Architecture of Enterprise Applications XIII Architectural Patterns – Session State Patterns

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

Session State Patterns

- System Requirements
  - E-learning
  - To manage all kinds of teaching activities of one of the top universities, including enrollment, courses selection, score management, assignment management, and so on.
  - This university has over 30,000 undergraduate students and graduate students, and over 4,000 teachers.
  - This university opens over 4,000 courses every semester.
  - We should keep all data for a long period.
  - It should be integrated with other e-campus systems, such as Financial System, Research Projects Management Systems.

Session State Patterns

- Client Session State
  - Stores session state on the client
- Server Session State
  - Keeps the session state on a server system in a serialized form.
- Database Session State
  - Stores session data as committed data in the database.

Client Session State-How It Works

- Even the most server-oriented designs need at least a little Client Session State, if only to hold a session identifier.
  - With some applications you can consider putting all of the session data on the client, in which case the client sends the full set of session data with each request and the server sends back the full session state with each response.
  - This allows the server to be completely stateless.

- The client also needs to store the data. If it's a rich-client application it can do this within its own structures, such as the fields in its interface.
  - A set of nonvisual objects often makes a better bet, such as the Data Transfer Object itself or a domain model. Either way it's not usually a big problem.

- With an HTML interface, things get a bit more complicated. There are three common ways to do client session state: URL parameters, hidden fields, and cookies.

Client Session State-How It Works

- URL parameters are the easiest to work with for a small amount of data.
  - The clear limit to doing this is that the size of an URL is limited, but if you only have a couple of data items it works well, that's why it's a popular choice for something like a session ID.

- A hidden field is a field sent to the browser that isn't displayed on the Web page.
  - You get it with a tag of the form <INPUT type = "hidden">. To make a hidden field work you serialize your session state into it when you make a response and read it back in on each request.

- The last, and sometimes controversial, choice is cookies, which are sent back and forth automatically.
  - Just like a hidden field you can use a cookie by serializing the session state into it. You're limited in how big the cookie can be.

Client Session State-When to Use It

- Client Session State contains a number of advantages.
  - In particular, it reacts well in supporting stateless server objects with maximal clustering and failover resiliency.

- The arguments against Client Session State vary exponentially with the amount of data involved.
  - With just a few fields everything works nicely.
  - With large amounts of data the issues of where to store the data and the time cost of transferring everything with every request become prohibitive.
- There's also the security issue. Any data sent to the client is vulnerable to being looked at and altered.
  - Encryption is the only way to stop this, but encrypting and decrypting with each request are a performance burden.
  - Without encryption you have to be sure you aren't sending anything you would rather hide from prying eyes.

- You almost always have to use Client Session State for session identification.
  - Fortunately, this should be just one number, which won't burden any of the above schemes.

Server Session State-How It Works

- In the simplest form of this pattern a session object is held in memory on an application server.
  - You can have some kind of map in memory that holds these session objects keyed by a session ID
  - All the client needs to do is to give the session ID and the session object can be retrieved from the map to process the request.

- This basic scenario assumes, of course, that the application server carries enough memory to perform this task.
  - It also assumes that there's only one application server--that is, no clustering--and that, if the application server fails, it's appropriate for the session to be abandoned and all work done so far to be lost in the great bit-bucket in the sky.

Server Session State-How It Works

- The first issue is that of dealing with memory resources held by the session objects. Indeed, this is the common objection to Server Session State.
  - The answer, of course, is not to keep resources in memory but instead serialize all the session state to a memento for persistent storage.
  - This presents two questions: In what form do you persist the Server Session State, and where do you persist it?

- The form to use is usually as simple a form as possible, since the accent of Server Session State is its simplicity in programming.
  - Several platforms provide a simple binary serialization mechanism that allows you to serialize a graph of objects quite easily.
  - Another route is to serialize into another form, such as text--fashionably as an XML file.
  - The binary form is usually easier, since it requires little programming, while the textual form usually requires at least a little code.
  - Binary serializations also require less disk space.
  - There are two common issues with binary serialization.
    - First, the serialized form is not human readable.
    - Second, there may be problems with versioning.

