Transfer Pattern

Christophe Addinquy and Frédéric Paulin

Abstract

Well-designed software separates the domain model from peripheral subsystems (like GUI, persistence or communication infrastructure). However, these subsystems still have to communicate with the domain model The domain model subsystem should not depend on peripheral subsystems that are susceptible to change. The most common way of having these peripheral subsystems communicate with domain model is to write dozens of access methods for each domain class.

We present a design pattern called Transfer that avoids these „bloated“ interfaces by isolating the set of specific basic types used in peripheral subsystems from the domain model.

Name:

Transfer

Problem:

The core question answered by Transfer is „how do you communicate between subsystems?“ Suppose an object from a subsystem needs to communicate with another from a different subsystem, How could this be achieved without exposing it's the internal data of the objects in the subsystem? In other words, how can we associate subsystems while keeping them as independent as possible?

Context:

This pattern is used in designing non-distributed software, where multiple subsystems are involved. Each subsystem handles data for a specific goal like displaying values in dialog boxes or storing data in databases.

Forces:

- **Moving data between subsystems**. Software is often structured into subsystems. Each subsystem is dedicated to a specific behavior. For example, the domain subsystem is in charge of the business model; a GUI subsystem displays the resulting values to users and a storage subsystem ensures data persistence. Each of these subsystems work on the same data at different times, which is why data needs to be transferred from one subsystem to another.
- **Connection between subsystems must be minimal**. We can't predict the evolution of each subsystem, so each subsystem must be designed independently of other ones. Additionally, a minimal interface allows to change or to test each subsystem more easily.
- **Avoid exposure of object's internal data**. We want also to avoid „ access methods “ to this object, because these methods are not true services. Even read-only access methods (like `GetName()`) show internal data types and structures, so they must be avoided whenever possible. Moreover, a large number of access methods leads to „bloated“ interfaces.
- **Keep track of object value's evolution**, in order to restore previous state during „ undo “ operations.
- **Warrant visibility rules between subsystems**. We often want to avoid some visibility between subsystems. For example, the GUI subsystem is allowed to know the business model, but the business model should not see the GUI subsystem. A fundamental redesign of the GUI subsystem should have no consequences on the business model.
- **Adaptation of elementary types used in the domain model to the types known by subsystems**. When a peripheral subsystem is built upon a framework, it deals with the set of types known by the framework. The business model deals with two kinds of elementary types : language basic types or user defined elementary classes. Each kind of these basic types needs to be adapted.
  - **Language basic types**: They can be either built-in basic types or standard library basic types. For some languages like Smalltalk or Java, theses basic types doesn't need to be converted in something else. For some others, like C++ or Object Pascal, framework's basic types often don't exactly match with standard elementary types, for many reasons. Such a framework needs conversion functions between standard basic types and frameworks basic types and reverse.
  - **User defined elementary classes** : Often, one want to build his own elementary classes, in order to handle easily business concepts. For example, currency, value of shares or stock in financial domain

can be represented by elementary classes. These classes are Abstract Data Types and their instances can be used as business model object attributes. As far as these elementary classes are very specific to a domain, third part frameworks can't deal with these new elementary classes. These user elementary classes must be broken into basic types known by the framework, in order to be processed by the subsystem.

-
- **Limit the overhead of smart communication between subsystems**. For the most part, software remains non-distributed, even as distributed computing becomes more common. Communication between subsystems using attribute access methods tend to be badly designed solution, though also quite efficient. On the other hand, a well-designed solution is clearly not the customer's main interest if it implies highest costs. We need a solution which is nearly as efficient as access method design.
- **Consistency control of modifiable objects**. Modification of an object by unitary operation avoids partial modification of the object. Allowing access methods for each attribute can lead to partial modification of objects. The consistency of an object may depend on the interaction between attribute values. In a multi-threaded environment, it's much more difficult to avoid race condition through multiple methods calls than in a single method call. It may be necessary to forbid the modification of several attributes if the set of new values is not consistent.

Solution:

Create a set of classes, called Transfer classes, between the business model subsystem and each subsystem where data transfer is required. Each class of this set is used to store and transfer data from a class of the business model to one or more classes of the other subsystem. Only two methods are necessary on the business class:

- `Get`: To put the data from the business object inside the transfer object.
- `Set`: To write data from the transfer object into the business object.

