The Model-View-Controller (MVC)
Its Past and Present

Trygve Reenskaug, University of Oslo
(trygver@ifi.uio.no)

Abstract.

MVC was conceived in 1978 as the design solution to a particular problem. The top level goal was to support the user's mental model of the relevant information space and to enable the user to inspect and edit this information.

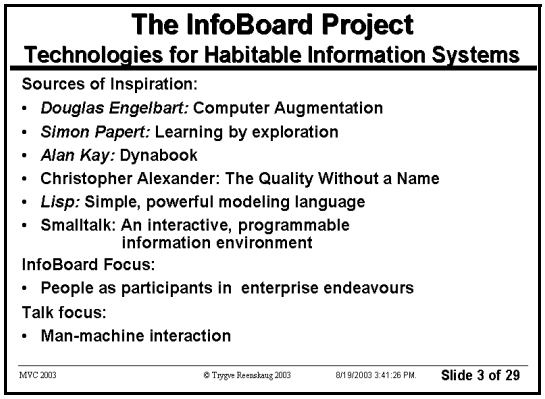
The first part of the talk describes the original problem and discusses the chosen solution.

The second part elaborates the original ideas and extends the scope to include current day challenges to the original goal. We examine some ideas related to MVC that are found in the literature and select those that appear to be particularly relevant to the top level goal.

It is all summarized in a condensed MVC pattern language.

1. Introduction

This presentation is part of the **InfoBOARD** project where I explore IT technologies that help create habitable information systems. The scope of this project is indicated by its main sources of inspiration:



This talk focuses on bridging the gap between man and machine.

2. An MVC Pattern Language

The following is the first draft of a pattern language for a systems architecture based on the MVC ideas.

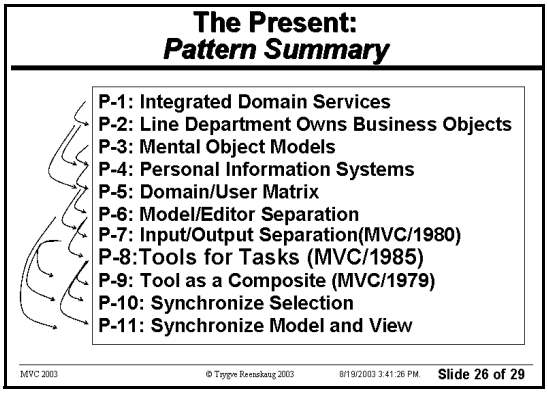
The patterns may not satisfy the stringent requirements set up by the patterns community. In particular;

- Several patterns represent ideas rather than implemented experience
- References to interesting, public domain patterns are TBD (To Be Done)

- The patterns have been written by a single author and have not been discussed in the community

On the other hand, all the patterns are in Alexander's spirit; every one is motivated by the needs of people and the desire to create habitable information systems.

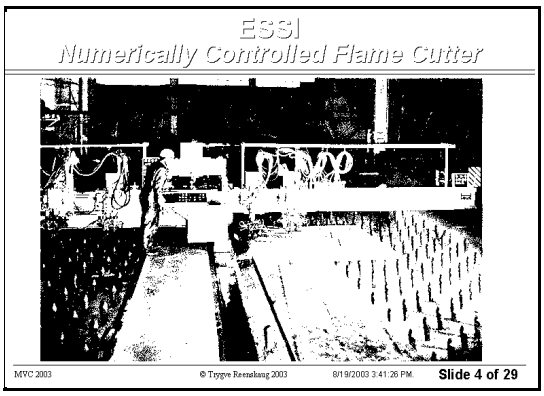
The patterns of this draft version of the MVC pattern language have been developed together with the presentation. The patterns are therefore decorated with the corresponding presentation slides.



This first pattern language consists of the following patterns:

- P-1: INTEGRATED DOMAIN SERVICES (page 2)
- P-2: LINE DEPARTMENT OWNS DOMAIN COMPONENTS (page 3)
- P-3: MENTAL OBJECT MODELS (page 4)
- P-4: PERSONAL INFORMATION SYSTEMS (page 7)
- P-5: DOMAIN/USER MATRIX (page 8)
- P-6: MODEL/EDITOR SEPARATION (page 9)
- P-7: INPUT/OUTPUT SEPARATION(MVC/1980) (page 10)
- P-8: TOOLS FOR TASKS (MVC/1979) (page 11)
- P-9: TOOL AS A COMPOSITE (MVC/1979) (page 12)
- P-10: SYNCHRONIZE SELECTION (page 13)
- P-11: SYNCHRONIZE MODEL AND VIEW (page 14)

P-1: INTEGRATED DOMAIN SERVICES



Context

An enterprise handles a number of different business functions a.k.a. *domains*. Examples are design, materials management, planning, control, and finance.

Problem

The enterprise needs to support such domains with integrated information systems.

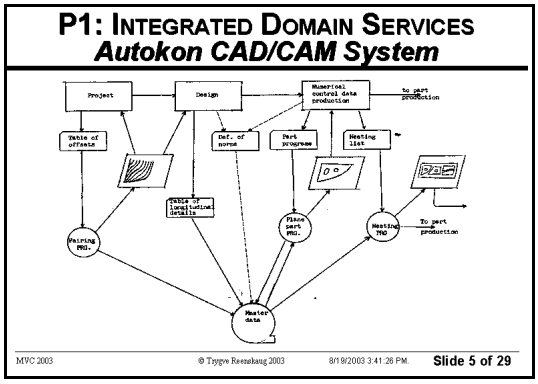
Solution

Create separate domain services for each of the different business domains. Each of these services should be tightly integrated internally, e.g., through a common data base or through tightly coupled interacting services. Integration between domains will be through mechanisms that are outside the domain services.

