Software Development and Reengineering outside of the IT Industry using a Procedural Workflow Framework

Wilfried Abels, TU Hamburg-Harburg, Hamburg/Germany,  w.abels@tu-harburg.de
Lars Greitsch, TU Hamburg-Harburg, Hamburg/Germany,  lars.greitsch@tu-harburg.de

Abstract

This paper presents an approach of a software framework to support software developing engineers by handling procedural software. The framework offers practical assistance for separating the control flow from the numerical functionality. Especially by long-lived and growing software tools, it allows to implement a flexible and clear documentable workflow, based on a configurable state machine. A rudder design tool illustrates the approach. This tool has been continuously developed over years with several changes of the responsible engineer.

1. Introduction

Much of the engineering software in use was developed for specific and highly specialized applications. The focus was in transposing physics into numerical engineering models. Especially for a prototype dominated industry (such as shipbuilding), it is very important to get reliable information about technical products within a short time. In contrast to mass production industries, the design stage of a ship is relatively short, Krüger (2003). The prototype character and strict time requirements impose rapid adjustments of the supporting software to the technical design process. As a result, there is often no commercial software package to solve the special problem and designers have to develop their own software to gain a competitive advantage.

Engineers are specialists for the technical product and the associated physics, but not for designing maintainable and scalable software. The programming skills of engineers are focused on numerical mathematics applied in engineering. The resulting software is mostly procedural, which works fine for individual problems. Problems occur when combining several stand-alone tools to more complex packages. At this point, software design, data storage and user interfaces become relevant. Many of the technologies available within the IT community are too difficult for the engineers responsible for developing the engineering design tools.

This paper describes an approach to support a software engineer to structure technical software tools, using a module to describe the dependencies between single numerical functionalities and separated user interactions. This task is performed by a free configurable state machine. This state machine allows separation of the structure and workflow from the functionality. Besides the structuring of new projects, a main focus is to reengineer existing software tools. This aspect will be illustrated for a rudder design tool in chapter 3. This tool has been developed over many years and is used successfully in industry and academia. It is continuously updated incorporating latest research.

2. Concept of a configurable state machine

Typical engineering tasks are procedural. An example is the design of ship rudders. Here, a CAD model has to be designed, the hydrodynamic aspects have to be calculated and the structural strength has to be analysed. For all three steps, dedicated software support is needed. The three tasks is a largely separate. A separated task will be called “functionality” in the following. Functionalities may differ in complexity, starting from a simple empirical function expressed in perhaps ten lines of source code to complex CFD (computational fluid dynamics) methods. Further functionalities address user interactions and data import/export.
The design engineer sees these as software tools to support him in his technical design process. He cares about the handling within his work process, not about the underlying complexity. Functionality must be used in the correct order. They frequently need output from other functionalities and generate in turn input for the next. The work process can be seen as a sequence of functionalities, where each functionality has a specific position. The order can not be changed at will.

This sequential way of interacting with software, is often used to guide a user. The software paradigm is a wizard based on a modal dialog. The user has only one interaction window and can deal with the problem step by step. Examples for this approach are the configuration tools for computer hardware, applications for online tax computation or online shops in the internet. A design process for a technical product can be structured in the same way.

![Fig.1: Example of state oriented workflow](image)

A classical state machine, i.e. a graph consisting of nodes and directed edges, Fig.1, is well suited to map these dependencies. The nodes represent functionalities, the edges possible ways to follow functionalities. This approach allows defining a network of functionalities describing the workflow in a design process. The approach allows separating the structure of dependencies from the numerical functionality. The restriction to a state machine ensures a clear interface between functionalities and the structure of the workflow. Each functionality is strictly isolated from all others. Only a selector specifies the directed edge, which defines the connection to the following functionality. But the functionality does not know which functionality the selector is connected to. A wizard implements three possible actions:

- NEXT : the next functionality should be executed
- BACK : the user wants to go one step back
- END : the user wants to cancel the application

Now we have two fundamental aspects needed to implement a generic state machine for a configurable workflow:

- A SELECTION : this is an integer number (1:n)
- An ACTION : one element from the set of [ NEXT | BACK | END ]

One problem of this approach should not be hidden. Because of the strict separation between functionalities, they cannot exchange data directly. Thus data interactions have to be implemented via the project’s database. Sometimes this is more complicated than to implement special parameter lists for a procedure. If we see a functionality as a special tool to manipulate the information of a project, this approach is consequent, because manipulations on local data have no effect on the actual project. I.e. a functionality, that does not manipulate the project’s database, has no effect on the design process.
2.1. Implementation of the generic workflow module

