Documentation of software systems

Jørgen Steensgaard-Madsen

Students are frequently asked to report in writing on assignments. Whether the actual work takes half a day or several weeks, it is important to realise that the quality of reports is very influential on the assessment of efforts.

Major rôles of a report are to provide

> a summary of what has been achieved and what not
> an overview of options and justification for choosing among them
> a roadmap for readers who might have to zoom in on selected details

Technical details should be provided, of course, but often in appendices to emphasise these rôles.

Contents

Documentation of software systems

Jørgen Steensgaard-Madsen

Writing plays an important rôle in the professional life of every engineer. Hence it is important that students write many reports during their study, so that a great deal of experience with technical writings is obtained.

This document focuses on writings related to software systems. One special aspect in this area is that programs can and must be considered as documents in their own right: they will be read very carefully by others than the original author. Readers will want easy access to detailed technical information from the program text, so that modifications can be made quickly and with little risk of creating problems.

Although a program text by itself is insufficient as documentation for a realistic software system, this is where a freshman student should focus first. The section about program texts provides some general advice, independent of what programming language is used. Since other sections address issues that to a large degree constitute a description of a program as an object, it is appropriate to ensure initially that it is well organised.

Acknowlegments

Several people have influenced this document in various ways. Hans Rischel and Robin Sharp have previously written a document which was designed to be short. Some paragraphs in the "Advice on style" have been adopted literally from that. My colleagues Paul Fischer, Anne Haxthausen, Jens Thyge Kristensen, and Hans Henrik Løvengreen, as well as a student Torben Gjaldbæk have contributed with valuable comments.

Programs considered as documents

A program text expresses of course a computation, but its use as a document implies that it has to express other things as well. Some are related to form, in the sense that readers expect to find certain kinds of information there, just as one might expect to find a table of contents in some books. Others arise from experience in how to tell other people about essential ideas in a program. And finally there is an aspect of appearance, which depend on the technology used to present the document to readers.

Form

Any document has at least one author, which the document itself should identify as early as possible. This applies also to program texts, even when it constitutes just one of several parts of a larger program. Thus the basic rule says

A program text starts with an identification of its author

of course included as a *comment* like any other text that helps readers to understand the program.

An experienced author works towards a final product through a sequence of *versions*. As the sequence can be quite long it is wise to identify each version separately, e.g. by using a hierarchical numbering system including perhaps an id like 3.6.4 and a date.

Provide a scheme of version identification

A new version is sometimes generated by an author who has to account for changes, so

Expect supplementary authors to identify themselves in the text

In the extreme one might find it useful to maintain a *history* of changes to the program text, possibly including a short summary of justifications for the changes. Of course one has to execute a reasonable judgement with respect to the counting of versions: ephemeral texts are not to be counted.

Technical writings are not literature in the artistic sense. Hence texts should be prepared for easy reading. Since technical matters require an extraordinary attention to details, proper concern for readability implies that a long text should be broken down into smaller meaningful parts.

Readers can understand just 1/2 – 2/3 a page of program text at a time

so authors should structure their programs accordingly. The coherent pieces of text should be clearly delineated by various graphical means: blank lines, block comments, single line comments, etc. A simple corollary is

Use subroutines with telling names to make complex program parts readable

But a telling name is not enough, so add to every subroutine a comment that can help other programmers to use it appropriately. Of course the same requirements apply to other major parts of a program, whether they are called classes, modules, components, or something else.

Adequate grouping of program text should not be restricted to statements. Variables typically have relations to one another, and if so, their declarations should be written to emphasise it: use blank lines between groups — at least. Variable declarations deserve to be commented, with information about the meaning of each variable name. Often a comment on the same line as the declaration suffices, but for groups of related declarations more extensive comments may be appropriate. Anyway

The meaning of every name in a program should be described in a comment

whether the name denotes a variable or a computation. Programmers tend to excuse themselves for violating this rule, mostly justified by what they consider self evident — and mostly by themselves! Your author is definitely a programmer in this sense, and not proud of it.

Objectives and means

Comments should tell *what* a piece of program achieves, so that the program text tells *how* it is done. This kind of information also helps readers to understand the program lines proper.

The full text of a program can easily span many pages. So can parts of a program that constitutes a logical entity from the programming language point of view. The text of a `class` of a Java program can be quite long if it contains many methods, but language concepts (e.g. inheritance) often make it possible to limit the length of text nevertheless.

Any justifications to write long program texts does not change the fact that human readers can understand only small pieces at a time. Such justifications then also calls for more comments to help human readers to get the overall picture and to help them identify the appropriate subparts that can be understood separately. Thus writing a program text is often a serious challenge of writing clearly for humans about complex matters.

There should be at least one serious, explanatory comment on every page

A program construct that implies repeated computations constitutes a challenge for the author to explain, especially if it involves a number of repetitions that cannot be given as a simple number initially. Typically such repeated computations evolve around some insight that the programmer has achieved, and that insight must be passed on to readers in a comment. The notion of an *invariant* can be a very useful statement of such an insight.

The notion of an invariant is not reserved for programs. A simple example of an invariant provides a strategy for winning (if possible) a very simple game

Play: *Let two players alternately remove pebbles from one of two piles*
Win: *The one to take the last pebble wins*

The invariant that leads to a win is: *leave two piles with the same number of pebbles to the opponent.* The unlucky one who has to remove from piles of equal size may try to delay the pain by removing only a few pebbles in the vain hope that the other makes a mistake.

Several algorithms likewise depend on some kind of invariant, and it should of course be stated in a comment. Since this presentation is not concerned with programming, readers are expected to learn more about this elsewhere. But the little example above provides a nice vehicle for an illustration

```
/* PRECONDITION: a > 0 and b > 0 */
while (a != b)
{ /* INVARIANT: result == greatest common divisor of a and b */
  if (a < b) b = b - a;
  else a = a - b;
}
/* RESULT: result == a (== b) */
```

The invariant is here the key to understand the method used to find the greatest common divisor of two non-negative integers.

Appearance

Many students get trapped by document processing software into believing that program texts should be printed with a font as ordinary text. Not so: it will be easier for programmers to work with texts written with a fixed-spaced font, and programmers are the primary audience for program texts. This illustrates the point

Write program texts for modification, not publication

This applies to long program texts as well as to shorter ones, and do not forget that habits from working with shorter program texts may stay with you as programs become more complex. Long program texts, millions of lines long, will most likely never be printed.

Of course a program text can be prepared so that it prints nicely with special words in boldface etc. but it is probably not worth the author's troubles. A program text should be prepared so that it can be read and modified easily with an editor. This is one essential reason to keep related parts of the text short. Furthermore, programmers may choose an editor that emphasises the structure in the program text using other means than fonts.