There are no other public access methods in the domain model. The external interfaces of the Transfer classes are in conformity with the basic types known by the business subsystem. This external interface has access methods, but this interface is strictly restricted to the methods „`Get`" and „`Set`" of corresponding business classes. It must not be handled elsewhere.

On the other hand, the internal attributes of Transfer objects use the framework's subsystem basic types. To translate basic external interface types into a framework's basic types and back, we need converter functions. These functions can be collected in one class called Translator. As long as it is possible to use this class through a separate instance referenced by each Transfer class, we prefer to use this class as a base class for a set of Transfer objects. On doing so, we achieve two goals:

- We have a common base class for each set of Transfer classes. Therefore, we can handle each set by means of this base class.
- Converter methods of the Translators can have protected visibility. This ensures these methods to be used only by subclasses, called Transfer classes.

If a subsystem is built upon a framework, subsystem classes generally subclass the framework's classes. The underlying connection structure between Transfer objects and peripheral subsystem objects is close to the Adapter pattern connection between Adaptee and Target objects [Gamma+95]. Peripheral subsystem objects act as Adapter between a framework base class (Target) and the Transfer class (Adaptee).

Like the Adapter Pattern, association between Transfer class and SubsystemObject class allows two variants as shown on structure diagram below. The real semantic of association between Transfer class and SubsystemObject class is a composition, with SubsystemObject class as owner. This structure exposes clearly the intent, and can be used by languages which does not allow multiple inheritance like Java or Smalltalk. The drawbacks are a poor efficiency because of multiple copy involved during transfer, and a public visibility for both interfaces (the interfaces for ConcreteBusinessObject and for SubsystemObject) on Transfer class.

In order to improve efficiency, the multiple inheritance structure can be used if the language used allows this feature, like C++ or Eiffel. As consequence, Transfer class interfaces are separated, with a public visibility for the ConcreteBusinessObject interface, and a protected visibility for the SubsystemObject interface. in both structures the Transfer class acts as a data storage, so a subsystem class can use native data as its own.

At initialization, the subsystem class calls the business class's `Get` method, passing itself as a Transfer class, to update its own data. The business class performs this update. After a subsystem class is modified, the business class needs to be updated. The subsystem class does this modification by calling the `Set` method on the business object, passing itself as a Transfer object.

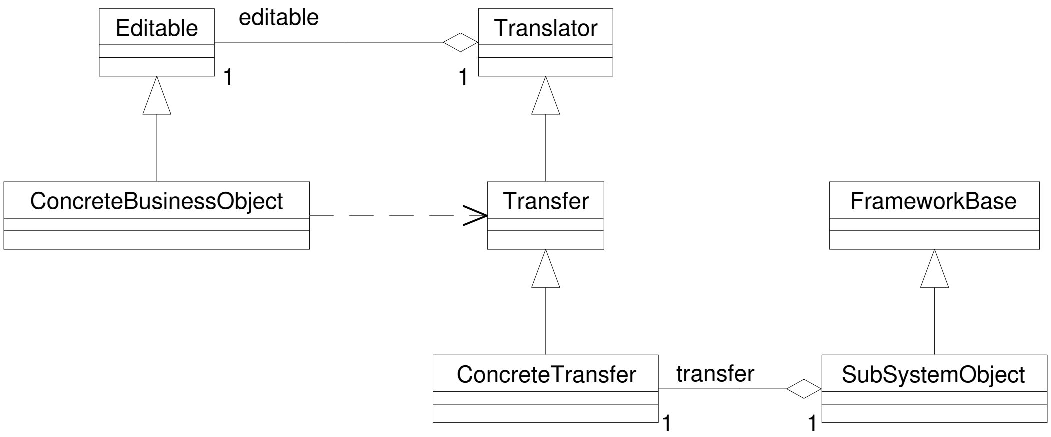
At this point, an undo / redo mechanism needs a Memento [Gamma+95] to keep track of all these changes. Fortunately, the Transfer object owns all data that may change, both before and after these changes. Only additional `Clone` methods, which duplicate Transfer objects, are required on Transfer classes to make it become

a Memento. By calling these `Clone` methods before changes, we can take a snapshot of the corresponding business class.
The initialization / update protocol is uniform for all business classes that need to be transferred, so this protocol can be abstracted in an Editable interface within the business class. Every business class that needs to be transferred must support the **Editable** interface. Extension object [Gamma98] is a way for class to support this kind of interface.

