THE DISCRIMINATION NET: CATEGORIZING AND MANIPULATING DOMAIN OBJECTS

In some applications it is common to receive a specification that says, "For all <objectName> that are <categoryName>'s do <some calculation>". It would be convenient for the code to mimic the structure of the specifications because when the specifications change, one can easily identify what code needs to be changed. Furthermore, it is common for specifications to split existing categories into sub-categories so that the algorithm can treat the objects differently. That is, in the old specification it was treating a group of objects uniformly, but the new specification requires some objects to be treated differently than others.

Here is some code you may have seen in various places throughout your programs:

```
if (theAccount.type == Account.TAXABLE) {
      //Do something
} else {
      //Do something else
}
```

This code is everywhere in the application, for example, the user interface for displaying accounts and the domain objects for doing calculations. The underlying problem here is that the knowledge about the category to which a domain object belongs is spread throughout the system.

Applicability

This pattern language can be used when objects need to be categorized, or partitioned, in a dynamic way (i.e., not generally held in a collection throughout their lifetime) based on user-defined criteria or constantly-changing requirements. It is also useful when the application implements a complex algorithm that is itself in a state of flux.

The Pattern Language

The overall strategy of this pattern language is to avoid hard-coded references to categories so that they can be *extended* and *specialized*. Categories are extended when the user expands the scope of the product to include more categories than were originally considered. Categories are specialized when the user wishes to treat some objects differently in one version that were treated the same as other objects in the previous version.

One can use this pattern language to implement a framework for creating algorithms that are modular and extensible and refer to the objects that need to be manipulated (e.g., aggregated, calculated) in terms of their Category.

This pattern language is presented in bottom-up order because lower level patterns are often used as a vocabulary for higher level patterns.

Discrimination Net

An Example

Since the pattern language promises to reduce re-work throughout a system, I think it is appropriate to present a small slice of a system. The objective is to contrast the naïve approach with the prescriptions in the pattern language in order to understand why and how the pattern language provides benefit.

Let's suppose we want to display a customer's Account's, with the taxable ones first and the tax-deferred ones following. This might be the code snippet (in a controller class):

```
Vector taxableAccounts = new Vector();
Vector taxDeferredAccounts = new Vector();
Iterator iterator = customer.getAccounts();
Account account = null;
while(iterator.hasNext()) {
      account = (Account)iterator.next();
      if (account.isTaxable()) {
            taxableAccounts.addElement(account);
      } else {
            taxDeferredAccounts.addElement(account);
      }
}
```

The Account object would implement the method:

```
public boolean isTaxable() {
      return taxableFlag;
}
```

Suppose that the client (the person paying for the program, not the end-user) wants to handle tax-free accounts (e.g., Roth IRA). The following change would have to be made to the Account object:

```
public final static int TAXABLE_TYPE = 1;
public final static int TAX_DEFERRED_TYPE = 2;
public final static int TAX_FREE_TYPE = 3;

public boolean isTaxable() {
      return taxableFlag == TAXABLE_TYPE;
}

public boolean isTaxDeferred() {
      return taxableFlag == TAX_DEFERRED_TYPE;
}

public boolean isTaxFree() {
      return taxableFlag == TAX_FREE_TYPE;
}

public int getTaxableType() {
      return taxableFlag;
}
```

The controller class that displays the list of accounts would be changed to:

```
Vector taxableAccounts = new Vector();
Vector taxDeferredAccounts = new Vector();
Vector taxFreeAccounts = new Vector();
Iterator iterator = customer.getAccounts();
```

```
Account account = null;
while(iterator.hasNext()) {
      account = (Account)iterator.next();
      if (account.isTaxable()) {
            taxableAccounts.addElement(account);
      } else if account.isTaxDeferred() {
            taxDeferredAccounts.addElement(account);
      } else if account.isTaxFree() {
            taxFreeAccounts.addElement(account);
      } else {
            //log an error for invalid account type
      }
}
```

While this seems to be a simple fix, consider that this is just one additional category, and the problem may need to be fixed in many places throughout the program.

