised classification of objects

Here is a plan of the revised object hierarchy. As before, subconcepts are indented, and attributes are shown in brackets.

```
CADCAM-OBJECT
  UNIQUE-OBJECT
    DATA-OBJECT
      DRAWING
      FILE
      DOCUMENT
    UNIQUE-SCREEN-OBJECT
      LINE
        MULTILINE
        POLYLINE
      COMMAND-LINE-PROMPT
      STYLE-LIST
    SPECIFICATION
      STYLE-SPECIFICATION
      SCALE-SPECIFICATION
      JUSTIFICATION-SPECIFICATION
    PARAMETER-VALUE
      NUMBER
      JUSTIFICATION-VALUE
```

```
        TOP-JUSTIFICATION
        ZERO-JUSTIFICATION
        BOTTOM-JUSTIFICATION
    COMPONENT-OBJECT (OWNER)
      DATA-NAME (OWNER)
      SCALE (OWNER)
      JUSTIFICATION (OWNER)
      END-POINT (OWNER)
      POINT (OWNER NUMBER)
    LABELLED-OBJECT (LABEL)
      SCREEN-OBJECT (LABEL)
        WINDOW (LABEL)
        SCREEN-LIST (LABEL)
        MENU (LABEL)
        FIELD (LABEL)
        DIALOGUE-BOX (LABEL)
        OPTION (LABEL)
        ICON (LABEL)
        BUTTON (LABEL)
        FLYOUT (LABEL LOCATION)
        TOOLBAR (LABEL)
        COMMAND-LINE (LABEL)
      SOFTWARE-TOOL (LABEL)
        PROGRAM (LABEL)
        SOFTWARE-COMMAND (LABEL)
      KEY (LABEL)
    LIST-OBJECT (FIRST REST)
      JUSTIFICATION-LIST (FIRST REST)
```

3.2 Modelling actions

From Drafter we carried over the following hierarchy of concepts for actions.

```
USER-ACTION (ACTOR ACTEE)
  START (ACTOR ACTEE)
  SELECT-ACTION (ACTEE OPTIONS ACTOR)
    CHOOSE (ACTEE OPTIONS ACTOR)
  LOCATED-ACTION (ACTEE LOCATION ACTOR)
    ENTER (ACTEE LOCATION ACTOR)
    DOUBLE-CLICK (ACTEE LOCATION ACTOR)
    CLICK (ACTEE LOCATION ACTOR)
  DATA-ACTION (ACTOR ACTEE)
    EDIT (ACTOR ACTEE)
    PRINT* (ACTOR ACTEE)
    SAVE (ACTOR ACTEE)
    OPEN* (ACTOR ACTEE)
```

Again, some further concepts have been added in order to cover the material in the target texts..

3.2.1 Actions in text 1

Start the PLINE command

An instance of START.

choosing Polyline ...

An instance of CHOOSE.

Specify the first point of the polyline

Introduces SPECIFY, which can be classified as a straightforward USER-ACTION (inheriting ACTOR and ACTEE).

Press Return

Introduces PRESS, which also has ACTOR and ACTEE only.

end the polyline

Introduces END, which will be a sister of START.

3.2.2 Actions in Text 2

Enter st at the prompt

An instance of ENTER, with ACTEE and LOCATION specified.

Enter ?

Another instance of ENTER, this time with no LOCATION. The T-Box marks LOCATION as an optional attribute.

select a style

Instance of CHOOSE (so that "select" and "choose" are treated as synonymys). The OPTIONS attribute on CHOOSE is optional.

display the style list

Introduces DISPLAY, which has ACTOR and ACTEE.

justify the polyline

Introduces JUSTIFY, which has ACTOR and ACTEE.

change the scale of the multiline

Introduces CHANGE, which has ACTOR and ACTEE.