Structure

There are two variants :
Multiple inheritance variant :

Composition variant :



Participants

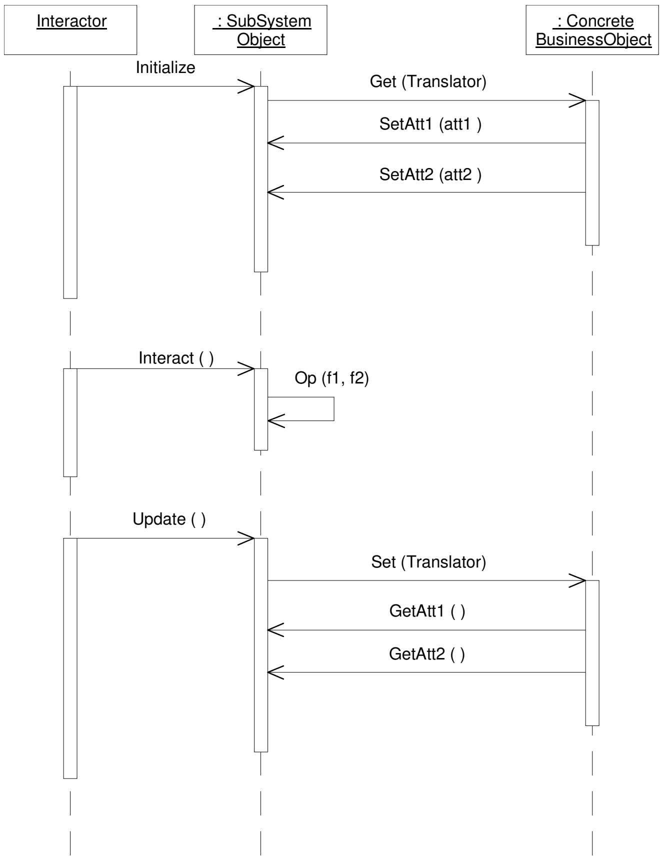
BusinessObject class: A class of the business model (also called the domain model) which is the „ source “ of the transfer.
Editable: A base class for each business class which needs to be transferred. This abstract base class acts as an interface, and defines the `Get` and `Set` methods. This base class can be improved in two ways.

- Using Extension object pattern [Gamma98].
- Using Template Method [Gamma+95] p. 325. Using this pattern, the Editable `Get` and `Set` are not abstract methods. In this case, some common behavior is centralized, such as notifications of change or calls to persistence subsystem, and primitive operations are delegated to subclasses.

SubsystemObject class: A class from one of the peripheral subsystems, which is the „ target “ of the transfer.
Transfer: This class ensures data exchange between a class of the business model and a class of a peripheral subsystem.

Translator: This class is the collection of the converter methods. It translates data from a business model type to the subsystem framework's type and reverse.

Collaboration



Resulting context

We finally obtain an architecture where subsystems are clearly identified. There's no more fear to use „as is" a framework's types or special features, because we know that Transfer classes ensure independence. Transfer classes become the link between the subsystems.

Implementation example:

Our example concerns the communication between a domain model and a GUI subsystem based on MFC. In order to focus on mechanisms, error checks are dismissed. We want to edit a `User` class via a dialog box. First, the domain model class must inherit from an `Editable` abstract class.

```
class Editable
```

```
{
public :
      virtual ~Editable() ;

protected :
      Editable(){}
      virtual bool Get(Translator&) const = 0;
      virtual bool Set(const Translator&) = 0;
};
```

Class `User`, defines two attributes, and behavior of methods `Get` and `Set`.

```
class User : public Editable
{
public:
      User(const  UserTransfer& t_user);
      ...
      bool Get(Translator& transfert) const;    // Overwrite
      bool Set(const Translator& transfert);    // Overwrite
...
private:
      string            name_;
      int               age_;
      EmployeeNumber    empNum_ ;
};
```