A program text should never appear as part of the main text in a technical report. It does so in texts you read when learning about programming, but do not use such a style in a report. Preferably a program text should be a document by itself, but it can be appropriate to put it in an appendix in a report. Fragments of the program text that appear in the main text should be particularly noteworthy — sparkling pearls if you like. Few programmers write that kind of fragments. Reference [1] contains several pearls and also an introduction (in Part 1, Column 4) to invariants. It is an outstanding book, strongly recommended.

A final point about program texts as documents concerns *indentation* i.e. the amount of blank space at the beginning of a line. Program constructs include many variations over the theme of parentheses. The *level of nesting* is the number of opened, but not yet completed, parenthetical constructs at a given point of writing. Adhere to the following rule for indentation of program texts

Indentation should be proportional to the level of nesting

with some trivial reservations, for instance to cater for a parenthesis fully contained on one line.

Here is an illustration of how **not** to write program texts and its recommended form

No good:	**Recommended:**

`...`
`while (a != b)`
`{`
`  if (a < b) b = b - a;`
`else a = a - b;`
`}`
`...`
	`...`
	`while (a != b)`
	`{`
	`  if (a < b) b = b - a;`
	`  else a = a - b;`
	`}`
	`...`

If the indentation rule is violated during program construction, one can use standard programs (e.g. the Unix program `indent`) to impose a standardised indentation. It may not be precisely the author's favourite rules, but to the reader the resulting text will be more pleasing than any that does not follow a set of consistent rules.

This document contains program text in a few appendices, partly to illustrate the points made above. More need to be said about them, so just be aware that they exist for the time being. A later section will focus on techniques that can (and should) be used to convince a reader that a program, or part of a program, works satisfactorily. Like program text such technicalities belong best in appendices, so for the sake of variation, let us have a look on reports in a wider perspective.

Advice on style

One general rule of writing, the *soothe rule,* is fairly obvious, but often violated:

Do not undeliberately displease your readers.

Technical writings should not in any way distract a reader's attention from the technical contents. The rule's mild reservation is due to the fact that humans can violate rules, if they are ready to suffer the consequences.

There are many kinds of writings: novels, scientific articles, applications for funds, PhD theses, newspaper columns, user's guides, letters, technical descriptions, etc. Students are required to work on projects and write reports on their work. One of the reasons for this is that *writing* is hard work, so assignments are not only given for students to learn skills specific for a particular line of study.

The rule emphasises the importance of identifying who the readers may be. The soothe rule can be specialised as detailed in the following.

Do not confuse readers

The *terminology* is the set of terms used to denote concepts pertaining to the software product together with their precise meaning when used in the documentation. Graphical symbols used in figures are also part of the terminology.

A software product should, whenever possible, borrow the terminology of the *application domain* (e.g. the documentation of a text processing system should use the terms "paragraphs" and "sections" if these terms are commonly used in describing the composition of a letter). However, the terms from the application domain have to be used in a more stereotyped and specific way, as the terminology is now used to document communication with a program (while the terminology of the application domain is used in communication among people). Often it is also necessary to express finer distinctions with more concepts, so one will have to invent new terms – or to use terms which are otherwise considered synonymous with different meanings.

The basic rules for terminology in software documentation are

Consistency Always use the same term (or symbol) to denote the same concept.

Differentiation Use different terms (or symbols) to denote different concepts.

Note that these rules are in conflict with a stylistic rule requesting the use of a varied language to create a fluent text. This stylistic rule does *not* apply to the *terminology* of software documentation where consistent use of terms (and symbols) is much more important than stylistic qualities of the language.

Literary skills are rather useful when documenting software, but don't get carried away – you are *not* writing a novel. For students at a technical university the most relevant kinds of writing are for the theses and the reports on assignments given in courses. Both typically fall in a general category of technical writing for which the soothe rule implies

- Distinguish between the application domain and your professional domain
- Use well-known professional terms correctly without giving definitions
- Define terms that the intended readers may not know
- Make definitions precise, each one easy to identify and find
- Use figures when appropriate.
- Elements of figures may need defined meanings, just like terms
- The use of figures must be consistent with the written text.
- Use examples instead of verbose explanations in tutorial material.
- Use tables in the text to arrange referential material in a systematic way.

Do not expect readers to read everything

A project report may be organised with a main text, which is meant to be read in detail, and a number of appendices, which supply details and provide evidence for claims, e.g. proofs, programs, tests. A user's guide can also be an appendix, motivated by its intended use in isolation. The main text should contain

- an overview of the entire document
- information about what has been achieved
- decisions that have been made (with justifications)

Of course it can be a nuisance for readers to swap frequently between the main text and the appendices, but that is just a challenge to the author: do not displease your readers. In other words it can be justified to include some technical details in the main text, and the author demonstrates with the actual choices a current professional outlook — something that may influence the evaluation of the report.

The *structure* of a report is the division of the documentation into sections with titles and intended contents. The structure provides the means for the user to

access a selected part of the documentation (via the table of contents). There can be sections at several levels (e.g. chapters, sections, subsections, etc.).

The following rules should guide the design of the document structure

- Each section should address a specific subject and different sections should address different subjects.
- Each (low-level) section should be written to address a specific user demand.
- Avoid repetitions (also to ease the maintenance of the documentation).
- Avoid forward references (to ease reading linearly from beginning to end).
- Parallel sections at the same section level should have comparable contents e.g. a book on zoology should not have two such sections entitled "animals" and "monkeys".

Do not be fancy

Often the context will specify some requirements that should be met meticulously. Here are a few general guidelines

- State the title and identify the authors on a front page
- Include a table of contents, except for short documents
- Use A4 paper and wide margins allowing holes for binding and margin notes
- Font changes in a paragraph disturb reading
- Avoid long lines — they are hard to read (60 characters may be too long, but paper often dictates otherwise)
- Do not emphasise repeating elements like headers and footers
- Number pages consecutively throughout the document, including appendices
- Refer uniformly to points within the documents
- Include a section with a list of referred documents (when appropriate)
- Refer indirectly to other documents through the list of referred documents

Do not provoke

Several specific versions of the soothe rule can be expressed

- Use correct spelling and language syntax
- Adhere strictly to prescribed conventions
- Use short sentences.
- Remove sentences that are not essential for conveying the current message.
- Find out what readers expect
- Make it easy for readers to find the essentials

It is important that a report does not contain contradictions of any kind. A special case: be very reluctant to edit output from a program for inclusion in a report. A reader will be more than displeased to see output that cannot be generated by a given program, but is claimed to be. Thus

- Be open about failures
- Include raw data and give an edited version separately only if needed

Do not forget the readers' situation

A special version of the soothe rule applies to reports written on assignments in courses with many students, the *evaluation rule:*

- A teacher should be able to evaluate a good report in X minutes

Reports resemble applications for funds: the readers are busy and soothe-worthy, and they are obliged to assess many documents within a limited time. The value of X is not universal, but it is probably lower than any author expects! If longer time is needed, it is likely to influence the assessment unfavourably.

