The MUSA design methodology

S. Pemberton, L.G. Barfield

Computer Science/Department of Algorithmics and Architecture

CS-R9265 1992

The MUSA Design Methodology

Steven Pemberton and Lon Barfield

CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
&
SERC
P.O. Box 424, 3500 AK Utrecht, The Netherlands
Email: Steven.Pemberton@cwi.nl, Lon.Barfield@cwi.nl

Abstract

MUSA is a multi-user authoring system, planned as an application in the Views system. MUSA's aim is to support concurrent development of 'products' being produced by several people cooperatively or by one person alone. The products are documents in a wide sense: articles, books, software, drawings, etc. This article presents the methodology behind MUSA.

The
VIEWS
System

1991 Mathematics Subject Classification: 68N99, 68U99.

1991 CR Categories: H.4.2, H.5.3.

Keywords and Phrases: Types of systems, decision support, group and organization interfaces.

1 MUSA

MUSA is a multi-user authoring system; its aim is to support concurrent development of 'products' being produced by several people cooperatively or by one person alone. The products are documents in a wide sense: articles, books, software, drawings, etc.

The effort involved in 'producing a document' can range from relatively small, for a small document being produced by one person, to very great, involving a large group of people, many interacting documents, many interacting decisions, with a lengthy planning phase, different versions at different stages, and experimental 'what-if' versions. The aim of MUSA is to make the life of the authors as simple as possible, by supporting all stages of the production process.

For a complex set of documents, there will first be a lengthy planning phase: requirements must be specified, design issues identified, alternative solutions enumerated, and decisions reached. The interaction between these elements should be made explicit, so that if at any stage some requirement is changed, decisions can be reviewed, and if necessary changed, without any unforeseen interference with other parts of the design. This process then produces the design for the product, which can be used as the basis for a workplan and designation of tasks.

The subtasks of producing the requirements, the design, and the workplan can be seen as another co-operative production of a document, and indeed any of these elements can iterate through different versions, be subject to 'what-if' trials, and so on, so that the production of these elements should be treated as MUSA documents in the same way.

MUSA thus offers its users support for the design, management, and production of documents:

Design:

- ♦ recording the design process of the system
- ♦ reviewing the decisions taken, and the reasons for them
- ♦ producing a rationale for the design.

Management:

- ♦ identifying and planning the tasks to be done
- ♦ resource allocation
- ♦ keeping track of progress

Production:

- ♦ allowing 'what-if' experiments
- ♦ supporting incremental definition of the structure of the product
- ♦ supporting incremental construction of the product
- ♦ supporting versioning, including merging of versions
- ♦ supporting concurrent working
- ♦ giving a production methodology with computer support.

2 Designing a product

The design of a product starts with a specification of what is required of the product. This can be anything from a simple statement of intent, such as “*produce a timetable for moving house*” to a complete set of documents carefully documenting all necessary parts of the final product.

The task of creating the final product is then a process that transforms this specification into something that matches the specification (see Figure 1.)

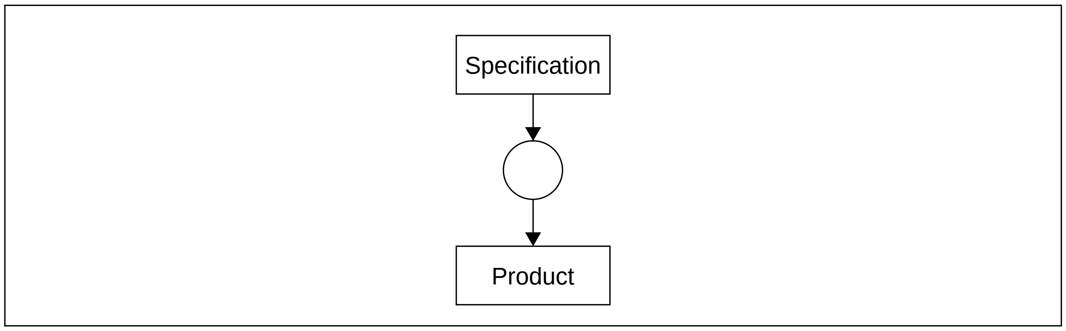

Figure 1. Transforming a specification into a product.

The circle represents the production process, and is itself structured, as will be shortly explained[1]. It is the design of what goes into this circle that the MUSA design methodology is principally about.

While designing the production process, many decisions have to be taken: there are many aspects of the product that are typically not defined in a specification that have to be resolved before the product is finally built. These sorts of decisions are typically taken in meetings after discussions of various merits. An important aspect of designing a product is to keep a record of these decisions, *and the reasons for them.* Very often a decision has to be changed, and having a record of the decision process can shorten the time needed to make new decisions, and obtain an overview of the effects on the whole of the change of decision.

Finally, in order to actually produce the product, a plan has to be created to allocate resources (time, people, equipment) to the production process; then production can begin (see Figure 2.)

As will be expanded on later, many aspects of this process can be computer-supported: the network of decisions, the structure of the product that arises from the decisions, the allocation of resources, monitoring the progress of production according to the plan, and the actual construction itself.

3 The elements of the MUSA production cycle

As can be seen from the diagram, the MUSA production cycle consists of five elements: *the specification, the process, the plan, the argumentation* and *the product itself.* We will now treat these one by one.

1. The specification and product are represented as boxes to suggest a book, or a box containing a product; the production process is represented as a circle to suggest a wheel or cog.