Server Session State-How It Works

- Where to store the Server Session State. An obvious possibility is on the application server itself, either in the file system or in a local database.
  - This is the simple route, but it may not support efficient clustering or failover.
  - To support these the passivated Server Session State needs to be somewhere generally accessible, such as on shared server.
  - This will support clustering and failover at the cost of a longer time to activate the server

- This line of reasoning may lead, ironically to storing the serialized Server Session State in the database using a session table indexed by the session ID.
  - This table would require a Serialized LOB to hold the serialized Server Session State.
  - Database performance varies when it comes to handling large objects, so the performance aspects of this one are very database dependent.

Server Session State-How It Works

- At this point we're right at the boundary between Server Session State and Database Session State.

- If you're storing Server Session State in a database, you'll have to worry about handling sessions going away, especially in a consumer application.
  - One route is to have a daemon that looks for aged sessions and deletes them, but this can lead to a lot of contention on the session table.
  - Another approach: partitioning the session table into twelve database segments and every two hours rotating the segments, deleting everything in the oldest segment and then directing all inserts to it.
  - While this meant that any session that was active for twenty-four hours got unceremoniously dumped, that would be sufficiently rare to not be a problem.

- All these variations take more and more effort to do, but the good news is that application servers increasingly support these capabilities automatically.
  - Thus, it may well be that application server vendors can worry their ugly little heads about them.

Server Session State-When to Use It

- The great appeal of Server Session State is its simplicity.
  - In a number of cases you don't have to do any programming at all to make this work.
  - Whether you can get away with that depends on if you can get away with the in-memory implementation and, if not, how much help your application server platform gives you.
- Even without that you may well find that the effort you do need is small.
  - Serializing a BLOB to a database table may turn out to be much less effort than converting the server objects to tabular form.
- Where the programming effort comes into play is in session maintenance, particularly if you have to roll your own support to enable clustering and failover.
  - It may work out to be more trouble than your other options, especially if you don't have much session data to deal with or if your session data is easily converted to tabular form.

Database Session State-How It Works

- When a call goes out from the client to the server, the server object first pulls the data required for the request from the database.
  - Then it does the work it needs to do and saves back to the database all the data required.
- In order to pull information from the database, the server object will need some information about the session, which requires at least a session ID number to be stored on the client.
  - Usually, however, this information is nothing more than the appropriate set of keys needed to find the appropriate amount of data in the database.
- The data involved is typically a mix of session data that's only local to the current interaction and committed data that's relevant to all interactions.

Database Session State-How It Works

- One of the key issues to consider here is the fact that session data is usually considered local to the session and shouldn't affect other parts of the system until the session as a whole is committed.
  - Thus, if you're working on an order in a session and you want to save its intermediate state to the database, you usually need to handle it differently from an order that's confirmed at the end of a session.

- So how do you separate the session data? Adding a field to each database row that may have session data is one route.
  - The simplest form of this just requires a Boolean isPending field.
  - However, a better way is to store a session ID as a pending field, which makes it much easier to find all the data for a particular session.
  - All queries that want only record data now need to be modified with a sessionID is not NULL clause, or need a view that filters out that data.

Database Session State-How It Works

- Using a session ID field is a very invasive solution because all applications that touch the record database need to know the field's meaning to avoid getting session data.
- Views will sometimes do the trick and remove the invasiveness, but they often impose costs of their own.

- A second alternative is a separate set of pending tables.
  - So if you have orders and order lines tables already in your database, you would add tables for pending orders and pending order lines.
  - Pending session data you save to the pending table; when it becomes record data you save it to the real tables. This removes much of the invasiveness.
  - However, you'll need to add the appropriate table selection logic to your database mapping code, which will certainly add some complications.
  - If you use pending tables, they should be exact clones of the real tables.

Database Session State-How It Works

- You'll need a mechanism to clean out the session data if a session is canceled or abandoned.
  - Using a session ID you can find all data with it and delete it.
  - If users abandon the session without telling you, you'll need some kind of timeout mechanism.
  - A daemon that runs every few minutes can look for old session data.
  - This requires a table in the database that keeps track of the time of the last interaction with the session.

Database Session State-When to Use It

- Database Session State is one alternative to handling session state; it should be compared with Server Session State and Client Session State.
  - The first aspect to consider with this pattern is performance.
  - The second main issue is the programming effort, most of which centers around handling session state.

