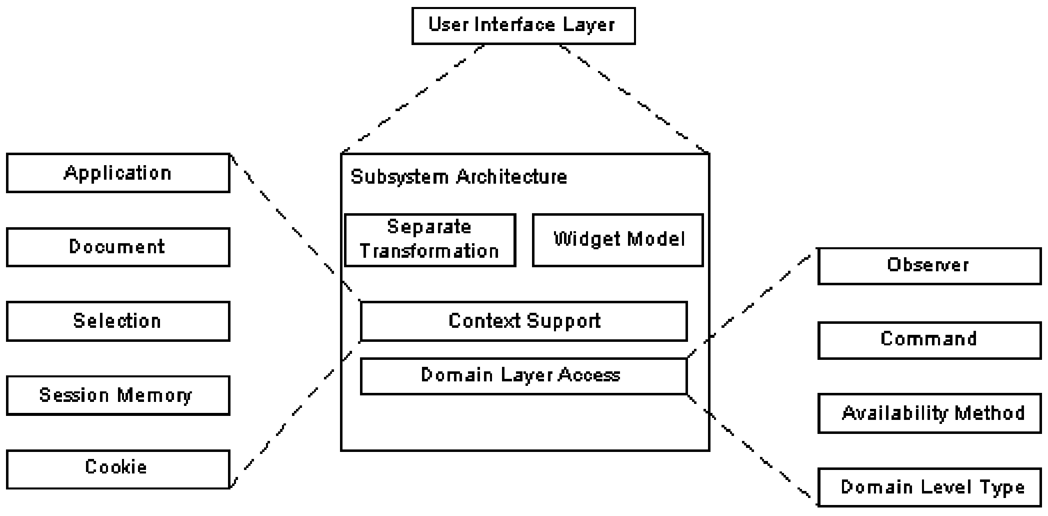
User Interface Software

Jens Coldewey,
Coldewey Consulting,
Uhdestraße 12, D-81477 München, Germany
Tel: +49-89-74995702; Fax: +49-89-74995703
Email: *jens_coldewey@acm.org*

This pattern language digs step by step into the design of a user interface architecture. I do not mean the layout of a user interface. This is a matter of ergonomics and bears enough potential to form a complete set of pattern languages on its own. If you are interested in these issues, refer to books such as *[Tog92]*, *[Coo95]*, or *[Col95]*.

Instead, this paper is about the software that drives the user interface. The figure below shows the overall landscape of the pattern language. It starts with the most fundamental pattern, the *User Interface Layer*. Two patterns describe the architecture of this layer: *Separate Transformation* explains how to deal with complex interactions while *Widget Model* helps to structure presentation. Though both patterns seem to describe different philosophies, they are often combined to form the basic architecture of the *User Interface Layer*.

Still you need to deal with more issues. At first you have to provide *Context Support* between different interactions of the user. Depending on the requirements and architecture of your system you can apply several patterns for this. For the sake of brevity this document contains only thumbnails of them. Another area of interest concerns *Domain Layer Access*, which again lays the foundation of several other patterns. While some of them are well-known design patterns, others are special to user interfaces.



This set of patterns forms the coarse landscape of most architectures around.

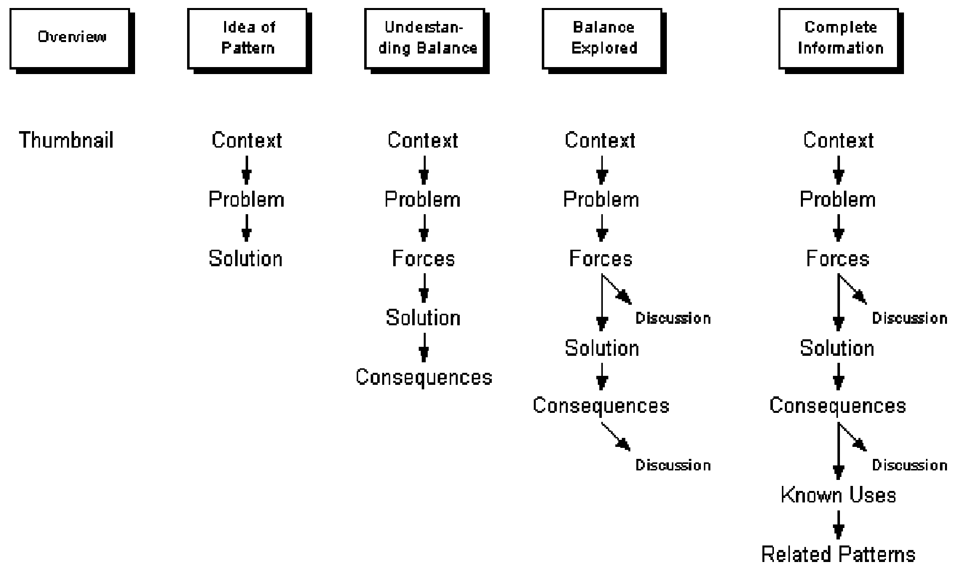
Some Terms Explained

I have used several terms in this paper not everyone may share. I give a rough definition here and some further hints where to find detailed information about it.

Object-Oriented User Interface	I use the term "Object-Oriented User Interface" to describe an interface paradigm where the user first selects an object and then chooses an action from a context menu or via direct manipulation. The Macintosh user interface is an example of such an interaction paradigm: The user first selects a document and then decides what to do with it. She can either manipulate it via double click or drag it over the printer icon to print it or do other fancy things. The term "object-oriented" here refers to the interaction style rather than to the implementation technique – though it is usually a good idea to use object-oriented design and programming to build this kind of interface. For more details refer to [*Col95*] for example.
Form-Based User Interface	The term "Form-Based User Interface" denotes interfaces where the user starts with selecting a certain business process. The application then guides the user though this process with a series of forms. Most user interfaces on mainframe systems work this way. See [*CoK97*] for further discussion.
Object-Oriented Systems	I use this term to refer to applications that have been designed and implemented using object-oriented techniques. In the context of this paper the most important feature of these systems is a set of objects with their own life-cycles that interact to perform the functionality. Hence single use cases or transactions are not relevant to the architecture.
Transaction-Oriented Systems	In these systems the architecture usually is organized around transactions. Every transaction invokes a different program that manipulates a database. Most programs using transaction processors, such as CICS, are build that way. See [*GrR93*] for further discussion.