Notice that a domain service may span several business organizations and even several enterprises.

Forces

- Large domain services may be unwieldy, hard to design, hard to implement, and hard to modify.
- Small domain services lead to fragmentation of the total system and makes integration across domains harder.



Known Uses

This is the common approach to systems architecture. A domain service is often called an *application*. Integration between functions is often ad hoc.

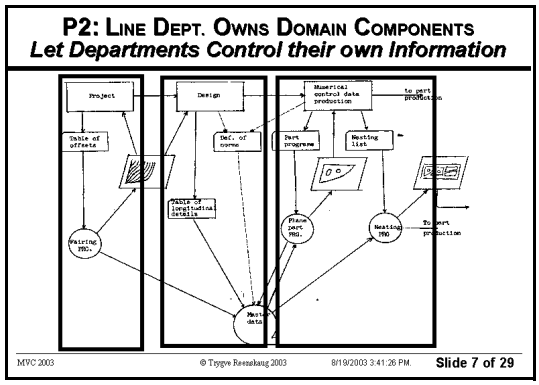
See also

P-2: LINE DEPARTMENT OWNS DOMAIN COMPONENTS (page 3)

History

August 2003: First draft of this pattern

P-2: LINE DEPARTMENT OWNS DOMAIN COMPONENTS

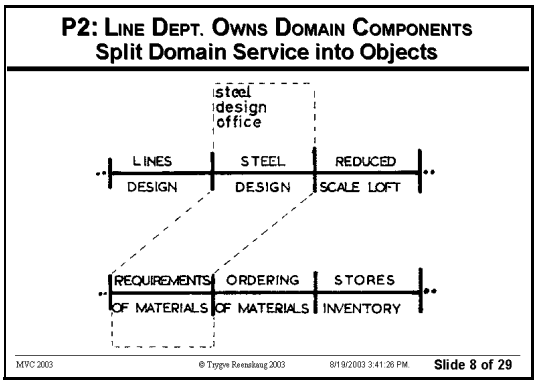


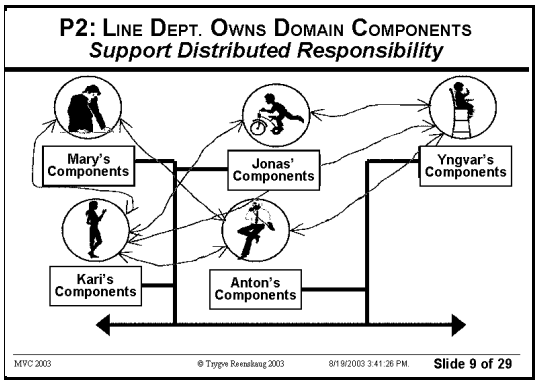
Context

Business domains frequently span several responsible departments in the line organization. A ship construction domain could, for example, span contract, main design, detailed design, and workshops. The design domain service is a whole, but spans many line departments and areas of responsibility.

Problem

The different departments will normally be consulted during the specification of a new or modified domain service. The influence of an individual department tends to be limited because the overall specification is often a compromise between conflicting interests. It seems desirable to give departments and individuals better control of their information so as to match their responsibility and authority.





Solution

Decompose each domain service into several interacting *Dopmain Components* so that each component has a single owner in the line organization. This way, each department controls the part of the total service that is important to it. The interests of the whole are taken care of through interface control, while the internals of each component is controlled locally.

Forces

•The decomposition of a business domain service strengthens the individual owners while weakening the whole.

•A domain service architecture that is tightly coupled to the current line organization makes it hard to change the organization.

•A domain service architecture that is tightly coupled to personal idiosyncrasies can cause difficulties when people move.

•There could be decreased computational efficiency, and also added security risks.

- A multi-tier solution with central ownership of low tiers and local ownership of upper tiers could be considered. (Also see P-5 on page 8

Known Uses

There is no known implementation of this pattern.

See also

http://www.c2.com/cgi/wiki?DistributionOfComponents.
P-4: Personal Information Systems (page 7)
P-5: Domain/User Matrix (page 8)

History

1973: Ideas first presented at the ICCAS Conference in Tokyo, Japan. [Ree-73]
August 2003: First draft of this pattern

P-3: Mental Object Models

Context

Most products are developed in incremental steps of product releases. There is no explicit overall model, many products even lack a reference manual encompassing all product features. Many developers find it hard to create a model up front. The system structure evolves as it is being shaped directly in code. This supports the notion of systems development as an excursion into uncharted territory. Many versions of a

system have probably been released before the developers feel they understand the requirements. And this understanding is a moving target anyway, since the user's understanding and requirements deepen and evolve.