- In a choice between Database Session State and Server Session State the biggest issue may be how easy it is to support clustering and failover with Server Session State in your particular application server.
  - Clustering and failover with Database Session State are usually more straightforward, at least with the regular solutions.

Architecture of Enterprise Applications XIII Architectural Patterns – Other Issues in Server Side

Haopeng Chen

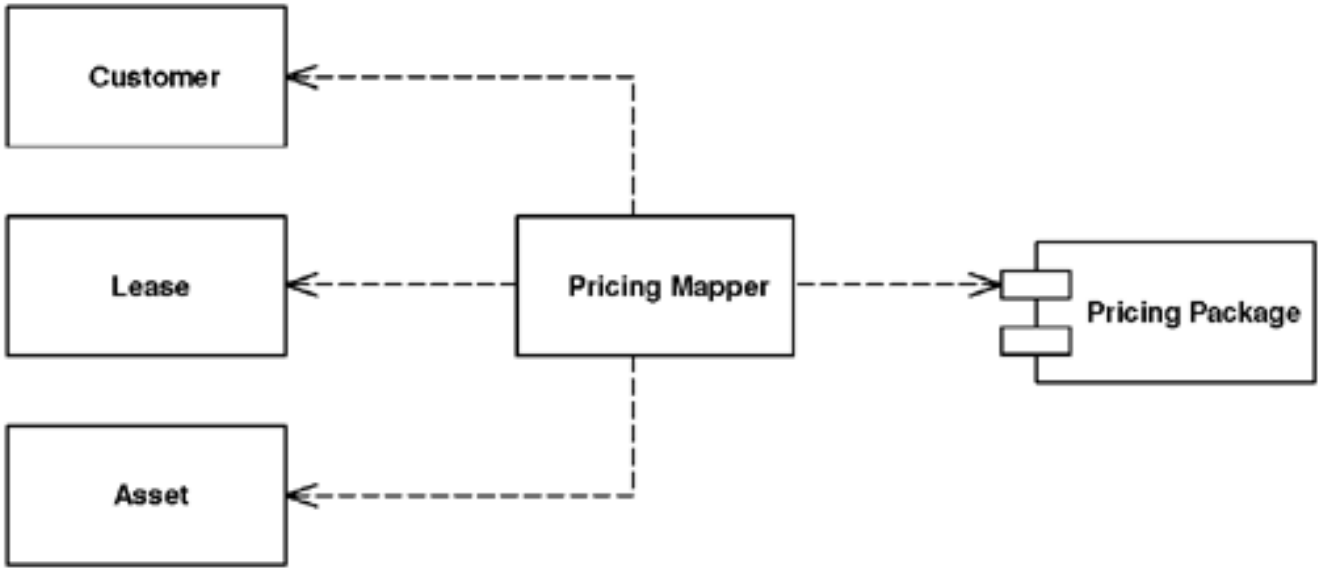
***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

Other Issues in Server Side

- Have been running for a long period, E-learning system is required to interact with E-campus system without modifying any code of E-learning and E-campus. What can we do?
- The key is the interaction can't implement by invasive way.
- Mapper is an alternative design

Mapper

- An object that sets sup a communication between two independent objects.



- Sometimes you need to set up communications between two subsystems that still need to stay ignorant of each other.
  - This may be because you can't modify them or you can but you don't want to create dependencies between the two or even between them and the isolating element.

Mapper-How It Works

- A mapper is an insulating layer between subsystems.
  - It controls the details of the communication between them without either subsystem being aware of it.
- A mapper often shuffles data from one layer to another.
  - Once activated for this shuffling, it's fairly easy to see how it works.
- The complicated part of using a mapper is deciding how to invoke it, since it can't be directly invoked by either of the subsystems that it's mapping between.
  - Sometimes a third subsystem drives the mapping and invokes the mapper as well.
  - An alternative is to make the mapper an observer of one or the other subsystem. That way it can be invoked by listening to events in one of them.
- How a mapper works depends on the kind of layers it's mapping.

Mapper-When to Use It

- Essentially a Mapper decouples different parts of a system.