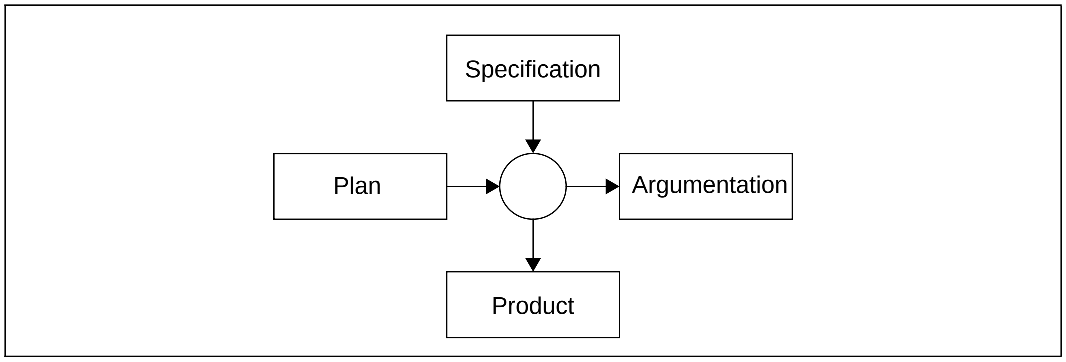

Figure 2. The complete MUSA production diagram.

The specification

As mentioned before, the specification can be as little as a single sentence, and as much as several volumes of close detail. Note in particular that the specifications could be themselves a product of a MUSA cycle, whose own specification was "*produce specifications for product*". This can either be seen as a part of the higher-levels of the production process, or as a refinement of the task (see Figure 3.)

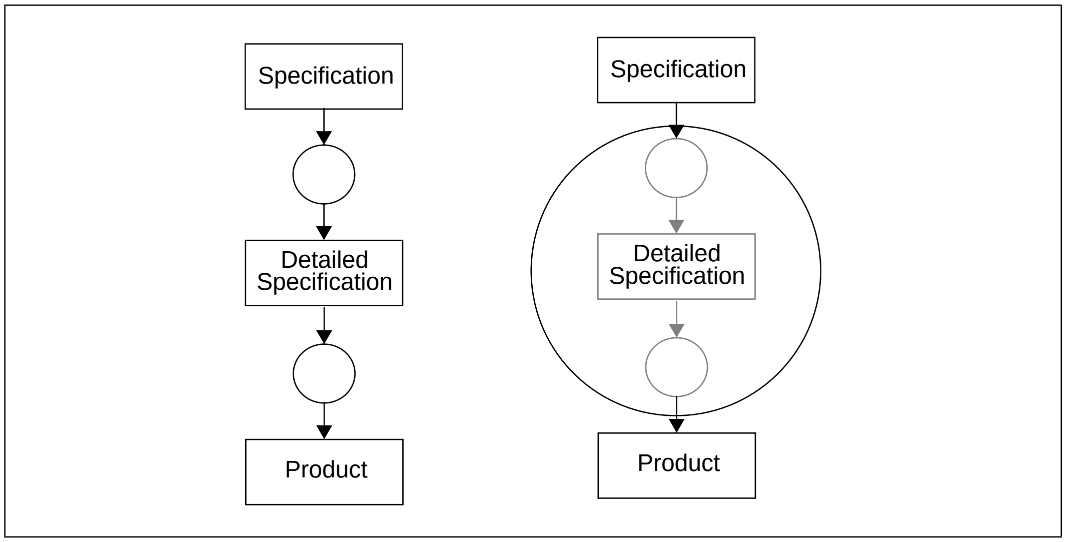

Figure 3. Specifications as products.

The specification gives a functional description of the final product: how it will work (or what it is to achieve) without going in to detail of how to achieve that end. In particular, one part of the production process is deciding which of the many methods to use for implementing the different parts of the functional description.

The specification will often contain references to other documents (such as language definitions for a compiler project) considered to be part of the specification, but not physically included in the specification.

Note that the specification need not be a frozen document: it is quite usual in the life-cycle of a product that the specifications change to match changing requirements or resources, and it is necessary for any design methodology to support this.

The argumentation

The aim of the MUSA design process is to break a complex task into a number of smaller simpler ones. Thus an important first step is to create the structure of the underlying process. This is typically done with a *brainstorming session*, where a number of high-level *issues* are identified that need to be resolved, and as many as possible *positions* are enumerated with regards to these issues, along with argumentation about the relative advantages and disadvantages of the different positions. Different positions may create new sub-issues that then in themselves need to be resolved. This argumentation method is based on the IBIS method [1], and will be treated in more detail later.

This brainstorming generates a basic structure for more detailed reasoning about the final product. At each level as an issue is resolved, a *decision* can be recorded for the issue. This hierarchy of issues, positions, argumentation and decisions is then a record of the discussions around the product, and also records the relationships between different decisions, so that later changes can refer to the original argumentation[1].

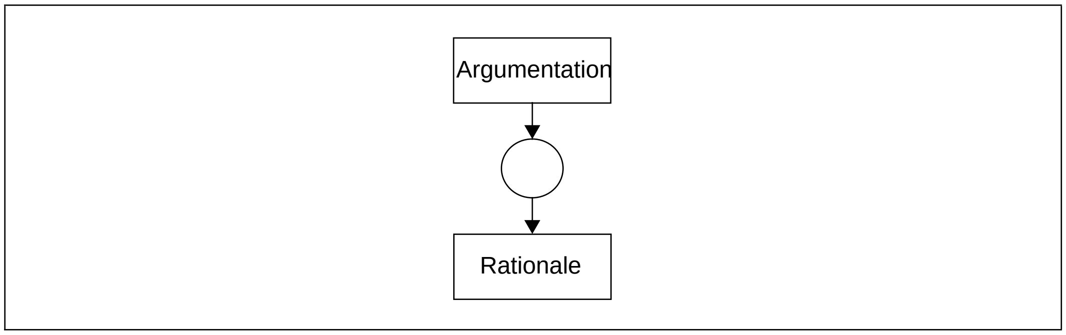

Figure 4. The production of a rationale.