We take advantage of the `Transfer` class to provide the constructor's initial update through a single parameter. In order to allow the transfer of the two attributes, we need a subsequent transfer class. Such as:

```
class  UserTransfer : public Translator
{
public:
      UserTransfer(Editable *pEditable) : Translator(pEditable){}
      string GetName() const;
      void SetName(const string& name_i);
      int GetAge() const {return t_age_;}
      void SetAge(int age_i){t_age_ = age_i;}
      EmployeeNumber GetEmpNum() const;
      void SetNumber(EmployeeNumber empNum);
protected:
      CString     t_name_;
      int         t_age_;
      int         t_number_ ;
};
```

Where the class `Translator` will be defined further on.
Management of attribute `t_age_` is rather simple, and the associated `Get` and `Set` methods are defined inline. Neverless the type of attribute `t_name_` doesn't match the corresponding parameters in the access methods. The type `CString` belongs to MFC, the GUI subsystem framework. The code for the access method is not defined yet, because they need a translation, which is carried out in a class `Translator` that is also a base class for all Transfer classes. We now define the interface class.

```
class Translator
{
public:
      virtual ~Translator(){}
protected:
      Translator(Editable *pEditable)
      {
            pSource_ = pEditable ;
            pSource_->Get(*this);
      }
      bool Update() { return pSource_->Set(*this); }
      void StringToCString(const string& val_i, CString& val_o) const
      {
            val_o = val_i.c_str();
      }
```

```
	void CStringToString(const CString val_i, string& val_o) const
	{
		val_o = (const char*)val_i;
	}
	void EmployeeNumberToInt(const EmployeeNumber& val_i, int& val_o)
	const
	{
	...
	}
	void IntToEmployeeNumber(int val_i, EmployeeNumber& val_o) const
	{
	...
	}
private :
	Editable	*pSource_;
};
```

We only give the converter methods code in order to show how simple it is. In most cases, these methods can be specified inline, so they do not induce a loss of performance compared to „direct translation coding". Now, the access methods for the attribute name can be written.

```
string  UserTransfer::GetName() const
{
	string		name;
	CStringToString(t_name_, name);
	return name;
}
void  UserTransfer::SetName(const string& name)
{
	StringToCString(name, t_name_);
}
EmployeeNumber  UserTransfer::GetEmpNum() const
{
	EmployeeNumber	numRet ;
	IntToEmployeeNumber(t_number_ ,numRet);
	return numRet  ;
}
void  UserTransfer::SetNumber(EmployeeNumber empNum)
{
	EmployeeNumberToInt(empNum, t_number_)
}
```

Concrete business classes like `User` need to invoke these access methods in `Get` and `Set`. A downcast is needed; C++ allows to perform safely this downcast using the `dynamic_cast` mechanism, by the `RTTI` features. At least one virtual method is necessary for this mechanism, which is provided by virtual destructor of the base class `Translator`. We now give the implementation of methods `Get` and `Set`.

```
bool User::Get(Translator& transfert) const
{
	bool bResult = false;
	 UserTransfer*	t_user = dynamic_cast<TransfertUser*>(&transfert);
	if (t_user != 0)
	{
		t_user.SetName(name_);
		t_user.SetAge(age_);
		t_user.SetEmpNum(empNum_) ;
		bResult = true;
	}
	return bResult;
}

bool User::Set(const Translator& transfer)
{
	bool bResult = false;
```

```
	const  UserTransfer*    t_user =
		      dynamic_cast<const TransfertUser*>(&transfer);
	if (t_user != 0)
	{
		name_ = t_user.GetName();
		age_ = t_user.GetAge();
		empNum_ = t_user.GetEmpNum();
		bResult = true;
	}
	return bResult;
}
```
It seems that these two methods do the "dirty work". Indeed, object design teaches us that the remaining "dirty code" should be hidden inside the former classes, which is exactly what we did.
Now, let's have a look on the GUI side. As previously stated, Transfer classes are parents for those classes. We can write UserPropertiesDlg class as follow :
```
class UserPropertiesDlg: public CDialog, private  UserTransfer
{
public:
	UserPropertiesDlg(User *pUser_i) ) :  UserTransfer(pSource_i) {}
	virtual void OnInitDialog();
	virtual void DoDataExchange(CDataExchange *pDx);
	virtual void OnOK() {
		...
		Translator::Update();
		...
		}
};
```
The method `OnOK` is inherited from MFC. The framework calls this method when the OK button is pushed. By doing so, the business object must be updated. A simple call to the `Set` method on the business object achieves this goal. Now, the concrete user property dialog is rather simplified:
It’s not useful to know the entire framework for understanding the mechanism. `UserPropertiesDlg` redefines two `CDialog`’s methods, called according to Hollywood principle [Gamma+95] p. 327.