- As a result you should only use a Mapper when you need to ensure that neither subsystem has a dependency on this interaction.
  - The only time this is really important is when the interaction between the subsystems is particularly complicated and somewhat independent to the main purpose of both subsystems.
  - Thus, in enterprise applications we mostly find Mapper used for interactions with a database.

- Mapper is similar to Mediator in that it's used to separate different elements.
  - However, the objects that use a mediator are aware of it, even if they aren't aware of each other;
  - the objects that a Mapper separates aren't even aware of the mapper.

Other Issues in Server Side

- If all services are require have some methods to register themselves to a service registry, and log their lifecycle. What is a feasible solution to it?
- The key is all classes in service layer have some common behavior
- Layer Supertype can be adopted as a solution.

Layer Supertype

- A type that acts as the supertype for all types in its layer.

- It's not uncommon for all the objects in a layer to have methods you don't want to have duplicated throughout the system. You can move all of this behavior into a common Layer Supertype.

Layer Supertype-How It Works

- Layer Supertype is a simple idea that leads to a very short pattern. All you need is a superclass for all the objects in a layer
  - for example, a Domain Object superclass for all the domain objects in a Domain Model. Common features, such as the storage and handling of Identity Fields , can go there. Similarly all Data Mappers in the mapping layer can have a superclass that relies on the fact that all domain objects have a common superclass.

- If you have more than one kind of object in a layer, it's useful to have more than one Layer Supertype.

Layer Supertype-When to Use It

- Use Layer Supertype when you have common features from all objects in a layer.

Layer Supertype-Example

- Domain objects can have a common superclass for ID handling.

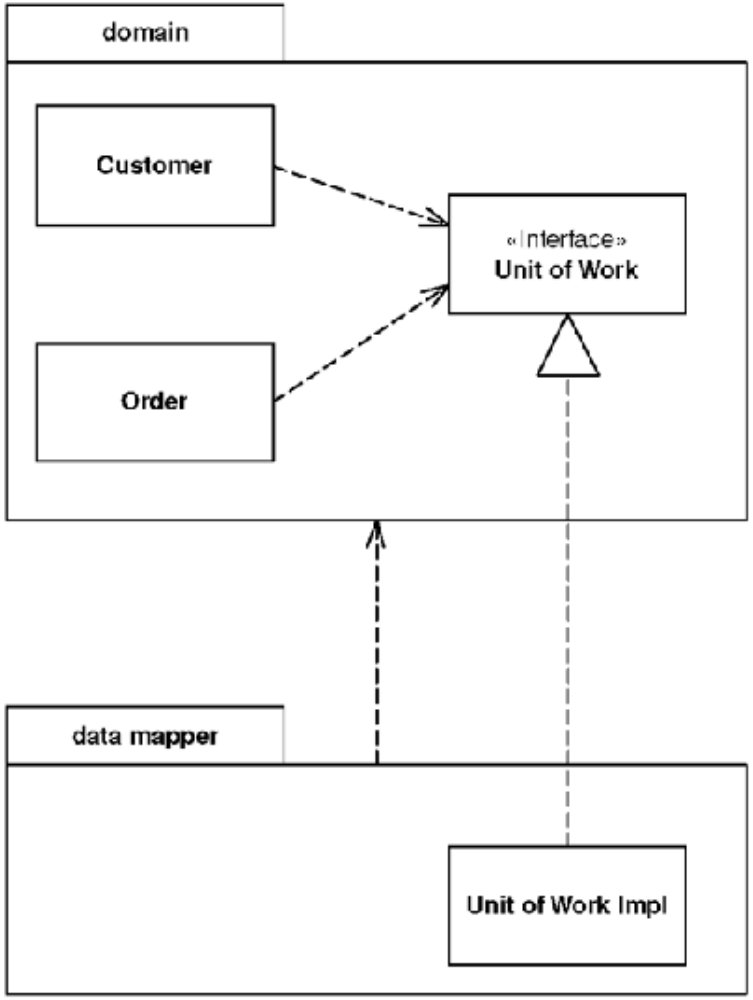
```
class DomainObject...
 private Long ID;
 public Long getID() {
  return ID;
 }
 public void setID(Long ID) {
  Assert.notNull("Cannot set a null ID", ID);
  this.ID = ID;
 }
 public DomainObject(Long ID) {
  this.ID = ID;
 }
public DomainObject() { }
```

Other Issues in Server Side

- For each service, we have two implementations which are different in QoS, because we want to provide suitable version to each university.
- But we don't want users to be aware of the difference when they develop their client applications.
- We should provide the unified interfaces to users.
- Separated Interface is a feasible pattern.