How to Read this Paper

You rarely read a pattern language cover to cover. Structure and layout of this paper help you to dig into the patterns as deep as you want to. For a short abstract of what the pattern is about, read the Thumbnail. You can easily get an overview over the complete language by just reading every thumbnail section. If you are interested in the detailed idea of the pattern read Context, Problem, and Solution. If you are interested in the forces that drive the pattern and the balance the solution chooses, you should also read the Forces and Consequences of a pattern. Both sections contain the main statements in standard font. A small font indicates a thorough discussion of these statements. The image below demonstrates the suggested paths through this paper.



User Interface Layer

Thumbnail

Driving a user interfaces is complex and more prone to changes than the domain usually is,...therefore separate domain level issues from user interface software and encapsulate the user interface in a separate layer.

Context

When you design the architecture of a complex business system you usually have to start with the coarsest architectural decisions: What are the large chunks I have to design? In former days technological considerations drove the first decision, ending up in large applications with a technical interior structure. Those were the days of 'The Standard Architecture' for certain families of systems, and the days of 100 people working on one project. Today many architects first use domain level aspects to chop the system into smaller parts they call business objects, before they care for technical considerations.

No matter which of these two ways you go, sooner or later you have to consider technological aspects to further subdivide your system. One of these technological aspects is presentation of the system towards the users. That is where this pattern steps in. It applies to object-oriented systems as well as to transaction-oriented designs, to systems using the tool-material metaphor *[Rie97]* as well as to form-based applications *[CoK97]*. So...

Problem

...how do you place user interface issues of large systems into the overall architecture?

There are several things you have to take care of:

Forces

The user interface presents the same functional requirements as your analysis model,...
...but it has different non-functional requirements that lead to a lot of details the user does not want to be molested with.

> The user interface is only an *interface* to your system, so it fulfills exactly the same functional requirements your analysis model does. A naive approach would be just to present all the analysis objects on the user interface with their methods as actions. *However, is this really a good idea?* The goals of a good analysis model are to reduce redundancy and to find abstractions that support flexibility. When you proceed with the design of the domain objects, additional aspects join into the game: Performance and decoupling are major considerations as well as concurrency and all the other aspects that prevent a software engineer from getting bored. Yet, none of these issues are topics the user wants to be molested with. Good user interfaces take care to follow the user's mental model of her domain and to support her workflow as good as possible. It is no surprise that a large share of change requests concern the user interface and have no effect on the analysis model. A new group of users with a different mental model of the domain may need a completely new user interface without even the slightest change in the analysis objects. Hence, the non-functional requirements of a user interface are quite different from those of the domain objects.

A standard technique to fulfill different non-functional requirements is to separate the different concerns into subsystems on their own,...
...but a user interface needs a lot of domain level information to meet high ergonomic standards, so separated subsystems would be coupled closely.

The standard reflex of a software architect is to put different non-functional requirements into different subsystem. This gives her the freedom to optimize every subsystem on its own. In the case of user interface this decision would mean to have a subsystem managing the user interface and another subsystem managing domain level issues. *However, is this really a good idea?* One of the main criteria of subsystem boundaries is the amount of information you have to transfer between the clusters. Modern user interfaces support the user with a vast of information: List boxes provide possible choices, navigation trees help to address specific objects, and grayed menus and buttons prevent you from doing something stupid. All this causes high coupling between the subsystems.

Modern development environments offer convenient ways to access the user interface in a particular way,...
... but complex systems often need several different user interfaces to satisfy different needs and these interfaces may need different designs to meet their respective requirements.

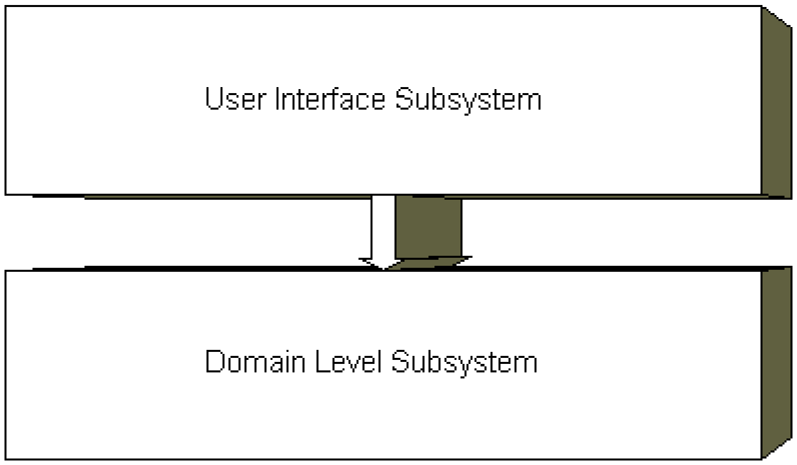
In languages such as Smalltalk or Java, it usually needs a single line of code to disable a menu choice or to pop up a simple dialog box prompting the user. Therefore it is tempting just to prompt the user if something strange happens in your *domain* code and ask him "Shall I continue to do what I intend to do? (Yes / No / Help)". *However, is this really a good idea?* Sooner or later you may find that different users want to operate the system. They have to do the same job a thousand times a day and they want the system to support this particular job with as few keystrokes as possible. A dialog popping up disturbs their work. Instead they want the system to record the problem in an error list so they can care for it later. You need a completely different user interface layout. Or consider using your system in batch mode: It is surely an unpleasant idea to have the system waiting for a user decision from 2:30 in the morning until the operator comes at 9 o'clock with an unproductive prompt on the screen. Finally your system may run in a context without any user interface at all. Consider a telephone switching system somewhere in the middle of nowhere: There is no operating personal on site but the system is operated remotely. It would be no good idea to show up a prompt at the local terminal while the operator waits 300 miles away for the system to answer. Similar situation appear more and more with global business systems. So remotely operated user interfaces need a different architecture then systems to enter mass data, which again need a different architecture than highly interactive systems. Most development environments do not support all of these architectures optimally.