The approach was implemented as a FORTRAN90 module, consisting of some 500 lines of source code. This small implementation can be regarded as a concept demonstrator, which can be adjusted to different software environments. The practical use will be demonstrated in the following. A workflow is defined as a FORTRAN90 type definition. In a similar way, the concept can be implemented in any other computer language with type or class definitions. A workflow can be initialised by defining a variable of the following type:

```fortran
  type ty_Workf
    ... integer :: actualState,
    beginState, endState
    integer :: action, selection
    ... type(ty_State),
    dimension(1:WORKF_MAX_STATE)::
    StateList
  end type ty_Workf
```

This extract of the type definition illustrates how workflow acts. The most important part of this type is the integer array StateList. Each state within a workflow has to be a unique identifier. This identifier is implemented as an integer value. Additionally the attribute actualState is needed to specify the active state within the workflow. The attributes beginState and endState are used to define begin and end of a workflow.
The generic interface is defined for each functionality as follows:

```fortran
subroutine Func_1(action, selection, error)
    integer, intent(inout) :: action
    integer, intent(out)   :: selection
    integer, intent(out)   :: error
end subroutine
```

If the interface of functionality is strictly defined in the same way, these functionalities can be handled by a generic workflow module as shown in Fig.2. The `action` is going into the functionality and `selection` returns, which can be used by the workflow to get the next state. The identifiers of all possible states are stored within the array. By using the variable `selection` the following state can read from this list. Corresponding to the returned `action`, the next state will be the selected following state, the last state or the end state of the workflow. If `action` is ‘NEXT’ the actual state will be stored to an internal stack. This stack allows retracing the complete history of executed states.

A problem by implementing a generic workflow in FORTRAN90 is that the workflow does not know the calls to the functionalities it has to use. In object oriented languages as C++ or Java, there are mechanisms of abstract classes which could be used within such generic modules. The implementation of own functionalities could be realised by a derived class. In FORTRAN this is impossible. But as the focus here was to support engineers with existing software environments, FORTRAN was chosen, because this language frequently is used in engineering projects. Therefore a solution was necessary to handle abstract functionality. The solution was to demand the implementation of a subroutine for calling the functionality. While admittedly not elegant, this solution is practical. When the workflow should be started, the subroutine for calling the user implemented functionalities has to be passed by the parameter list to the workflow module:

```fortran
type(ty_Workf)  :: wf
...
call WORKF_run(wf, subState, ierr)
```

The parameter `subState` is a subroutine programmed by the user of the workflow module. This subroutine is the one called within the endless loop shown in Fig.2. The statement “exec actualState” is handled by calling the procedural-parameter `subState`! It is important to implement this subroutine correctly. Otherwise the whole workflow cannot work. But the implementation is easy and has only the aim to fulfill formal aspects. The structure of such a state manager is the same in each case, as shown in the following example:

```fortran
subroutine subState(state, action, selection, ierr)
    integer, intent(in)            :: state
    integer, intent(inout)         :: action
    integer, intent(out)           :: selection
    integer, intent(out)           :: ierr
select case(state)
    case(state_function_1) :
        call function_1(action, selection, ierr)
    case(state_function_2) :
        call function_2(action, selection, ierr)
    case(state_function_3) :
        call function_3(action, selection, ierr)
    ... 
    default : ierr=1
end select
end subroutine
```
end subroutine subState

This convention allows storing all needed functionality in complete isolated parts. The control flow is strictly separated from the numerical tasks. New functionalities, which are implemented later, do no affect existing ones. The control flow is only handled by the workflow. Because of the strict separation of control flow from the functionality, it is necessary to configure the connections between all functionalities independently. This means before the workflow can be started, an initialization part must be implemented. The following subroutine can be used to add a functionality as a state to a workflow:

subroutine WORKF_addFuncState(wf, stateNum, descr, nSel, sel_states, ierr)

The most important information is within the array sel_states. By defining this array, the connections between a state and the following states are implemented. After “adding” all functionalities to the workflow, the whole directed graph of the workflow is configured. Because the initialization of the directed graph is strictly separated from the implemented functionality, restructuring or extending the workflow is now simple. Only some following states within the array sel_states need to be changed to change the control flow. The real functionality has not been touched. The error potential of restructuring an existing control flow is significantly reduced. This is particularly beneficial for old and long living software systems. In chapter 2.3 this aspect will be discussed in more detail.

