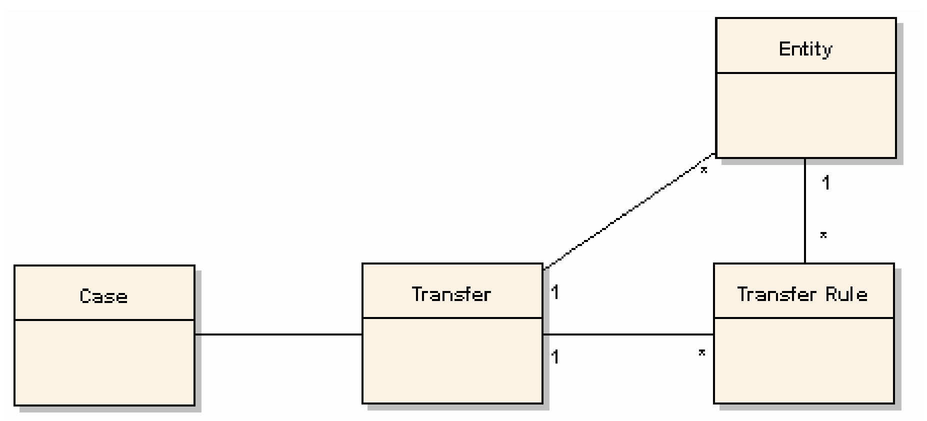
CASE TRANSFER

Case ownership is transferred from one organizational entity to another



Example: *A periodic case load rebalancing determines that Jack's caseload is significantly larger than Jill's and that caseloads need to be rebalanced (transfer event). Since Jack and Jill are in the same county, the transfer rules for transferring a case between local organizational entities, which are relatively simple, are applied prior to completing the transfer.*

Example: *The non-custodial parent on a child support case moves from New York to California (transfer event) to be closer to his children. Because the case transfer is interstate, the rules governing the transfer of the case between organizational entities are more complex.*

Example: *A member on a multi-jurisdictional case moves to a new state. Management of the demographic data will continue to reside in the current state while management of the financial data will move to the new state.*

Although many analysts regard case transfer to be one of the most complex components of a case management system to implement, the underlying approach is relatively straightforward. In the most elemental sense, events happen that cause cases to get transferred from one organizational entity to another. Simple, right?

The difficulty that so many people attribute to this activity is due to the inherent complexity of the case transfer rules that govern under what circumstances cases may be transferred. Using this approach, this complexity is encapsulated in a set of transfer rules that understand how to interact with particular transfer types.

Making it Work

The rules that define the transfer are complicated for several reasons. First and foremost, they are complicated because different rules exist for different transfer types. Secondly, each entity may have its own transfer rules applying to both the transfer type and the other entity involved in the transfer. Finally, case transfers may need to be handled piecemeal; with one part of the case being transferred and another part of the case not being transferred. The sequence diagram in figure CT1 illustrates a simple scenario for the creation of transfer type and entity specific business rules and the application of these business rules to make a case transfer determination.