The evaluation rule implies that there is an upper limit to what will be read in detail. It does not imply that a report should be thin. In fact all relevant details should be available, and readers should be able to dive into the details when in doubt or just for spot-checking.

A particular assignment should provide more information about what is expected. Of course reports should adhere strictly to expressed expectations, even if they are in conflict with these interpretations of the soothe rule.

Documentation of quality

Customers for any product are concerned with quality. Producers consequently have to praise their products, and customers know that they must be on guard for unwarranted praise. Serious customers will look for documentation of quality, preferably something that can be checked by someone other than the producer and cover the expected use of the product.

What is the use of a car with an excellent engine, if the brakes do not work?

Programs are products and reliability is one essential quality attribute. Others are efficiency, ease of use, modifiability, and portability, but few will like to use a very efficient program if it is not reliable. One face of reliability is correctness of results, another is absence of failure in operation.

Program testing is an activity aiming to ensure reliability. Reference [2] is strongly recommended, and it has heavily influenced this section, which can be seen merely as a summary of some of it with a minor supplement about testing components of a program, rather than an entire program.

A supplementary approach to ensure reliability is *formal verification* which draws on mathematical proofs in argumentation for program correctness. It will lead too far to go into this vast topic, but the notions of *invariants, preconditions, post-conditions,* and *assertions* are essential for verification. Once again Column 4 of Reference [1] is recommended as an introduction.

Knowledge of a program's application domain is needed to assert the correctness of results obtained for various sub-domains. A simple example is the computation of square roots of real numbers. Computed values can be checked against tabulated values, or some computed by other means. The experienced customer will know that it is prudent to check the computations in various sub-domains:

$$x > 1,\ x = 1,\ 0 < x < 1,\ x = 0 \text{ and } x < 0$$

Note the inclusion of a sub-domain expected to generate an error. The dependency of application domain knowledge makes it hard for a programmer to perform the checks without consultations with customers.

Failure of operation may be caused by variables attaining values not foreseen by the programmer. Other failures may be unaligned tables in output, missing values in output, premature program termination, and more. Although it is only human to expect absence of failures in one's own products, it is vital that programmers can look with suspicion on their own 'child' during development, and try to identify situations in which they may fail. *Testing* is the term used for that activity, *bug hunting* might be more descriptive and fair.

One thing that makes testing boring is that it can never show the absence of conditions for failures, only their presence. The activity is time consuming and hence expensive.

It can require as much time to test a program as to write it initially

Some programming methods can be used to reduce the time for testing. One fine method is to identify (small) parts of a program that can be developed and tested in isolation. Let the term *components* denote such parts. Component testing fits nicely with the advice that program texts should be understandable by humans in small bites. Likewise it fits with use of *abstract data structures* as for instance classes in object-oriented programming languages. Although it may at first require more time to develop a solution for a specific subproblem, it pays to strive for reuse in any situation, since then one may later draw on a set of well-tested solutions to subproblems developed earlier.

Distinguish between trusted and untrusted components during development

The more trusted components, the less time needed to test the entire program.

Trust is something that builds up, just as among people. You may never be entirely sure, but you learn when you dare rely on trust. A program is not a person, so you can without problems apply a program in various artificial situations.

Now we are going to describe some such situations that may help increase confidence. Consider the following outline of a program fragment

```
initialisation;
while (condition) iterand;
follow_up;
```

The *condition* denotes a branch-point in the control flow, in the sense that either an *iterand* or a *follow_up* computation may follow it immediately. Since there are two paths leading to the branch-point the following combinations are particularly interesting

```
1  initialisation; condition; follow_up;
2  initialisation; condition; iterand;
3  iterand; condition; iterand;
4  iterand; condition; follow_up;
```

Its programmer should observe and accept the fragment's effects involving each combination at least once. Note however, that the fourth combination can only arise when the second has occurred, so the usual mnemonic rule for this fragment is

Test for zero, one, and more than one iteration of a while-loop

The general rule is, of course, that a programmer for every branch-point should observe and accept a program's behaviour in n *path combinations,* where n is the number of control paths leading immediately to the branch-point times the number of control paths that may immediately follow the branch-point.

Convenience then dictates the following advice to programmers

Reduce the number of branch-points as much as possible

Programmers should note that they run the risk of writing programs that are easier to understand, are more general, and probably also use better data structures by following this rule — all at the price of reduced work on testing.

How can a program be forced through a prescribed path combination? Since the behaviour at branch-points depend on the current values of variables, a programmer must control these values, but has to do so by actions executed earlier in the control path. These actions can be *input* of values or *calls* to subroutines.

If input actions are used, the corresponding data are called *test data,* and a programmer should choose test data so that the program actions are easy to mimic by hand. At least that gives the programmer a way to check the results of the computation. It may be necessary to write several sets of test data to ensure that all combinations actually are covered. This is where the main emphasis is in [2] .

If calls are used, these typically appear in a special *test program* written to test a component intended for use in an application program. It may be necessary to write several test programs to ensure that all path combinations in the component are covered.

Make every path combination occur at least once for the test as a whole

An appropriate way of documenting a test is to tabulate as in [2]. The key entry is an identification of a branch-point. This can be a chosen identifier that appears as a comment in the program text next to the condition that is associated with the branch-point, or it can be the condition itself, if no misunderstanding is possible. Furthermore, the pair of paths coming in and going out should be identified as well, but of course with due regard to the particular circumstances — e.g. there is no reason to identify an incoming path, if there is only one, and the number of times the condition in a while loop has been true (0, 1, more than 1) is also a good identification.

Reference [2] is primarily focused on the use of test-data. It illustrates the principles mentioned above and shows concretely how a test can be documented nicely. The present generalisation to test programs is very small indeed. It is strongly recommended to follow the tabulation format used in that document.

Test documentation in university project reports

- Testing is as time consuming as programming.
- We need to cut down on routines to allow more room for new topics
  – but please do not use this as a standard excuse
- Test requirements typically mean that you should focus on principles.
- In practice you should test part of the program completely:
  – typically a smaller routine that is not trivial.
  – always include *a plan* for a structural test.
  – always include a functional test for the program at large.
  – sometimes it can be justified not to carry out a plan completely.