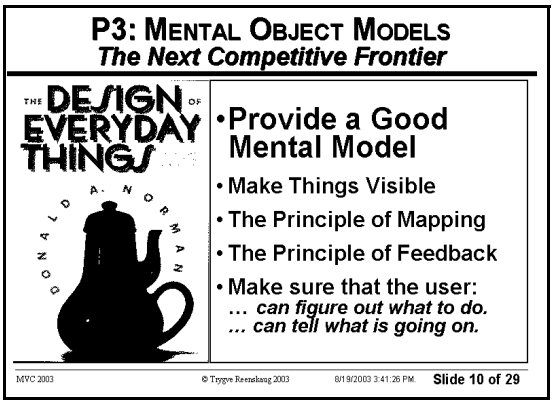
There are three approaches to making information systems manageable for the user:

1) Provide a wizard for each task the user could possibly want to perform. The problems with this approach is that novices may not understand the vocabulary: *"Enter your fuzzywog ID now"*. More experienced users often have tasks never contemplated by the system designers. And experts find wizards cumbersome and slow to use.
2) Provide a help system. We find the same vocabulary difficulties. In addition, it is usually not feasible to provide help for every detail. Many help systems even lack an entry for every menu item and dialogue box element.

Neither of these solutions is satisfactory.

[Norman-90] claims the only way to make artifacts manageable is to help the user build a mental model of the system. But this is impossible if the mental model hasen't been designed into the artifact from the beginning. As an afterthought, it is doomed to be a failure.

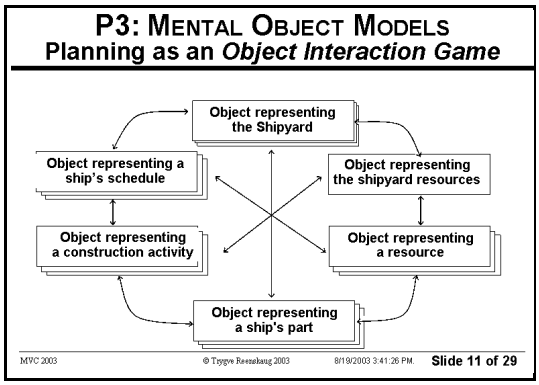
Some standards are generally accepted in our business. UML[UML 2.0] is a standard modelling language for systems architecture and design. But I do not know of a modelling language specifically aimed at building and implementing users' mental models.



Problem

The problem is to empower the user to build a workable mental model. For information systems, this implies that the user must internalize some modelling language that can be shared between the user and the system designer. There are two issues. First, what is the nature of this modelling language. Second, how can we design a system so that the user sees a reflection of his mental model in the system without any of the details that happen to be of no interest to him. We will deal with the first here, and relegate the second to other patterns (More at "See also" below)

A mental model can only be buildt from elements existing in the person's mind. Apple's desktop metaphor is an example of tying a well-known environment. This is insufficient for the complex information systems of today. A large number of people are now users of information systems, but their mental models are fragmented and frequently inaccurate. The challenge is to establish a modelling language as "knowledge in the world" that system architects can use when designing the systems and the users can use for building powerful and accurate mental models.



Solution

Let the user's modelling language be based on the concepts of interacting objects. (Note: The UML Component is a subclass of the UML Class, so objects can be instances of components as well as classes. This means that objects can be composite; encapsulating other objects).

As an example, consider an activity planning and control system. Resources, products and activities can be represented as objects in the computer. The planning and other processes can be realized by interactions between these objects.

The language used for building the user's mental model must be within the user's passive vocabulary, i.e., "self-evident". The sophistication of this language could be increased if a sophisticated language became "knowledge in the world". Such a generally known language could be a UML[UML 2.0] profile or, if necessary, a new language.

As mentioned in the Context paragraph above, it is notoriously hard to design a new system up front since the requirements evolve over time. A solution could be to let the modelling language be the programming language. UML is very close to becoming a programming language. This modelling/programming language could be used in all phases of product life from the first experiments, through the evolution of requirements and numerous release cycles. OMG develops MDA (Model Driven Architecture)[MDA]. This initiative is promising and could be the beginning of a generally known language and method for evolving user mental models and programs in parallel.

Finally, if a sufficiently powerful language became sufficiently well known, the road would be open for users reading and even writing the top tiers of their programs themselves.

Forces

Users tend to be sophisticated within their field of expertise. Real requirements are, therefore, usually quite complex. This puts a heavy strain on the models and the modelling language.

People use their whole being to master a subject. Formal modelling languages are strongly left brain. This makes it hard to model right brain phenomena. This mismatch should be carefully considered during the development of a universal modelling/programming language.

Known Uses

There are two known indications that an object based modelling language is acceptable to people. One was an experiment at Xerox PARC in 1978 where we developed a planning system for a new semiconductor production facility The facility manager was thinking in terms of silicon wafers, processes and equipment while the Smalltalk developer was thinking in terms of interacting Smalltalk objects. The communication between manager and developer went very smoothly and confirmed our belief that an object model can give users effective control over their information systems.

Another indication is that our company, Taskon, did several process modelling projects in banks during the nineties. We invariably found that object models felt natural to the user community and empowered them to think precisely about their processes.