Finally, the domain level software may contain abstractions the users do not understand,...
...but the domain level structure always shows through to the user interface.

During the analysis most systems go through a metamorphosis starting with the very concrete ideas of the domain experts and ending up with an abstract object model that also respects issues such as reusability, maintainability, and ease of design. If the system has to satisfy a broad variety of requirements, the analysts may even have decided to set up a meta model of the domain. These are the decisions of software experts, not of domain experts. Often you end up with a model that is incomprehensible for domain experts. To ensure the usability of your system you have to transform the abstract domain model back into the user's world. *However, is this really a good idea?* Transforming the object structure to a different structure is additional effort at least. If the domain level model is not compatible with the model you want to present to the user, the job may become a nightmare. The best example is an object-oriented user interface on top of a transaction oriented domain model: These systems usually suffer from unintuitive navigation and poor performance.

Solution

Therefore separate user interface issues from domain level aspects and put them into two different subsystems. Make sure the domain level subsystem has no notion of the user interface part, thus forming a layered architecture.



It is important to note the layered aspect of the solution: The user interface layer is allowed to call the domain layer but not vice versa. When a 'callback' is inevitable, we need special mechanisms to ensure the layered approach. *Access Domain Layer* addresses these issues.

Consequences

With this architecture it is easy to optimize the two layers for different non-functional requirements,...
...still you will have to take care not to implement functional requirements twice.

Both layers may now have a totally different design. For instance, the user interface layer may use a *Widget Model* with the various views as the backbone of the design while the domain layer is based on the analysis model. With this design changes of the layout result in local changes of the user interface layer while changes of performance requirements concentrate on the domain layer. *However*, a new topic comes up you have to take care for: There is a high risk to implement functional requirements in both layers redundantly. For instance, consider an OK button that has to be disabled as long as the dialog box does not contain consistent data with respect to some business rules. It is tempting to put this rule into the dialog box class. On the other hand you also need the criterion somewhere in your domain logic. Now the mess happens as soon as the rule changes: You have to change on two places or even more. To fight this risk, the domain layer usually provides meta information about the domain. *Availability Method* and *Domain Level Type* deal with these problems.

The layered design provides separation of concerns,...
...still you have to take precautions to keep the coupling between both layers manageable.

With a disciplined use of this pattern you can make sure, that presentation aspects are strictly separated from domain level aspects. If you are in doubt, where to put a certain responsibility, the key question is: If I replace the user interface with a batch control, do I still need this functionality? As a rule of thumb, if the answer is "yes", it is probably a good idea, to place the responsibility in the domain layer. If the answer is "no", you probably snatched a user interface responsibility. *However*, you will still end up with a broad interface between both layers. You can assign the information you need to several categories:

- Read-only context information about objects the user currently works with,
- Rules for formatting,
- Information whether a particular action is allowed in the current context,
- Information on status and progress of the domain layer,
- Triggering (trans)actions, and
- Error information

Domain Layer Access discusses these issues.

The actual amount of traffic between the two layers depends on the user interface style. If your user interface relies on dialogs which consist of a series of forms, you usually need less context information than with a tool-material metaphor. This is the deeper reason, why web-based applications and mainframe systems usually use forms while good PC based systems often use the tool-material metaphor.

Not all modern development environments support this architecture - and if they do, it is usually more work,...
...still the layered approach gives you the freedom to support several presentations of the same system.

Many popular frameworks take a user interface centered approach. When the user clicks a button, the framework calls a `clicked` method of the corresponding `Button` object. It is the programmer's task to take the appropriate action in this method. Often there are no standard processes or even classes that manage the communication with the domain layer. Hence, it is a matter of discipline to ensure the layering - not an easy job, especially when you are under pressure of a tight schedule. *However* a clean layering saves you a lot of work if you need additional representations: You "just" have to plug off the graphical user interface classes and replace them with batch classes or with a CORBA interface. Even if you do not plan to have a CORBA interface, you may soon end up with different representations for the same domain level aspect when you analyze manipulation. For instance, many user interfaces support copying via the clipboard and drag & drop. Both require different user interface activities but the same domain level functions.

The extra layer can adapt the user interface to the specific needs of a user group,...
...still it cannot turn butter into gold.

If the domain layer uses a meta model of the domain, the top layer can use the information to assemble an interface that complies to the user's mental model. For example you can present hard coded attributes of domain objects the same way you present property lists, thus hiding the property list from the user. *However,* the domain layer has to support the transformation. Consider "GUIfication" of transaction-oriented systems as a counterexample: Many clients think, it is enough to adapt the 3270 forms of a host-based system with a sequence of HTML forms to get a 'modern' user interface. Yet, the success of the windows-based user interfaces stems from a completely different paradigm. Modern interfaces do not model business processes but offer a toolkit to modify the business 'material' as the user wants to. Most information on the screen does not contribute to the specific business process the user currently is in - it is additional information supporting *the user* for other processes or if she changes her mind. From a technical perspective this means that the system has to provide much more information than in a traditional system, where the user knew her transaction code and entered exactly the data *the system* needed. Consequently the domain layer of these systems is optimized to process transactions, not to provide all the additional information. In the best case you will run into performance problems. In the worst case you have to rewrite the complete system.