Category

Problem

The requirements often refer to different categories of domain objects, explicitly or implicitly. These categories can change and be extended as the requirements evolve. How can we avoid making broad changes when categories are added or specialized?

Forces

As the requirements and the software evolves, the developer becomes more aware of how the domain expert thinks. The code that manipulates domain objects often does so vis-à-vis the ways the domain objects have been categorized. There are two types of knowledge: the criteria for how an object is categorized and the meaning of each category.

Solution

Make knowledge about what categories are being used, and their possible values, global and explicit. Keep private the knowledge about the criteria for assigning an object to a category. This pattern is a way to model categories in the real world independent of their representation in the domain objects internal state. They are a *lingua franca* for objects that need to apply calculations on or manipulate other domain objects based on their category. A domain object should know how to map its internal state into standard Category objects. When the categories are extended only this mapping needs to be updated. Please note that while these examples represent Categories as discrete values, they could also be continuous without affecting the validity of the pattern.

You can see that the knowledge about how an object is categorized is simply scattered throughout the code. This pattern, at the simplest level suggests making the categories explicit, and simply asking the object for the category in which it belongs. While this was a rather simple example, imagine a real system with a realistic number of objects that need to be categorized across a realistic number of dimensions (called “CategoryUniverses” in this pattern language), and a realistic number of places where the rest of the system needs to know an object’s category.

If the software is structured similar to the mental model of the domain expert, it seems intuitive that it is easier to change.

Discrimination Net

Implementation

`interface Categorizable` – implemented by domain objects so they can interact with the Discrimination Net.

> `boolean Categorizable.categoryOf(Category)` – returns true if the domain object is in the given Category.

An Example

The Account class would be implemented as:

```
public class Account implements Categorizable {
	…
	//It’s a matter of style whether you define the Category’s here
	//or in a centralized class (or interface).
	public static final Category TAXABLE_CATEGORY = new Category(…);
	public static final Category TAX_DEFERRED_CATEGORY = new 
Category(…);
	public static final Category TAX_FREE_CATEGORY = new Category(…);

	//when loading from persistence, set the taxableCategory

	public boolean isCategoryOf(Category category) {
		return taxableCategory.equals(category);
	}
	…
}
```

The class that displays the accounts would be implemented as:

```
//This is the only part that has to change when new categories are added
Vector displayCategories = new Vector(3);
displayCategories.addElement(Account.TAXABLE_CATEGORY);
displayCategories.addElement(Account.TAX_DEFERRED_CATEGORY);
displayCategories.addElement(Account.TAX_FREE_CATEGORY);

Hashtable accounts = new Hashtable();
Iterator categories = displayCategories.iterator();
Category category = null;
while(categories.hasNext()) {
	category = (Category)categories.next();
	accounts.put(category, new Vector());
}

Iterator iterator = customer.getAccounts();
Account account = null;
while(iterator.hasNext()) {
	account = (Account)iterator.next();
	categories = displayCategories.iterator();
	while (categories.hasNext()) {
		category = (Category)categories.next();
		if (account.isCategoryOf(category)) {
			((Vector)accounts.get(category)).addElement(account);
		}
	}
}
```

This certainly reduces amount of code changes required when adding a new category. The `displayCategories` variable essentially represents the categories to be displayed and the order in which to display them.

The implementation of `Account.isCategoryOf(Category)` is relatively simplistic, and you might argue that you can create the same effect with constants. That would be true if there were only type of category. But it is possible for an object to have values lying in many dimensions, which is where CategoryUniverse and MultiCategory come into play.

Related Patterns

Every Category must live in a Category Universe. A Category could be a Multi-Category.

Category Universe

Problem

Categories can be compared to any other Categories, but it only makes sense to compare ones that "fit". For example, comparing "Heavy" to "Blue" is indeterminate because "Heavy" and "Blue" are not in the same dimension, or taken from the same domain (the mathematical concept "f(domain) → range").

Forces

We need to ensure that categories are compared to each other in a meaningful way.