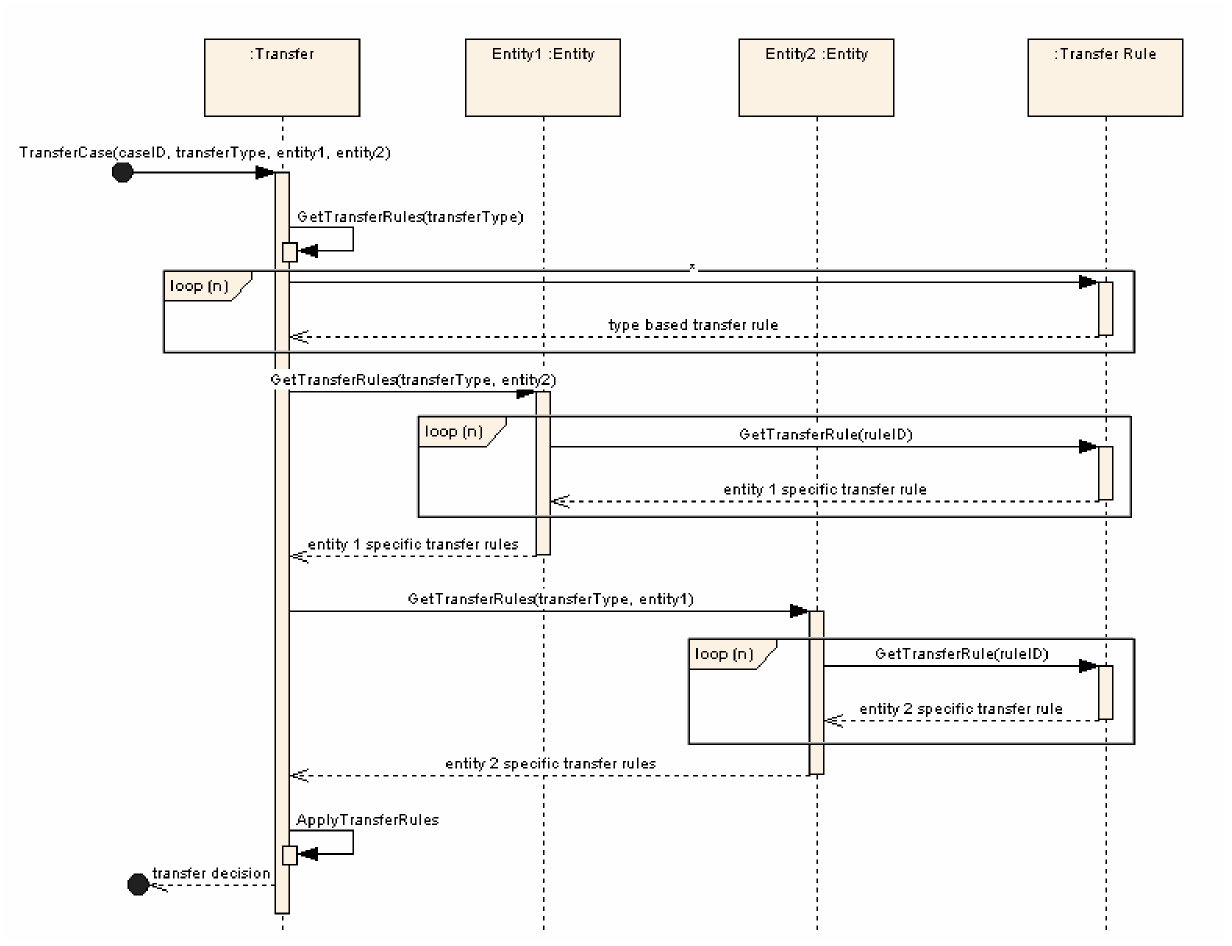

:Transfer
Entity1 :Entity
Entity2 :Entity
:Transfer Rule
TransferCase(caseID, transferType, entity1, entity2)
GetTransferRules(transferType)
loop (n)
*
type based transfer rule
GetTransferRules(transferType, entity2)
loop (n)
GetTransferRule(ruleID)
entity 1 specific transfer rule
entity 1 specific transfer rules
GetTransferRules(transferType, entity1)
loop (n)
GetTransferRule(ruleID)
entity 2 specific transfer rule
entity 2 specific transfer rules
ApplyTransferRules
transfer decision

Figure CT1 Applying rules to a case transfer

Another important consideration is that rules have the potential to change over time. Therefore transfer rules may use the *Effectivity Period [Fowler]* for each period that a rule is in affect. For the effective period of a particular transfer rule, it is considered authoritative, with no contradicting rules being permitted to exist.

When to use it

Case transfer rules are a very effective way to organize transfer logic to react to complicated and often changing transfer rules. When dealing with very simple conditions and a handful of transfer rules, conditional logic may do the trick. When a greater number of interdependent rules or volatile rule conditions are likely, the benefits of using this approach offset the initial complexity of setting up the transfer rule infrastructure.

It should be noted that this approach applies strictly to case transfer and not to case referrals. Referrals result in a reference to an existing case being passed to seed the creation of a new case, often one of a different type and/or in a different jurisdiction. Transfers imply that the transferor is relinquishing all or part of the case ownership to the transferee. Due to the lack of ownership change in referrals, they are significantly easier to deal with than transfers.

