

95.540 Project

Hats: Separating Roles of an Object in OOD

Marcin Pilat
marcin1@home.com

December 10, 2001

Abstract

The Hats pattern tries to improve the structure and efficiency of an Object Oriented System Design. It separates the roles that an object has ("Hats") from the internal state of the object ("Data Object"). The Hats implement the behavior of their roles and use the Data Object to get its internal state. Hats can be used by different Clients in the application that do not need to know or care about the other roles the object has. This abstraction provides an effective separation of roles, both structurally and behavior-wise.

Problem

To separate the various roles an object could play from the internal representation of the object.

Context

The Hats Pattern is a design pattern that applies to Object Oriented Design. The pattern is useful when applied on an object in one or more of the following circumstances:

- *The object has multiple behavioral roles in the system.* While we usually want to stay away from defining complex objects with multiple behaviors/roles, we are sometimes faced with objects that are used differently by various parts of the application. Each such use can be thought of as a role the object plays and it can be described by a set of functions and their implementation that defines the behaviors of that role.
- *The object is copied often.* Objects that are copied often must be minimized in order to minimize the time for copying of those objects. Sometimes only a part of the object needs to be copied because other parts are usually unchanged or are not useful in some situations. The overhead in copying the entire object instead of a part of it can be large when copying is done often and/or the object is large.
- *Many instances of the object class exist in the system at any time.* Similar to the frequent copying situation, the instances share common behavior and might share some common data. Some of this behavior and data is only useful for a certain situation (a role the object might be in) and is not needed for others. By the use of inheritance and internal function tables, the overhead in behavior is minimal, but the overhead in data might still remain.
- *A part of the object is used by another thread or process.* Usually, this part of the object must be extracted and passed to the thread/process. This extraction can be accomplished by specialized Factory Methods Pattern [5] which create objects that are based on parts of the object. This can be avoided if the object can be broken down so that the required part can be copied using normal means.

Forces

- *Objects with many behavioral roles tend to be too complex.* In an OOD framework, objects with one role are often preferred. The addition of extra

roles to an object often adds extra behaviors and often internal state to the object. Small object with simple behaviors are easier to maintain and quicker to reproduce.

- *Development of new objects is expensive.* In a Software Development environment, the number of objects in the system often counts towards the complexity of the system, which is used for time management decisions. Development of several classes is usually more time consuming than the development of a single class. The development time depends on the complexity of the class and a balance between those items must be reached.
- *The number of methods in a class should be minimized.* Objects with many methods are harder to maintain and reuse. The methods of an object should represent related behaviors. It is often discouraged to place methods that are for unrelated behaviors in a single object.
- *Usually, only one role of an object is used in a part of the system.* It is usually the case that parts of the application are only interested in one role the object could play in the system. Therefore, it is not worth using a multi-roled object in all circumstances, but need to only use parts of it at a time.
- *Adding levels of indirection slows down the execution of methods.* Creating wrappers for methods or encapsulating an object in another object often leads to decrease in system performance. The overhead of calling more functions may be minimal in itself, but can add up to lower the performance of the system if done frequently.
- *Classes should have meaningful names.* When developing a large system, it is important to use a class naming convention that will let developers quickly determine the purpose of the class and its function.

Solution

Provide separation of the roles of an object from its internal presentation by using objects called Hats.

The participants in the solution are: Clients, Data Object class and Hat classes. The designed object in question (from now called the multi-roled object) can be split into a series of Hat classes and a Data Object class. Figure 1 shows a structure diagram of the Hats Pattern.

Separate classes called Hats represent the roles of the multi-roled object. The Data Object is the core of the multi-roled object that stores the internal state of