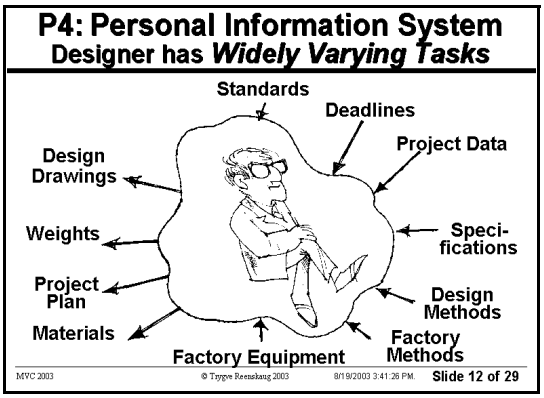
See also

P-4: PERSONAL INFORMATION SYSTEMS (page 7)

History

1977: Ideas first presented at the IFIP Conference in Toronto, Canada. [Ree-77]
August 2003: First draft of this pattern

P-4: PERSONAL INFORMATION SYSTEMS



Context

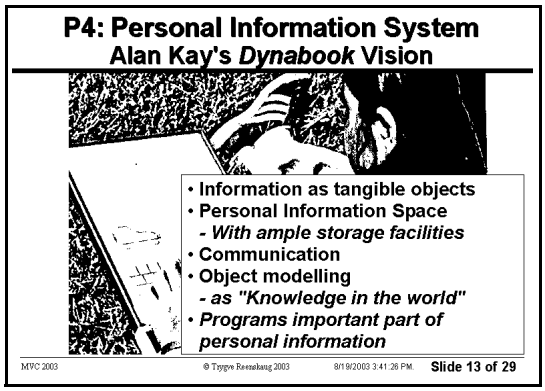
An enterprise has a line organization that defines the roles played by people and machines. To each role there is an associated set of responsibilities, authorities and capabilities. Examples of roles are designers, accountants, machine operators.

Problem

An individual performs tasks, where each task often involves several domains. A designer, for example, needs full access to the part of the design domain he is responsible for, but also to other parts of the design domain for reference purposes. He may also check materials availability in the Material Management service, fill in time sheets in the Accounting service, report progress to the Planning and Control service, etc. The individual needs task oriented tools that supports him in his various tasks.

Solution

Give the individual *tools* that support his various tasks. Let these tools be embedded in a personal computer that supports the user in all his or her tasks.[Kay-77]. Let each tool access one or more domain services as required.

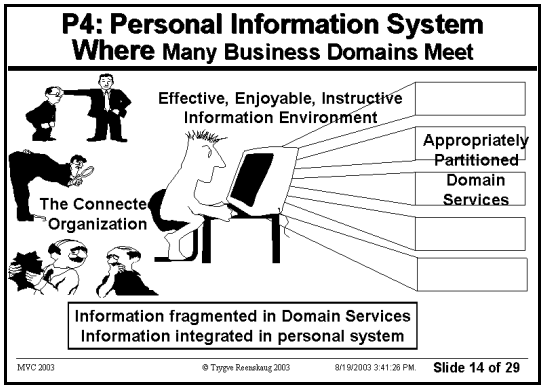


Forces

- Tasks tend to change frequently, and emergencies that may need special tools are known to occur.
- "The only stable feature of our company is that it changes".

Both forces call for rapid and cheap tool development. This can be realized through simplicity or through (semi-)automatic tool generation.

People's tasks are frequently complex. This calls for powerful tools.



Pople's tasks often involve several domains. The tool is, therefore, a point of integration across domains. (Also see FourLayerArchitecture[pattern links], where the four layers are called View, Application Model, Domain Model, andInfrastructure.)

Known Uses

The Norwegian Exie[Exie] product offers rapid implementation of flexible administrative solutions where personalized and automatically generated tools connect the users to background information systems for tasks, resources and production.

See also

P-5: DOMAIN/USER MATRIX (page 8)

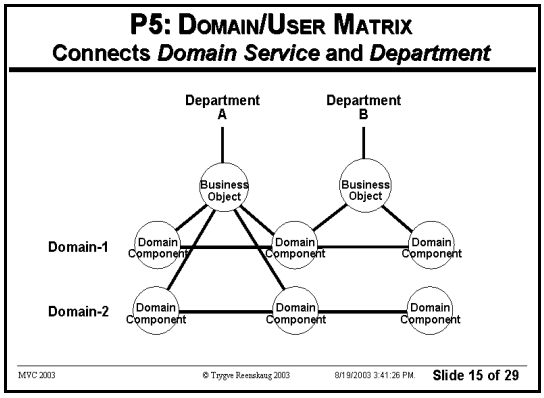
History

June 1999: TOOLS Europe '99 Keynote: "T. Reenskaug: *Component-Based Development - the True Object Orientation.*
August 2003: First draft of this pattern

P-5: DOMAIN/USER MATRIX

Context

P-2: LINE DEPARTMENT OWNS DOMAIN COMPONENTS (page 3) told us to decompose each domain service into a number of domain components. P-3: MENTAL OBJECT MODELS (page 4) and P-4: PERSONAL INFORMATION SYSTEMS (page 7) suggest that the user's mental model should be an object model that is accessible through the user interface and that may or may not correspond to the actual nature of the domain services.



Problem

The domain services are implemented for efficiency and information integrity while the optimal mental models depend on the user and his tasks. There is no guarantee that the domain services correspond to the mental models of the different users. If they do, there is no problem. If they do not, we need to bridge the gap.

Solution

Extract common information and behaviour from the domain components. Implement common information and behaviour in centrally controlled background domain services.

Create a new layer of *Business Objects* above the domain service components as illustrated in the slide. Let each business object create the illusion that the system implements the user's mental models, using the domain services as required.