- Strive to automate test and test acceptance as much as possible.

  An automated test routine should emphasise unusual cases, so in the long run it should be silent when actual and expected results agree.

Organisation of reports on software

Although the primary focus is on reports on software, the suggested structure may apply to many other fields of writing as well. What is important is the description of the contents of sections, not so much as their titles and their sequence. A frequently useful organisation of a report:

Cover page

Authors, abstract, index

Introduction

What has been achieved, acknowledgements, document structure.

Analysis

Points to be clarified, issues that deserve special attention

Clarifications: list alternatives, describe pro.s and con.s, finally justify one choice

Program documentation

Pseudocode, links to sources (in appendices)

Documentation of tests (and verification)

Kinds of tests performed, links to test data and - results

Conclusion

Details of incompleteness, assessment of quality,
suggestions for improvements and new projects

References

[1] Jon Bentley. *Programming pearls*. Addison–Wesley, 1986. ISBN: 0–201–10331–1.

[2] Peter Sestoft. Systematic software test.
http://www.imm.dtu.dk/courses/02130/struktur.pdf.

Programming Problem: Topological sorting

A *relation* on a set S is a subset of $S \times S$. A *partial ordering* on S is a relation R_p that satisfies three conditions

Reflexivity: $\forall x \in S\ ((x, x) \in R_p)$

i.e. for every x in S the pair (x, x) belongs to R_p

Anti symmetry: $\forall x_1, x_2 \in S\ ((x_1, x_2) \in R_p) \wedge (x_2, x_1) \in R_p\ \Rightarrow\ x_1 = x_2)$

i.e. for every x_1 and x_2 from S:
if both (x_1, x_2) and (x_2, x_1) belong to R_p it must be the case that $x_1 = x_2$

Transitivity: $\forall x_1, x_2, x_3 \in S\ ((x_1, x_2) \in R_p \wedge (x_2, x_3) \in R_p) \Rightarrow (x_1, x_3) \in R_p$

i.e. if (x_1, x_2) and (x_2, x_3) belong to R_p, then also (x_1, x_3) belongs to R_p
for every choice of x_1, x_2, and x_3 from S

A *total ordering* R_t in addition satisfies

Totality: $\forall x_1, x_2 \in S\ ((x_1, x_2) \in R_t \vee (x_2, x_1) \in R_t)$

i.e. either (x_1, x_2) or (x_2, x_1) or both belong to R_t for any x_1, x_2 from S

Given a partial ordering R_p on a *finite* set S, one can be interested in a total ordering R_t such that $R_p \subseteq R_t$ (i.e. every pair that belongs to R_p also belongs to R_t). Note that more than one total ordering may satisfy the requirement.

Algorithm

Topological sorting is the name for the following algorithm to determine a total ordering R_t that contains a given partial ordering R_p. First note that a total ordering on a finite set S corresponds to a sequence that contains all elements of $S:\ [x_0, x_{0+1}, ..., x_{n-1}]$. $(x_i, x_k) \in R_t$ if and only if $i \leq k$.

If S is non-empty, there must be at least one element u without predecessor: $\forall v \in S\ \neg((v, u) \in R_p)$. Pick any such u as the first in the sequence of elements ordered according to the resulting total ordering.

Furthermore, let $R'_p = R_p \backslash \{(u, y) \in R_p\}$, i.e. the set that remains after every pair with u as its first member has been removed and let $S' = S \backslash \{u\}$. If S' is empty the total order is given by the picking sequence, otherwise the next element can be determined by applying the same algorithm for the partial relation R'_p on the set S'.

Assignment

Write a program by which topological sorting can be illustrated, for instance by writing results for particular partial orderings to standard output.

The program should be written in Java and make appropriate use of the class concept.

Write a report documenting your efforts and discuss adequate program reactions when a given ordering is not partial as assumed.

Report on the Programming Problem

The following appendices could be reorganised to form an autonomous report on the programming problem (page 13). The front page is omitted, since the front page of this document serves as a model. This appendix should be considered the main text of the report, the subsequent ones as appendices in the report.

Topological Sorting

Jørgen Steensgaard-Madsen

This reports on the solution to the given programming problem. Readers are supposed to be familiar with the problem statement and the concepts used in it. Furthermore, readers are expected to have worked on the problem themselves. An abstract data structure has been developed successfully as the Java class `POset`. The report tells about analysis prior to program development, the program itself, and its test.

Analysis

The problem statement has no requirements about how a partially ordered set may be given. Once given, the algorithm provided in the problem statement should be easy to implement, and the intended output should be produced.

An abstract reflection on our understanding of a partially ordered set is

> *`Printable Printable Set Map`*
> i.e. a finite function that takes a *`Printable`* into a *`Printable Set`*

The finite function should map each element into a set of known successor elements. The transitivity may imply more successors than the known ones. A smallest set of known successors that allows the full set of successors to be determined exists, and it will be called the set of *immediate successors*.

Since maps and sets are fairly complex structures in general, we shall use a simplified version, which suffices here:

> *`Printable Printable List Pair List`*
> i.e. a *list* of *pairs* (`Printable`, *list* of `Printable`)

where each pair identifies an element in the set and associates it with an unordered list of known successors. A list of such pairs represents the entire set of elements.

Building a partially ordered set

Two approaches have been considered:

> reading a written representation of a set and its ordering
>
> building it by calling members of an object

If data are going to be read, it will be best to do so independently of building an internal representation. A possible reader of data would consequently need call routines provided by a builder, so the second approach is chosen.

Building a pratially ordered set might be based solely on identification of its elements as seen in paris of the ordering. This is not quite satisfactory, since some elements may not be related to others. Consequently, a member routine to identify set elements seems required, e.g.

```
new_item(Printable x)
```

with `x` being some printable object. The routine may return an object, e.g. `Q`, with a member that can record relation with it, e.g.

```
Q.precedes(R); //
```
to indicate $(Q, R) \in P$

This implies that `new_item` must return an appropriate object. Note that if $(Q, R) \in P$ and $(R, S) \in P$ have or will be recorded there is no reason to record also $(Q, S) \in P$, which is a consequence of the partial ordering.