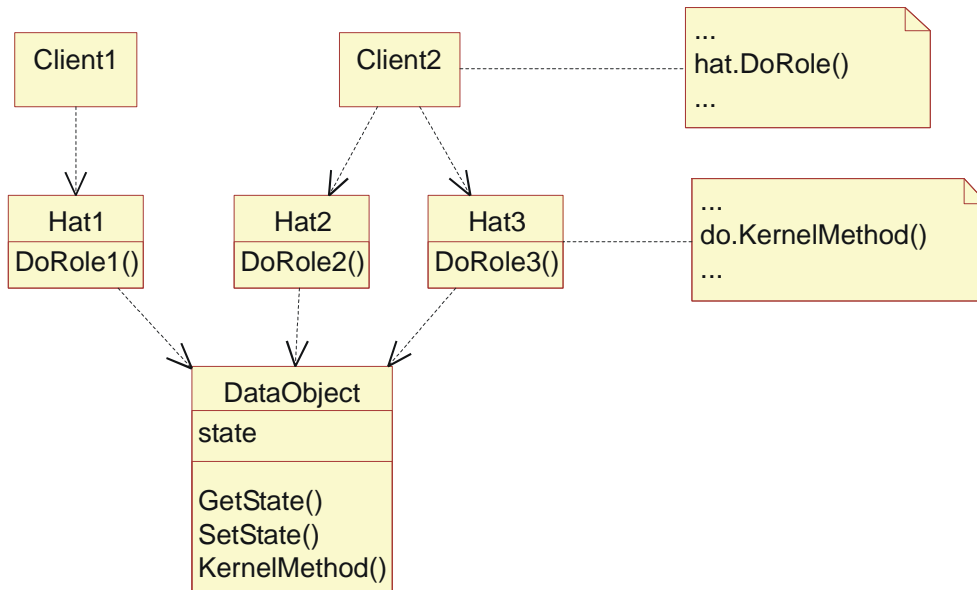


Figure 1: Structure of the Hats Pattern

the object and offers methods to modify the internal state. The methods of the Data Object are the Kernel methods of the multi-roled object as in the *Identify Message Layers Pattern* [1]. The Clients are parts of the application that need to use a particular role of the multi-roled object. In most systems, a Client will only need to use one particular role of the multi-roled object at a given time.

When a Client needs to use the multi-roled object in a particular role, it first acquires the appropriate Hat (defining the role) and uses the Hat on the Data Object to access the behavior associated with that role. Figure 2 shows the collaboration diagram between the Participants in the Hats Pattern.

The meaning of the name of the pattern - Hats - comes from the fact that the Hats represent the roles the multi-roled object plays in the system. When the multi-roled object needs to be a particular role, it puts on a certain Hat and becomes the role it is to play. Taking the analogy further, we can also define a Hat Rack as a place where Hats can be stored when they are not being used by the system.

Implementation

The application of the Hats Pattern must rely on a few implementation details required for efficient use. The idea of Hats can be designed and implemented in a variety of ways depending on the type of system and the use of the Hats. We

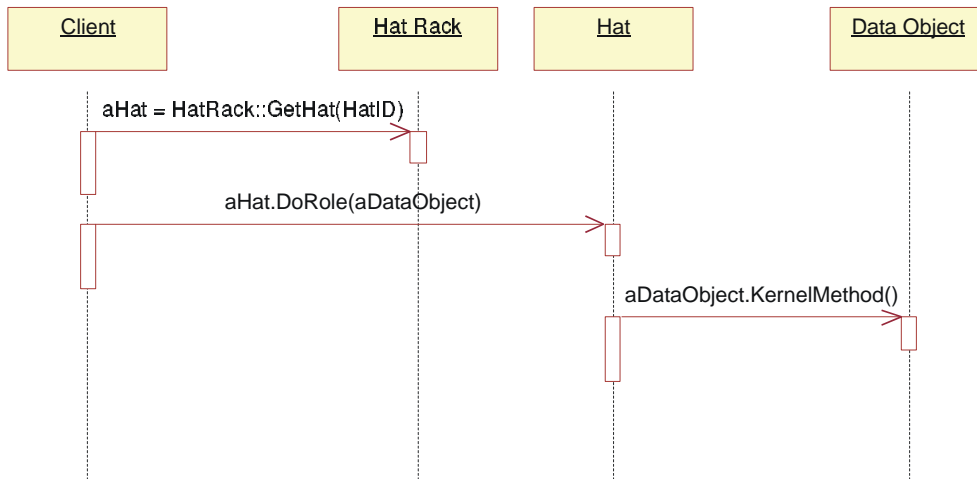


Figure 2: Collaboration diagram between the Participants in the solution.

present a variety of implementation details and considerations that will use Hats to maximum capacity.

Structure

Following are implementation considerations regarding the structure of the Participants in the Hats Pattern:

1. *Hats as collections of static functions.*

It is most often the case that Hats only provide the behavior of the role they represent and do not need to contain any internal state. The state of the multi-roled object is stored in the Data Object and is used by the Hat when necessary.

When this situation arises, Hats can be implemented as a collection of functions that are encapsulated in an abstract class without instances.