Solution

A CategoryUniverse partitions Categories so that they can compared to each other, but only to other Categories in the same CategoryUniverse.

Implementation

`CategoryUniverse Category.categoryUniverse()` – returns the CategoryUniverse for this Category.
`Category Category.categoryFor(CategoryUniverse universe)` – returns the Category object that has the given `universe` as its CategoryUniverse, which could be null if the Category does not have a value in the given CategoryUniverse.

An Example

In the example of the "Taxable" and "TaxDeferred" Account's, we are really modeling the CategoryUniverse called AccrualTaxability, which has the not-so-interesting member Categories called "TaxableAccrual" and "NonTaxedAccrual". Every Category must have a CategoryUniverse so that it is not compared against Category's from another CategoryUniverse.

Let's take a look at the implementation of Account again. Notice how the parameters for the Category constructor had been left out. Now we can fill them in.

Discrimination Net

```
public static final CategoryUniverse TAXABLE_ACCRUAL_CU
      = new CategoryUniverse(“TaxableAccrual”);
public static final Category TAXABLE_CATEGORY
      = new Category(TAXABLE_ACCRUAL_CU, “Taxable”);
public static final Category TAX_DEFERRED_CATEGORY
      = new Category(TAXABLE_ACCRUAL_CU, “TaxDeferred”);
public static final Category TAX_FREE_CATEGORY
      = new Category(TAXABLE_ACCRUAL_CU, “TaxFree”);
```

Related Patterns

Category’s require a CategoryUniverse to be compared correctly.

Example (Continued)

Suppose that the requirements changed to display the customer’s accounts separated from the spouse’s accounts. Here is how the original code would be altered (assume also that we don’t know about tax-free accounts yet).

```
Vector taxableCustAccounts = new Vector();
Vector taxableSpouseAccounts = new Vector();
Vector taxDefrdCustAccounts = new Vector();
Vector taxDefrdSpouseAccounts = new Vector();
//getAllAccounts() returns all accounts, owned by both the customer
//and spouse
Iterator iterator = customer.getAllAccounts();
Account account = null;
while(iterator.hasNext()) {
      account = (Account)iterator.next();
      if (account.isTaxable()) {
            if (account.getOwner().equals(customer) {
                  taxableCustAccounts.addElement(account);
            } else {
                  taxableSpouseAccounts.addElement(account);
            }
      } else {
            if (account.getOwner().equals(customer) {
                  taxDefrdCustAccounts.addElement(account);
            } else {
                  taxDefrdSpouseAccounts.addElement(account);
            }
      }
}
```

The Account object would implement the method:

```
public Person getOwner() {
      return owner;
}
```

Notice that the number of variables have doubled, and that the caller has determined the category to which the different accounts belong. Adding tax-free accounts also exacerbates the problem.

Discrimination Net

Multi-Category

Problem

Many objects can simultaneously be categorized with different values (i.e., a Category) in different CategoryUniverse's (but never different Categories in the same CategoryUniverse). When comparing these objects we would like to be able to ask, "What Category are you within this CategoryUniverse?"

Forces

We need to model the real world because objects can have attributes drawn from various domains (the mathematical definition of "domain").

Solution

A domain object would create a Multi-Category that contains all the Category's in all the CategoryUniverse's that apply to it. It is convenient for an implementation of Multi-Category to be a subclass of Category in order to make use of the Composite pattern; that is, a Multi-Category can implement the same interface as Category, but using a number of objects to do so.

Implementation

`Collection Category.categoryUniverses()` – returns a collection of CategoryUniverses in which the Category exists. For a Category, this is a collection of one element, for a MultiCategory the result will probably have many elements.

`MultiCategory extends Category` – Composite pattern

`Collection MultiCategory.categories()` – returns a collection of Category objects that make up the MultiCategory.