Known Uses

Nearly all popular user interface architectures use this pattern. I will pick three examples:

1. The *Seeheim* user interface architecture features an application kernel as domain layer and a user interface layer. It subdivides the latter in two more layers, called the presentation layer and the dialog control layer. Because this architecture has its roots in transaction oriented systems, there are no callback mechanisms from the application kernel back to the user interface.

2. Perhaps the most popular example is the *Model-View-Controller* architecture *[BMR+96]*. The View and the Controller classes form the user interface layer while the Model classes form the domain layer. Views and Controllers are allowed to send specific messages to Models but not vice versa. An *Observer* serves to provide a callback mechanism from the Model to the Views.

3. IBM Visual Age for Smalltalk lets the programmer assemble the application from *Parts*. There are *Visual Parts*, such as menus, tree views, and windows, and there are *Nonvisual Parts* that usually wrap ordinary Smalltalk classes. Used in a disciplined way, the Visual Parts correspond to the user interface layer while the Nonvisual Parts correspond to the domain layer.

See Also

[*BMR+96*] discusses layered architectures in depth.

Subsystem Architecture

After you have decided to use a *User Interface Layer*, the work has just begun. Remember the main purpose of the separate layer: Encapsulate the complex user interface. Hence, it is usually not a good idea to implement the complete layer as one monolithic part. So the next step is to look for subsystems. There are four patterns that generate the overall architecture: *Separate Transformation* and *Widget Model*, *Domain Layer Access*, and *Context Support*. While you find variants of *Separate Transformation*, *Widget Model*, and *Domain Layer Access* in nearly every user interface architecture, *Context Support* depends on the environment.

Separate Transformation

Thumbnail Presenting information on the screen and processing user actions are two different complex tasks, ... therefore separate the transformation of domain information to the screen from the transformation of user actions to domain layer calls.

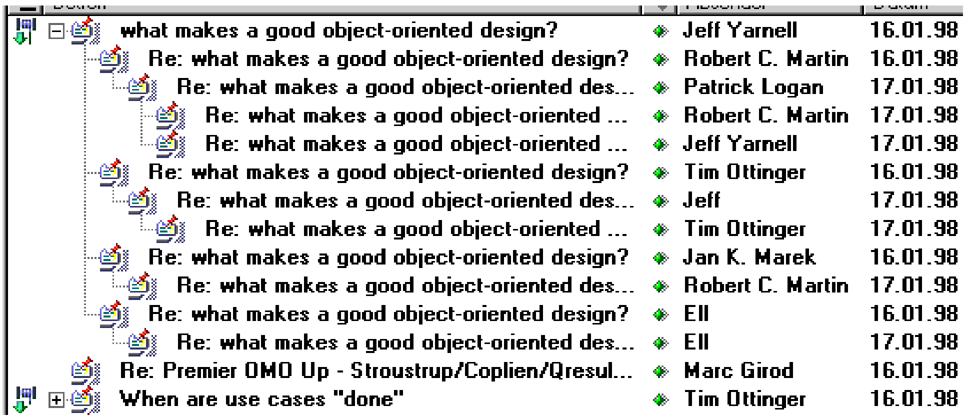
Context After you have decided to use a *User Interface Layer*, the work has just begun. You have to define the detailed structure of the user interface. Often input and output are clearly separable concerns. Especially when you are using direct manipulation and the Tool-Material metaphor, displaying the information and processing manipulations can both become quite complex. So...

Problem ...how do you assign input and output processing to subsystems?

To decide for a solution you have to take several forces into account:

Forces Visualization and manipulation are complex tasks,...
...but they are closely coupled.

> Consider a tree view on the different threads of a news reader below. It consists of several different graphical elements, such as lines, icons, and so on. All these graphical elements show domain level information, such as status. Controlling this presentation calls for some pretty complex software. In addition to its appearance this window also offers the opportunity to read a posting, mark it, answer it in several ways, or archive it. The user accesses most of these actions with pop-up or pull-down menus and with drag-and-drop techniques. To control these actions and forward them to the domain layer are also quite sophisticated tasks. So it might be tempting to split visualization and manipulation. *However, is this really a good idea?* Both parts are closely coupled on two sides: At first they work on the same domain level objects, such as a news posting. On the other end the same interface entity may represent both visualization and manipulation. Consider the small diamonds in front of every author. On the visualization part it shows that I have not read this posting by now (although I should have). If you click this diamond, the reader marks the corresponding posting as read, which is clearly a manipulation task.



what makes a good object-oriented design?	Jeff Yarnell	16.01.98
Re: what makes a good object-oriented design?	Robert C. Martin	16.01.98
Re: what makes a good object-oriented des...	Patrick Logan	17.01.98
Re: what makes a good object-oriented ...	Robert C. Martin	17.01.98
Re: what makes a good object-oriented ...	Jeff Yarnell	17.01.98
Re: what makes a good object-oriented design?	Tim Ottinger	16.01.98
Re: what makes a good object-oriented des...	Jeff	17.01.98
Re: what makes a good object-oriented ...	Tim Ottinger	17.01.98
Re: what makes a good object-oriented design?	Jan K. Marek	16.01.98
Re: what makes a good object-oriented des...	Robert C. Martin	17.01.98
Re: what makes a good object-oriented design?	Ell	16.01.98
Re: what makes a good object-oriented des...	Ell	17.01.98
Re: Premier OMO Up - Stroustrup/Coplien/Qresul...	Marc Girod	16.01.98
When are use cases "done"	Tim Ottinger	16.01.98

Visualization and manipulation address different domain layer features,...
...but usually work on the same domain level objects.