In languages like C++ and Java, Hats can be implemented as abstract classes that only contain static functions. Those classes would have hidden constructors, so that they cannot be instantiated. The methods specifying the behavior of the role would be static member functions of the class.

In languages such as Smalltalk, Hats could be implemented using classes with only class methods. The new class method can be overwritten to deny the creation of instances of the class.

It is possible that the Hats require the storage of some data which is class specific and does not change between the instances of the Hat class. This data can be stored as static variables of the Hat class in C++ and Java, or as class variables in Smalltalk.

2. *Hats as regular objects.*

In some OOD environments, the use of classes defined only using static functions is frowned upon and considered insufficient reason for existence of the classes. The Hats can also be implemented as regular objects with member functions and/or internal state.

Care must be taken to ensure the Hat objects are only used in conjunction with their Data Objects. A mechanism for storing the Hats - the Hat Rack - also needs to be considered.

3. *Hats as singletons.*

In the case where the Hats are implemented as regular objects but do not have internal state (or the internal state of a Hat does not change between instances of the Hat class) we only need one instance of each Hat class to exist in the system.

The Hats can then be implemented as singletons, using the *Singleton Pattern*[5]. This pattern guarantees that there is only one instance of each Hat class in the system. Singletons make the system space-efficient and reduce its complexity by minimizing the number of objects.

In languages that do not contain garbage collection mechanisms, the system implementors must take care that the singletons are properly deleted from the system when they are no longer necessary, thus removing memory leaks. A Hat rack might also be necessary to store the Hat singletons in an accessible place in the system.

4. *Data Objects as data storage and access classes.*

The Data Objects should not provide any role related functionality in the system. This prevents the Data Objects from being used directly by Clients to maximize the separation between the roles of the multi-roled object and its internal state. The Clients in the application should only be interested

in the roles of the multi-roled object and it is the responsibility of the roles to access the data from the Data Object.

Data Objects should store the internal state of the multi-roled object and should provide only Kernel methods as in *Identify Message Layers* [1] and *Encapsulate Concrete State* [1]. This ensures that the Data Objects have the optimum number of interface methods.

Data Objects and Hats

We need to consider how the association between the Data Objects and their Hats can be accomplished in the system. In each scenario, the Hats use Delegation [4] to the Data Object in order to retrieve the appropriate information from it and use the data for a particular purpose. We provide a few implementation considerations below:

1. *Data Objects as Hat function parameters.*

When Hats do not have internal state and are defined as a form of a singleton or collection of static functions, we can link the Data Object to the Hat by specifying the Data Object as an argument to the member functions of the Hat. The Client would retrieve the proper Hat and execute the required functions on the Hat passing the Data Object to each function. Sample Client code is shown.

```
Client::Function()
{
    ...
    aHat.DoRole(aDataObject);
    ...
}
```

This requires the Client has access to the Data Objects and the Hats. Since the Hats do not internally store the instance of Data Object they are working on, only one instance of the Hat can exist in the system (singleton) or static functions can be used.

A nice separation between the Data Object and the Hats is established where the Hats can be used by any number of Data Objects. The human analogy would be that any person can put on a miner's Hat and then

become a miner. The Hat does not know whose head it will be on until it is put on somebody's head to do a particular task.

2. *Hats know which Data Object they are assigned to.*

It is often the case that when a Hat is being used on a Data Object, the Client will call many methods on the Hat pertaining to the same instance of the Data Object. To minimize the impact of supplying the Data Object as a parameter to every function that is run on a particular Hat, the Data Object associated with a Hat can be referenced in the internal state of the Hat. Sample Hat method call is shown.

```
Hat::DoRole()  
{  
    ...  
    myDataObject.KernelMethod();  
    ...  
}
```

This means that there needs to be a function in each Hat that would set the Data Object that is currently using the Hat. A pointer or reference to the Data Object would be a member variable of the Hat class. Whenever the Data Object instance would change, the Data Object reference set function would be called on the Hat.

The Hats can still be implemented using singletons or static classes since the Client will be able to use only one instance of a Data Object at any time, notifying the Hat when switching the instances. If the Hats are not singletons, this methodology would still work the same way.

Care must be taken that before any block of function calls is done on a new Data Object, the Hat must be notified of this change, so that the proper Data Object is used by the Hat. If only a few function calls are done between each Data Object change, this method will be less efficient than specifying Data Objects as function parameters since the setting of the new Data Object instance introduces extra overhead.