Finally some words about the parameter descr. This parameter is not really necessary for the workflow module. It is only a textual name for the added state. But it is very useful to describe the state with a meaningful name. The workflow module uses this string for logging the state before executing it. With this logging file it is possible to analyze exactly the way of the workflow through the directed graph of the state machine, a useful feature for debugging.

2.2. Principle implementation of a functionality

After explaining the workflow algorithm and the calling mechanism of functionalities, the functionality implementation shall be explained in more detail. The paradigm of a wizard controlled user interaction leads to two types of functionalities:

1. GUI functionalities which allow an interaction with a human user of the workflow.
2. Numerical functionalities for all calculations.

From the view of the workflow, they are the same. Therefore both have to handle the same mechanism of action and selection. But depending on type, they have to be handled differently. Especially the input/output parameter action has to be explained in more detail. As defined, it can have the values ‘NEXT’, ‘BACK’ and ‘END’. The value ‘END’ will be never passed to a functionality because this value is handled directly by the workflow module and has the task to end the execution of the workflow. Therefore the value ‘END’ is only possible as a return value of a functionality. This means, a functionality has only to handle the values ‘BACK’ and ‘NEXT’. ‘NEXT’ means that the functionality should execute its normal task. The value ‘BACK’ is a bit more complicated as an input parameter for action. In this case the functionality has to know how to react. For of a numerical functionality, the task is often simple because the functionality has nothing to do but to return with action ‘BACK’. The workflow will go automatically a further step back.
To understand this mechanism, remember the handling of a wizard dialog. If a user interacts with a wizard dialog pressing the back button, he expects to return to the last user dialog. All numerical actions between two user dialogs must then be ignored. Only if a functionality has manipulated and stored project relevant information, an undo mechanism has to be implemented sometimes. But in most cases, a simple statement is enough:

```python
if action == BACK then return
```

The action ‘BACK’ within a GUI functionality is handled as follows: If a user presses the back button, the last GUI has to appear showing the last configuration the user did. This means the GUI functionality has to initialize all entries with default values if the functionality is called with the action ‘NEXT’ and with the last values if it is called with ‘BACK’.

The `selection` parameter can be used freely by the programmer of a functionality. It defines how many output items a functionality has. A numerical functionality has normally only one output item. A possible second output item could be a non-fatal error. In this case it could be reasonable to pass the user to a special GUI functionality for handling such an error. But the normal use of the `selection` parameter is to use it within a GUI functionality allowing the user to select different possibilities within the wizard driven workflow.

### 2.3. Using the state-machine approach for reengineering

The approach of separating the control flow from the numerical tasks has a further advantage. By designing a new software application, the focus is to implement the required numerical functionality. The software structure can be designed from scratch. Aspects of software maintenance and possible future extensions are important aspects for the implementation and they can be realized by regarding the actual state-of-the-art of science and technology.

But in engineering, often software is not designed from scratch. Existing tools have to be improved or adjusted to new problems. In this case, the existing structure of the software has to be taken into account. It is impossible to re-implement the complete tool. In practice, several developers implement different parts of the tool. Each individual works on some feature to improve and soon it becomes difficult to understand all existing features. Because new developers have no longer the complete overview, the software structure degenerated increasingly. Every new person has more problems to implement his task and the effort for new features becomes uncontrollable.

At this point there are only two possibilities. The first one is to delete everything and to start from scratch. Normally this is impossible due to excessive required financial and personal resources. The second one is to restructure the existing tool and to implement a new maintainable structure to the existing software. This approach is still expensive, but often there is no other way than to reengineer an existing tool. After the initial decision for reengineering, there is the question of how to do it. A seeming trivial demand is that the reengineered software has to calculate the same values as the original. But in practice it is difficult to ensure the same behavior after a complete restructuring of the source code. Here the approach of separating the control flow from numerical parts is very useful. Often differences in behavior result from small changes to the control flow, not taken into account. It is very difficult to find these errors. But the control flow is often a main aspect of a reengineering project. Developers tend to implement little features by putting some functionality within a source code and this code is only executed under special conditions which need to be understood. After analyzing this behavior, a main part of the work is done. For this task, the careful separation of the control flow is useful. If you try to extract too many control sequences in one step, errors are almost guaranteed. Therefore an approach is needed to support stepwise restructuring, namely employing by the state-machine base workflow. Because the complexity of a functionality is not defined, it is possible to put the complete existing application in one functionality and the workflow is only used to start it. In this case there is only one state and after execution, the state machine terminates. Because
there is no new functionality and no changes within the existing source code, the application will act in the same way as before. But now there is a starting point for further reengineering tasks.