Separated Interface

- Defines an interface in a separate package from its implementation.



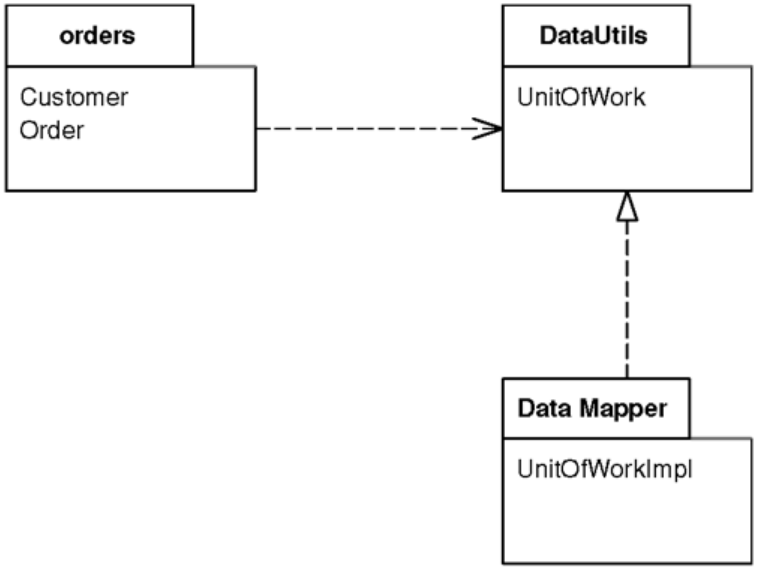
Separated Interface

- As you develop a system, you can improve the quality of its design by reducing the coupling between the system's parts.
  - A good way to do this is to group the classes into packages and control the dependencies between them.
  - You can then follow rules about how classes in one package can call classes in another.
  - for example, one that says that classes in the domain layer may not call classes in the presentation package.

- However, you might need to invoke methods that contradict the general dependency structure.
  - If so, use Separated Interface to define an interface in one package but implement it in another.
  - This way a client that needs the dependency to the interface can be completely unaware of the implementation.

Separated Interface-How It Works

- This pattern is very simple to employ.
  - Essentially it takes advantage of the fact that an implementation has a dependency to its interface but not vice versa.
  - This means you can put the interface and the implementation in separate packages and the implementation package has a dependency to the interface package.
  - Other packages can depend on the interface package without depending on the implementation package.
- You can place the interface in the client's package (as in the sketch) or in a third package



Separated Interface-How It Works

- If there's only one client for the implementation, or all the clients are in the same package, then you might as well put the interface in with the client.
- A good way of thinking about this is that the developers of the client package are responsible for defining the interface.
- Essentially the client package indicates that it will work with any other package that implements the interface it defines.
  - If you have multiple client packages, a third interface package is better.
  - It's also better if you want to show that the interface definition isn't the responsibility of the client package developers. This would be the case if the developers of the implementation were responsible for it.
- One of the awkward things about separate interfaces is how to instantiate the implementation.
  - It usually requires knowledge of the implementation class.
  - The common approach is to use a separate factory object, where again there is a Separated Interface for the factory.

Separated Interface-When to Use It

- You use Separated Interface when you need to break a dependency between two parts of the system. Here are some examples:
  - You've built some abstract code for common cases into a framework package that needs to call some particular application code.
  - You have some code in one layer that needs to call code in another layer that it shouldn't see, such as domain code calling a Data Mapper.
  - You need to call functions developed by another development group but don't want a dependency into their APIs.

- For applications I recommend using a separate interface only if you want to break a dependency or you want to have multiple independent implementations.
  - If you put the interface and implementation together and need to separate them later, this is a simple refactoring that can be delayed until you need to do it.