3. *Data Objects know their Hats.*

The opposite situation to *Hats know which Data Object they are assigned to* can also be used. The Data Objects can store a list of Hats, one for each role the multi-roled object might play in the system.

When the Client would want to use a role of the multi-roled object, it would query the Data Object for a Hat corresponding to the role. The Client would then use this Hat either by setting its reference to the Data Object (which could be done automatically by the Data Object) or by specifying the Data Object as a function parameter to the Hat functions. This is shown by a code example.

```
Client::Function()
{
    ...
    aHat = aDataObject.GetRole1Hat();
    aHat.DoRole(aDataObject);
    ...
}
```

The list of Hats associated with a Data Object needs to be stored by the Data Object. This could be done by storing a collection of references to Hats in the Data Object.

This solution might not always be desirable in that it adds extra state information to the Data Object that links it to its Hats. This weakens the separation between the Hats and Data Objects and may be viewed as contrary to the Hats Pattern itself.

4. *Hats through Data Object types.*

A variant of the *Data Objects know their Hats* method is to keep the separation between the Data Object and its Hats at maximum and store the association between the Data Object and its Hats outside the Data Object class.

Each Data Object can be given a type. This type would uniquely identify the Data Object class. Storing of this type could be done by static

data members (C++, Java) or class variables (Smalltalk) of the Data Object. Unique class enumeration can be tricky in most languages, thus a string/symbol identifier is preferred.

The Client would ask a Data Object about its type. The type would be then used by the Client to query some global dictionary for the particular Hat corresponding to the given type. The dictionary linking the type to a set of Hats needs to be registered. This registration can take place during construction of the Data Object. The Data Object would then register its type with the Hats that it can use. A code sample is shown.

```
Client::Function()
{
    ...
    hatType = aDataObject.GetRole1Type();
    aHat = HatRack::GetHat(hatType);
    aHat.DoRole(aDataObject)
    ...
}
```

5. *Hats identified by types.*

In order to retrieve a reference to a Hat from a Data Object instance (either by *Data Objects know their Hats* or *Hats through Data Object types*), the Hat also needs to be uniquely identified.

This Hat identification can be done by the Hat class name when possible or by a Hat type that uniquely identifies the Hat. As in the case of Data Object types, the type is best to be represented by a string or symbol.

Hat Rack

The implementation of the Hat Rack (the way to store the Hats in the system) needs to be addressed. The following implementation considerations apply to the structure of the Hat Rack:

1. *Hat Hierarchy.*

As with every collection of similar classes, we want to organize them as best as possible. We can use a Hat class hierarchy to organize the Hat classes. An abstract Base Hat class can serve as the root of the Hat hierarchy from which all other Hat classes would be derived.

In C++ a Hat hierarchy can be created by making the Base Hat class a collection of pure virtual function declarations that are shared among all Hat classes. Interfaces can be used in Java in a similar manner to define a Hat interface that all Hat classes must derive from.

We can also introduce other abstract classes that some Hat classes will derive from. Those abstract classes would join classes that share similar behavior, functions, and/or internal state. Those abstract Hat classes would all derive from other abstract Hat classes, forming an organized hierarchy.

2. *Globally accessible dictionary.*

To facilitate the finding of particular Hat instances, a globally accessible dictionary of Hats can be used. This dictionary can be keyed by the Hat class type. The key can also contain Data Object class type information to facilitate the search for a Hat of a specific Data Object.

In Smalltalk and Java, a Dictionary can be used. In C++, a map of strings to pointers can be used (MFC or STL). Dictionaries can be keyed by strings or symbols.

The location of this dictionary is also an issue. We want to avoid having global objects in an Object Oriented System. Using a Hat class hierarchy, the dictionary can be placed in an abstract class using static data members (class variables in Smalltalk). This would provide one point of reference to the dictionary.

3. *Manager class.*

The Hat Rack can be a class in the system that uses the Manager Pattern [6]. The class can be made a singleton and it would provide a way of storing and retrieving Hats in the system.

The Hat instances would be added to the manager. The Client would query the Hat Rack class for a Hat using its type. The manager can be

implemented to take into account the Data Object types as well. This would enable the Client to query for a Hat instance that can be used by a particular Data Object.

Class Naming

The naming of classes is an important aspect in Object Oriented Design. We need to make sure that our solution uses a naming system that easily separates the Participants. This could be accomplished as follows:

1. *Hat class naming convention.*

It is important to separate the Hats from the other classes in the system. The name of the Hat classes must convey the meaning of a Hat class and/or its corresponding role.