Printing results

One might of course let an object that represents a partially ordered set provide a method to print a total ordering that agrees with it. However, it will be more general (i.e. less tied to the suggested printing) to have a class that implements something like the `Iterator` interface. Since the `remove` method will make no sense, the following methods for iteration will be implemented

```
get_first()     // get one element with no predecessor
has_next()      // tell if there are more elements
next()          // procede to a lesser element
contents        // the (printable) element id.
```

Usage is illustrated in the test program `POtest.java` shown page 22

Implicit computation

The computation of a total ordering that extends a given partial one will be performed when `get_first` is called after some modifications. Thus it will be possible to iterate over the set several times. However, one has to consider what happens when an explicit addition to the partial ordering is provided (i.e. when `new_item` is called):

outside an iteration the effect can be well defined

during an iteration the effect may depend on the history

It has been decided to throw an exception in the latter case, and accept the former.

When `new_item` is called, no check is performed to see if the partial ordering conditions are met. A consequential error will then occur during an iteration over elements, and at that point an exception will be reported. The only check that could reveal the problem early seems to be to iterate implicitly over elements, which seems a waste and can be done explicitly by users who want to do so.

Use of class POset

An object of class `POset` represents a partially ordered set of elements represented by objects of class `POset.POitems`. Element objects must be instantiated with the `new_item(x)` method of `POset`, where `x` is an object representing the 'value' of the element and must be of type `Printable`.

```
interface Printable { public void print(); }
```

Users decide what kind of 'identification' should be printed, for instance complete information about `x`. `POset` itself implements `Printable` and calls the `print` method of every element when a user calls a `POset`'s `print` method.

Except while iterating over the elements, a new element may be added by calling the method `new_item`, for instance `eX = mySet.new_item(x)`. If this rule is violated, a `RunTimeException` will be thrown. Note the distinction between the value `x` and the object `eX` to hold it. Given two elements, their mutual relation can be stated by calling the method `precedes`, e.g. `eX.precedes(eY)`.

Elements like `eX` have type `POset.POitems`, which means that `POitems` is an inner class of `POset`. The type name can be used to declare an array, for instance, but instantiation of objects of this type requires use of the method `new_item(x)` as explained above. The test program (page 22) illustrates the use of the described methods.

Elements of a set may be accessed in a program as in the following outline:

```
POset mySet = new POset();

// ... enter elements into mySet

  tmp = mySet.get_first();
  if (tmp != null)
   while (tmp.has_next()) {
     //   tmp.contents   identifies the current element
     tmp = tmp.next();
   };
```

Internals of POset

`POset` is an abstract data structure that represents a finite set with operations to generate elements and to define a prescribed partial ordering of elements. Other operations allow elements to be accessed in a sequence corresponding to a total ordering that has the prescribed partial ordering as a subset. A final operation allows the current state to be printed, illustrating the prescribed partial ordering.

Every object of class `POset` represents a partially ordered set. Private fields are:

first is (a reference to) a list of all elements of the set

modified is *true* if a new edge has been added to the prescribed partial ordering after the latest total ordering has been determined

Elements are themselves objects of a public class `POitems`. These elements are linked together in a list of objects of private class `Succ`.

The simple version of the representation of a partially ordered set

`Printable Printable List Pair List`
i.e. a *list* of *pairs* (`Printable`, *list* of `Printable`)

i.e. the inner structure of `POset` is more specifically represented as

a *list* of `POitems` **where** `POitems` = *pairs* (`Printable`, *list* of `POitems`)

The rôle of element identifications is to represent the the recursive nature of this definition conveniently in input. The representation in a Java program is here more efficient in terms of references to objects.

Use of an inner class, like `POitems`, is natural in a class that has methods to handle a collection of objects. One only has to remember two things:

- objects of an inner class cannot be instantiated directly outside the containing class, so a method of the containing class may take care of that
- the name of the inner class can be used as a type name when qualified by the name of the containing class

The interface to `POset` with a short description of methods:

```
public class POset extends java.lang.Object implements Printable
{
   public POset();

   public POset.POitems new_item(Printable x);
   public POset.POitems get_first() throws Exception;
   public void print(); // displays internal representation (for testing)

   public class POset.POitems extends java.lang.Object
   {
       public Printable contents;
       public void precedes(POset.POitems);
       public POset.POitems next() throws java.lang.Exception;
       public boolean has_next();

       private Succ successors;       // recorded list of less elements
       private POitems next;         // next pair (element,successors)
   }

   private POset.POitems first;      // first pair (element,successors)
   private boolean modified;      // must stay unset during iterations
   private class Succ
   {
       Succ next;                                  // the next successor
       POitems item;          // less than the element in the list's head
   }
}
```

Implementation details

An object of type `POitems` represents an element of a `POset` and consists of:

contents a printable object

successors a list of objects $[y_0, y_{0+1}, ..., y_{n-1}]$ for which *this*`.precedes(`y_i`)` has been called

predecessors an integer count of (remaining) objects that must precede *this* in a total ordering

next a link to the rest of the list of all `POitems`

It is actually the private methods of `POitems` that implements topological sorting. A private method, `count_predecessors` exists to compute the count of predecessors whenever a topological sort is required. Another private method, `get_entry` performs the identification and removal of an element with no predecessors as described in the problem statement.

The test program POtest

An application of `POset` has been written to illustrate its use. The application serves also as a structural test of the methods `print` and `topsort` in `POset`.

Testing `print` can be summarised as follows

Condition	Character	Remark	`POtest`	`POdata`
while	0 times	Empty set	l. 45-46	l. 1-3
(temp != null)	1 time	Singleton set	l. 50-53	l. 5-6
(line 89)	> 1	Most common	l. 62-63	l. 9-19
while	0 times	No successors	l. 72-73	l. 22-27
(aux != null)	1 time	One successor	l. 72-73	l. 29-30
(line 95)	> 1	Several successors	l. 72-73	l. 28
if (aux != null)	True	At least two successors	l. 72-73	l. 28
(line 98)	False	Last successor	l. 72-73	l. 28-30
if (temp != null)	True	More than one element	l. 72-73	l. 22-30
(line 102)	False	After last element	l. 72-73	l. 31

Testing `topsort` follows the same pattern

Condition	Character	Remark	`POtest`	`POdata`
while	0 times	Empty set	l. 45-46	l. 1-3
(first != null)	1 time	Singleton set	l. 50-53	l. 5-6
(line 244)	> 1	Most common	l. 62-63	l. 9-19
while	0 times	Empty set	l. 45-46	l. 1-3
(temp != null)	1 time	Singleton set	l. 50-53	l. 5-6
(line 256)	> 1	Most common case	l. 62-63	l. 9-19

One error was detected and repaired during the test. The case of an empty set was not handled correctly by the `show`-routine in the test program. No errors were found in `POset` itself. Of course many were found before the actual systematic test.