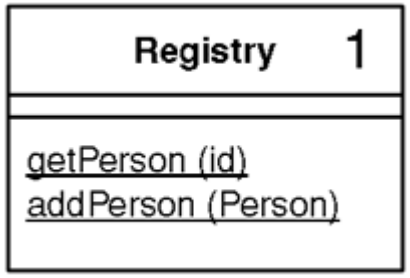
Other Issues in Server Side

- How can we get a reference to the object we want in a fast and unified way?

- Registry, a pattern similar with Identity Map, is a good choice

Registry

- A well-known object that other objects can use to find common objects and services.



- When you want to find an object you usually start with another object that has an association to it, and use the association to navigate to it.
- A Registry is essentially a global object, or at least it looks like one--even if it isn't as global as it may appear.

Registry-How It Works

- The first thing to think of is the interface, and for Registries my preferred interface is static methods.
  - A static method on a class is easy to find anywhere in an application.
  - Furthermore, you can encapsulate any logic you like within the static method, including delegation to other methods, either static or instance.

- Some applications may have a single Registry; some may have several.
  - Registries are usually divided up by system layer or by execution context.
  - My preference is to divide them up by how they are used, rather than implementation.

Registry-When to Use It

- Basically, you should only use a Registry as a last resort. There are alternatives to using a Registry.
  - One is to pass around any widely needed data in parameters.
  - Another alternative to a Registry is to add a reference to the common data to objects when they're created.
- One of the problems with a Registry is that it has to be modified every time you add a new piece of data.
- So there are times when it's right to use a Registry, but remember that any global data is always guilty until proven innocent.

Other Issues in Server Side

- How to determine whether the two PERSON objects represent the same person?
- We should compare the content of PERSON objects but not the reference of them.
- Value Object is the right way to do it.

Value Object

- A small simple object, like money or a date range, whose equality isn't based on identity.

- With object systems of various kinds, I've found it useful to distinguish between reference objects and Value Objects.
  - Of the two a Value Object is usually the smaller; it's similar to the primitive types present in many languages that aren't purely object-oriented.

Value Object-How It Works

- Defining the difference between a reference object and Value Object can be a tricky thing.
- The key difference between reference and value objects lies in how they deal with equality.
- This difference manifests itself in how you deal with them.
  - Since Value Objects are small and easily created, they're often passed around by value instead of by reference.
- If you're doing a lot of binary serializing, you may find that optimizing the serialization of Value Objects can improve performance, particularly in languages like Java that don't treat for Value Objects in a special way.

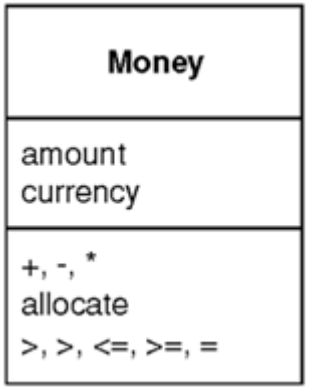
Other Issues in Server Side

REin
REliable, INtelligent & Scalable Systems

软件学院08年1月工资	薪金类酬金	RY00001	工行	3.5	86.7	5.8
评建评审费	薪金类酬金	AB509007	工行	0	220	80
软件学院发07年年薪	年终奖酬金	RY00001	工行		4.1	9.9
合计				.8	49.55	669.24999999999

Money

- Represents a monetary value.



- Money isn't actually a first class data type in any mainstream programming language.
  - The lack of a type causes problems, the most obvious surrounding currencies.
  - The more subtle problem is with rounding. Monetary calculations are often rounded to the smallest currency unit. When you do this it's easy to lose pennies (or your local equivalent) because of rounding errors.

Money-How It Works

- Money is a Value Object, so it should have its equality and hash code operations overridden to be based on the currency and amount.

- Money needs arithmetic operations so that you can use money objects as easily as you use numbers.
- But arithmetic operations for money have some important differences to money operations in numbers.
  - different currencies
  - rounding

Money-How It Works

- Storing a Money in a database always raises an issue, since databases also don't seem to understand that money is important (although their vendors do.)

Money-When to Use It

- Use Money for pretty much all numeric calculation in object-oriented environments.
- The primary reason is to encapsulate the handling of rounding behavior, which helps reduce the problems of rounding errors.
- Another reason to use Money is to make multi-currency work much easier.
- The most common objection to Money is performance.