In a second step, it is possible to analyze the primary ways of the control flow. Mostly it is easy to find some primary paths. Perhaps there is a special initialization part, which is executed before the main calculations. This is a typical candidate for extraction to a separate functionality. In the same way, a post-processing part for storage and visualization can be extracted to a separate functionality. The first trivial functionality can be cut stepwise in two, three and more functionalities. After each cut, it is possible to check whether the reengineered tool still acts as before. New functionalities can be analyzed whether a further separation is useful or not. In combination with the automatic logging mechanisms of the workflow module, it is easy to control the steps of execution.

In the same way as a civil engineer has to brace a ceiling of a building before a wall can be destroyed, the control flow needs bracing, before the existing structure can be removed. This can be done by the discussed state-machine approach. The workflow module allows supporting the application during the reengineering process by keeping alive a fully functional control flow. Therefore a smooth extracting of the control flow is possible without long periods of a software tool without function.

3. Reengineering of an existing rudder-design software

Within the design process of a vessel the prognosis of the manoeuvrability is a very important subtask. The fulfillment of the requirements is essential for a successful delivery of the vessel. One major aspect is the calculation of rudder forces for different rudder angles and vessel speeds as input parameters for the manoeuvring simulation. Within the ship design framework E4 a panel code is used to gain these characteristic diagrams of manoeuvring forces for different design stages of the rudder regarding the influence of the propeller slipstream for different propeller loads. The calculations are validated in Abels and Greitsch (2008) and in Haack (2006). In addition to these force calculations the existing panel method has been extended for the prognosis of cavitation aspects for single operating points.

Due to latest research activities further extension of the method became necessary. The aim was the prognosis of the risk of cavitation within the operational profile of the vessel of interest. Thus the cavitation risk distribution has to be calculated for a huge number of operating points. This new feature requires the implementation of a new batch mode in combination with a modified adjustment of the propeller load state. The rate of modifications led to the necessity of an extensive restructuring of the program.

3.1. Rudder force calculation

For calculating the rudder forces in the slipstream of the propeller the correct gathering of the propeller model is essential. This is carried out by a sequential procedure of user interactions and calculating functionalities. An overview about a typical workflow of a rudder force calculation for a single operating point shows figure 3.

Steps of the workflow printed on grey bars represent user interactions; green bars indicate data operation or calculation. After initializing the database access and relevant variables the first GUI functionalities allow the selection of rudder type, rudder and skeg. With the present method full spade rudders, flap rudders and via additional corrections semi-balanced rudders can be calculated. After selecting the rudder and skeg of interest, the geometrical data is prepared.

The next set of GUI functionalities deal with the selection of hull efficiency and resistance curves and the choice of the considered propeller. Subsequently the operating point of the propeller for the entered vessel speed is determined. After generating the calculation grid for a chosen rudder angle the velocities on the rudder within the slipstream according to the resulting propeller load can be
calculated. The pressure distribution on the panel grid derived from the velocity distribution lead to the quested rudder force.

This schedule is exemplary and shows the adjustment of the influence of a fixed pitch propeller (FPP) on the fluid flow around the rudder. In the FPP case the equilibrium position regarding the propeller thrust is found by varying the propeller revolution speed; in the case of a controllable pitch propeller the adjustment of the thrust is carried out by changing the propeller pitch.

3.2. Restructuring the established CFD code

In order to realize the necessary extensive modifications a successive restructuring of the existing CFD code has been pursued. For that reason the software has been cut into some manageable well-defined subroutines. For keeping the method executable the variables which are needed by more than one subroutine has been declared in a declaration module as global variables. So the parameter passing was reduced to one logical variable for the identifying of errors in the subroutines.

In the next step the execution chronology of the subroutines has been implemented with the new workflow approach. The parameter passing now has been extended by the selection and the action parameter as introduced above. The option of setting the selection parameter now allows crossroads in the execution chronology. The reconstruction of the complete method with all reasonable branches has become possible.