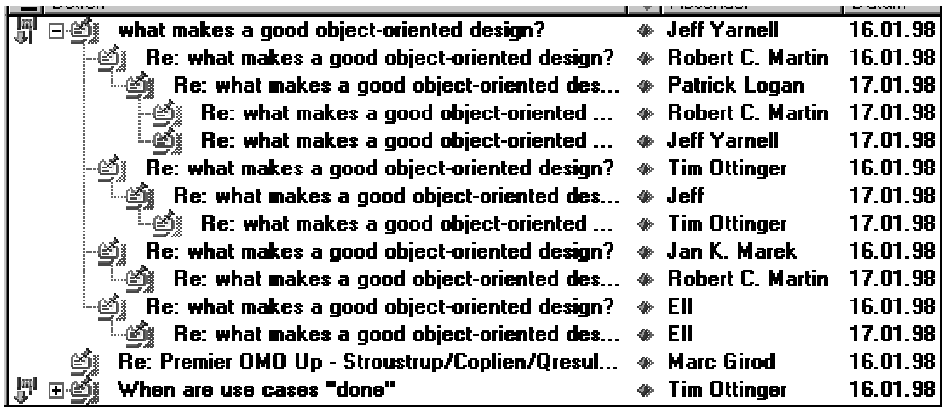
Although visualization and manipulation may look nearly the same on the interface, they usually address different features of the domain kernel. At least they call getter and setter methods respectively. In a context-sensitive pop-up menu of a tree view the menu items may address methods of the node itself, of its father, or of a factory. So considering visualization and manipulation as being completely different seems to be a tempting approach. *However, is this really a good idea?* In a well-designed user interface, the manipulation usually is at least related to the selected object: It may effect the object itself, its class or a directly related object. So you usually start the navigation at the current selection. Maintaining the selection redundantly surely is prone to errors. Hence if you separate visualization from manipulation you are going to have a lot of communication between both.

Your design may suggest a straight forward separation,...
...but often the framework already dictates a certain separation

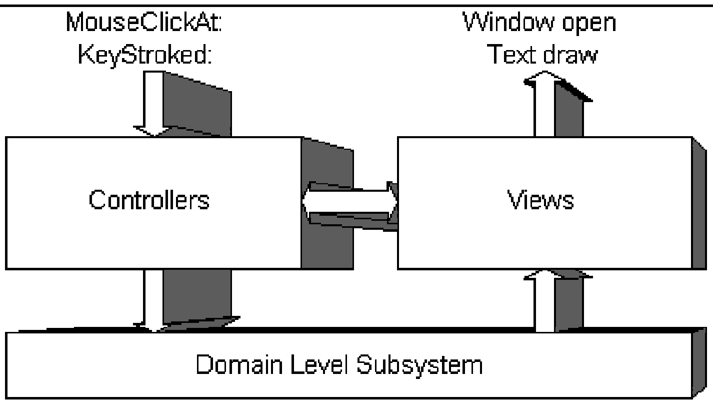
Sometimes it may be easy to identify a certain separation as the best one, based on the other forces. So you might be inclined to implement it in any case. *However is this really a good idea?* In most cases you use a class or function library to address the awkward details of user interface control. Most of them are organized as frameworks, promoting a certain separation. Using a different one may be feasible but most often is just a bunch of additional code with all the additional software you have to code, to test, and to debug. As a rule of thumb it is unwise to change the architecture your framework supposes, though it may be wise to extend or refine it.

Solution

Therefore, separate user interface processing into two subparts: A *View* displays the data on the screen and a *Controller* handles user input. Let the current *Selection* define the context both objects work upon. Usually every single view has its own controller.



what makes a good object-oriented design?	Jeff Yarnell	16.01.98
Re: what makes a good object-oriented design?	Robert C. Martin	16.01.98
Re: what makes a good object-oriented des...	Patrick Logan	17.01.98
Re: what makes a good object-oriented ...	Robert C. Martin	17.01.98
Re: what makes a good object-oriented ...	Jeff Yarnell	17.01.98
Re: what makes a good object-oriented design?	Tim Ottinger	16.01.98
Re: what makes a good object-oriented des...	Jeff	17.01.98
Re: what makes a good object-oriented ...	Tim Ottinger	17.01.98
Re: what makes a good object-oriented design?	Jan K. Marek	16.01.98
Re: what makes a good object-oriented des...	Robert C. Martin	17.01.98
Re: what makes a good object-oriented design?	Ell	16.01.98
Re: what makes a good object-oriented des...	Ell	17.01.98
Re: Premier OMO Up - Stroustrup/Coplien/Qresul...	Marc Girod	16.01.98
When are use cases "done"	Tim Ottinger	16.01.98



Consequences

The design successfully separates the different tasks ,...
...still both parts are closely coupled

Both classes now have clearly defined responsibilities and implement two different concepts. If you have different views of the same object with the same mechanisms for manipulation, it is often enough to have a single Controller class but several View classes. *However*, they are coupled quite closely. Changes to the appearance usually also change the possible interactions and vice versa. Changes to the domain level object may now even result in changes of two other classes.

The Controllers mainly address the setter methods of the domain level objects while the views usually use the getters and navigate though the network of objects,...
...still both objects tend to work with the same domain level instance, so you have to provide *Context Support* for the context common to both - an extra effort.

In a naive approach (and many 4GLs) the controllers just set the attributes of the domain level object. In a more sophisticated system, manipulations are not that easy. Before the user chooses any action you have to set up menus and to enable menu items and buttons depending on domain level states and the current selection, and you may have to restrict parameters. After the user has submitted an action, you may have to retrieve additional parameters, check them for validity, save undo information, and so on. As you might already guess, this often is too much for a single class. Hence the Controller often uses *Command*, *Availability Methods*, and *Domain Level Type*.
Frequently The task of the View is more complex than just initializing a label with a string retrieved with a getter method: The result has to be formatted and the presentation may vary depending on the concrete subclass of the object retrieved. For instance, consider a view for a circuit diagram: The symbol depends on whether you are showing a transistor or a capacity. *Domain Level Type* help to do the formatting.