- `OnInitDialog` is called when dialog’s widget can be updated for the first time. We can take advantage of this method to initialize all fields with `UserTransfer`’s values.
- `DoDataExchange` is called each time necessary, to update widgets from data and reverse.

Variants:

Generic implementation of subsystem classes in C++. This language allows multiple inheritance variant. In this case, Transfer classes are parents for subsystem classes. Using the generic paradigm, we can write a base class for all dialogs based on Transfer. Such a class uses a C++ idiomatic form :
```
template<class TTransfer> class TransferDialog : public FmkDialog, public
TTransfer
{
public:
	virtual void OnOK() {
		...
		Translator::Update();
		...
		}
protected:
	TransferDialog(Editable *pSource) : Translator(pSource) {}
};
```

Now, the concrete user property dialog is rather simplified:
```
class UserPropertiesDlg : public TransferDialog< UserTransfer>
{
public:
	UserPropertiesDlg(User *pUser)
		: TransferDialog< UserTransfer>(pUser){}
	virtual void OnInitDialog();
```

```
	virtual void DoDataExchange(DataExchange *pDx);
};
```

Concrete „editable“ class invocation, from the subsystem side: In our pattern, deals between business class and subsystem classes go through the editable interface. This interface only has access to the base classTranslator. So, in the concrete business class, the `Get` and `Set` methods need a downcast to use Transfer’s access methods. Another solution is to avoid polymorphism through Editable interface. An interesting feature of this solution is that is uses a generic mix-in subsystem class. A direct call to the concrete business class may be made, and this business class can take the concrete Transfer type as a parameter. Using this technique, C++ method overloading allows a single business class to deal with multiple Transfer classes within the same set of Transfer classes. The example bellow shows a part of the generic mix-in subsystem class. In this example, template parameter `Trans` is the Transfer class, and the parameter `TBCl` is the business class.

```
template<class Trans, class TBCl> class MixIn: public dialog, public Trans
{
public:
	MixIn(TBCl *pSource_i) : pSource_(pSource_i) {pSource_I->Get(this);}
	bool Update() {pSource_->Set(this);}
protected:
	TBCl  *pSource_;
};
```

Exact matching between business classes and concrete subsystem concrete mix-in is solved at link-time.

Friendship relation between business classes and Transfer classes: As we want to limit the use of Transfer’s interfaces to the methods `Get` and `Set`, we could give a friendship privilege to this two methods. So Transfer’s interface could be `private`. We thereby ensure that Transfer’s interface can only be called by the business class to which it belongs. The Memento class is implemented in a similar way in the Memento pattern [Gamma+95p.287]. A drawback is that `Get` and `Set` methods can access any other methods or attributes of Transfer object.

Access methods for subsystem classes: Some designers prefer to avoid any direct access to attributes. There are several arguments for it: direct access to attributes does not ensure migration to a thread safe behavior, nor does it allow operational control like invariance or state transition check. We can give a `private` visibility to attributes in Transfer classes and write protected access methods. These access methods must be in conformity with framework’s basic types instead of business model basic types, because only the subsystem classes are clients of these access methods. The wish of designers is more satisfied with this design than inside one class, because attributes are thereby not visible from subsystem classes.