Step-by-step the complete method has been cut into more granular subroutines with separated functions. All user interactions have been separated from the calculating or data operating parts. The identifying of a possible value ‘BACK’ of the parameter action is captured within the first line of the numerical functionalities, which allows the replacing of all goto statements. The ‘BACK’ command of the user is carried out at every location of the code by going back to the next user interaction. An enhancement with further functionalities automatically fulfills this mechanism. It’s not longer programmer’s task to adjust the goto statements.

![Workflow of a typical rudder force calculation](image_url)
3.3. Crossroads within the workflow

The presented workflow concept allows a reliable upgrading of the method in terms of additional features without daring the existing performance. As an example the implementation of a new rudder type is demonstrated in the following. After selecting the rudder type for the calculation (figure 3, GUI A) the workflow is divided into different lanes for the further run. In case of a flap rudder additional information is needed to be entered by the user. Afterwards the workflow is continuing with the data base access (FUNC A) like in the default case. The existing rudders and skegs in the data base are listed in the next selective user interfaces (GUI C and GUI D). The information which rudder type was selected in the beginning is saved within the variable rudder_type, which can be interpreted at every other position in the procedure.

The matter of distinguishing different types of rudders regarding attributes, that have an effect on the geometrical modeling, stands exemplary for all supposable distinctions between different calculation cases. The decision whether the default rudder setting or the flap rudder setting is to be executed is identified by the different integer values of the selection parameter. The direction of termination is destined by setting the parameter action.

Fig.3 Example of a typical crossroad
The workflow module itself calls the state manager only with the parameter value of the variables selection and action. There is no more conjunction between workflow managing part and the functionality executing part. For this reason the implementation and extension of a workflow is only carried out in the method itself.

**3.4. Combining workflow functionalities with batch-mode processing**

The used CFD code for calculating the rudder forces is a suitable scope for combining a procedural execution as shown in figure 4 for a single operating point calculation and a batch-mode processing needed for an efficient use of the code for computing the complete characteristic diagram of manoeuvring forces for the investigated rudder shape.

Instead of a consecutive process in the case of a single point calculation the batch mode is executed in the functionality “Calc: rudder forces batch mode” in terms of two nested loops. This allows the calculation of all combinations of vessel speeds and rudder angles. Having a look at the state machine for the single point calculation all calls of functionalities are figured as bidirectional arrows, meaning that the information selection and action are passed between functionality and state manager and interpreted for the procedure. The calls from the batch processing illustrated as dashed arrows only point in one direction. There is no feedback and interpretation of the information selection and action. In this case the functionalities just function as subroutines. The workflow itself only realizes the functionality “Calc: rudder forces batch mode” as a workflow element.
3.5. Implementing the cavitation risk prognosis method

In order to calculate the cavitation risk within a given operational profile, the cavitation distribution on the rudder surface has to be determined for the specific operating points. The risk itself is calculated by weighting the operating points with their relative frequency. Unlike the procedure of calculating the required propeller thrust for the observed operating situation within the rudder calculation method now the thrust demand of the propeller is predetermined by a manoeuvring simulation in order to consider the environmental influences like wind and sea state.

It is easy to imagine that the changed procedure of determining the slipstream of the propeller within the specific operating points can be implemented in the simple way of exchanging the existing functionalities of calculating the propulsion equilibrium with a new functionality that prescribes the pre-estimated thrust demand from the manoeuvring simulation. The existing method features are irrespective from these modifications.

4. Conclusions

A new concept of reengineering procedural software has been developed. The new approach allows the step-by-step reconstruction of existing code into a sorted collection of functionalities, whose execution is managed by a state machine. The order and branching is defined in a workflow. This proceeding allows the extension of the existing methods in a very short time with a low risk of errors in the execution scheme. Thereby the software stays upgradeable. The capability of this approach is shown by means of the modification of a panel method for the calculation of rudder forces and the cavitation prognosis.

References

KRÜGER, S. (1999), Offenes Methodenbanksystem für den Propulsorentwurf, Flensburger Schiffbau-Gesellschaft mbH& Co. KG. Flensburg, Technischer Bericht

KRÜGER, S. (2003), The Role of IT in Shipbuilding, 2nd Conf. Computer and IT Applications in the Maritime Industries (COMPIT), Hamburg


ABELS, W. and GREITSCH, L. (2008). Using a bridge simulator during the early design-stage for evaluation of ship design manoeuvring ability, 7th International Conference on Computer Applications and Information Technology in the Maritime Industries, Liege, Belgium