An Example

Let's see how to display the customer's and spouse's accounts, further partitioned as tax-deferred and taxable.

```
//This is the only part that has to change when new categories are added
Vector displayCategories = new Vector(6);
MultiCategory category = new MultiCategory("TaxableCustomer");
category.addCategory(Account.TAXABLE_CATEGORY);
category.addCategory(Person.CUSTOMER_CATEGORY);
displayCategories.addElement(category);
category = new MultiCategory("TaxDeferredCustomer");
category.addCategory(Account. TAX_DEFERRED_CATEGORY);
category.addCategory(Person.CUSTOMER_CATEGORY);
displayCategories.addElement(category);
category = new MultiCategory("TaxableSpouse");
category.addCategory(Account. TAXABLE_CATEGORY);
category.addCategory(Person.SPOUSE_CATEGORY);
displayCategories.addElement(category);
category = new MultiCategory("TaxDeferredSpouse");
category.addCategory(Account. TAX_DEFERRED_CATEGORY);
category.addCategory(Person.SPOUSE_CATEGORY);
displayCategories.addElement(category);
```

```
Hashtable accounts = new Hashtable();
Iterator categories = displayCategories.iterator();
Category category = null;
while(categories.hasNext()) {
      category = (Category)categories.next();
      accounts.put(category, new Vector());
}

Iterator iterator = customer.getAccounts();
Account account = null;
while(iterator.hasNext()) {
      account = (Account)iterator.next();
      categories = displayCategories.iterator();
      while (categories.hasNext()) {
            category = (Category)categories.next();
            if (account.isCategoryOf(category)) {
                  ((Vector)accounts.get(category)).addElement(account);
            }
      }
}
```

If the requirements change to include tax-free accounts, notice that the only part that changes is the first section that sets up the MultiCategory objects. It would be relatively simple to add two more MultiCategory objects to contain “Tax-Free Customer” and “Tax-Free Spouse”.

Related Patterns

A Multi-Category is a Composite of a Category. Every Category in a Multi-Category must have a different Category Universe.

Example (Continued)

Let’s move to a different part of our application. Suppose there is a graph that displays the asset values of the various accounts owned by the customer and his/her spouse. Without using Categories, the code would look very similar to the previous example, except the action statement (within the conditional) would read “`taxableCustAccounts += account.getBalance()`” instead of “`taxableCustAccounts.addElement(account)`”. And all of the difficulties about adding new categories would remain.

Category Visitor

Problem

Sometimes we want to display objects in a very detailed way and sometimes we want to show an aggregate of the objects’ values. Objects need to be partitioned based on the Category to which each belongs.

Discrimination Net

Forces

The code should be simple to change in the face of dynamic requirements. There are two areas we would like to change: 1) the criteria for partitioning the domain objects; or 2) the operation applied to domain objects in a given partition (or all partitions).

Solution

Use a Category Visitor to aggregate domain objects based on their Category (which is commonly a Multi-Category). For each partition, create a "criteria" Category object that describes the domain objects that belong to the partition. Give the Category Visitor this collection of Category objects, one for each partition) and have it visit a collection of domain objects (that can respond with a Category or Multi-Category). The intermediate result would be a collection of partitions that contain the domain objects that match the criteria Category. The final result would include whatever operation we need to apply to domain objects in each partition.

An Example

If you look at the code from the Multi-Category example, you'll find that it is actually an implementation of a CategoryVisitor. The first part sets up of the criteria Categories, and the second part partitions the domain objects into the right "bucket". The "`accounts`" variable is the collection of partitions (a `Hashtable` of `Vector`'s) of `Account` objects. In order to do graphing, we simply want to add together the balance from each account in the partition.

Notice that if the requirements about what to display in the graph change, we can simply change the criteria Categories.

Related Patterns

The Discrimination Net uses a Category Visitor to partition objects. The Category Visitor must retrieve the Category from the domain objects in order to partition them.