As a bottomline, Views and Controllers address different parts of the domain level object's protocol. *However*, they still address the same domain object, usually the one the user has selected. There are two possibilities: The selection can be a direct selection in a navigation widget, such as the tree of postings in the example above. In a form-based user interface, the user often has to select the object using some form of query, such as match code search. In general there is an additional context that defines on which objects the View and the Controller works. Therefore you will have additional effort for *Context Support*.

In any case, be sure your architecture conforms to the separation.

It is usually very unwise to implement a design which does not comply to the architecture your framework supports. If you think that *Separate Transformation* is the right choice for your application but your framework supports a *Widget Model* you can either use another framework or change your user interface. Do not try to build a skyscraper on the foundations of a cathedral. Only rather low-level frameworks, such as AWT, permit both architectures.

Known Uses

1. Model View Controller *[BMR+96]* is the classic example of this pattern. The Model is the domain level object and the context at the same time while Views and Controllers exactly conform to the pattern.

2. http, the Hyper Text Transfer Protocol also uses this pattern in a slightly different variant. Consider the html pages as the Views, while the CGI scripts are the controllers. The arguments of the script contain the context. Whether you have a clearly defined domain level object depends on the purpose of the page.
   More advanced pages use Java applets to perform complex user interfaces. Because the AWT supports *Widget Models*, this usually results in a mixed architecture. However, AWT is rather low-level, so it is still possible to use *Separate Transformation* using AWT.

3. CICS also offers two completely different mechanisms for visualization and manipulation: Transaction codes determine what program you start - and therefore what manipulation you do. Maps encapsulate output to the terminals. Because CICS is transaction-oriented rather than object-oriented, it does not maintain any context. There are several ways to provide *Context Support*, I will discuss below.

4. Dirk Riehle describes this pattern as "Trennung von Interaktion und Funktion" (Split between interaction and function) in the context of the tool material metaphor *[Rie97]*

Related Patterns

Widget Model is an Alternative architectural strategy that is useful if the coupling between View and Controller becomes too tight. In fact, user interface architectures often use *Widget Model* on an architectural level while they use *Separate Transformation* on a lower level. If manipulation becomes too complex you also find the opposite order.

Widget Model

Thumbnail Every widget on the screen has its own data and functionality ,... therefore let a set of objects of the user interface model the widgets on the screen.

Context Most graphic user interfaces consist of a hierarchy of widgets: Windows contain subwindows, which contain areas, which contain elementary widgets, and so on. Most widgets display some information *and* do some manipulation. Often every elementary widget presents a certain domain level object or an attribute of it. Aggregating widgets often present aggregations of domain level objects. So...

Problem ...how do you model the user interface if the structure of your widgets is closely coupled to your domain model structure.

To decide for a solution you have to take several forces into account:

Forces You want the architecture to define as much of the structure as possible,...
...but it is usually hard to find a simple architecture that covers all topics.

The more an architecture defines the better are your chances to get a homogenous and easy-to-maintain system. Especially inexperienced architects tend to search for architectures that cover all requirements, which might show up in the next decades. *However, is this really a good idea?* Simplicity is one of the most important features of an architecture. The more complex it is, the higher its risk to have severe bugs. A simple but powerful concept is what most architects head for.

You want classes to be coupled loosely,...
...but you would also like them to have as few responsibilities as possible.

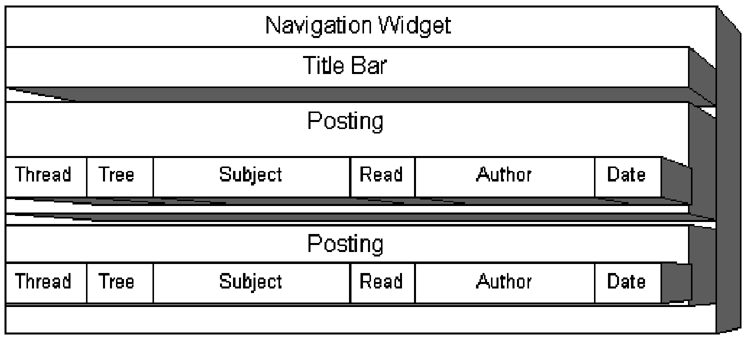
This sounds like one of these all-time force pairs, but it is one of the main force that characterizes this pattern. Besides bad design, there are two major sources of coupling in user interfaces: Classes that collaborate to present the same domain level objects and user interface items that have to collaborate to form the complete interface the user sees. You can fight the first source by putting all knowledge you need to present a domain object into a single class, while you can fight the second source with generic protocols for all the user interface items. *However, is this really a good idea?* Putting all the domain presentation of a single class into one object may lead to monster classes with a shopping list full of responsibilities. Not exactly a one-class-one-concept approach. Generic protocols are also hard to define given the flexibility you usually need.

