IWCIT'01

A Design Pattern of the User Interface of Safety-Critical Systems

Zsigmond Pap, Dániel Petri
Department of Measurement and Information Systems
Budapest University of Technology and Economics

Abstract: This paper describes a design pattern for safety-critical user interfaces. This design is a version on of the Model-View-Controller architecture, but accepts safety and ergonomics criteria based on the literature.

Introduction

A safe system never processes such an operation, which can cause incidents, accidents, human or environment harm. Our life relies on more and more on the safety of embedded computer control systems, and as the complexity of these systems (especially the programs) increases, the assurance of the system safety becomes increasingly difficult.

Most often the accidents that are caused by computer, happens due to software faults or operator errors. These two reasons have two common parts:

- many of the problems occur in an unusual situation;
- the operator errors can be a result of an unadvised program behavior.

Since nowadays can the automated software designer tools highly help the process of software verification, the programs can correspond to the specification. Unfortunately when the specification has got errors already, the program cannot be good. This is why the verification of the specification is very important.

47 safety criteria for software specification has been published by N. Leveson. These criteria are mainly for the specification completeness and consistency, including to the user interfaces as well.

The specification can be written by using UML [1], the de-facto industrial standard modeling language of the object-oriented systems. Unfortunately the UML has not got any effective mechanism to specify the user interface of the program. Naturally somebody can use the Use-Case model form to specify the human-computer interface, but nowadays the programs have very difficult and complex user interfaces, so they need a more precise methodology, like the State chart or the Class diagram.

These methodologies are very effective, but have not got any support to specify user interface models. To verify the use of these criteria and general rules we need some additional constraints in the use of UML. If the designer uses a restricted form to specify the human-computer interface, the checker program can work more effi-

ciently, and some criteria do not need checking: the restricted form structurally satisfies them.

This paper describes a design pattern using UML semantics, which supports the design of the user interface of safety critical systems with applying many of the criteria mentioned above.

User interface problems

In every computer-human interaction the operator has a mental model of the controlled system. This model is different from the real state of the computer, especially the user interface. This difference, called the "semantic gap" is the main source of the operator errors, and some property of the user interface can increase this.

Based on real and simulated accidents of airplanes [2] N. Leveson and her team has identified the following factors as problem sources of operator errors:

- Ambiguous or misunderstood display, which highly increases the semantic gap.
- Mode changing as a side effect. The side effects –independently of their characteristic- are always very dangerous. They can cause unexpected situations, in particular when the operator has no feedback information about the unwanted operation.
- Unexpected context changing,
- Unexpected side effect at a function. They are asynchronous operations, which can cause slips, since the operator cannot recognize the context changing in time, and continues the intervention. This looks like as an inconsistent or indeterministic behavior.
- Inconsistent operation; In this case the operator gives a sequence of operations to the system twice from the same starting state, but the result in the two cases are not the same. Note that, the basic indeterminism (without unexpected context changing) has two types. The first is the mathematical indeterminism. In this case the automaton has a state from where starts a set of transition into another state at the same event, and a machine chooses one randomly. This type of indeterminism is well checkable.

  The second type is the virtual indeterminism. In this case the automaton is mathematically deterministic, but the operator cannot observe one of the internal state variables. As a result, the behavior of automaton can be different according to this hidden variable. For the operator this looks like an indeterminism.
- Lack of feedback of operator events. If the operator does not receive some kind of feedback information about the command, they will repeat it. This can result in unwanted multiple commands.
- Lack of operator rights. This means that, the operator should access one of the system functions, but he is unable to do this due to a problem. For instance the operator can switch ON the engine, but can't access the OFF menu.

Design Patterns

At the development of a system it is a natural demand not to plan it from the principles but reuse as much proven solutions and structures as possible. The design pattern is suitable for this purpose. It is intended to document a problem in a context, it

gives a core solution to this problem, which can be configured or customized, and it also names the consequences of applying the solution.

Design patterns have four essential parts:

i) The **name** of the pattern, which can describe the problem and the solution in one or two words.
ii) The **problem** introduces the context and describes when to apply the pattern.
iii) The **solution** describes the participants of the solution, their collaborations, relationships, and responsibilities. It is not an exhaustive solution and implementation to the problem but rather a template which can be applied in many different situations and contexts.
iv) And finally the **consequences**, which include the results and trade-offs of applying the pattern.

We chose the **Model-View-Controller pattern** [7] showed by the Figure 1 as a basis, because it is the most popular user interface pattern and many of the other interface patterns are using similar philosophy.