In C++, one naming convention is to start a name of a regular class with a capital C followed by few capital letters specifying the type of the class and then by an underscore and the real class name. We can use this naming scheme to construct names for Hats. A `CHT_Name` naming scheme will immediately distinguish Hats from other objects in the system.

Another naming convention, often used in many OO-Languages is to append the name of the type of class to the class name. Thus a `ClassNameHat` name would suggest that the class is a Hat class.

Hats can also be named by the role they play in the system, and instead of naming each Hat class a Hat, we could name it as a special type of Hat. Some useful class naming patterns [2] exist for Smalltalk and can also carry on to other OO-Languages.

2. *Data Object class naming convention.*

It is also important to separate the Data Object classes in the system from other classes. This can be done by naming each Data Object class correspondingly.

The naming conventions used are similar as in the case of Hats above. In C++ we can use the `CDO_Name` convention to name Data Object classes. We can also apply this to the end of a class name to form `ClassNameDO`.

After application of proper naming conventions to the Hats and Data Objects, we will end up with classes such as `CHT_Name` (`ClassNameHat`) and `CDO_Name` (`ClassNameDO`) which provide a clear identification of the purpose of each class.

Examples

In order to show the usefulness of the Hats pattern, we present two examples. The first example talks about a CAD application, the second about Dialog Boxes.

Example 1: CAD Application

Computer Aided Design (CAD) Applications are usually big and complex systems. Those systems, when implemented as Object Oriented Systems, use a variety of classes. It is very important that the internal structure of such a system is easy to understand and maintain.

CAD Applications are used to design complex structure from smaller structures. Those smaller structures are usually called components. For the purpose of the example, we consider a CAD Application that can be used to design optical systems using small optical components. A sample view of a CAD Application is shown in figure 3 below.

CAD Applications usually work together with Simulation Software that simulates the working of the complex structure created using the CAD Application. In the design of the CAD Application, we must consider the ease of data and code sharing between the CAD and Simulation Applications. Suppose our CAD Application example integrates the Simulation into the CAD Application.

Now, we consider the component objects of the CAD system. Those objects store data pertaining to the internal structure of each optical component. Each component plays a few roles in the system:

- *User Interface Component.* Component needs to be visually represented in the design layout. Each component must be drawn on the screen. The user must also be able to interact with the component in the design layout. The component can connect to other components in the layout and it can be moved around.

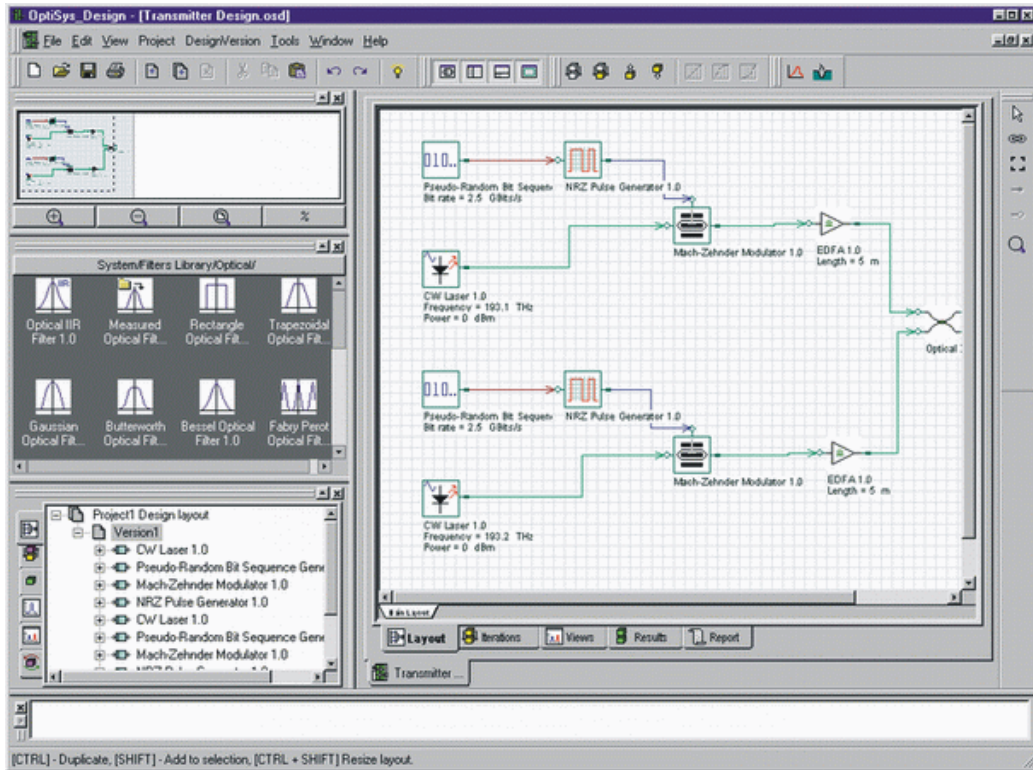


Figure 3: CAD Application - working with components to create a complex structure. Compliments of Optiwave Corporation [7]

