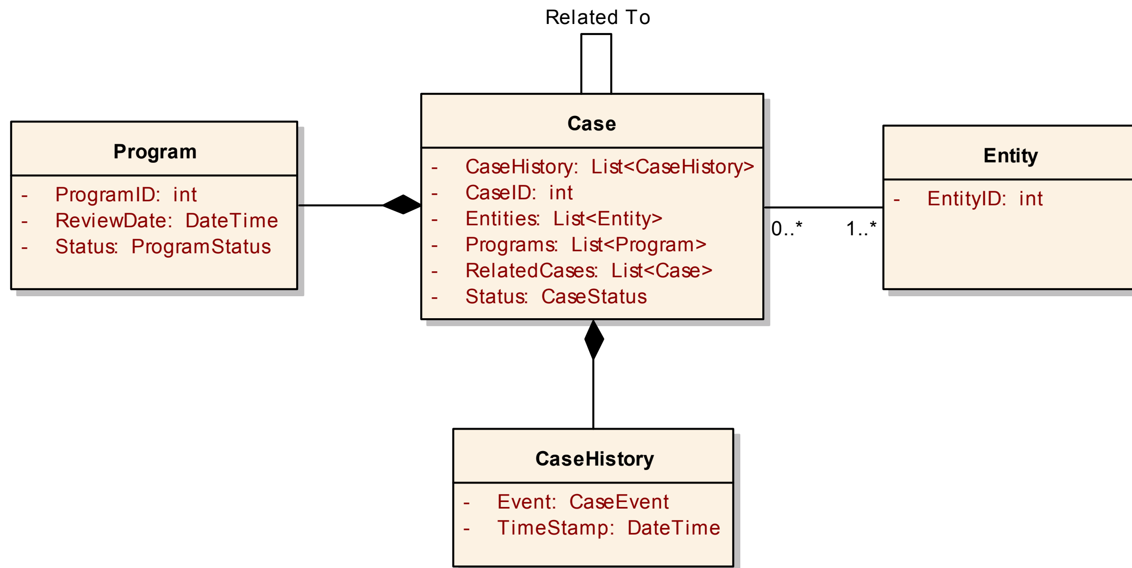
CASE

A container that encapsulates a set of programs and relates these programs back to an owning entity or entities



Cases rear their heads in every corner of state government systems; from health and human services to court and judicial systems. There are various ways to think about what a case is. None of these are necessarily wrong; some just better reflect a broader basis of reality than do others.

There are several commonalities that we have observed in cases across the systems that we've worked on. Although these should by no means be considered the gospel, these commonalities recur frequently enough to warrant including them in the baseline definition. The first of these defining attributes is the case as a container of programs. That is, a case represents information about benefits programs, commitments, legal proceedings or other multi-party state government activities. For the case to make logical sense, there must be some commonality between these programs that warrants grouping them into a case structure. Usually this is the presence of a shared person or legal entity throughout all of or the majority of programs. As an example, a case could include child support and alimony program elements containing a common defendant.

The second common attribute that we've observed across systems is the ownership of a case by one or more entities. It is not uncommon for cases to have more than one owner. Even if the case contains only a single program, it is possible that the case is split across multiple organizations (inter-State or intra-State) or that ownership is divided within an organization to align with organizational specializations (e.g. split case management of financial and program data). As you might imagine, if the case contains multiple programs, the ownership mapping can quickly get pretty hairy.

Making it Work

As this pattern's diagram suggests, information brokering is one of the case's most vital characteristics; it is the owner of very little information but possessor of very much information. The case's relationship with Program and Entity is bidirectional in nature. The relationship with CaseHistory is decidedly unidirectional. Perhaps most interesting is the reflexive relationship to the case itself. In combination with case status, this allows complex case mergers, divergences, and associations to be accounted for over the life of a case.

There are several things that a case is clearly not. Due to the fact that we've heard the following definitions or arguments more than once, we believe that these misconceptions warrant attention as full-fledged anti-patterns. The first of these anti-patterns surrounds the confusion of the terms *case* and *caseload*. The caseload refers to the number of cases that are ascribed to a particular entity. This number can be derived by the entity using the association with the cases but otherwise has no relationship to a proper case. The second case anti-pattern is mistakenly associating a case with an individual or legal entity directly. This mistake is easy to understand since, as mentioned above, commonality of individuals or legal entities is usually the determining factor in the grouping of programs into cases. The additional layer of *indirection through the program* is absolutely necessary, however. Whereas this level of indirection may be accounted for with alternate designs in systems that are responsible for managing only a single program, these designs become brittle as responsibilities for additional programs are added to the system.