Conclusion

We have successfully written a Java class `POset` with members that can be used to store a partially ordered set, and to retrieve its element in a sequence corresponding to a total ordering that contains the partial one. The class has been tested and used in a program that writes the elements of a partially ordered set as suggested in in the problems statement.

Our implementation uses several concepts that are characteristic for use of classes in Java: inner classes, exception handling, and interfaces.

The testing technique advocated by Peter Sestoft has been used succesfully.

```
1
2   /********************** Author: J. Steensgaard-Madsen ************
3    *
4    * A partial ordering is a transitive and asymmetric relation.
5    * A total ordering in addition relates any two values.
6    *
7    * Problem:
8    *   find a total ordering which contains a given partial ordering
9    *
10   * Topological sorting is one method to solve the problem.
11   *
12   * The class below provides means to state and solve particular
13   * problems of this kind.
14   *
15   ******************************************************************/
16
17  public class POset implements Printable {
18
19      /*************************************************************
20       * POset provides means to define a partially ordered set
21       *
22       * public class POset extends Object implements Printable {
23       *     public POset();
24       *     public POset.POitems new_item(Printable);
25       *     public POset.POitems get_first() throws Exception;
26       *     public void print();
27       *     public class POset. POitems extends Object
28       *     {
29       *         public Printable contents;
30       *         public void precedes(POset.POitems);
31       *         public POset.POitems next() throws Exception;
32       *         public boolean has_next();
33       *         public POset.POitems(POset,Printable);
34       *     }
35       * }
36       *
37       *************************************************************/
38
39      private POitems
40          first;      // The first element of the list of all
41
42      private boolean
43          modified;   // Set to true with a new order requirement
44
45      private class Succ {
46
47          /* A class of objects for a list of required successors    */
48
49          Succ next;      // the next in the list of successors
50          POitems item;   // the element required to follow
51      }
52
```



```
53
54      public POitems new_item(Printable x) {
55          /**
56           * Creates a new element of type POitems and enters it
57           * into the local data structure with no relations
58           ************************************************************/
59          return new POitems(x);
60      }
61
62      public POitems get_first() throws Exception {
63          /**
64           * Prepares for scanning the items in successor order
65           ************************************************************/
66          if (modified) first.topsort();
67          return first;
68      }
69
70      public void print() {
71          /**
72           * Format:
73           *   [ (x_1,[y_1_1,y_1_2,y_1_3,...]),
74           *     (x_2,[y_2_1,y_2_2,y_2_3,...]),
75           *     ...
76           *   ]
77           *
78           * i.e. a list of pairs consisting of a node and a list of
79           * immediate successors.
80           *
81           * A pair (x_i,y_i_k) represents u.precedes(v) with
82           * x_i corresponding to u.print() and y_k to v.print()
83           *
84           ************************************************************/
85
86          POitems temp = first;  // to traverse the list of all
87          Succ    aux;           // to traverse the successor list
88
89          System.out.print("[");
90          while (temp != null) {
91              System.out.print("(");
92              temp.contents.print();
93              aux = temp.successor;
94              System.out.print(",[");
95              while (aux != null) {
96                  aux.item.contents.print();
97                  aux = aux.next;
98                  if (aux != null) System.out.print(",");
99              }
100             System.out.print("])");
101             temp = temp.next_item;
102             if (temp != null) System.out.print(",\n ");
103         }
104         System.out.print("]\n");
105     }
106
107
```

```
108
109     public class POitems {
110
111         /*********************************************************
112          * Objects of type POitems represents elements of POset.
113          * Member functions:
114          *
115          *   a.prededes(POitems b)   requires b to succeed a
116          *   a.next()                a's successor in the list of all
117          *   a.contents              the user's identification of a
118          *
119          *********************************************************/
120
121         private Succ
122             successor;    // A list of required successor POitems
123
124         private POitems
125             next_item;    // The next element in the list of all
126
127         public Printable
128             contents;     // Provided by users
129
130         private int
131             predecessors; // The number of required predecessors
132
133         public void precedes(POitems x) {
134
135             /**
136              * Enters x in a new element in the list of
137              * required successors of this
138              *********************************************************/
139
140             Succ y = new Succ();  // the new element
141
142             y.item = x;
143             y.next = successor;
144             successor = y;
145             modified = true;
146         }
147
148         public POitems next() throws Exception  {
149
150             /**
151              * Returns the next element in the list of all, provided
152              * no new element or no new required order.
153              *********************************************************/
154
155             if (modified)
156                 throw new RuntimeException("Untimely modification");
157
158             return next_item;
159         }
160
161         public boolean has_next() {
162             /**
163              * False for the last in the total order only
164              *********************************************************/
165             return next_item != null;
166         }
167
```



```
168
169         private void count_predecessors() {
170
171             /* An element with no required predecessors may be first
172              * in the list of all.  Later one can then simulate the
173              * removal of that element so that the next may be found
174              * likewise
175              *********************************************************/
176
177             POitems
178                 item = first;  // to traverse the list of all
179             Succ
180                 temp;          // to traverse the successors list
181
182             while (item != null) {
183                 temp = item.successor;
184                 while (temp != null) {
185                     temp.item.predecessors += 1;
186                     temp = temp.next;
187                 }
188                 item = item.next_item;
189             }
190         }
191
192         private POitems get_entry() throws Exception  {
193
194             /* Finds and removes an element for which no
195              * required predecessor remains.
196              *********************************************************/
197
198             POitems
199                 entry = first, // to traverse the list of all
200                 pred = null;   // the predecessor of entry
201             Succ
202                 temp;          // to traverse the successors list
203
204             while (entry != null && entry.predecessors != 0) {
205                 pred = entry;
206                 entry = entry.next_item;
207             }
208
209             // No such element exists: no partial ordering
210             if (entry == null)
211                 throw new RuntimeException("No partial ordering");
212
213             // Remove the element
214             if (pred == null) first = entry.next_item;
215             else pred.next_item = entry.next_item;
216             entry.next_item = null;
217
218             // Reduce the counts of predessors correspondingly
219             temp = entry.successor;
220             while (temp != null) {
221                 temp.item.predecessors -= 1;
222                 temp = temp.next;
223             }
224
225             return entry;
226         }
227
```

Notions about aggregation of values

Astronomers depend on a notion of *Riemann space* to develop their models of the universe, engineers use the notion of *vector spaces* to compute the proper dimensions of bridges and depict the trajectories of rockets sent into outer space, and several professionals use the notion of *graphs* for various purposes, e.g. production planning. These notions are important and independent of tools used to manipulate particular instances.