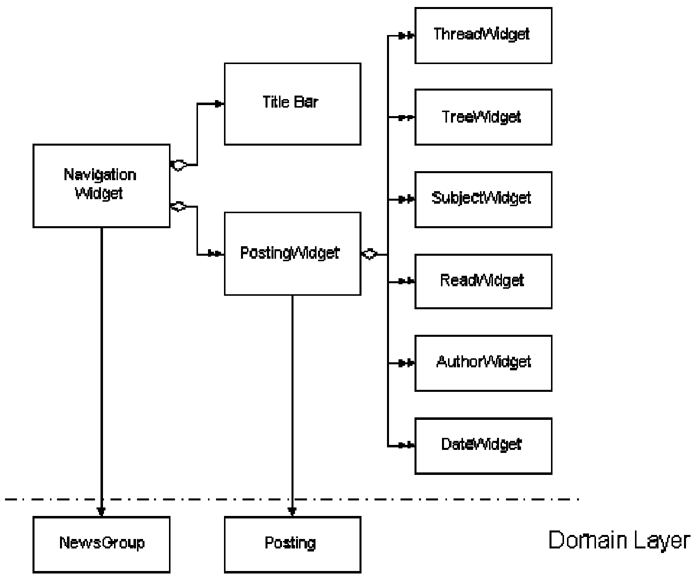
You want to reuse classes directly if you encounter a similar widget,...
...but sometimes you achieve the best reuse by exploiting meta information.

This also sounds like a very generic force, but it is still worth to be examined in depth. Given the complexity of user interface classes the best code is the one you do not have to write (according to Kent Beck). Especially if you look at form-based user interfaces of large system, you often find the same subforms over and over again with only slight variations. Naturally you want to use the same classes for all of these subforms. *However, is this really a good idea?* Reusing the classes means that these are flexible enough to cover all the uses you have. If all your uses are the same, things are fine. If you have a bunch of variants you can either configure a class to a certain variant or you can attach several strategies to make it behave the way you want to. Both easily blow up to a mess. If you have forms which vary in the fields they show, you may also use a completely different approach: You can generate the classes from meta information about the form. Note that generation may also happen at runtime in a reflexive environment. Still, this requires an architecture which defines a small, unified protocol for all classes you would like to generate.

Solution

Therefore, have a single Widget class for every item on the user interface. The Widget class is linked to a complete domain level object or a single attribute of it. It handles both presentation and manipulation of the domain level object. The Widget objects form a hierarchy that corresponds to the item hierarchy on the user interface with the root object modeling the main window. Every node in this Widget hierarchy controls the lifecycle of its children. The figure below demonstrates this design for the news reader.





Consequences

This architecture smoothly defines a fine grained structure of the user interface,...
...still there are important issues left open.

If you can draw a picture of your user interface, this pattern defines most of the structure of its software: Transform the picture into a hierarchy of items, such as windows, subwindows and so on. This is the instance tree you have while the window is active. *However*, there are still some topics left: Most applications have a context that spans the windows. Consider the Clipboard as an example. There is no visible item you can attach this clipboard to, so you need additional classes to provide *Context Support*. Another issue left open is an overall control that cares for initialization before you open the first window and clean-up after you have closed the last window. A separate *Application* class addresses this topic.

You only have to change a single class if you change the domain object,...

...still this class has a bunch of complex responsibilities.

With an object-oriented user interface the widget model also means to reflect domain objects. A widget usually presents a single domain object or parts of it. Because the widget is responsible for all interaction with this object, the knowledge about it is quite concentrated on this class. *However*, this may lead to complex classes. To make things even worse, a widget not only has to care for both directions of domain object communication, but it also has to manage its children's lifecycles - including the layout. With a complex widget this usually leads to classes that suffer from elephantiasis. Hence, complex widgets often use *Separate Transformation* as micro architecture. Another measure is to use *Automatic Layout* to support layout management. Other patterns that help to break up the classes are *Domain Layer Access*, *Command*, *Availability Method*, and *Domain Level Type*.

A Widget Model leads to a natural reuse of widgets,...
...still it is not suited very well for variants.

Reusing a widget is straight forward: You just have to create a new instance of the same class from another parent widget. *However*, the more variants of the widget you have, the more complex the class becomes, because the more flexibility it needs. You have to implement hooks and strategies to conform to all the variants. The simplest way to fight this is to redesign the user interface avoiding all the variants.

It is hard to generate widgets from meta information about the domain object you have to present

With some form-based interfaces it is sufficient to know the objects and attributes you want to include in a form to generate it. You give the form class a property list, run an *Automatic Layout* and display it. You can generate a large share of "insert, update, delete" forms this way. It is important to note that the manipulation part here is nearly stable, only the data presented varies. This approach also needs highly standardized protocols for every form. A Widget Model does not encourage both requirements. For this approach *Separate Transformation* is the better architecture.

Known Uses

1. Application Document View is the most popular example of this approach. The Macintosh first used this architecture and most window libraries have adopted it. While Application and Document manage the context, the Views are the roots of the opened Windows. Every View consists of subviews, consisting of widgets, and so on.

2. Presentation Abstraction Control *[BMR+96]* extends this concept into the domain level with a hierarchy of collaborating "Agents".

3. AWT, Java's GUI library, assembles the user interface of "Components" which manage presentation as well as manipulation of the data they display.

4. Visual Basic is the most popular non-OO environment that uses this architecture: A "Form" recursively consists of "Subforms" which finally contain "Controls". All of them have certain events that are called when the user chooses to do something. The lack of mechanisms to implement further micro-architectures for every form is one of the main problems with Visual Basic for large projects.

Related Patterns

Separate Transformation is an alternative if you can separate input and output clearly. Often both patterns are combined, forming a hierarchy of View-Controller pairs.
Chain of Responsibility *[GHJ+94]* is the classic design pattern to manage the communication between widgets.
Mediator *[GHJ+94]* is a common way to reduce coupling between several subwidgets.

Domain Layer Access

Thumbnail

The interface to the domain layer sometimes is quite sophisticated,... therefore designate a special set of objects to access the domain kernel.

Context

If you have decided to use a *User Interface Layer*, the structure of this layer not only has to tell you how to construct the classes for user interface control, but also has to contribute to the functional requirements of the system. Fancy windows and forms help the user to access the system but they do not represent any value in itself. So...

Problem

...how do you access domain layer functionality from the user interface software?