Multiple set of Transfer classes, using „traits“: This C++ specific technique [Myers95] allows the definition of various implementation or type definitions using generic programming. Using this technique, it is possible to write only one set of Transfer classes, even if multiple sets are necessary to deal with multiple subsystems. The various elements like framework’s type or translator methods are designed using the trait class. For example:

```
template<class T> class Trait_translator{};

template<class T> class MyTransfer : public Trait_translator<T>
{
public:
	typedef Trait_translator<T>::t_string    TString;
	typedef Trait_translator<T>::t_bool      TBool;
	...
	void SetName(const string& name_i) {
		StringToTString(name_I, t_name_);}
	...
protected:
	TString    t_name_;
};
```

Each time a new family of Transfer classes is needed, only one Translator is necessary for defining all symbolic types and translator methods. Such a trait translator is defined as follows:

```
class Trait_translator<CString>
{
public:
	typedef CString   t_string;
	typedef BOOL      t_bool;
	...
protected:
```

```
	StringToTString(const string& val_I, t_string& val_o) {...}
	...
};
```
For each real trait class, a „selector“ type is required. In the example above, we choose a framework string type as selector (`CString` is the MFC string), because it’s a typical basic type that changes between frameworks.
Now, as the generic set of Transfer class and each specific trait Translator are created, each concrete set of translator needs only some `typedefs` to be declared :
```
//-- First set
typedef MyTransfer<CString>   MFC_MyTransfer;
...
//-- Second set
typedef MyTransfer<string>    STD_Transfer;
```
The Editable interface can deal with undefined number of Transfer set using C++ template member functions.
```
class Editable
{
public:
	template<class U> bool Get(Trait_translator<U>& translator
		) const;
	template<class U> bool Set(const Trait_translator<U>&
		translator);
...
};
```

Rationale:

Move data between subsystems: Editable is the interface, which defines the `Get` and `Set` methods. These two methods allow data to move from and to business objects. Transfer objects ensures the storage of moving data.
Adaptation of elementary types used in the domain model to the types known by subsystems. This goal is achieved in two points.
External interface of Transfer classes is consistent with the two kind of elementary types used by the business model.
Language basic types are translated in framework basic types by the Translator class if needed. In the same way, user defined elementary types are broken in basic types by the Translator class.
Minimal connections between subsystems: Only one `Get` method and one `Set` method are involved from the business model viewpoint. Transfer classes are not really an active part of subsystem classes, so we can say that subsystem’s methods involved are the initialization / update methods. On the other hand, all external interfaces of Transfer classes may be considered, and this interface can be large. However, this interface is restricted to the `Get` and `Set` methods, so a minimal connection is ensured.
Avoid exposure of object’s internal data: Instead of having a set of access methods, business objects handle Transfer objects. Therefore, the interface of the main classes is not „degenerated“ by this kind of design. As a drawback, we could say that these access methods are just shifted to the Transfer classes. In fact, shifting data access to the Transfer classes allows to keep only true services on the business classes. Doing so, we avoid direct data access in other contexts. If access methods are written on business classes, to allow data modifications from dialog boxes for example, we open the door to the use of these access methods for absolutely different usage. One member of development team working on other part othe business model, may use these access methods instead of existing services or instead of designing new true services.
One the other hand, the shifting of data access to the Transfer classes can really limit internal data access between team development members. For example, with C++ compiler, a developer who is not involved in the evolution of a business class only needs the corresponding object and the header file (.obj and .h) . There is no reason to give him the header file of the Transfer class. Without the Transfer class header, there is no way to get business class internal data.
The key point is that the Transfer access methods avoid exposure of Transfer attributes types. Changing the framework used to implement a peripheral subsystem has no effect on the business model. In Internet years, particular technology chosen today might become completely obsolete next year. The business rules and basic types will change more slowly than the frameworks on the either end.
Keep track of object’s value evolution: Transfer classes are mandatory channel to data modifications. Using Transfer classes as Memento when data are modified is a smart way to achieve this goal. Only a `Clone()` method is needed. As we have seen in the implementation section, this method was easily implemented using template techniques in C++.
Warrant visibility rules between subsystems: Domain model doesn’t deal with other subsystems directly. Instead of this, it works with the external interface of the Transfer classes exclusively, without any more

knowledge. Subsystem classes don't need to know a lot about business classes. They just deal with editable interface, so that concrete business classes may change in future releases without breaking down the whole architecture. Project managers often appreciate this issue because GUI subsystem may change suddenly before the end of a project, as customer or project managers themselves change their minds about ergonomics.