Example (Continued)

In our application, suppose we wanted to project forward in time the value of the accounts. We would calculate next year's balance based on this year's balance, adjusting for interest, taxes, and inflation. The code might look like this:

```
Collection thisYear = customer.getAllAccounts();
//These would probably be calculated elsewhere
float interestRate = 0.048;
float inflationRate = 0.026;
float taxRate = 0.28;
Collection allYears = new LinkedList(); //holds the state for each year
allYears.add(thisYear);
int yearsLeft = customer.getLifeExpectancy() – customer.currentAge();
Collection nextYear = null;
while (yearsLeft > 0) {
      thisYear = deepCopy(thisYear); //clone all accounts
      Iterator iterator = thisYear.iterator();
      Account account = null;
```

```
            while(iterator.hasNext()) {
                  account = (Account)iterator.next();
                  int interest = account.getBalance() * interestRate;
                  if (account.isTaxable()) {
                        //subtract for taxes on the interest
                        interest *= 1 – taxRate;
                  }
                  newBalance = account.getBalance() + interest;
                  //Adjust for inflation
                  newBalance *= 1 – inflationRate;
                  account.setBalance(newBalance);
            }
            allYears.add(thisYear);
            yearsLeft--;
      }
```

This code looks relatively simple, but it is not very extensible. For example, suppose that you want to model the fact that once the customer reaches retirement, s/he begins withdrawing a constant amount from all assets, where the order of preference is taxable, tax-deferred, then tax-free. To implement the new requirements you would have to sort the accounts according to withdrawal preference, and you would have to differentiate between taxable accrual and taxable withdrawal.

The Discrimination Net pattern allows you to build a modular, extensible algorithm based on Categories and the Category Visitor.

Discrimination Net

Problem

Since requirements can change frequently, you would like to design an algorithm that is modular and extensible. Modularity in a calculation algorithm implies that you can consider business rules independently from the over all algorithm. This is important because you may wish, for example, to change the control flow of the algorithm for performance reasons without losing the value of the underlying business rules. Extensibility means that you can add functionality or calculations without major changes to the rest of the algorithm. An extensible algorithm doesn't have dependencies on categories, and adding or subdividing categories doesn't cause a change to its fundamental elements.

Forces

Requirements change in unexpected ways. What may seem to be a simple change can have broad effects on an algorithm's logic and performance. Since performance is always an issue, you want to avoid re-categorizing objects.

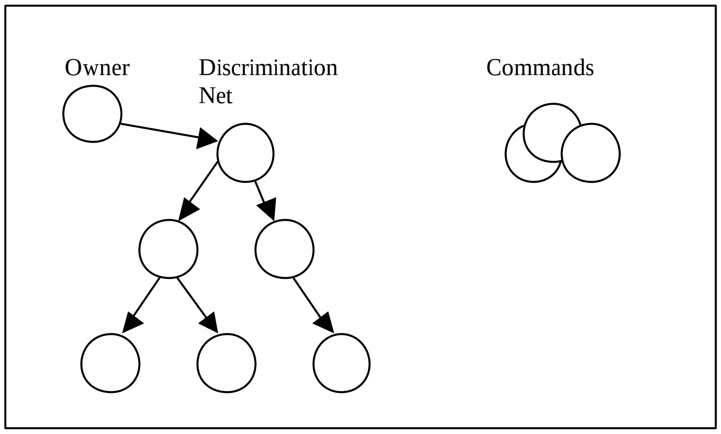
Solution

The general structure of a Discrimination Net is a tree of nodes with a specific root (a Composite). Each node in the tree represents a partition of domain objects that are treated the same by the algorithm. A domain object belongs to a node based on its Category. The tree structure can also be used for aggregating calculations based on the nodes' criteria Categories (similar to how a Category Visitor forms collections of domain objects).

Discrimination Net

Participants

- Discrimination Net – The tree of nodes, each containing a Category
- The collection of domain objects that act as input to the algorithm.
- Commands – the modular pieces of the algorithm
- Owner – The owner of the Discrimination Net, which is responsible for managing the control flow and issuing the Commands that make up the algorithm.



Discussion

The control flow is loosely determined by the structure of the Discrimination Net, and partially by the Owner object, that is, the object that created the Discrimination Net. The Owner controls the domain objects that are sent through the Discrimination Net. Each domain object ends up in the most detailed “bucket” that applies to its Category because each node (the root node included) can either accept or reject the domain object (based on its Category). If the node accepts it, the node presents it to each sub-node. This occurs recursively until it is either accepted by a leaf node, or by an intermediate node whose sub-nodes do not accept it.