- *Simulation Component.* The simulation part of the application uses the component in simulating the entire design of a system created using the components. This part is usually only interested in raw data stored inside the component.
- *Serialization Component.* A serialization mechanism in the application stores the components into a file so that the system design can be later reloaded for editing or simulation.

The above three uses for each component can be thought of as roles the component plays in the system. In the User Interface role, the component needs to be able to draw itself and interact with the user, while in the Simulation role, it needs to be able to simulate itself.

We could ask why the simulation part of the application does the simulating while only getting the relevant information from the data-based component. This is sometimes possible, but for a system with many components that have different structures and different behavior, the design and code for such a simulator would be very complex and difficult to maintain.

There are tens of different types of components in the system and each component can be used many times in the system design. Thus, the number of component object instances is quite large.

The simulation part of the application runs on a separate thread. This thread must run separately with the rest of the application, thus it needs to have a copy of the components that are used in the system.

As we have seen, the components play many roles in the application, there is many instances of them, and a part of the application needs copies of the component objects to function properly. It looks like a component is a perfect multi-roled object to which we can apply the Hats pattern.

We now identify the participants in the Hats pattern:

- *Clients*. We found the following clients: UI Client, Simulation Client, and Serialization Client.
- *Data Objects*. The basic data required to store a representation of each component is placed into a corresponding Data Object. The interface provided to the Data Object is a standard get and set interface for the data.
- *Hats*. Each component has a UI,Simulation and Serialization Hat. The Simulation and Serialization Hats do not need to store state information so can be implemented as singletons. The UI Hat cannot be implemented as singleton since it has to store information about its appearance and state on the design layout.

To make the example small and manageable, we consider two components: Laser and a Strong Laser (Note that the names have been created for explanation purposes and do not necessarily exist in the example domain). The Strong Laser is a kind of a Laser.

The simulation of a Strong Laser differs from the simulation of a Laser, so both components have two different simulation Hats. The components also differ in external presentation and so have two different UI Hats. The serialization of the two lasers is done the same way, so it can be done by one Hat.

A class diagram of the application of the pattern to the CAD application is shown in figure 4. The figure also shows the names chosen for the Hat and Data Object classes in the application.

The following code example shows how the Hats are used by the UI Client. The code is part of the drawing code and is written to show the use of the objects, not to show fully functional code.

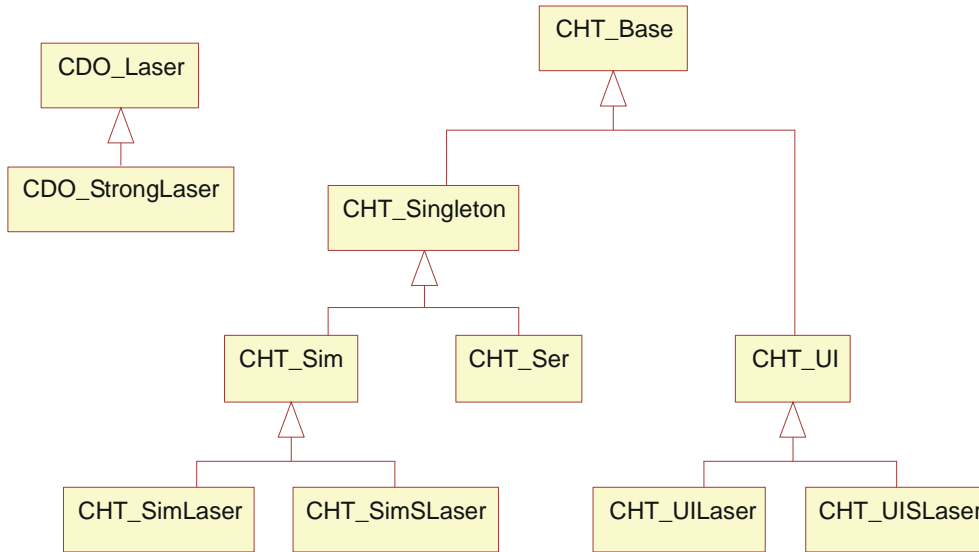


Figure 4: CAD Application - class diagram of two Components using the Hats Pattern.

```

CDO_Laser* pLaser = GetNextComponent();
CHT_UI* pUIHat = pLaser->GetHat(CHT_UI);
pUIHat->Draw(pLaser, x, y);
  