Here are A-Box fragments for "enter st at the prompt", "enter a scale", and "change the scale of the multiline". Note that the scale specification is distinguished from the actual scale.

```
enter
ACTOR: user
ACTEE: command-line
       LABEL: "st"
LOCATION: prompt

enter
ACTOR: user
ACTEE: scale-specification

change
ACTOR: user
ACTEE: scale
       OWNER: multiline
```

3.2.3 Revised classification of actions

```
USER-ACTION (ACTOR ACTEE)
  SIMPLE-ACTION (ACTOR ACTEE)
    START-TOOL (ACTOR ACTEE)
    QUIT-TOOL (ACTOR ACTEE)
    START-LINE (ACTOR ACTEE)
    END-LINE (ACTOR ACTEE)
    SPECIFY-COMPONENT (ACTOR ACTEE)
    PRESS (ACTOR ACTEE)
    DISPLAY (ACTOR ACTEE)
    OPEN-SCREEN-OBJECT (ACTOR ACTEE)
    CLOSE-SCREEN-OBJECT (ACTOR ACTEE)
    JUSTIFY (ACTOR ACTEE)
    DRAW (ACTOR ACTEE)
    CHANGE-COMPONENT (ACTOR ACTEE)
  SELECT-ACTION (ACTEE OPTIONS ACTOR)
    CHOOSE (ACTEE OPTIONS ACTOR)
  LOCATED-ACTION (ACTEE LOCATION ACTOR)
    ENTER (ACTEE LOCATION ACTOR)
    DOUBLE-CLICK (ACTEE LOCATION ACTOR)
    CLICK (ACTEE LOCATION ACTOR)
  DATA-ACTION (ACTOR ACTEE)
    EDIT (ACTOR ACTEE)
    PRINT* (ACTOR ACTEE)
    SAVE (ACTOR ACTEE)
    OPEN* (ACTOR ACTEE)
```

4. Conclusion

We have implemented the domain model in CLOS, providing an API for defining and interrogating T-boxes and A-boxes. The upper model has been transcoded to this formalism, and some concepts for representing instructions in the CAD/CAM domain have been added So far the coverage is just sufficient to model the target texts reproduced in the Appendix. During the next phase of the project, the domain model will be maintained regularly to meet any new requirements that arise from developments in text planning or tactical realization; its coverage will also be expanded considerably in order to model more complex texts.

References

Bateman, J, Kasper, R., Moore, J. and Whitney, R.. (1990) A general organization of knowledge for natural language processing: the Penman Upper Model. Technical Report, USC/ISI

Keene, S. (1989) Object-oriented programming in Common Lisp: A programmer's guide to CLOS. Adison-Wesley, Reading: Massachusetts.

MacGregor, R. and Bates, R. (1987) The LOOM knowledge representation language. Proceedings of the Knowledge-Based Systems Workshop, St Louis.

Paris, C., VanderLinden, K., Fischer, M., Hartley, A., Pemberton, L., Power, R. and Scott, D. (1995) A support tool for writing multilingual instructions. Proceedings of the International Joint Conference on Artificial Intelligence, Montreal: Canada.

Appendix

The initial coverage of the CAD/CAM domain model was guided by the following texts, adapted from the AutoCAD user manual.

Text 1

To draw a polyline

1. Start the PLINE command by choosing Polyline from the Polyline flyout on the Draw toolbar.
2. Specify the first point of the polyline
3. Specify the endpoint of the polyline.
4. Press Return to end the polyline.

Text 2a

To draw a multiline

1. Start the MLINE command by choosing Multiline from the Polyline flyout on the Draw toolbar
2. Enter **st** at the prompt to select a style.
3. Enter **?** to display the style list.
4. Enter **j** and choose a justification to justify the multiline.
5. Enter **s** and enter a scale to change the scale of the multiline.
6. Now specify the multiline.
7. Specify the first point of the multiline.
8. Specify the second point of the multiline.
9. Specify the third point of the multiline.
10. Press Return to end the multiline.

Text 2b

To draw a multiline

1. Start the MLINE command by choosing Multiline from the Polyline flyout on the Draw toolbar.
2. Enter **st** at the prompt, then enter **?** to select a style. The AutoCAD Text Window appears. Enter the name of the style.
3. Enter **j**, then enter a justification from top, zero and bottom to change the justification of the multiline.
4. Enter **s**, then enter a number to change the scale of the multiline.
5. Close the AutoCAD Text Window.
6. Specify the first point of the multiline.

7. Specify the second point of the multiline.
8. Specify the third point of the multiline.
9. Press Return to end the multiline.