The Owner can also send a Command object through the Discrimination Net. The standard behavior of a Command is that it visits all the nodes in the tree and applies an operation to each. However, similar to domain objects, each node has the opportunity to accept or reject the command. In the simple case, a Command represents the complete calculations required for the algorithm. In more complex cases there may be multiple Commands to execute different phases of the algorithm.

Applying a Command may even cause a new domain object to be created. The Owner would be responsible for sending the new object through the Discrimination Net.

Using a Discrimination Net to execute a complex algorithm is both flexible and modular. It is flexible because one can start from a relatively small tree, which treats domain objects somewhat uniformly, and extend it as one needs to apply more specialized handling. It is modular because one can add a new Command object, and it will only apply to objects for a specific node.

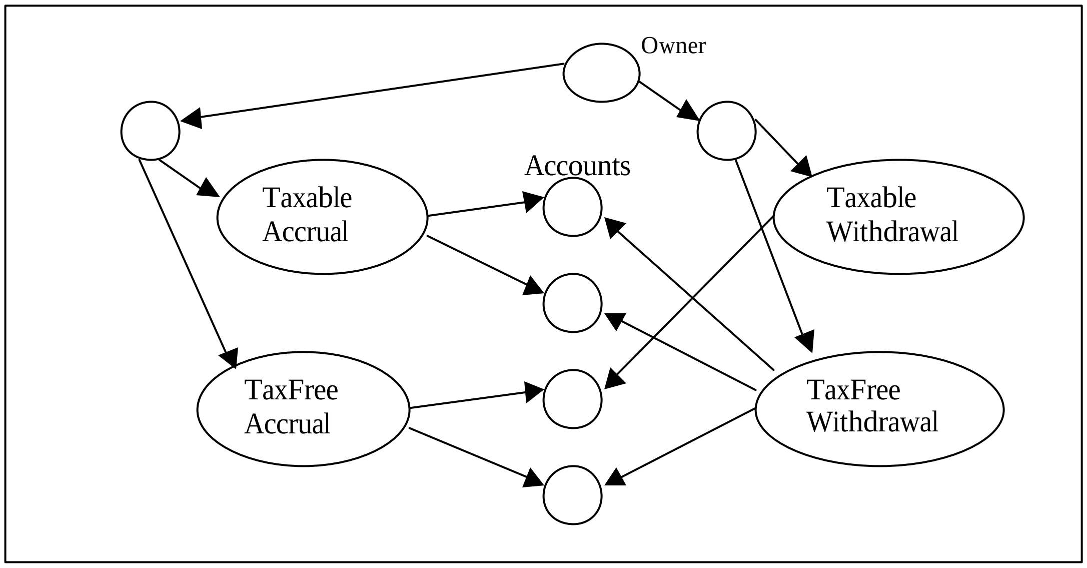
The Discrimination Net makes it possible to create a domain-specific language that the customer could use to modify the algorithm without developer intervention.

Discrimination Net

An Example

To implement our original algorithm (for calculating asset values with regard to taxes, interest, and inflation), we would only need three nodes and one command. The root node would accept all Account objects, and two sub-nodes that accept TaxableAccrual and TaxFreeAccrual, respectively. The command would match any Account and would calculate the earnings for the following year.

In order to add the withdrawal functionality, we would need to add another tree that contains the same domain objects. The new tree would consist of a root that accepts all Account objects, and two sub-nodes that accept TaxableWithdrawal and TaxFreeWithdrawal. We need the new tree because it is a different partitioning of the Account objects. There would be a Withdrawal command to model a post-tax withdrawal (which is calculated differently based on the account type).