(The term "Business object" is used in an OMG initiative. The use of the term here may not coincide with the OMG usage.)

Forces

There may be a tendency to move common domain logic and information into the business object and under the control of the line organization.

There may be a tendency to move local logic and information into the domain services to simplify maintenance and retain central control.

See also

P-6: MODEL/EDITOR SEPARATION (page 9)
P-7: INPUT/OUTPUT SEPARATION(MVC/1980) (page 10)

History

August 1973: The above figure first published in [ICCAS]
August 2003: First pattern version.

P-6: MODEL/EDITOR SEPARATION

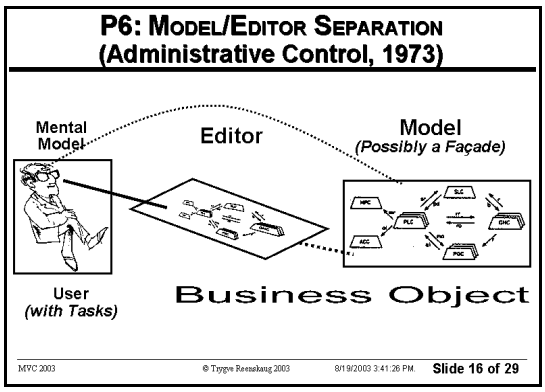
Context

A person wants to study and interact with a business object (P-5 on page 8).

Problem

The user may want to inspect and edit information that exists in a business object.

- These objects may not be directly accessible from the node holding the user interface
- The objects may be too complex to be viewed directly
- Different tasks may require different presentations and operations on the same information; each version highlighting some aspect and suppressing something else.



Solution

We split each Business Object into two parts; one close to the user and another close to the domain services. We call these objects *Editor* and *Model* respectively.

The *Model* holds the user's object model with its information and behaviour, reflecting the user's mental model. The *Editor* is responsible for presentation and user operations.

The Model can be implemented as a Facade as defined in [GOF].

The Editor can e.g., be hand coded as a Java Swing component, it can be a Java Bean, or it can be automatically generated from a GUI painter or through reflection on the model code.

Forces

- A powerful, yet understandable modelling language promotes the creation and realization of the user's mental model.
- Complex user needs lead to sophisticated Editors. Automatic or semi-automatic Editor generation promotes rapid adaption to varying user tasks.

See also

P-7: INPUT/OUTPUT SEPARATION(MVC/1980) (page 10)
P-9: TOOL AS A COMPOSITE (MVC/1979) (page 12)
P-11: SYNCHRONIZE MODEL AND VIEW (page 14)

History

1979: This idea was part of the original MVC. [MVC-1], [MVC-2]
May 2001: MVC pattern written for Mogul patterns workshop.
August 2003: Above pattern refactored and revised

P-7: INPUT/OUTPUT SEPARATION(MVC/1980)

Context

The Editor described in P-6 on page 9 combines input and output in the same object.

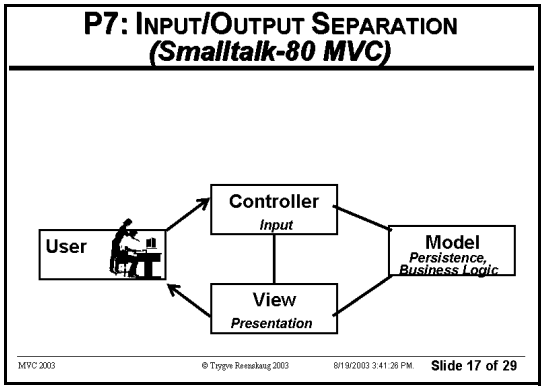
Problem

The input and output aspects of the Editor are technically very different with few interdependencies. Their combination in a single object tends to make this object unnecessarily complex.

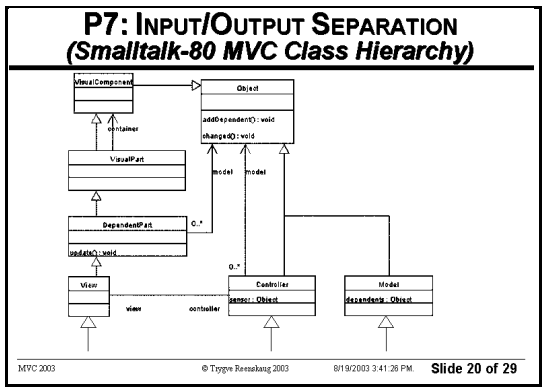
Solution

Let the Editor object contain two objects; a *View* object responsible for presentation, and a *Controller* object responsible for taking and interpreting input from the user.

The illustration shows a UML collaboration model describing the Smalltalk-80 solution.



This illustration shows an example class hierarchy taken from the Smalltalk class library.



Forces

- In simple cases, the Model, View and Controller roles may be played by the same object. Example: A scroll bar.
- The View and Controller roles may be played by the same object when they are very tightly coupled. Example: A Menu.
- In the general case, they can be played by three different objects. Example: An object-oriented design modeller.