Case history is a necessary partner of the case entity. Although case history is associated directly with a case, the historic entries contained in the case history may be written by any number of entities that process case-related transactions. Irrespective of the form the application design takes on, it is important to identify what case transactions will cause case history records to be written and how these entities will create the history records.

When to use it

If you look around in state government systems, you will observe that cases really are everywhere. The analysis patterns categorized in this document reflect best practices that can serve as the basis for the successful design of basic case management functions. It is worth reiterating here that even when this pattern seems somewhat too complex for your initial requirements, it is worth understanding why this complexity was modeled. Case management systems that initially handle single entity and program types often grow to accommodate a variety of entities and program types. If appropriate abstractions such as the `Program` are not accounted for when the system is designed and built, accommodating these abstractions at future points in time can be both costly and burdensome.

Sample Code

Although case-related patterns can get pretty complicated, cases are, at their core, pretty simple things. It's actually rather unlikely that you'll ever see a pure case making its way around a system. Cases are usually implemented as interfaces or abstract classes, depending upon the nuances of the domain being modeled. The actual implementation of case functionality is then left to the concrete inheriting class. This is illustrated in the example below.

```
public abstract class Case
{
    private int _caseId;
    private Status _caseStatus;
    private List<Case> _relatedCases;
    private List<CaseHistory> _caseHistory;
    private List<Entity> _entities;
    private List<Program> _programs;

    public List<Case> GetRelatedCases()
    {
        //Generic code for all case types
    }

    //Abstract implementation overriden by specific case types
    public abstract List<Participant> GetCaseMembers();
}
```

Here, `Case` is defined as an abstract class. It includes a concrete implementation of the specific functionality for `GetRelatedCases()` but defines `GetCaseMembers()` as abstract, leaving it to be overridden by the implementing class, as illustrated below.

```
public class MentalHealthCase : Case
{
    public override List<Participant> GetCaseMembers()
    {
        //Specific implmentation for a mental health case
    }
}
```

Although not shown here, the code for getting case members is worth mentioning. Notice that the method returns a collection of participants. However, in the diagram that we started with, participants are not included. The case must go through the program (that additional level of indirection we spoke of earlier) to get to the case's members. This is especially important since what constitutes a case's members varies widely across programs. As an example, mental health cases may include just a single citizen and multiple service providers whereas a Welfare case likely has no service providers but a multitude of citizens with complex relationships (a.k.a. a family). Responsibility for cobbling together the case members is best delegated to the entity that has knowledge of potential case structures for a particular program type, the `Program`.

Case history is another facet of cases worth illustrating. You will likely find that your requirements dictate that certain transactions require writing a record to the case history diary whereas others do not. The transaction to actually write the history record should be maintained independent of the underlying business transaction; such business concern independent transactions are often referred to as *cross-cutting concerns*. Cross cutting concerns are best implemented as aspects (attributes in .NET, annotations in Java).

Without going into the particular implementation of the aspect, it is worthwhile mentioning that the aspect would be applicable only at the method level. The code below reflects how this aspect could be used to decorate a method for which we would almost certainly want to write out a historic record, a *Case Transfer [Beck, Graeff]*.

```
[ApplyCaseHistory("CaseTransfer")]
public Case TransferCase(Case transferringCase, int newCaseManagerID)
{
    // Perform case transfer logic
    return transferringCase;
}
```

In the above code, the `ApplyCaseHistory` attribute is applied to the `TransferCase()` method, which should result in the creation of a historic record indicating when this transaction was completed.

There are several other facets of cases that are interesting and worth discussing, such as the use of the `RelatedCases` collection. However, to not overcomplicate what should remain a simple pattern, illustration of these aspects of the case will be left to related patterns, such as *Case Closure [not yet available] and Case Referral [not yet available]*, which make extensive use of related cases.

Author: Thomas Beck (thomas@beckshome.com)

Version: 0.1
Last Revision: 12/20/2006
Domain: State Government
Function: Case Management

License: Creative Commons Attribution-ShareAlike 2.5 License