The process

The process of reasoning about the product identifies or creates a sub-structuring of the task to be performed. These sub-tasks can themselves be seen as MUSA tasks with the same structure of specification, argumentation and plan, with as product some sub-part of the final product. The sub-tasks can be tasks that must be performed sequentially, when the later tasks need the output of an earlier task as input (the example above with the production of detailed specifications from a high-level problem statement is an example of this), or they can be tasks that can be processed in parallel, when each sub-task is independent of the others, followed by a merge step (see Figure 5.)[2]

The structuring of the process is also reflected in the argumentation: if the main task has been split into three sub-tasks, then there will be a section of the argumentation based at this level, extolling different possible divisions of the task, and explaining this decision, and then further discussion will be directed at the lower-level tasks. A similar sub-division will be found in the plan.

1. Note that the argumentation is not itself a MUSA product, since it is not the product of a MUSA production cycle but a by-product. It can be used however as the specification for a formal rationale of a product (see Figure 4.)

2. Note that the splitting of a task into parallel sub-tasks doesn't demand that the tasks be executed in parallel: that is only decided in the planning stage when assigning resources to each sub-task. The parallel sub-tasks only show that they *can* be executed in parallel.

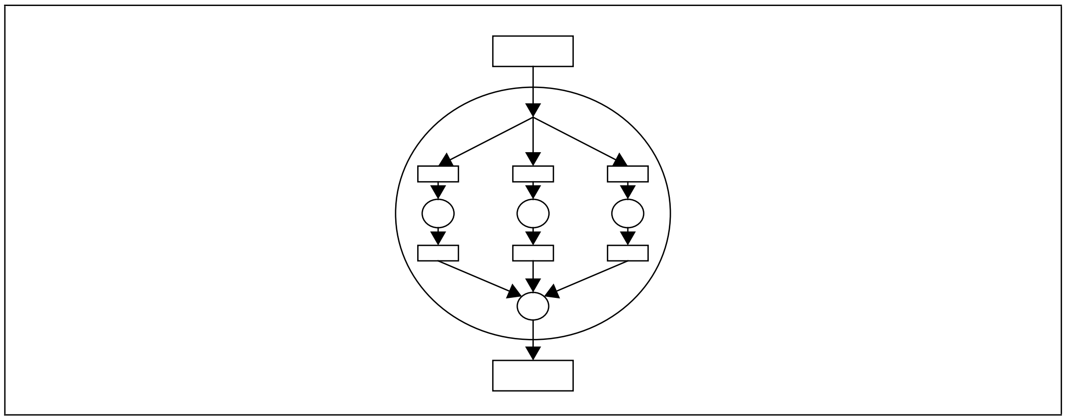

Figure 5. Parallel sub-tasks with merge.

The structuring of the process, then, is driven by the argumentation; but the argumentation is as already pointed out, subject to change and revision, due to changing requirements or circumstances; consequently the process structure isn't a fixed entity, but changes to match changes in the argumentation.

The plan

Once the structure of the process is decided, resources need to be allocated to the sub-processes: personnel, time, and possibly equipment. The plan has exactly the same sub-structure as the process itself, with each sub-task being allocated resources in the same way. The plan is the only place where the notion of time appears.

The simple structuring method used in breaking up tasks (serial decomposition and parallel decomposition) simplifies the allocation of resources to tasks: with serial decomposition the end date of one process is the start date of the next; in parallel decomposition the start date of the merge step is the latest of all the end dates of the to-be-merged tasks. Time allocation can be done bottom up — by allocating amounts of time to each sub-task, and specifying a start date for the whole — or top-down, by specifying a start and end date for the whole and working inwards. Computer support can then be used to identify slack times, or places where insufficient time is available for some sub-task.

The whole

Since so many of the MUSA elements can be sources or targets of further MUSA cycles, and each MUSA cycle can be split into further sub-cycles, the question arises: *where does it all end?*

Planning is only necessary to organise human tasks that are otherwise too complex to carry out in a number of simple steps. Tasks like "*Make a cup of coffee*" or "*Write a function to output the symbol table*" or "*Make a reservation at the hotel*" are sufficiently simple that they don't need to be further sub-divided or planned. So the aim of the MUSA process is to repeatedly sub-divide the production task to a level where no further subdivision is necessary, so that at the lowest level you have MUSA diagrams with no further sub-structure, and no more complex a plan than the start and end times.

4 Some examples

To illustrate the MUSA production process, there follow a number of examples. These examples have been chosen in most cases to focus on one particular part of the process.

The design of a file format

The problem statement is the following. We have a repeated need to send text files by electronic mail. The text files in question will often have lines very much longer than that accepted by some mailers, but there are no other properties that current mailers are unable to handle (such as very long messages, or control characters). Write a pair of programs to pack and unpack the files so that they arrive undamaged.

This task can immediately be split into two sub-tasks: *“Design a file format for packaging the files”*, and “*Implement the programs”*. We will focus on the first.

The first task is to brainstorm possible solutions, and to organise the possibilities into issues and positions, using the IBIS method.