Sample Code

Having established that the transfer rules and their groupings are the basis of the complexity when implementing this approach, we can now examine some of the underlying implementation details. Please keep in mind that these examples are purposely left simple to illustrate design intent without erecting unnecessary barriers to understanding. For our example, we will use a mental health case which is derived from a base *Case [not yet available]* type.

```
public class MentalHealthCase : Case
{
    // Normally, common case information would be inherited from Case
    private int _caseID;
    private int _caseManagerID;

    public int CaseID
    {
        get { return _caseID; }
    }

    public int CaseManagerID
    {
        get { return _caseManagerID; }
        protected set { _caseManagerID = value; }
    }

    public MentalHealthCase(int caseID, int caseManagerID)
    {
        this._caseID = caseID;
        this._caseManagerID = caseManagerID;
    }
 }
```

The most important thing to note here is that due to the infrequency of case transfer activities with respect to other activities, the concrete implementation of `Case` does not rely upon a dependency with `Transfer.` Instead, dependencies have been inverted and `Transfer` is now dependant upon `Case`. Ideally, we would do this via an interface containing dependant functionality, which `Transfer` would implement but we will forgo that now for simplicity's sake. That leaves us with the invocation of the transfer looking like:

```
MentalHealthCase MHCase = new MentalHealthCase(caseID,oldManager);
Transfer TransferManager = new Transfer();
TransferManager.TransferCase(MHCase, newManager);
```

The code for `Transfer` can be found below so that you can follow along. Keep in mind that this is just one possible implementation and is kept relatively simple to illustrate the principles behind the case transfer. The actual implementation may leave `Transfer` in control of the rules or may delegate responsibilities to other entities to make granular transfer decisions, returning only the decision (and not the collection of rules) back to `Transfer`.

```
public class Transfer
    {
        private List<TransferRule> _transferRules = new List<TransferRule>();
        private bool _isValidTransfer;

        public bool IsValidTransfer
        {
            get { return _isValidTransfer; }
        }

        public Transfer()
        {
            _isValidTransfer = true;
        }

        public Case TransferCase(Case transferringCase, int newCaseManagerID)
        {
            // Perform case validity check
            _transferRules.AddRange(GetCaseTransferRules());
            _transferRules.AddRange(GetEntityTransferRules());
            ValidateTransfer(_transferRules);
            if (IsValidTransfer)
            {
                transferringCase.CaseManagerID = newCaseManagerID;
            }
            return transferringCase;
        }

        private List<TransferRule> GetCaseTransferRules()
        {
            // Derive the case / transfer type based upon _case
            // MH Rules Collection derives from base rules collection class
            MentalHealthInstateRulesCollection MHRules = new
                  MentalHealthInstateRulesCollection();
            return (MHRules.TransferRules);
        }

        private List<TransferRule> GetEntityTransferRules()
        {
            // Derive the entities based upon _case
            // IC Rules Collection derives from base rules collection class
            IntracountyRulesCollection ICRules = new
                  IntracountyRulesCollection();
            return (ICRules.TransferRules);
        }

        private bool ValidateTransfer(List<TransferRule> transferRules)
        {
            foreach (TransferRule tr in transferRules)
            {
                if (tr.ApplyRule() == false)
                {
                    _isValidTransfer = false;
                    return _isValidTransfer;
                }
            }
            return _isValidTransfer;
        }
    }
```

The transfer class is significantly more complex than the case class and, as you'll notice in this first cut here, there are a lot of detailed processing steps which are not explicitly specified at this time. The decision making logic which is commented out, although not complex, might be fairly extensive and lends little to the understanding of the other more critical elements of this approach. Therefore, this logic will not be handled here.