```

Next code example shows an implementation of a Hat function - the SerializeIn function of the Serialize Hat.

```

void CHT_Ser::SerializeIn(CDO_Laser* pDO, CMyArchive& arc)
{
    arc << pDO;
}
  
```

As we can see, the serialization is very simple since it only serializes the entire Data Object. All variables of the component that are not needed for the serialization are not in the Data Object but in different roles in which they are necessary.

A similar situation happens with the Simulation Client. This client only cares about the data that belongs to the component object and about the method that is used to simulate it. The needed data is stored in the Data Object and only this Data Object needs to be copied to be used by the simulation thread. Other roles like the UI role of the multi-rolled object are completely ignored.

Example 2: Dialog Boxes

This example deals with dialog boxes. Many GUI development frameworks use some representation of dialog boxes. In Microsoft Foundation Class Library (MFC) the main dialog class is called `CDialog`, in MacApp it is called `TDialogView`. For the purpose of an example, we will focus on the MFC dialogs.

The primary purpose of dialogs is to display data and/or let the user modify this data using a GUI interface. Dialogs can represent a set of data that is related. An example of this would be the color and font settings dialog of the layout in a word processor. Dialogs can also represent properties of a certain entity in the application. Properties dialog of an graphics object in a word processor or component properties in the CAD Application from example 1.

For an object that needs to be able to provide editing and display of its properties through a dialog, we can treat this behavior as a role of the object. This object usually has more roles in the application. We can merge the behavior of this role with the multi-roled object, but in MFC, we tend to leave this behavior away from the multi-roled object and in a dialog. As we can see, it is an example of the Hats Pattern. A class diagram is provided in figure 5.

The participants in the Hats Pattern will be as follows:

- The client is the part of the application that displays the dialogs to the user.
- The Data Object is the multi-roled object that needs to be displayed to the user on a GUI.
- The dialog box is the Hat of the Data Object that is used to display the properties of the multi-roled object to the user and modify the multi-roled object based on user input.

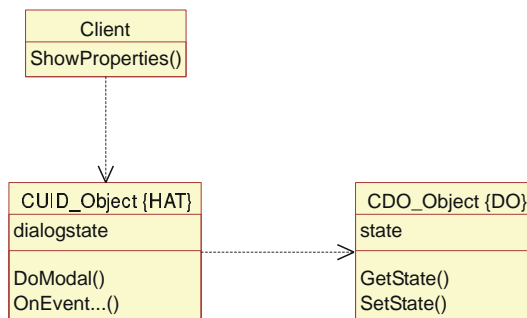


Figure 5: Class diagram of the Hats Pattern application in dialog boxes.

It is often the case in this situation that the dialog receives a reference to the object Data Object so that it can retrieve and modify its properties. The dialog Hat is not usually implemented as a singleton, but can be in situations when the dialog is reused (for instance a dockable dialog for properties of shapes in many drawing programs). It is often the case that a single dialog Hat class can be reused for many Data Objects that share similar internal structure.

As with every application of the Hats pattern, the Data Object can be used for other roles in the application and will not contain any data that pertains to displaying a GUI for it to the user. Dialogs provide a nice separation between the GUI and the data.

Resulting Context

After the Hats Pattern is applied to multi-roled objects in the Object Oriented System being designed, the result is a system in which the roles of the multi-roled objects are separate from each other. The Data Object is not filled with extra role data and behavior.

The application design will contain more classes since the Hat classes are separate from the Data Objects. If the Hat classes are placed into a proper hierarchy, the design will not be hard to read. The number of instances in the system will usually not increase much because most Hats do not need internal data and can be implemented as singletons.

The copying of Data Objects is usually much faster and more robust than copying of multi-roled objects with data and behavior from many roles the object plays in the system. This is especially evident in the case when a worker thread needs to use the data of the multi-roled object but not necessarily all of its roles.