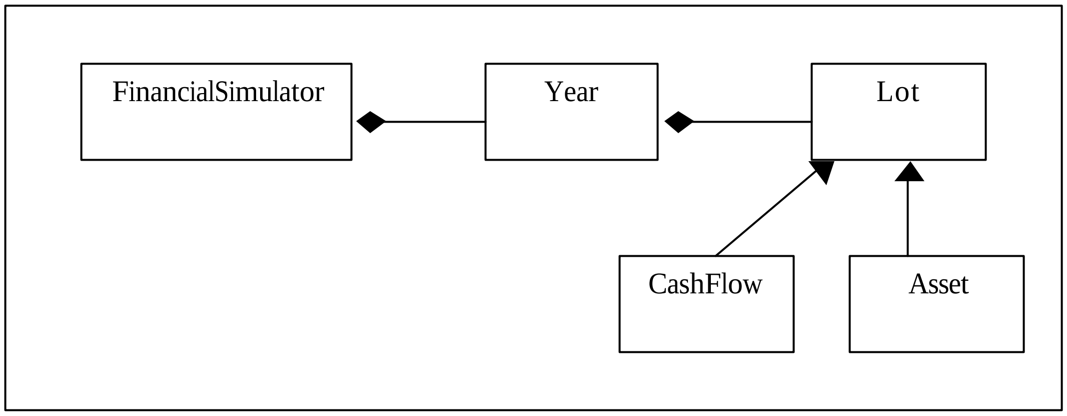
Related Patterns

The Discrimination Net uses a Category Visitor to partition the domain objects. Since every sub-node in the Discrimination Net is treated as the root of another Discrimination Net, it is an example of a Composite. A Command is used as a modular part of the algorithm.

Known Uses

The pattern language has been used in a financial services application. The requirements are to simulate a person's financial status forward in time, including retirement, until the death of both the person and his/her spouse. The user enters information about him/herself like retirement dates, salary, expected retirement income, expected life time, and assets. Here is the object model:

Discrimination Net



The Financial Simulator has the complete algorithm for simulating the financial status because the algorithm can change as it becomes more and more detailed. The Year object is essentially a container for all the Lot's for a specific year of the simulation. The Lot class can be an Asset (e.g., a bank account) or a Cash Flow (e.g., interest income). A Cash Flow object has a start and end date. For example, the user may expect to have a part-time job that starts on the retirement year and lasts for 4 years.

The initial state (i.e., the first Year) contains the current values of the customer's Asset's and all of the Cash Flow's that have been identified (e.g., a part-time job), even if they aren't yet active (i.e., not within the start and end date).

Once the simulation is complete, the system displays the results as a series of histograms. One shows each year's assets, the other shows the year's cash flows. Also, there are certain queries that the system does in order to display various information. For example, it would be instructive to know the year that all assets go to zero, if that happens.

There are three main parts to this system:

- Collect information
- Execute simulation
- Browse results of simulation

In the "Collect Information" step, there is a more complex object model for modeling what the user enters. For example, the user enters the balance (and optionally the investments) for every Account s/he holds, but in the simulation, there is a single Asset object for the aggregate of all taxable accounts.

The "Browse results of simulation" step shows two diagrams, one for assets and one for cash flows. The assets diagram shows a stacked histogram, one for each year, where the height of the histogram is total of assets, with taxable, tax-deferred and tax-free each a different color. The cash flow diagram shows a stacked histogram, one for each year, where the height of the histogram is the total of positive cash flows, with the different types of cash flows in different colors.

Using categories and the Discrimination Net makes the Financial Simulator very responsive to changing requirements, and it gives the end user a lot of control over how much detail to display in the histograms.

Existing Patterns

Fowler's Observation and Dimension patterns [Fowler97] are probably the closest existing patterns to the ones described here. They correspond loosely to Category and CategoryUniverse, respectively, even to the point of being hierarchical. The concepts of MultiCategory, CategoryVisitor and the Discrimination Net make these concepts more useful, in my opinion, especially regarding behavior and algorithms associated with measurements.

Peter Coad's Description [Coad96] pattern is a very abstract concept of associating attributes with objects, which corresponds loosely with Category. Jim Doble's Shopper [Doble95] pattern is similar to CategoryVisitor, but the condition for "shopping" is less well-defined than the concept of matching the Category. Also, the Shopper works over an arbitrary network of objects, while one usually applies a CategoryVisitor to a DiscriminationNet or a simple sequence of objects that can return a Category.