The top level **issue** in this case is: *How should message packing be implemented*, with two **positions**: *Use existing tools*, and *Write new tools*. The first position in its turn has several possible pro/contra **arguments**, such as *For: less work for us*, and *Against: the tools must be available at all sending and receiving sites*. The *Use existing tools* position itself raises a new issue: *Which tools should be used*, with itself more positions and those positions with arguments, and with more sub-issues. This structure can be seen as a tree (see Figure 6.)

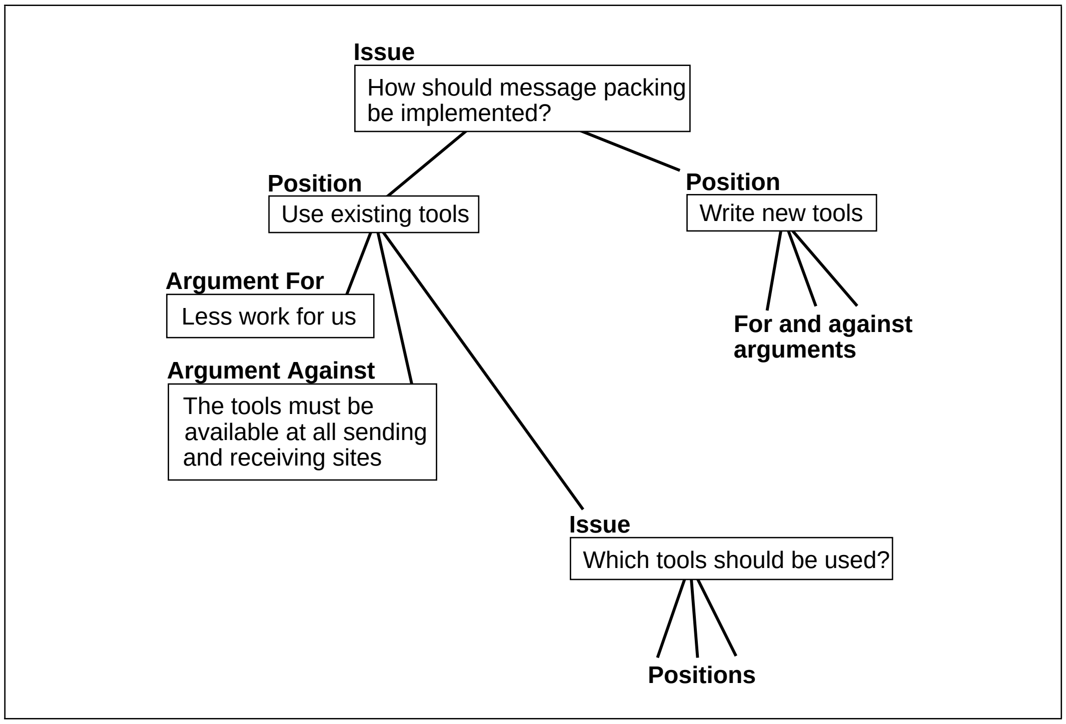
Issue
How should message packing be implemented?
Position
Use existing tools
Position
Write new tools
Argument For
Less work for us
Argument Against
The tools must be available at all sending and receiving sites
For and against arguments
Issue
Which tools should be used?
Positions

Figure 6. File design argumentation.

Of course, the tree doesn’t have to be fully worked out: at some point a decision may be obvious, or the alternatives unattainable, so that the tree can be pruned as it were, and only the useful parts fully worked out.

The advantages of organising the discussion this way have been discussed elsewhere, but briefly: it tends to keep the discussion on track, not easily allowing diversions or the use of tactics such as name-calling or argument by repetition, and tends to make missing alternatives obvious. More importantly, the explicit argumentation hierarchy permits easy browsing, and supports decision re-evaluation: you can see why decisions were taken, and the consequences of changing them.

A compiler

This example shows the splitting up of a task into lower-level tasks. The task is: "*Write a compiler for some given programming language*". The requirements include the definitional report on the language, and the product will be a compiler for the language (see Figure 7.)

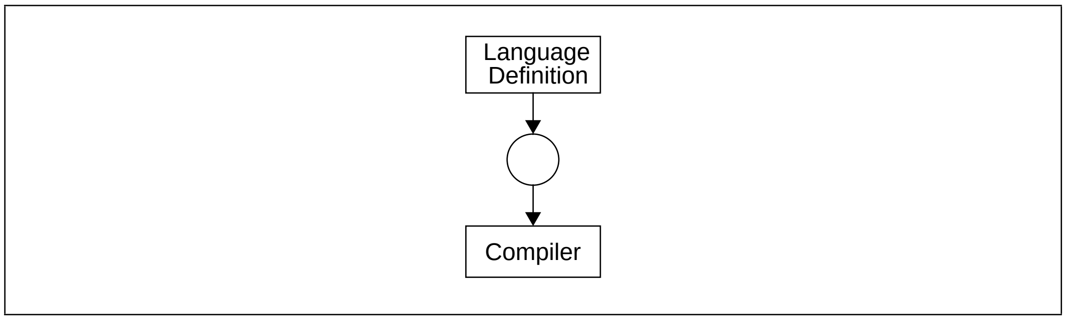

Figure 7. Producing a compiler.

Structuring the process cycle, we can split it into two sequential tasks: *"Design the functional specification for the compiler*", and *"Implement the compiler according to the functional specifications*" (see Figure 8.)