The short history of computer science and engineering has led to the identification of some notions that have proved very useful to represent facts and ideas in a computer. Representations involve forming aggregates of related entities, e.g. names and addresses of people. The kinds of aggregations have far less structure than the mathematical ones mentioned above, so they are typically introduced as part of programming language instructions. The ambition here is to introduce an appropriate collection independently of programming languages.

As with numbers our understanding of the representation in a computer is actually a two stage affair: a computer represents entities by some physical phenomena, like voltage levels, but humans need a textual representation for discussions and for teaching, and it has to match the computer's in a faithful way – faithful in a sense that cannot be stated satisfactorily here.

Text representations are particularly useful for humans, who can draw on experiences from various fields to justify or describe actions on values, when these can be described as texts. This *special font* will be used below to illustrate the proposed text representation.

Far from all programming languages offer text representations for the aggregations described in this section. Still the text representations are useful, partly for documentation, partly as guides to definition of libraries of routines to build and manipulate the aggregated values without special support from the programming language that is used.

Some of the notions described below may be well-known from other fields and are included for completeness and to help readers understand the formalities of the presentation.

Pairs and tuples

Name, age, and sex may be aggregated in a representation of a group of people. One individual might be represented as illustrated by

```
("Betty Nansen", 45, "female")
```

The term *tuple* is used for this kind of aggregation, sometimes specialised to *triple*, *quadruple*, *pentuple* etc. to reflect the number of parts in it. The shortest one is a *pair* having two parts, and pairs can be considered the fundamental building block, if one allows groupings as illustrated by

```
("Betty Nansen", (45, "female"))
```

i.e. a triple can be considered a pair having a right part that is a pair.

Some functions are characteristic for pairs, just as addition and subtraction are characteristic for numbers. Fortunately, the two for pairs, fst and snd (short for *first* and *second*), are extremely simple, characterised by the following property:

`(fst(x),snd(x)) = x`

meaning that if `fst(x) = "Betty Nansen"` and `snd(x) = (45, "female")` , then we can be sure that `x = ("Betty Nansen", (45, "female"))` . Furthermore, we will know that `fst(snd(x)) = 45` .

Compared to some of the following means of aggregation, pairing is unique because it provides the means to combine values of different types, e.g. integer and string values.

The functions fst and snd are characteristic for pairs, but not necessarily the only relevant ones. One might characterise a function to compare for equality:

`x = y` if and only if `fst(x) = fst(y)` and `snd(x) = snd(y)`

Variations exist between programming languages, not only in terms of whether pairs (and tuples) are supported, but also in terms of the names of the characteristic functions and whether the comparison for equality is (or can be) defined. Note that equality is essential in the abstract characterisation of the functions fst and snd, but that it does not mean that equality is always defined for use in programs.

It does not make sense to compare arbitrary pairs, since the comparisons between corresponding parts should make sense. In particular it does not make sense to compare a pair to a number or a string. Consequently, it does not make sense to compare `(45,"female")` to a triple.

Lists

A list is an aggregation of values all of the same type: all integers, all strings, all lists of pairs of integers and strings, etc. Apart from that property, there is a great similarity between pairs and lists. One important difference exists: a list may be empty, whereas a pair always has a first and a second part. Here are the important properties of lists, in terms of two characteristic functions, hd and tl (short for *head* and *tail*), and a `::` -operator symbol. The properties of the operator is defined by:

if `x` is not empty, `x = hd(x) :: tl(x)`

Again equality is essential in the abstract characterisation, but not necessarily defined for use by programmers. An empty list can be written as `[]` and the characterisation might be formalised as

$(x \neq [\,]) \Rightarrow (x = hd(x) :: tl(x))$

Any list can be given as $x_0 :: x_1 :: \ldots :: x_{n-1} :: []$ and an even more convenient notation will be introduced below. Not all languages support this notation, far from it, and a few comments are called for:

1. `hd(x)` is of some (named unknown) type T and requires `x` to be of type: list of T
2. `tl(x)` requires `x` to be of type list of T (for some named unknown type T) and yields a value of the same type
3. the operator symbol `::` requires the left operand to be of type T, the right operand to be of type: list of T and its result will then be of type list of T
4. the name `nil` is sometimes used for an empty list
5. the type of an empty list should be apparent from the context of use.
6. the notation `[`x_0`,`x_{0+1}`,...,`x_{n-1}`]` is used for ``x_0`::[`x_1`,`x_{1+1}`,...,`x_{n-1}`]`

Naturally things combine, so that a representation of a collection of people might be represented by a list of tuples, e.g.

```
[
("Betty Nansen", 45, "female"),
("Paul Reumert", 25, "male"),
...
]
```

It is tempting to define an indexing operation such that if `X = [`x_0`,`x_{0+1}`,...,`x_{n-1}`]` , then `X`$_k$ `=` x_k , but the conciseness of the notation would not reflect the complexity of the required composition of the functions `hd` and `tl` . Only few programming languages support indexing in this way by a primitive operation (but programmers will be able to define it themselves). The next subsection presents an aggregation that puts indexing high on the list of priorities.

Arrays

An array is an aggregate of values of the same type, organised in such a way that indexing is a primitive operation, i.e. not expressed in terms of other operations. An array of fixed length n can be visualised as a table with rows numbered $0, 1, 2, ..., n-1$. Each row may be used for aggregates of the same type, even arrays.

An array is not built from parts like tuples and lists, but rather comes into existence through one single operation. There is no commonly accepted way to indicate that operation, so here is a variant invented for this presentation:

- if `n` an integer, then `n->[`x_0`,`x_{0+1}`,...,`x_{m-1}`]` is an array with `n` elements $x_{i \bmod m}$ for `i = 0..(n-1)`

The *length of the array* is a fixed constant: the value of `n` . There is no way to directly build a new array from a given one. The choice between use of a list and an array should depend on the needs to access elements at various positions.

Note that `i mod m` denotes the remainder when integer `i` is divided by integer `m` , so elements of `[`x_0`,`x_{0+1}`,...,`x_{m-1}`]` are used cyclically to fill the entries of the array.

Records

As shown previously, a tuple may be used to represent a person. It implies that parts have to be extracted by combinations of functions fst and snd. Readability plays a role sometimes, and that is the justification for aggregation of records. They are similar to tuples, but provides more meaningful names for the constituting parts:

```
RECORD person(name:string,age:integer,sex:string)
```

is not itself a record, but a description of a family of records. Particular records may be built as illustrated by

```
person("Betty Nansen", 45, "female")
person("Paul Reumert", 25, "male")
```

and the properties that these records can be stated as

```
person(name(x),age(x),sex(x)) = x
```

The purpose of the description of a family of records is just to introduce the appropriate functions, which allows a corresponding characterising property to be stated.