See also

P-8: TOOLS FOR TASKS (MVC/1979) (page 11)
P-10: SYNCHRONIZE SELECTION (page 13)

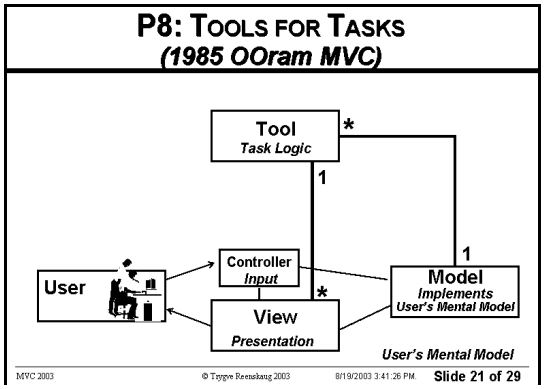
History

1980: This is the MVC of the Smalltalk-80 class library.
August 2003: First draft of this pattern extracted from the above

P-8: TOOLS FOR TASKS (MVC/1979)

Context

A user and a model object with a façade that reflects the user's mental model.



Problem

To give the user a *Tool* for performing one or more tasks. The Tool shall give the user an illusion of interacting directly with the model.

Solution

We create a user interface with a Tool object that contains the required Editors.

Some or all Editors can be split into a View-Controller combination as shown in the figure (see P-7 on page 10)

Forces

- In simple cases, the Tool may consist of a single Editor. This Editor then also plays the Tool role.
- Complex tasks may require several Editors, they then need a separate Tool object to coordinate them.

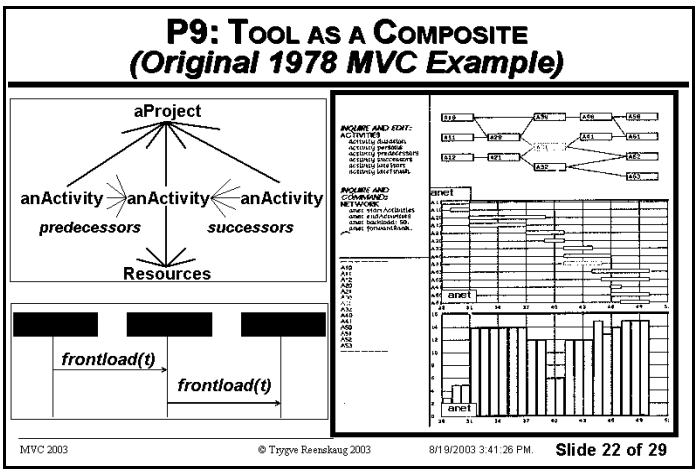
See also

P-9: TOOL AS A COMPOSITE (MVC/1979) (page 12)
P-10: SYNCHRONIZE SELECTION (page 13)
P-11: SYNCHRONIZE MODEL AND VIEW (page 14)

History

1979: This is the original MVC[MVC-1]
2001: First draft MVC pattern written for Mogul patterns workshop
August 2003: First draft of this pattern extracted from the above

P-9: TOOL AS A COMPOSITE (MVC/1979)



Context

A user may want to inspect and operate upon a complex model buildt as a structure of interconnected objects. As an example, consider a Project consisting of several Activities and Resources. The corresponding model is illustrated on the left of the slide, while a possible simultaneous presentation of various aspects is shown in the right half.

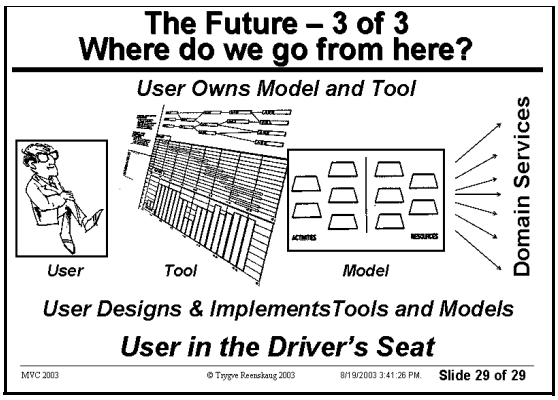
Problem

Structure the user interface so as to support this requirement in a simple and general way.

Solution

Separate the system into four, clearly distinguished parts that play these roles:

1) The *User* with his goals and tasks.
2) The *Model* that is responsible for representing state, structure, and behavior of the user's mental model.
3) One or more Editors that present relevant information in a suitable way and support the editing of this information when applicable.
4) A Tool that sets up the Editors and coordinates their operation. (E.g., the selection of a model object that is visible in several Editors).



Complex Editors may again be subdivided into a View and a Controller. (P-7 on page 10).

This solution is a composite pattern and can be regarded as the following "sentence" in the MVC pattern language: **(P-3) (P-4) (P-6) (P-7) (P-8)**

Forces

TBD

Known uses.

This pattern has been used extensively in the OOram role modelling Tool and other programs. It is also used in the Exie product[Exie].

See also

P-8: TOOLS FOR TASKS (MVC/1979) (page 11)
P-10: SYNCHRONIZE SELECTION (page 13)
P-11: SYNCHRONIZE MODEL AND VIEW (page 14)

History

1979: This is the original MVC[MVC-1]
2001: First draft MVC pattern written for Mogul patterns workshop
August 2003: First draft of this pattern extracted from the above

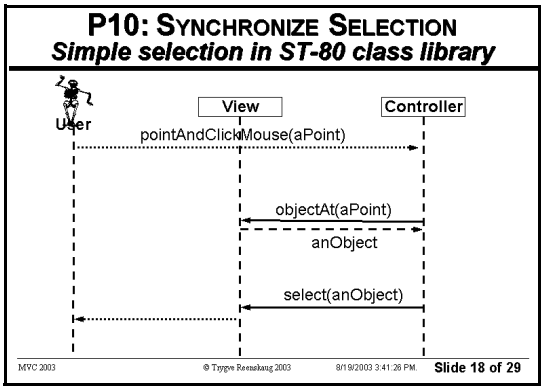
P-10: SYNCHRONIZE SELECTION

Context

We have a Tool that shows different Views of the same object.