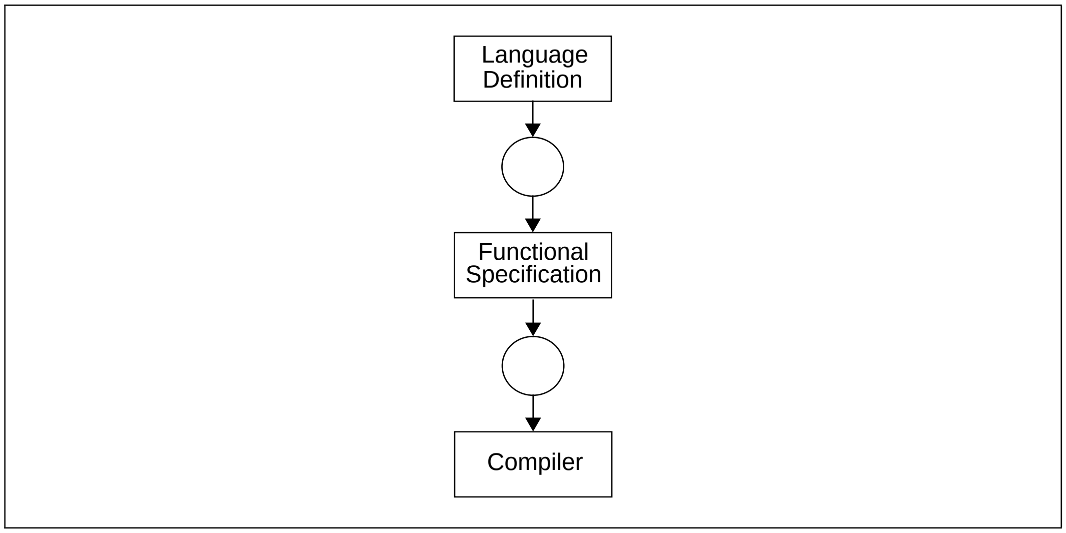

Figure 8. Compiler sub-tasks.

The functional specification in its turn may specify that the compiler has three passes, and then define the intermediate file formats between the three passes. The *"Implement compiler"* task would then be split into three independent tasks, followed by a merge (see Figure 9.) Note also here that there would have been a similar decomposition in the design of the functional specification.

If we keep breaking open the processes we finally come to a level such as *“Implement the function to insert an element in the symbol table”*, for which there may be no need for further sub-division.

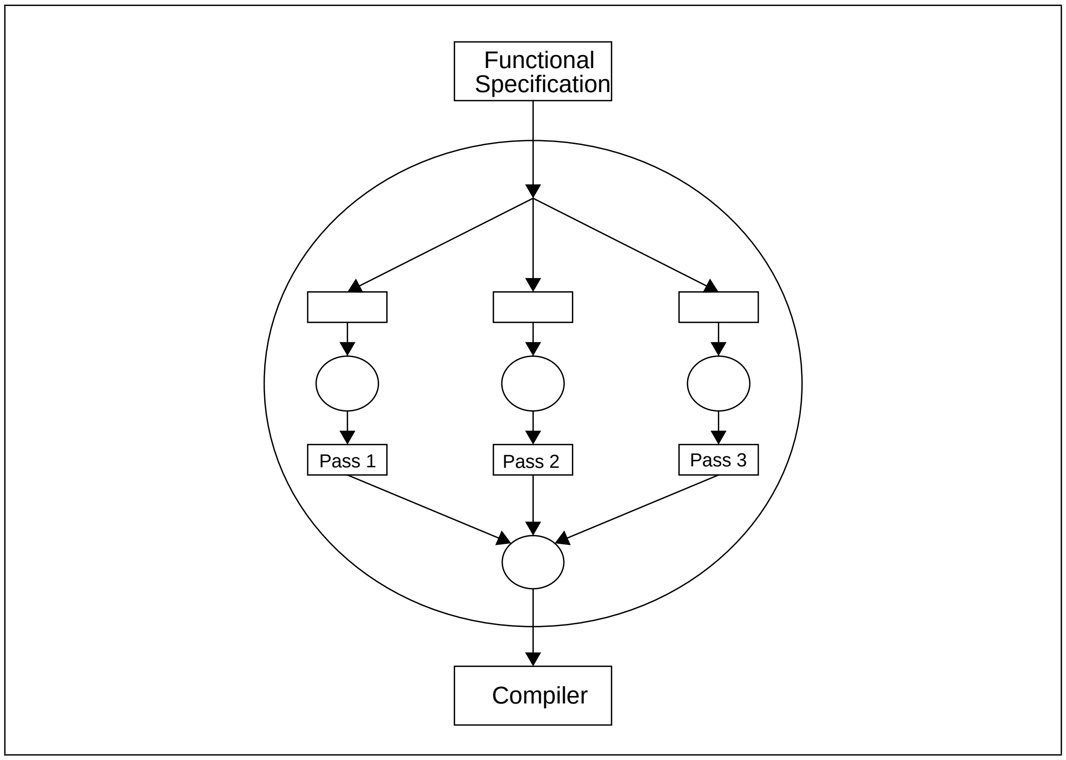

Figure 9. Implementation sub-tasks.

A travel plan

While the examples up to now have dealt with computer-related problems, the method isn’t restricted to them, but can be applied to general design. This example demonstrates this, and also demonstrates incrementally building the solution.

The problem statement here is: a researcher is going to a conference in the United States, and wants to take advantage of the trip to visit several companies and universities. Design a travel plan for the trip.

As with the other problems, the first step is to get a handle on the problem by finding the underlying structure of the solution. In this case, we need to initially fill in more detail in the specification: when is the conference, where is it, how long does it last, how many other sites need to be visited, where are they?

This is obviously a process of producing a more detailed specification. In the first case the researcher only has a list of sites she would like to visit. There are also some added constraints not specified in the high-level specification, such as: keep costs as low as possible, total time in the USA may not exceed 3 weeks, travel time should be minimised where possible.