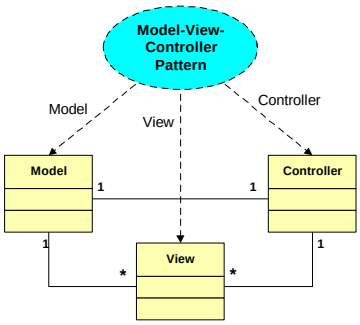

Figure 1. Structure of the MCV

The MVC pattern gives a solution to the following **problem:** Most of the applications handle some data (e.g. make decisions based upon them or run algorithms on them), display them to the user and receive user input, which influences the data. If these tasks are combined into a single object, they are tightly coupled which limits the reusability of the components and the changes in any part require changing the whole object. The MVC pattern shows how to decompose an application into orthogonal parts thus removes their tight coupling and allowing portability and reusability.

Solution: The participants of this pattern are: Model, View and Controller object (groups). The Model part contains the application objects; it holds the data of interest and updates the view and the controller if necessary. The View objects represent the data to the user and the controller objects receive and respond to events related to the view of the model.

Consequences: There are many advantages of this structure. If for example we want to change the way of the data that is displayed to the user, the modifications influence only the objects included in the View part.

The Safe User Interface Pattern

Leveson [2] and Paterno [4] have published criteria and rules for user interfaces. Based on these and the general rules of software ergonomics (see in [5]) our pattern should accept the following criteria-groups. Table 1. shows the detailed criteria.

- Using checkable form (for details see [3])
- Avoiding of operator leaps and misunderstanding
- Handling data modification and consistency
- Satisfying ergonomic criteria
- Time-related criteria

Description of the pattern and its functions

The **name** of our pattern is Safe User Interface pattern.

This pattern gives a core solution to the **problems** described above, and it is intended to help the developers to take care of the criteria and rules defined in [2] and [4].

The **solution:**

The structure of our Safe User Interface pattern can be seen on Figure 2. The collaborations and responsibilities:

Although this pattern is designed for safety critical systems, this architecture can be applied to other embedded systems too, where data manipulation functions and safety described below are achieved.

"Using checkable form (for details see [3])"

As it is based on the Model-View-Controller pattern, its participants (the objects) are divided into three groups (represented by packages). The most important object of the *controller* group is the UIHandler, which processes the user input commands received from the InputDriver and stores them in FIFOs according to the priorities of the commands. The OpEventFIFO is the common object, which handles the FIFOs.

The main part of the user interface is one or more final-state automaton, specified by the state charts in the ScreenHandler classes. The user interface can handle more than one ScreenHandler object. Such an object can be active or inactive. The setting of activity state is the task of the UIHandler object. The easiest method is the token-oriented operation: in the system there is a token, and that ScreenHandler is active, which has the token.

When active, the main handler class, the UIHandler can call the DrawSelf function and can send the keyboard events to this object.

"Avoiding of operator leaps and misunderstanding"

The objects shown in the *view* group are the interfaces between the user and the system. The ScreenHandler object holds display objects (e.g. a dialog box or the whole screen). It has a descendant called DefaultScreenwhere the system must return after an idle period. The task of the DisplayDriver object is to refresh the relevant information on the display device. This operation is also controlled by the UIHandler object.

When the data model is changing, the DataModel class sends an "Invalidate" message to the UIHandler, which forwards it to the active ScreenHandler objects, and they can refresh the information on the screen.

Every user event generates a "bip" sound (using the OutputDriver object), which is the feedback to the operator. The events can generate another sound if they were success or they were failed, and –as an alarm- some ScreenHandler object can play different sounds at the activation or refresh.

"Handling data modification and consistency"

Since the data model is handled by the controller and the user interface too, there must be a mechanism to allow mutual locking out. The databases use transaction-oriented data access to solve this problem. This method has the advantage of transaction rollback, which is important, when there is a problem with one of the user commands of a sequence.

The user interface can read data, but the write operation is dangerous, since the system can get into an inconsistent state. The best method is to allow data modifica-

tion only via user command messages, via a dedicated path (the FIFOs). This path contains one or more FIFOs to store the messages.

The commands stored in the FIFOs are represented by UserOp objects. The structure of this object allows handling coupled commands (via the TransitionNext pointer), which is important when canceling operations. The *model* part contains the data of the application. The DataModel object is responsible for handling the data (e.g. read, rollback) and the data is held in the StoredData objects.

"Satisfying ergonomic criteria"

In the embedded systems hardly ever exist a default screen. This is a special version of the ScreenHandler. The state chart of this class must have a "Default" state, which is the start state and the destination of the timeout transitions.

Most of the embedded systems have some special buttons, which are not usable for general purpose. These buttons should be handled on another level than the other buttons. To support this, the class ScreenHandler has two Keyboard handler functions: the OnKey for all keys and the HandleGenKey for the special purpose buttons.