Rationale

The Hats Pattern tries to resolve the forces mentioned above in the following manner:

- *Objects with many behavioral roles tend to be too complex.* After splitting the multi-roled object into the Data Object and its Hats, we spread the behavioral roles onto a number of objects, thus decreasing the complexity of the objects.

- *Development of new objects is expensive.* The extra overhead of creating Hat classes does add to object count in the system. However, the data and behaviors that the Hats contain would be contained within the multi-roled object as well. Thus, the implementation of a multi-roled object should take as long as implementation of the Data Object and its Hats combined. Since Hats can be shared between various Data Objects, it might be possible to decrease the amount of code that needs to be written.
- *The number of methods in a class should be minimized.* A class with less behavior will contain fewer methods. Since a multi-roled object is split into Data Object class and Hat classes, each such class will have fewer methods than the multi-roled object. Each object will also contain only the methods for a related set of behaviors.
- *Usually, only one role of an object is used at a certain time by a certain unique object in the system.* Since Hats split the multi-roled object into its corresponding roles, each Hat can be used independently of the other Hats in the system. Parts of the system that only use one role of the multi-roled object will use the corresponding Hat and will not be concerned about the other roles the multi-roled object might play.
- *Adding levels of indirection slows down the execution of methods.* Hats add a level of indirection to multi-roled objects since they have to go through their Data Objects to get the corresponding data. However, the Data Objects only provide Kernel methods that access their internal structure. The *Identify Message Layers Pattern* [1] states that every object should have Kernel methods, thus applying this pattern to the multi-roled object leads to the same indirection.

The question remains whether the need of Hat finding using the Hat Rack can slow down the execution of the system. There is an extra overhead involved in finding the proper Hats, but the dictionary data structures that should be used rely on hash tables which are efficient in searches.

- *Classes should have meaningful names.* We have discussed a naming scheme that will enable the designers and developers to name the Data Objects and Hats in a way that is readable and meaningful. The Data Objects and Hats can then be distinguished from other objects in the system.

Related Patterns

- **Skin.** The Skin Pattern [8] talks about the separation of the presentation style of an application from its internal logic. When applying the Hats

Pattern to a UI situation, a similar idea is used to separate the UI part of an object from its internal state. Thus, the Hats Pattern is in part the Skin Pattern applied to the structure of the objects in the application, not just the application itself.

- **MVC/DV.** The Model-View-Controller and Document-View patterns [3] also talk about the separation of classes in an application to several categories. The classes in each category play a similar role in the system. The Hats pattern might separate a multi-roled object into its Data Object (Model/Document), UI Hat (View), and another Hat that modifies the state of the object in some way (Controller).
- **Interface.** The Interface Pattern [9] is closely linked to the Hats pattern since each Hat can be thought of as an interface to an object. The roles of a multi-roled object can be split into various interfaces to separate the behaviors. This provides a separation of behavior, but the interface methods are still implemented in the multi-roled object. Thus, there is no structural separation between the roles of the multi-roled object which is acquired by the Hats Pattern.
- **Singleton.** The Hats Pattern often uses the Singleton Pattern when creating Hats. Most Hats can be made into singletons since they do not require instance-specific data.

Known Uses

The CAD Application example presented comes from an Optical Design and Simulation Software developed by Optiwave Corporation [7]. Hats were used extensively during the project.

Dialogs are used very often in a variety of applications. One use for dialogs that applies the Hats pattern is seen in many applications, including: CAD, Graphic Suites, Word Processors.

The Microsoft Foundation Class Library Document-View framework is an example of the Hats pattern. The main Document class (CDocument) is a Data Object class while its Views (CView) acts like Hats presenting the Document to the user or modifying it on user requests. Any other such framework using the Document-View pattern [3] would also be an example.

References

- [1] Auer, K., Reusability Through Self-Encapsulation, PLoPD1, p505-516.
- [2] Beck, D., Smalltalk Best Practice Patterns, Prentice-Hall, 1998.
- [3] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, Wiley, 1996.
- [4] Deugo, D., Foundation Patterns, PLoP 98.
- [5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA.
- [6] Eds. Martin, R., Riehle, D., Buschmann, F., Pattern Languages of Program Design 3 (PLoPD 3), Addison Wesley, 1998.
- [7] Optiwave Corporation: <http://www.optiwave.com>.
- [8] Pinchuk, R., Sharon, Y., The Skin Pattern, PLoP 00.
- [9] Riehle, D., Basic Class Patterns in Java, PLoP 00.