These parts of the specification are likely only to come to light when analysing the pros and cons of different decisions, and this highlights one of the advantages of the IBIS stage: while a major advantage is structuring discussions and decisions in a group context, it is just as useful for a person working alone by making reasons and decisions explicit.

Suppose then that the IBIS stage has discovered that the conference is in New Orleans, and that the companies can be split geographically into West Coast (Silicon Valley), and East Coast (Boston and New York).

The travel plan can now be split into 3 (related) sub-tasks: a travel plan for New Orleans (where the dates are fixed), and travel plans for the West and East Coasts (where the dates are flexible); similarly, the East Coast plan, in order to satisfy the requirement to minimise travelling could be split into two sub-plans for New York and Boston (see Figure 10.)

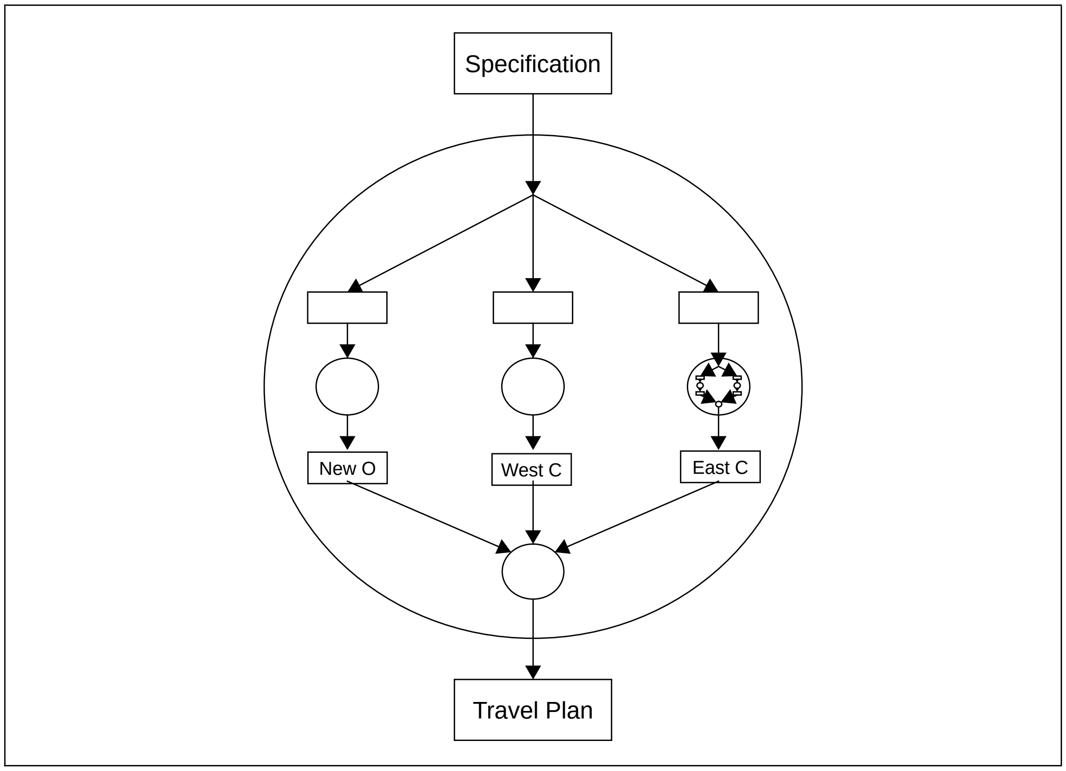

Figure 10. The Travel Plan.

At this point, the companies can be approached for possible dates, and as replies come in, the travel sub-plans filled in, allowing you to incrementally build the product.

The travel plan is actually then the specification for a new task “*Make bookings for trip”*, and this may feed back to the specifications for a revised version of the product, for instance if all flights are booked between New Orleans and San Francisco on the required day.

5 Computer support for the MUSA process

Up to now we have emphasised the methodology. However, there are many places where the MUSA design process can be supported by the computer, and this is the real reason for the existence of MUSA.

Here we present the ways in which the computer can support the method, using the Views environment as production platform.

Views

Views [2] is an open-architecture computing environment designed to make new applications easy to add, and all applications easy to use, by addressing some of the central problems of multi-application computing environments.

Diversity of user interfaces

In current workstation environments, the user is confronted with many applications. Typical usage involves swapping between applications at unpredictable moments: it is rare that one sits for hours running one application, then stops that application and starts another. More usual usage is to swap between applications: as someone comes in asking for some information; as someone phones and you have to refer to some files; when you want to include some information from one application in another.

It is also usual that each application has its own user interface, and that swapping between applications means that you have to consciously remember the context you are in so you use the correct commands for the current application. Even in environments where the user interface is standardised, such as the Macintosh, interfaces differ, sometimes in large ways, sometimes in annoying little details.

This diversity is due to the fact that the user interface has to be separately implemented in each application, using some toolbox of user interface tools or gadgets.

Applications also separately implement the import and export of objects (whether they be text or drawings, or whatever). This means that applications have to be aware of each other, and each other’s file formats, or that import and export has to be at a very low level, such as at the bit-map level, so that all structural information is lost. Once objects have been imported from another application, it is usually the case that they are frozen from then on: you can only delete them and reimport them, but you cannot change them in any useful way.

All this is accompanied by a huge programming effort for each application. Each application has to effectively reinvent the wheel just in order to get the basic elements of the user interface. The code just for uninteresting administrative aspects of the application can swamp the code for the true functionality of the application by a factor of 10.

Diagrammatically, you have the situation below (see Figure 11.)