"Time-related criteria"

The class ScreenHandler is responsible for timeout implementation when there is no user activity for a long time (using the member variable m_LastKeyTime and member function InternalTimeOut).

If the system cannot process one of the user commands, it can wait for a long time. To remove unaccepted commands from the FIFOs, every message has a time stamp. If a message is getting rather old, the FIFO can remove it, with the associated other messages too.

The data model object can drop the operator event (clearing it from the FIFO), when it comes too quick after the last data-change.

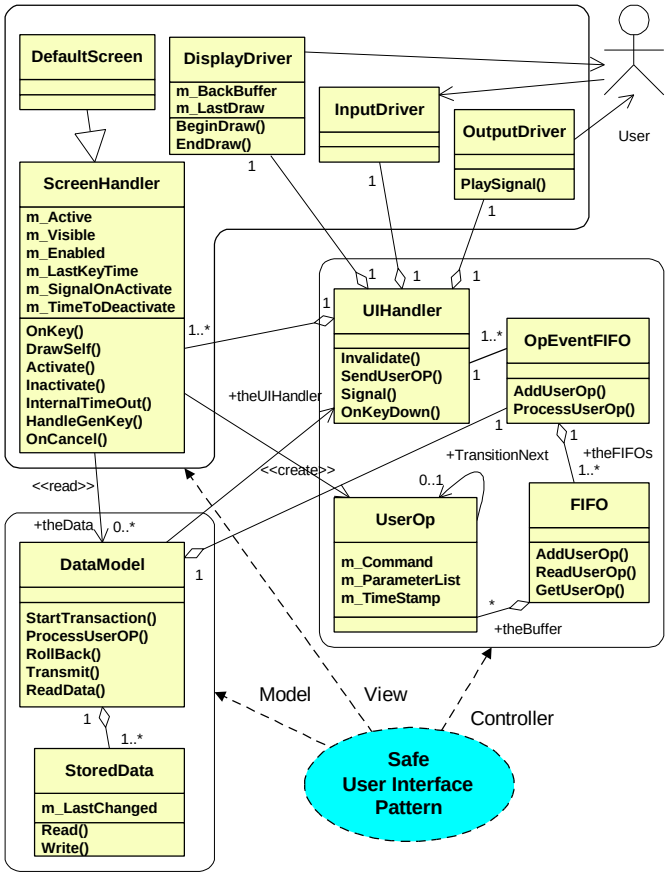

Figure 2. The structure of Safe User Interface Pattern

Problems solved by the pattern (the consequences)

Table 1. Properties of the design pattern.

Principle	R	S	P
using of state chart and class model (for UML)	✓	✓	✓
fully specified, and consistent.		✓	
for all operator event there should be a feedback mechanism	✓		
all internal states of the currently handled user interface mode should be displayed for the user (to avoid virtual indeterminism)			✓
no side effects allowed	✓		
avoiding the overloading of the operator	✓		
when the information is changing the operator should be warned	✓		
automatic data refreshing and clearing mechanism	✓		
the operator commands should not refer to external state variables			✓
the internal data model must be protected from direct manipulation	✓		
the operator commands should be grouped into roll-backable transactions	✓		
in every UI states, (except the defaults) should be a "Cancel" function		✓	
shortly after a data-change all operations on the data should be disabled	✓		
after a time of no user activity switching to the default screen		✓	
no automatic context switching, except in the former case		✓	
every operator event should have a time stamp	✓		

Legend: N = number = Reference number
R = realize = the structure performs the rule
S = support = the structure allow performing the rule
P = possible = the structure does not prohibit performing the rule.

Unfortunately the design pattern does not solve all safety-related problems, but at least supports them, and allowing the verification of the rule by external checker programs. Naturally the design pattern should be converted into the particular system, and this operator can modify the properties shown in the Table 1. Unfortunately the design patterns have no methods to protect the model.

References

1. Object Management Group: Unified Modeling Language Specification v 1.3. (1999).
2. N. G. Leveson: Safeware: System Safety and Computers. Addison-Wesley (1995)
3. N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese: Requirements Specification for Process-Control Systems. IEEE Trans. on SE, pp. 684-706 (1994)
4. F. Paternó, C. Santoro, B Field: Analysing User Deviations in Interactive Safety-Critical Applications (1998)
5. SHNEIDERMAN, B. 1987, 1996. Designing the User Interface. Reading, MA: Addison-Wesley. ISBN 0-201-57286-9
6. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley, Reading Mass. 1994.
7. Bruce Powel Douglass: Real-time UML: Developing Efficient Objects for Embedded Systems. Addison-Wesley Object Technology Series. 1997.