Problem

A user selects one or more objects in one of the Views. The selection should appear in all Views where the selected object is visible in order to maintain the object model illusion.



Solution

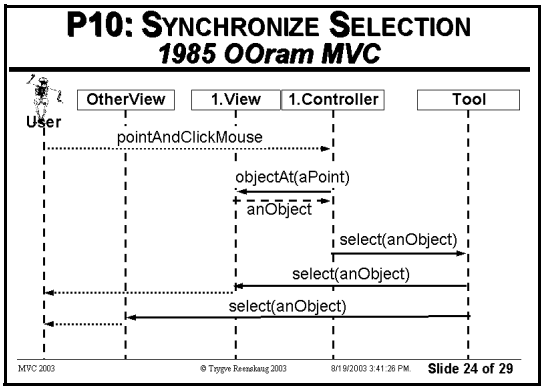
Let the selection be the responsibility of an object that knows all the relevant Views. The Tool is an example of such an object.

A simple example is the division of responsibility in the ST-80 MVC, a selection MSC is shown in the slide to the left.

Below is another MSC that illustrates how OOram implements this pattern.

Forces

- The implementation can be quite complex if the deselection and selection of many objects.



Known uses

This pattern was used extensively in the OOram modelling Tool and is also used in the Exie[Exie] product.

See also

Not Applicable.

History

August 2003: First version.

P-11: SYNCHRONIZE MODEL AND VIEW

Context

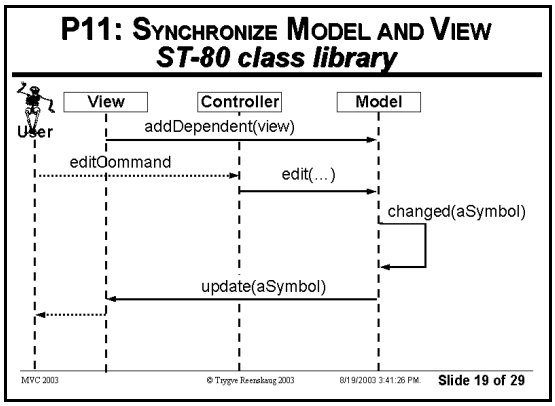
A View presents information that it retrieves from one or more model objects.

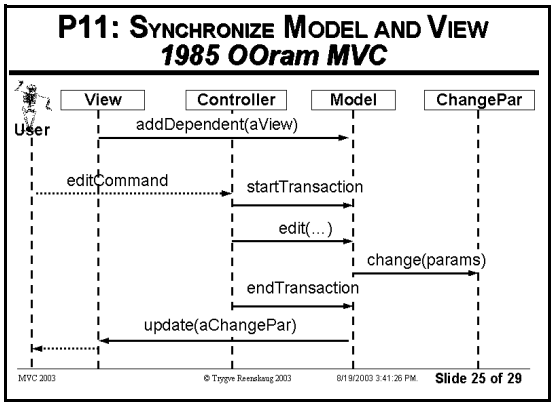
Problem

The View cashes model data in the screen buffer and/or in its private memory. These data need to be updated whenever the model changes.This is a special case of the Observer pattern in [GOF]

Solution

Let the View register with the Model as being a dependent of the Model, and let the Model send approipriate messages to its dependents whenever it changes. Two examples are shown on the right. The first is the simple changed-update solution found in the ST-80 class library. The second is a summary of the OOram solution which uses transactions to accumulate changes before releasing them to the Views.





Forces

The Smalltalk changed/update can lead to a great deal of annoying flashing on the screen.

The use of transactions can reduce the number of updates to one for each (composite) user operation.

The use of a ChangeParameter can further reduce flashing by identifying the object property that has changed. (The View and the Model has a common vocabulary in the Model's information retrieval interface).

A more sophisticated ChangeParameter can also identify the area affected by the change if the model has a notion of model geometry. (This will be the case for a drawing, for example)

Known uses

This pattern was used extensively in the OOram modelling tool. It is also used in the Exie product[Exie].

See also

Not Applicable.

History

August 2003: First version.

3. The future

Where are we? And where do we go from here?

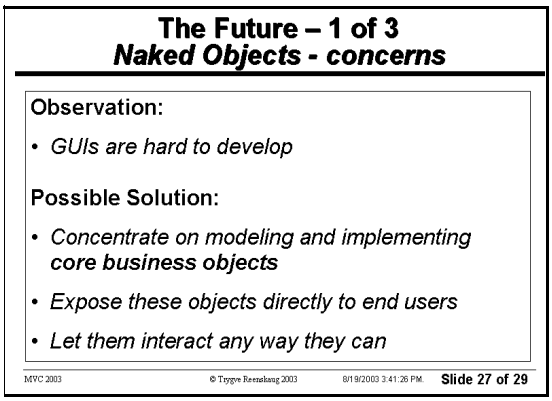
1) We understand many user interface issues (and much more can be found in the literature)
2) We need to automate tool programming. The *Naked Objects* project[naked] is a promising start.
3) We must do more work with architecture and a language common to users and programmers.