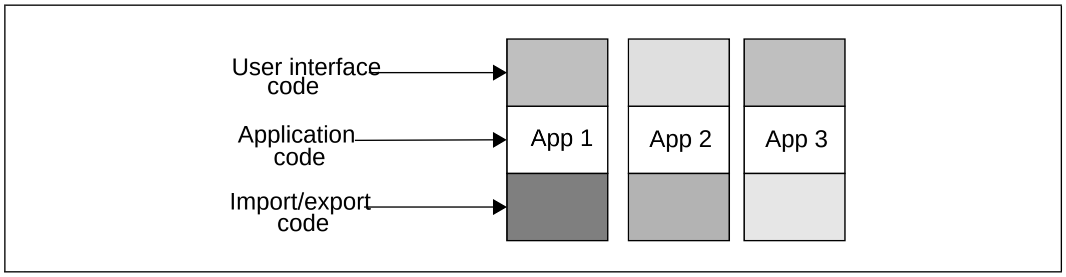

Figure 11. Current application organisation.

6 Views

Views is a computing environment with a system supplied user interface layer, and a system supplied persistent data layer (see Figure 12.)

This gives advantages:

- Since the user interface is localised in one place, each application has a uniform and consistent user interface.

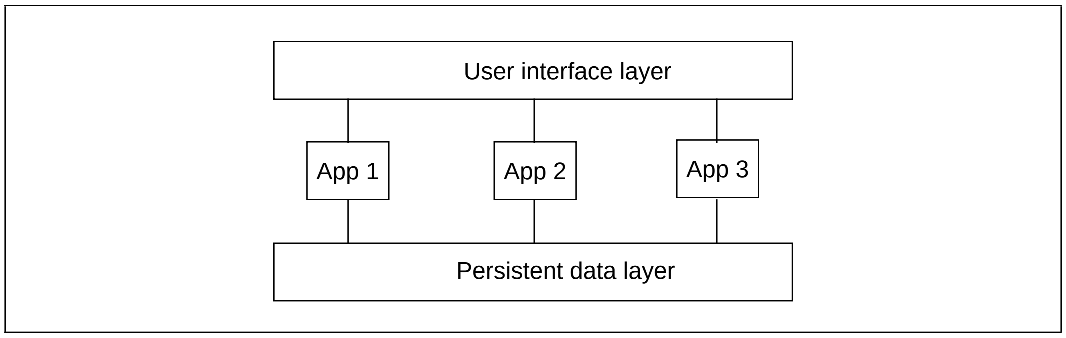

Figure 12. The Views application organisation.

- ♦ Once a user has learnt the basic user interface, it should be obvious how to run a new application.
- ♦ Since objects are managed by the system, they can be imported and exported structurally, meaning that they can still be changed in their new position.
- ♦ Application builders have much less work since they only have to deal with the functional part of their applications, the bit that does the real work (see Figure 13.)

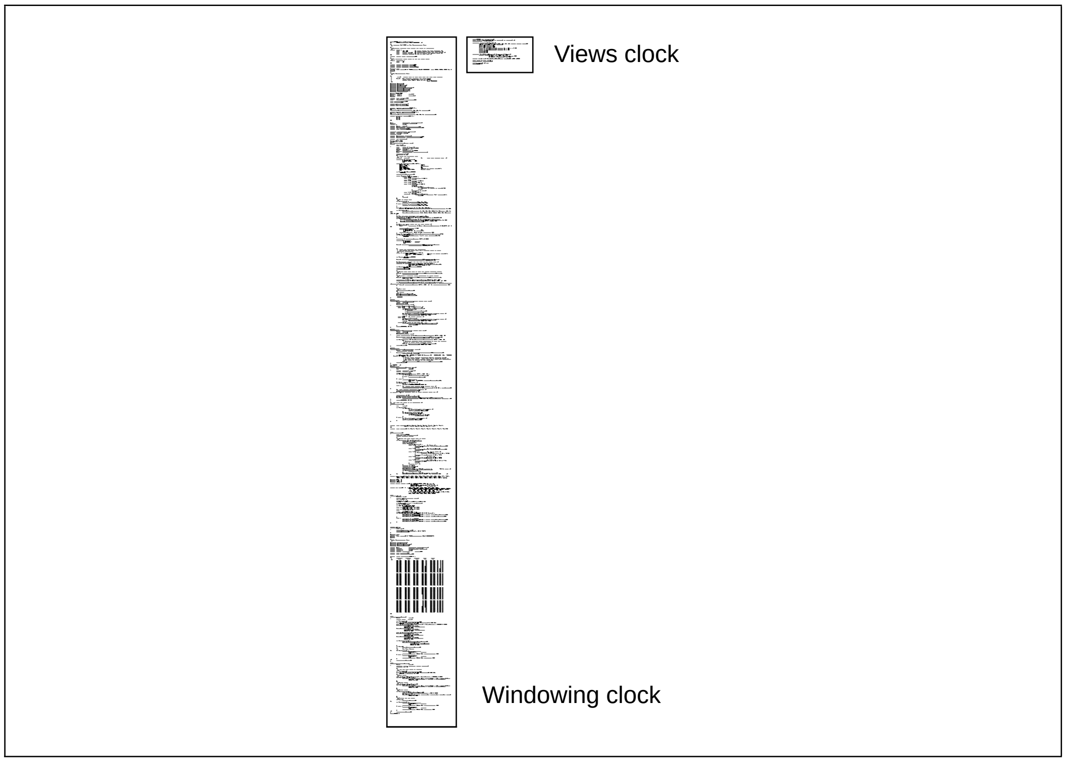

Figure 13. Traditional vs. Views Applications.