Limitation of the overhead in communication between subsystems: a Data transfer using simple access methods on domain model objects costs only one method call by transferred attribute. Using the multiple inheritance variant, Transfer Pattern is nearly as efficient as the simple access method alternative.

During subsystem class initialization the induced cost is :

- One call to the Get method on the business class.
- One method call on the Transfer class for each data to transfer.
- One conversion method call on the Translator class for each data to transfer.

The number of access method calls is the same using Transfer Pattern or using access methods directly on business classes (raw structure). The number of calls is twice in the composition variant.

The Transfer pattern adds one call to the Get method on the business model class .

If you have two set of basic types, you need a conversion. There is no severe difference between raw structure and Transfer Pattern, because conversion methods in Transfer Pattern can be optimized (with inlining in C++, for example). On the other hand, raw structure can deal with framework basic types on the overall system, leading to no conversion and a better efficiency. But there is a real difficulty to deal with a number of frameworks with such a choice.

The same Get method may be called from several subsystems using the same basic types, or from several classes of one subsystem. If some subsystem classes only need part of information stored in a business class, it is possible to optimize the amount of data to transfer by providing an „ origin information “. The same mechanism may be used with the Set method, in order to write only the data that the involved subsystem class can modify.

When a business object must be updated because of a change from a subsystem, only data really modified should be rewritten. A set of flags on concerned Transfer class can „ keep in mind “ each data really modified. There is two main ways to set up this set of flags.

- Data access on the subsystem side is a raw access, without access methods. This is the proposed solution. In doing so, modified data are detected by the subsystem class (a dialog box, for example). As shown before, a Transfer class may act as a Memento, and current state can be compared with the state before editing. With this mechanism, the subsystem class is responsible for updating the set of flags.
- Data access is secured by access methods on the subsystem side, as proposed on the variant „ Access methods for subsystem classes “. In this case, the Transfer class can update the set of flags because each change only occurs by its access methods.

In all cases, on the business object side, the rewritting of each data depends on the state of the corresponding flag on the Transfer class.

Consistency control of modifiable objects: All the operations for reading or writing the object's data are enclosed in the methods `Get` and `Set`. In multi-threaded environment, race conditions can be avoided using a critical section from the beginning to the end of each method. Consistency of business objects can be checked with preconditions at the beginning of the methods and post-conditions at the end.

Related Patterns:

- Three Subsystems, [Cockburn95, p.325] in PLoPD'2, describes the type of structure that a well-designed application should have. When applying this pattern you enter the context of Transfer Pattern.
- Observer, [Gamma+95, p.293] and Transfer Patterns are complementary. In the Observer Pattern, the concrete Subject is read and written by the concrete Observer through two methods: `SetState()` and `GetState()`. Transfer could be used to do this. An Observer could be a Transfer object or can use a Transfer object to get and to set the subject's state.
- Memento, [Gamma+95,p.283]. The Memento pattern is used to externalize an object's state, without violating encapsulation, so that the object can be restored later. A Transfer object could act as a Memento.
- Command, [Gamma+95,p.233]. The unification aspect of access methods leads to generic Command for creating and modifying Business objects. The Transfer Pattern can store an object's state between two operations and can be used to restore previous state. Using templates in C++, the creation/modification process of business objects can be generalized and encapsulated in generic Command objects [Gamma+95,p.233] thereby avoiding Command objects proliferation.

```
template<class TConcreteTransfer, class TEditable, class TCreator>
class CmdCreateEditable : public Command
   {
    public :
      CmdCreateEditable(TConcreteTransfer& concreteTransfer,
```

```
                        TCreator& creator) :
                creator_(creator),concreteTransfer_(concreteTransfer)  {}
        bool Do(){ return creator_->Make(concreteTransfer);}
        bool UnDo(){return creator ->Suppress(editable ) ;}

      private :
        TEditable * editable_;
        TConcreteTransfer * concreteTransfer_;
        TCreator * creator_;
     };
```