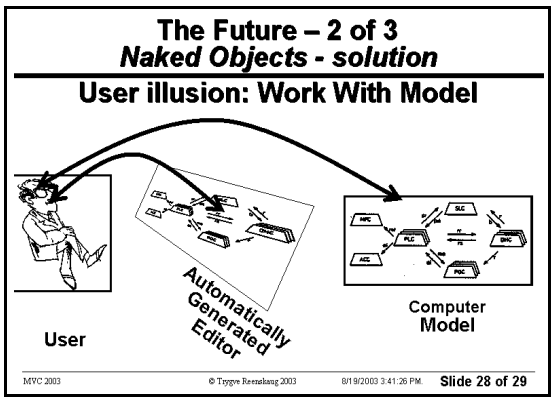
Many mainstream developers believe that all we need to support users is to ask them nicely about their requirements and preferences.

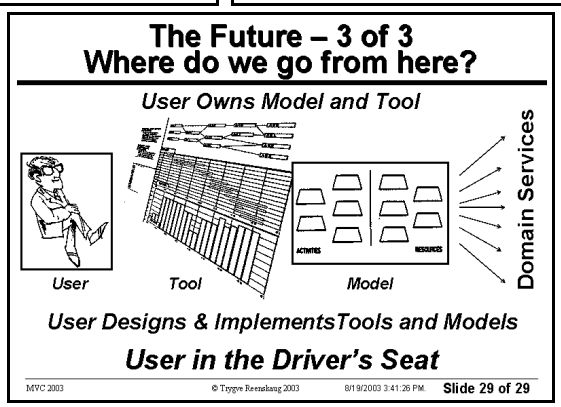
I have now shown that we need much more. We need system models that the users can understand, work with and evolve over time.

In short, every user needs his own habitable information environment.

The next stage of the InfoBoard project will endeavour to create a humane modelling and programming language and environment.







4. References

[home] mailto: trygve.reenskaug@ifi.uio.no
http://heim.ifi.uio.no/~trygver

[pattern links] http://hillside.net/patterns
http://theserverside.com/patterns/
http://www.c2.com/cgi/wiki?WelcomeVisitors
http://www.c2.com/cgi/wiki?EjbRoadmap
http://www.c2.com/cgi/wiki?FourLayerArchitecture

[Ree-73] Trygve Reenskaug: Administrative Control in the Shipyard. ICCAS Conference, Tokyo, August 1973. A scanned version at http://www.ifi.uio.no/~trygver/mvc/index.html.

[Ree-77] Trygve Reenskaug: *Prokon/Plan -- A Modelling Tool for Project Planning and Control.* Information Processing 77, B. Gilchrist, Editor. IFIP, North-Holland Publishing Co. (1977)

[GOF] Gamma, Helm, Johnson, Vlissides: *Design Patterns.* Addison-Wesley, Reading 1995. ISBN 0-201-63361-2

[ICCAS] Trygve Reenskaug: Administrative Control in the Shipyard. ICCAS Conference, Tokyo, August 1973. A scanned version on http://heim.ifi.uio.no/~trygver/mvc/index.html

[Prokon] Trygve M.H. Reenskaug: Prokon/Plan -- A Modelling Tool for Project Planning and Control. IFIP Congress, Toronto 1977. North-Holland, 1977 A scanned version on http://heim.ifi.uio.no/~trygver/mvc/index.html

[MVC-1] Trygve Reenskaug: *THING-MODEL-VIEW-EDITOR - an Example from a planningsystem.* Technical note, Xerox PARC, May 1979. A scanned version on http://heim.ifi.uio.no/~trygver/mvc/index.html

[MVC-2] Trygve Reenskaug: *MODELS - VIEWS - CONTROLLERS.* Technical note, Xerox PARC, December 1979. A scanned version on http://heim.ifi.uio.no/~trygver/mvc/index.html

[Exie] See http://www.exie.com

[Norman-90] Donald A. Norman: “The Design of Everyday Things.” Doubleday/Currency 1990. ISBN 0-385-26774-6.

[Alexander-77] Alexander, Christopher, et al. *A Pattern Language*, Oxford University Press, New York, 1977.

[Alexander-79] Alexander, Christopher. *The Timeless Way of Building*, Oxford University Press, New York, 1979.

[UML 2.0] The Unified Modeling Language, version 2.0. See http://www.omg.org/uml/

[MDA] OMG Model Driven Architecture (MDA). See http://www.omg.org/mda/

[Kay-77] Alan C. Kay: *Microelectronics and the Personal Computer.* Scientific American **237**, 3. Sept. 1977

[Cope] Coplien, James O. and Douglas C. Schmidt, ed. *Pattern Languages of Program Design*, Addison-Wesley, 1995.

[Roles] Reenskaug, Wold, Lehne: *Working With Objects.* Manning/Prentice Hall 1996. ISBN 0-13-452930-8 The reference work. This book is out of print. A buggy *.pdf* version can be downloaded free from above website.

[naked] Naked Objects home page: http://www.nakedobjects.org/home.html