The user’s model

Since all applications have the same user interface, a model has to be found for a user interface that is suitable for all applications. Views has the following model, that we call the TAXATA (Things Are eXactly As They Appear) model:

- All information is presented to the user as ‘documents’, in a broad sense
- All objects are in principle editable
- All actions are achieved by editing
- The state of the screen exactly reflects the state of the ‘world’: this is a stronger form of WYSIWYG (What You See Is What You Get): as you edit objects, the ‘world’ gets updated to match, on the fly.

As an example of this model, consider filestore navigation and manipulation in a Unix context. There you have many different commands, with many different options. In Views, a directory would be an object that can be ‘visited’. This would display the contents of the directory on the screen, as a list of filenames, or graphically as icons, or however. Nested directories can be visited; ‘files’ similarly can be visited to be read or edited; entries in a directory can be deleted by deleting the entry with the editor; similarly files and directories can be renamed by editing the names; files and directories can be moved and copied by using the copy and paste facilities of the editor.

Similarly, a mail reader can present the mail messages as an index of unread messages, that can be visited, copied, deleted, in exactly the same ways. Similarly the printer queue can be modelled in the same way: documents are copied to the printer ‘document’ to be printed. Visiting the printer document presents the list of documents in the queue, updated by the system as the queue progresses; to delete an entry in the queue the user uses the delete operation of the editor in the same way. Thus once the user has mastered the editor, all applications work in the same way.

The implementation model

The persistent data layer maintains a world of data objects. Objects are hierarchically structured and contain only information: there are no details in the objects themselves of how they should be presented on the screen; in particular there is no distinction in the system between graphics and textual objects: they are just presented in different ways.

For each type of object, there is a document that describes how objects of that type should be presented. This document is of course also editable, which means that the presentation of objects is in principle accessible to users, and changeable on-the-fly.

Objects may be linked together by (two-way) invariants (a restricted form of constraints), so that if an object gets changed (by the user or otherwise) any attached objects get changed to match.

So, what an application builder has to do to build a new application is:

1. define the types of objects involved
2. describe the presentation for the new types
3. create the particular instances needed
4. define the invariants connecting the objects.

and the system does the rest.

If the user visits a document, the system causes it to be displayed on the screen; the application itself is unaware of any of this, or even the existence of the screen. If the user edits an object, the system ensures that any linked objects are updated to match.

The object and invariant mechanism is used throughout the system itself to implement many sub-parts. For instance, the invariant mechanism is used to drive the display: the object to be displayed and the presentation document for that object are linked by an invariant to a ‘screen object’. If either the object itself or the presentation document get changed, the invariant goes out of date, and the screen object gets updated.

As another example, the menus used in the system are also accessible as a document; editing it changes the menus on the screen: they can be renamed, re-ordered, deleted, the shortcuts can be changed, and new menus can be created.

7 Views for MUSA

Views is clearly an ideal platform for MUSA development: it already has a structured document architecture, with hypertext-style access, and wide integration between applications, and allows incremental and interactive definition and implementation of applications.

Apart from the use of Views for the actual creation of documents, Views applications will also be created to support IBIS-like discussions and browsing, and plan creation, browsing, and management.

The basic model sketched above is a single-user's view of the system. There is an obvious extension for multi-user access to documents for sharing and co-operation. Current work is on making a multi-user version where simultaneous access to documents is transparent as long as there are no clashes.

MUSA IBIS

The first place the computer can support the user in the design process is in the brainstorming and decision-making phase.

Reference [3] describes a computer-aided IBIS system. This system resembles a news-reader or bulletin-board system where users (the designers of the proposed system) post messages as part of an ongoing discussion about the product. The discussion is structured by enforcing messages to be of certain types that we have already seen: issue, position, argument. These messages are then related to each other by certain types of labelled arcs, and users may read the messages time-wise, as they are submitted, or they may browse in the graph of messages.

MUSA IBIS follows the general scheme of the original, but with some changes. In general the labelling of the arcs between IBIS messages adds no useful information: almost all messages are 'in response' to another message, and the type of that parent message already gives enough information to deduce the type of the new message.

As we have already pointed out, the IBIS graph largely forms a tree; it is true that some lower-level node may cause a higher-level node to be spawned, but that new node can always be places in a suitable place in the tree. MUSA IBIS therefore emphasises the tree-structure of the discussion hierarchy, and offers — in a folding editor manner — an overview of the hierarchical structure and ways of zooming in on nodes and discussions, and adding new nodes at relevant places.

Plans

Plans as we have described them are actually just views on the process structure (that is to say, different ways of presenting the information contained in the process structure). With each (sub-)task there is information about the resources involved, and the (planned) start and end dates. Clearly computer support can be used here, as mentioned earlier, to check that dates are consistent and to identify slack times; furthermore, when production starts, it can be recorded (in the process structure) as each task is started and ended. This gives easy browsing facilities to see how production is progressing, and can quickly identify bottlenecks and slippage.

8 References

[1] H. Rittel and W. Kunz, *Issues as Elements of Information Systems*, Working paper 131, Institut fur Grundlagen der Planung, Stuutgart, 1970.

[2] Steven Pemberton, *Views: An Open-Architecture User-Interface System*, in proc. int. conf. *Interacting with Computers: Preparing for the Nineties*, Noordwijkerhout, The Netherlands, 1990.

[3] Jeff Conklin and Michael L. Begeman, *gIBIS: A Hypertext Tool for Exploratory Policy Discussion*, ACM Trans. Office Information Systems, Oct. 1988.