Solution

Introduce a set of methods and classes to access the domain layer. Be sure you consider the following topics when you design these classes:

- Commands to domain objects. The *Command* pattern addresses this topic.
- Handling errors during the execution of commands. *Command* also is a good place to care for errors. Error Handler *[Ren96]* is suitable for a more sophisticated error handling.
- Retrieving information about the availability of commands. You can use *Availability Method*s as a unified protocol.
- Retrieving information about the possible values of entry fields. *Domain Data Types* discuss a solution for this problem.

Observer

Thumbnail

Sometimes you want to enable the user to display several views of the same domain object at once but you want to avoid to call the user interface from the domain object if the object changes,... therefore let the views register at the domain object, which sends a change notification to all registered objects whenever it changes.

See Also

[*GHJ+94*]

Command

Thumbnail Issuing a command to a domain level object often is related with additional responsibilities, such as determining availability of the manipulation, retrieving additional parameters, error handling, undoing, and setting the boundaries of transactions,... therefore model the commands as separate classes with these responsibilities.

See Also [*GHJ+94*]

Availability Method

Thumbnail In most user interfaces you have to check whether a certain action is legal in the current context without actually performing the action,... therefore add an Availability Method for every manipulation method to the domain object, taking the same parameters as the manipulation method but answering a Boolean without any change of the domain layer.

Domain Level Type

Thumbnail Formatting and legal values of entry fields are often determined by the nature of the data to be displayed or entered,... therefore use separate classes with a common protocol for your domain level types that have these issues as their sole responsibility.

Also Known As Whole Value [*Cun95*]

Context Support

Thumbnail Especially in a transaction based environment the system does not store context information you need to support the user optimally,... therefore take care to support a context for every user.

Context User interfaces usually have to maintain some context while the user stares on the screen and scratches his head. You have to store information about the displayed objects, the current selection and the state of dialogs. So...

Problem ...where do you store this information?

Solution Introduce mechanisms that store the context. The mechanisms depend on your distribution architecture, the available bandwidth between client and server, and the scalability your system has to provide. For single-user systems and fat client architectures *Application*, *Document*, and *Selections* help to deal with specific context information. For thin-client architectures you can either use *Session Memory* to optimize bandwidth on the expense of scalability or *Cookie*s to optimize for scalability on the expense of bandwidth. For highly sophisticated systems you can combine both to enable fine-tuning.

Application

Thumbnail In a window environment you need a place to store general information you cannot assign to a window,... therefore have an Application object as a Singleton [*GHJ+94*], caring for initializing, cleaning up, and storing general information.

Document

Thumbnail In most window environments the user is able to open several views on the same set of domain objects,... therefore assign each window to a Document which manages the root of the presented set of domain objects.

Selection

Thumbnail Because the current selection controls most elements of a user interface, it is hard to assign it to any other element,... therefore introduce a selection class that contains the references to the domain objects currently selected. Send all manipulative actions to domain objects using these references.

Session Memory

Thumbnail In a transaction based environment the system does not maintain context information between two transactions, but often dialogs span several transactions,... therefore allocate a session memory for every client on the host that stores the context.

Cookie

Thumbnail In client/server environments with a very large or unpredictable number of clients the resources of the server may not suffice to store the context information of every client,... therefore send the status as data to the client and let the client store it. These little chunks of context information are called Cookies.

Some Special Patterns for Form-Based User Interfaces

Centralized Control

Thumbnail Form-Based user interfaces usually feature a complex state model with many common events and a high change rate on the states while the single steps of a dialog are quite generic,... therefore introduce a centralized dialog control to maintain the state.

Automatic Layout

Thumbnail The mutual placement of widgets in forms is an awkward work if you have a lot of forms, but it follows computable rules,... therefore implement these rules in an layout component rather than drawing each form by hand.

Acknowledgements

I'd like to thank everyone who has supported me in writing this document. Wolfgang Keller, Andreas Hess, Uli Zeh and Klaus Renzel have provided support for earlier versions of this pattern language, Jim Coplien provided significant insights to find the structure of it and long discussions with Alistair Cockburn helped to eliminate process patterns and find pattern names. Particularly I'd like to thank my PLoP shepherd Bob Hanmer for his help and his patience.

References

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Michael Stal, Peter Sommerlad: *Pattern-Oriented Software Architecture - A System of Patterns*; John Wiley & Sons, Chichester, 1996

[CoK97] Jens Coldewey, Ingolf Krüger: *Form-Based User Interfaces - A Pattern Language*; Proceedings of EuroPLoP '97, Siemens AG, TR 120/SW1/FB, 1997

[Col95] Dave Collins: *Designing Object-Oriented User Interfaces*; Benjamin Cummings, Redwood, California; 1995

[Coo95] Alan Cooper: *About Face - The Essentials of User Interface Design*; IDG Books Worldwide, Foster City, California, 1995

[Cun95] Ward Cunningham: The CHECKS Pattern Language of Information Integrity in Coplien, Schmidt: Pattern Languages of Programming, Addison Wesley, 1995

[GHJ+94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software*; Addsion-Wesley, 1994

[GrR93] Jim Gray, Andreas Reuter: *Transaction Processing - Concepts and Techniques;* Morgan Kaufmann Publishers, San Francisco, California, 1993

[Ren96] Klaus Renzel: *Error Handling - A Pattern Language*; sd&m GmbH&CoKG; 1996; available via http://www.sdm.de/g/arcus/

[Rie97] Dirk Riehle: *Entwurfsmuster für Softwarewerkzeuge - Gestaltung und Entwurf von Anwendungen mit grafischer Benutzungsoberfläche*; Addison-Wesley, Bonn; 1997

[Tog92] Bruce Tognazzini: *TOG on Interface*; Addison-Wesley, Reading, Massachusetts, 1992