This pattern was originally written in 1997 for use in a financial services application. It has been updated to refer to work that has been done since then.

Implementation Summary

`interface Categorizable` – implemented by domain objects so they can interact with the Category Visitor and Discrimination Net.

- `boolean Categorizable.categoryOf(Category)` – returns true if the domain object is in the given Category.

`class Category`

- `boolean Category.isCategoryOf(Category criteria)` – returns true if the this Category's CategoryUniverse matches the `criteria`'s CategoryUniverse, and the `criteria`'s Category (see Appendix B).
- `CategoryUniverse Category.categoryUniverse()` – returns the CategoryUniverse for this Category, the MultiCategory returns a special instance of CategoryUniverse which means that the comparison should continue.
- `Collection Category.categoryUniverses()` – returns a collection of CategoryUniverses in which the Category exists. For a Category, this is a collection of one element, for a MultiCategory the result will probably have many elements.
- `Category Category.categoryFor(CategoryUniverse universe)` – returns the Category object that has the given `universe` as its CategoryUniverse, which could be null if the Category does not have a value in the given CategoryUniverse.

`MultiCategory extends Category` – Composite pattern

- `Collection MultiCategory.categories()` – returns a collection of Category objects that make up the MultiCategory.

APPENDIX A
BIBLIOGRAPHY

[Coad99] Peter Coad, Eric Lefebvre, Jeff De Luca, *Java Modeling in Color with UML: Enterpise Components and Process*, Prentice Hall, 1999.

[Doble95] Jim Doble, “Shopper” in *Pattern Languages of Program Design 2*, ed. Vlissides, Coplien, and Kerth, Addison-Wesley, 1996

[Fowler97] Martin Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

APPENDIX B
MATHEMATICAL SUPPORT FOR CATEGORY VISITOR

Definition: isCategoryOf(Object ***obj*** , Category ***c***)
Object ***obj*** belongs to Category ***c***

isCategoryOf(Object ***obj***, Category ***c***) iff ∀(CategoryUniverse ***j*** ∈ { categoryUniverses(***c***) ∩ categoryUniverses(category(***obj***)) },
categoryFor(category(***obj***), ***j***) == categoryFor(***c***, ***j***))

Definition: categoryFor({MultiCategory | Category}, CategoryUniverse) ➔ Category
Returns the Category within the given CategoryUniverse

categoryFor(Category ***c***, CategoryUniverse ***u***) ➔ ***c*** iff categoryUniverse(***c***) == ***u***

categoryFor(MultiCategory ***m***, CategoryUniverse ***u***) ➔ ***c*** iff ∃ (Category ***c*** ∈ categories(***m***), categoryUniverse(***c***) == ***u***)

Definition: category(Object) ➔ Category | MultiCategory
Returns the Category, which could be a MultiCategory, of an Object

Definition: categories(MultiCategory) ➔ { Category }
Returns all the categories within a MultiCategory

Definition: categoryUniverses(MultiCategory) ➔ { CategoryUniverse }
Returns all the CategoryUniverse's of the MultiCategory

categoryUniverses(MultiCategory ***m***) ➔ CategoryUniverse ***cu*** ∈ { CategoryUniverse }
where ***cu*** ∈ categoryUniverses(MultiCategory ***m***) && ∃ (Category ***c*** ∈ categories(***m***), ***cu*** == categoryUniverse(***c***))

Definition: categoryUniverse(Category) ➔ CategoryUniverse
Returns the CategoryUniverse of the Category

Axiom: There can only be one Category for a given CategoryUniverse within the same MultiCategory

∀ (Category ***c***,***d*** ∈ categories(Multi-Category ***m***),
c != ***d*** ➔ categoryUniverse(***c***) != categoryUniverse(***d***))

Axiom: Every Category has a CategoryUniverse
∀ (Category ***c***, ∃ (CategoryUniverse ***u***, categoryUniverse(***c***) == ***u***))