Four very important assumptions about a `transfer`'s responsibilities are being made. These assumptions are enumerated below:

1. `Transfer` can derive a set of generic rules and will apply these generic rules prior to any more specific transfer rules. An example of a generic rule is *that the receiving entity cannot contain more than 50 cases after the transfer is complete*.
2. `Transfer` can derive a set of rules specific to the transfer type and involved.
3. `Transfer` can derive a set of rules specific to the entities involved in the transfer. Due to the fact that multiple entities can be involved in the transfer, `Transfer` may need to marshal more than one set of rules.
4. Evaluation of the rules occurs by chaining together the results of the rule evaluations with logical conjunctions. The result itself is a Boolean and implies that there is no weighing or subjectivity in evaluating a transfer, the transfer will either occur or it does not.

Managing Rule Complexity

Pursuant these assumptions, there is a significant amount of latitude left in managing the rules. From the basic implementation of `Transfer`, you can see that we have at least three different categories of rules: transfer rules, case type rules, and entity type rules. The initial inclination might be to model these rule groupings hierarchically, as illustrated in figure CT2.

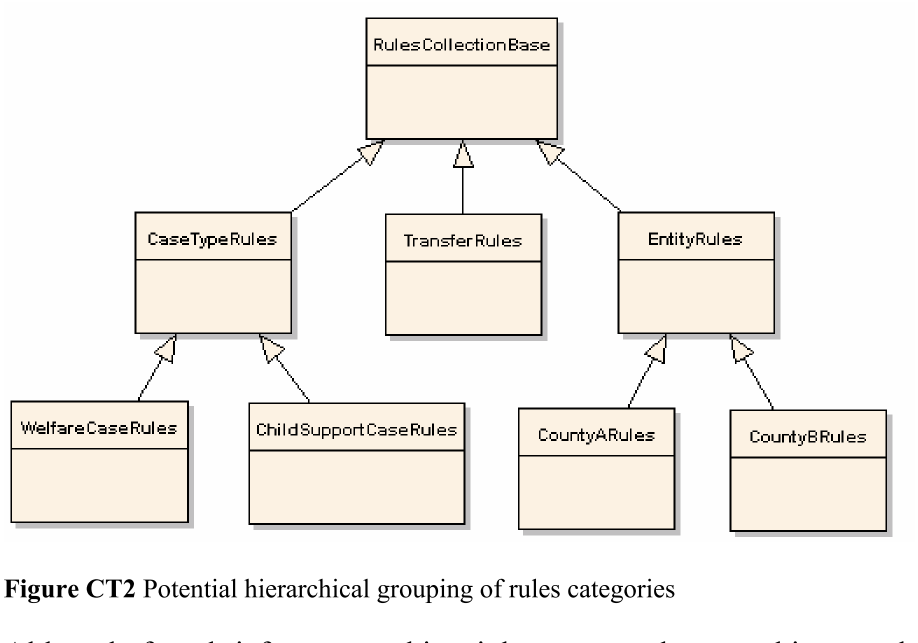


Figure CT2 Potential hierarchical grouping of rules categories

Although, for a brief moment, this might appear to be something worth entertaining, the honeymoon is short-lived. The hierarchical structure is far too terse to capture the complexities of the rule types. The easiest way to think about this is to consider what happens when, as an example, county A has specific rules for child support cases. Oops, those two rule groups are in two entirely separate branches, not allowing the expression of layered rules that many organizations are likely to require.

As an alternative to subclassing to extend the functionality of existing classes, we can decorate the primitive transfer categories with the appropriate rules and responsibilities. This leaves us with the following runtime establishment of a particular set of rules:

```
ChildSupportRulesCollection CSRules = new ChildSupportRulesCollection();
CSRules = new SuffolkCountyRulesCollection(CSRules);
CSRules = new InstateTransferRulesCollection(CSRules);
return (CSRules.GetTransferRules());
```

Note again that the above code represents a simple example. Things become significantly more confusing when the case represents several types and has multiple owners. Some of this is discussed in *Case [not yet available]*. However, by and large, the approach of ruleset creation through decoration provides a elegant way of handling even the most complex of cases.

Authors: Thomas Beck (thomas@beckshome.com), Damien Graeff (dgraeff@gmail.com)
Version: 0.1
Last Revision: 9/13/2006
Domain: State Government
Function: Case Management