Object-oriented programming languages generalise the notion of records to objects, but many other languages support the simpler notion. Note that object-oriented languages prefer a notation `x.name` over `name(x)` and generalise the notion so that elements of an object can be a routine (called a *method* in some languages).

Types and operations in general

The characteristic names: pair, list, array, and record, are called *type operators* in the literature, whereas integer and string are names of *types*. The type operators can be compared to the usual operators on integers: given integer operand values they can be combined by an operator into an expression that yields an integer value. Likewise: given operands that are types they can be combined by a type operator that then 'yields' a type. Actually it is customary to let the names of type operators begin with a capital letter, which will be done henceforth.

Type operands are combined by a type operator that is written after all its operands. Thus

`integer List` is the type of values that are lists with integer elements

`integer string Pair` is the type of values having a fst-part that is an integer value, and a snd-part that is a string value

`string integer string Pair Pair Array` is the type of values that can be considered tables with every entry containing a triple.

This notation is convenient when complex aggregates have to be described. Records are special: it has already a formal description. Note that every part of a record (often referred to as a *field*) is indicated by according to the pattern

name : *type*

where *type* is stated according to the rules above.

Another aspect of aggregations should be noticed: there are some means to build an aggregate and some to decompose it into parts. Sometimes these are referred to as *constructors* and *decomposers*, respectively. The kinds of aggregates we have considered are simple in the sense that their decomposers in particular are simple functions.

As there are differences between programming languages' ways of representing the simple aggregates, there are even bigger differences in their way to handle complex aggregates. We shall for completeness just hint at a particular complex kind of aggregates and recommend that you skip the rest of the section in a first reading.

A collection of people could be represented as a list or as an array. The alternative to be considered now is called a *sequence:*

```
Iterate {
  Item("Betty Nansen", 45, "female");
  Item("Paul Reumert", 25, "male");
  ...
}
```

What is special for sequences is that an appropriate decomposer would be an *iterator*, i.e. something that is closely related to statements of the form *for every ...*. A decomposer is a special kind of operation which accesses the elements of a sequence one by one. This is hard to understand for non-programmers, but here is an example that presumes that *persons* denotes the sequence outlined above

```
program{
  ...
  sequence persons;
  stdout << ~"Iterate{\n";
  persons.all{
    stdout << ~"  Item(" << name << ~", "
                         << age  << ~", "
                         << sex  << ~");\n";
  };
  stdout << ~"}\n";
}
```

Much has had to be omitted, but the idea is to regenerate in print the text representing the sequence. For sequences, this is as close as one can get to the mathematical statement of properties in an equation. It has been stated as a program in a suitable language, and if you have no programming experience or if this confuses you, it does you no harm to skip to the next section.

Iteration over the sequence is expressed by the 'decomposer' *all* , which implies that the computation expressed in the braced part after *all* will be executed for every member of the sequence, with suitable names associated with parts of each item. The omitted part is where the naming of parts is stated, but this is in fact of no

interest here. The tilde (~) before a string indicates that it should be printed without quotation marks. In other words: name and sex will be printed with quotation marks!

Sequences are most likely described differently in a particular programming language. The essence is that the destruction of a sequence requires more than a simple decomposer function as defined above. Sequences are closely related to data stored as *files* in a computer's permanent store. The complexity is reflected in the pattern programmers must learn to express the treatment of data from a file in a program.

Applications

Two examples will be presented to illustrate the use of textual representations of aggregates in practice. The first one arises in a system for information about students in a particular course. The second one discusses various possibilities for representing the notion of a *directed graph*. Note that one relates to actual facts, the other to abstract ideas.

Course attendance

A number of students attend a course. Teams of attendees consist of students who have registered to work together on exercises during the course.

Every student is known by name, study number, a sequence number in an alphabetical listing of attendees, and a team number. This has been organised as outlined in

```
teams(60 -> [[31,5,27],[2,82,58],[89,30],[17,110],[22,83],
             [102,50],[20,45,114],[71,39,25],[33,7],[57,47],
             [100,69,84],[6,112],[28,117],[15,52,96],[61,56],
             ...
             [88]]);

attendees(119){
  student(1,"Andersen, Ben Hur","s841996",32);
  student(2,"Andersen, Peter","c682046",1);
  student(3,"Andersen, Søren","c684187",29);
  ...
  student(118,"Østerbye, Tomas","c691451",21);
};
```

It should be easy to recognise an array in the parenthesis following the name `teams` and the `attendees(){...}` expands slightly on the sequence example in the previous subsection by providing a constant that is one more than the length of the sequence.

The two, `teams` and `attendees` , can be used to provide data for the actual application. The value of type `integer List Array` , which `teams` provides, contains an entry for each team, i.e. a list of sequence numbers that identify team members. The sequence of attendees is represented as explained in the previous subsection.

Directed graphs

A *graph* is a very simple structure studied in one branch of mathematics and quite often used in practice. Here the focus is on *directed graphs*.

Mathematics say that a directed graph consists of two finite sets called *nodes* and *edges*. Two functions, *from* and *to*, are defined, such that for every edge x, *from(x)* and *to(x)* are nodes. The intuition is that every node can be drawn as a circle, say, and edge x can be drawn as an arrow leading out of node *from(x)* and into node *to(x)*. In general graphs the edges do not have a direction.

One way to represent a graph is to follow the mathematics as closely as possible. If nodes are appropriately represented by strings, edges can be represented by a set of pairs of strings. However, sets have not been introduced as a standard aggregate, which it probably deserved, so some representation for a set is needed. We shall use a list of pairs of nodes, and satisfy ourselves with the fact that two different edges may in fact be represented by the same pair of nodes, so duplicate pairs in a list will be acceptable.

A simple graph, similar to a triangle may thus be represented as

```
(["X","Y","Z"],[("X","Y"),("Y","Z"),("X","Z")])
```

An alternative is to use just the edges part of this, since it appears that the set of nodes can be computed from it. However, one may leave out any two edges from the set of edges in the example, and the result is still a graph, whereas the set of nodes can no longer be computed from the edges alone. The alternative is useful only if it is known that every node is incident with an edge.

A third possibility is to represent a graph by a list of pairs consisting of a node and a list of nodes:

```
[("X",["Y","Z"]),("Y",["Z"]),("Z",[])]
```

It may be desirable to identify edges by a unique name. This can be achieved easily for each of the representations illustrated above, e.g. the latter

```
[("X",[("z","Y"),("y","Z")]),("Y",[("x","Z")]),("Z",[])]
```

The advantage of having a text representation of aggregates is that it is easy to present examples and to discuss their merits among team members or in a report about a system development.