- A `CmdModifyEditable` template class can be designed in the same way. Object Recovery [Silva+98, pp.261-276] in PLoPD'3 shares some similarities with Transfer Pattern but its intent is the recovery after a failure. The example presented suits the Transfer Pattern context: an application designed with three spaces (application, domain and persistent spaces), i. e. three subsystems, and need for undoable actions. It shares common forces: Encapsulation and Modularity. However, the primary goal of Object Recovery Pattern is to separate object recovery policies from object-specific functionality.
- Template Method, [Gamma+95,p.325]: The `Set` and `Get` methods respect the Template Method design pattern.
- Informational role [Kerth95, p.310] in PlopD'1 achieves for a part the same goal, but Kerth's pattern is essentially oriented to abstract specific information delivered by classes. No structure is given, the main force was avoiding `if` and `switch-case` statements.
- Adapter [Gamma+95, p.139] is used on the peripheral subsystem side to adapt Transfer objects to subsystem objects. Both forms of this pattern with multiple inheritance or composition are applicable. But, the composition variant often implies another copy between Transfer objects and subsystem objects. Therefore, it can be less efficient.
  Design Pattern Adapter can be used without Transfer objects as an alternative solution. For instance in [Hollub96], the author adapts business object with GUI objects by using multiple inheritance and friendship between nested classes. This kind of solution avoids all public access methods but the implementation given is C++ idiomatic. It can't be translate to Java because of the lack of multiple inheritance. Using composition, the public access methods would have reappeared.
- Transfer Pattern uses Memento and Adapter to move data between subsystems. The ICM (Interface-Control-Model) architecture [Brown96b] uses Mediators[Gamma+95, p.273] and Facades[Gamma+95, p.185] to translate messages from User Interface components into messages that are understood by the objects in the domain model. These Mediators and Facades belong to the „Control layer". A Mediator acts as a coordinator. It is more useful when the relations are many to many, for instance between widgets in a complex dialog box, i. e. within the GUI subsystem. As stated by [Brown96a] „it doesn't make much sense to involve the domain layer in these purely user interface actions". The protocol considered in this paper is simpler, just moving data between subsystems in a one to one relationship, for instance between a domain model object and a dialog box. Transfer objects are passives, they don't coordinate any interaction. They are handled by business objects and subsystem objects.

Known Uses

Transfer pattern has been widely used in a gas network supervision project at EDF-GDF, the french national company of electricity and gas.

Acknowledgments

We would like to thank Kyle Brown for shepherding our work and Philippe Foucard for his useful contribution on reading and reviewing this pattern.

References:

[Brown96a] Kyle Brown, „Remembrance of things past: Layered Architecture for Smalltalk applications" **Fehler! Verweisquelle konnte nicht gefunden werden.~~Error! Reference source not found.~~**.
[Brown+96b] Kyle Brown and Liang Shi „Using the Distributed ICM architecture to achieve client-server independence" **Fehler! Verweisquelle konnte nicht gefunden werden.~~Error! Reference source not found.~~**.
[Cockburn96]: Alistair Cockburn „Prioritizing Forces in Software Design", in *Pattern Languages of Program Design* vol.2 - J. Vlissides, J. Coplien and N. Kerth ed. – pp. 319-333.
[Gamma+95]: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides – *Design Patterns, elements of reusable Object Oriented Software* – Addison Wesley 1995.

[Gamma98]: Erich Gamma – „ Extension Object “, in *Pattern Language of Program Design* vol.3 - R. Martin, D. Riehle and F. Bushmann ed. – pp. 79-88
[Hollub96] : Allen Hollub – „ Hide Your Data and Make Objects Responsible for their Own User Interfaces “, *Microsoft System Journal*, vol.11, n°8, August 1996.
[Kerth95]: Norman L. Kerth – „ Caterpillar's Fate : A Pattern Language for the Transformation from Analysis to Design “, in *Pattern Language of Program Design* vol. 1 – J. Coplien, D. Schmidt ed. – pp. 293 – 320
[Myers95]: Nathan Myers - „ A new and useful technique : "Traits" “, in *C++ Report June 1995 vol. 7 / No 5*
[Silva+98]: Antonio Rito Silva, Joao Dias Pereira, and Jose Alves Marques – „ Object Recovery “, in *Pattern Language of Program Design* vol.3 - R. Martin, D. Riehle and F. Bushmann ed. – pp. 261-276