Money-Example

- The first decision is what data type to use for the amount. If anyone needs convincing that a floating point number is a bad idea, ask them to run this code.

```
double val = 0.00;
 for (int i = 0; i < 10; i++) val += 0.10;
  System.out.println(val == 1.00);
```

- With floats safely disposed of, the choice lies between fixed-point decimals and integers, which in Java boils down to BigDecimal, BigInteger and long. Using an integral value actually makes the internal math easier, and if we use long we can use primitives and thus have readable math expressions.

```
class Money...
  private long amount;
  private Currency currency;
```

Money-Example

- It's useful to provide constructors from various numeric types.

```
public Money(double amount, Currency currency) {
  this.currency = currency;
  this.amount = Math.round(amount * centFactor());
}
public Money(long amount, Currency currency) {
  this.currency = currency;
  this.amount = amount * centFactor();
}
private static final int[] cents = new int[] {1, 10, 100, 1000 };
private int centFactor() {
  return cents[currency.getDefaultFractionDigits()];
}
```

Money-Example

- Although most of the time you'll want to use money operation directly, there are occasions when you'll need access to the underlying data.

```
class Money...
 public BigDecimal amount() {
   return BigDecimal.valueOf(amount, currency.getDefaultFractionDigits());
 }
 public Currency currency() {
   return currency;
}
```

- If you use one currency very frequently for literal amounts, a helper constructor can be useful.

```
class Money...
 public static Money dollars(double amount) {
   return new Money(amount, Currency.USD);
 }
```

Money-Example

- As Money is a Value Object you'll need to define equals.

```
class Money...
 public boolean equals(Object other) {
   return (other instanceof Money) && equals((Money)other);
 }
 public boolean equals(Money other) {
   return currency.equals(other.currency) && (amount == other.amount);
 }
```

- And wherever there's an equals there should be a hash.

```
class Money...
   public int hashCode() {
     return (int) (amount ^ (amount >>> 32));
}
```

Money-Example

- We'll start going through the arithmetic with addition and subtraction.

```
class Money...
 public Money add(Money other) {
   assertSameCurrencyAs(other);
   return newMoney(amount + other.amount);
 }

 private void assertSameCurrencyAs(Money arg) {
   Assert.equals("money math mismatch", currency, arg.currency);
 }

 private Money newMoney(long amount) {
   Money money = new Money();
   money.currency = this.currency;
   money.amount = amount; return money;
 }
```

Money-Example

- With addition defined, subtraction is easy.

```
class Money...
 public Money subtract(Money other) {
   assertSameCurrencyAs(other);
   return newMoney(amount - other.amount);
 }
```

- The base method for comparison is compareTo.

```
class Money...
 public int compareTo(Object other) {
   return compareTo((Money)other);
 }
 public int compareTo(Money other) {
   assertSameCurrencyAs(other);
   if (amount < other.amount)
     return -1;
   else if (amount == other.amount)
     return 0;
     else
      return 1;
}
```

Money-Example

- Now we're ready to look at multiplication. We're providing a default rounding mode but you can set one yourself as well.

```
class Money...
 public Money multiply(double amount) {
   return multiply(new BigDecimal(amount));
 }
 public Money multiply(BigDecimal amount) {
   return multiply(amount, BigDecimal.ROUND_HALF_EVEN);
 }
 public Money multiply(BigDecimal amount, int roundingMode) {
   return new Money(amount().multiply(amount), currency, roundingMode);
}
```

Money-Example

- If you want to allocate a sum of money among many targets and you don't want to lose cents, you'll want an allocation method. The simplest one allocates the same amount (almost) amongst a number of targets.

```
class Money...
 public Money[] allocate(int n) {
   Money lowResult = newMoney(amount / n);
   Money highResult = newMoney(lowResult.amount + 1);
   Money[] results = new Money[n];
   int remainder = (int) amount % n;
   for (int i = 0; i < remainder; i++)
     results[i] = highResult;
   for (int i = remainder; i < n; i++)
     results[i] = lowResult;
   return results;
}
```

Money-Example

- A more sophisticated allocation algorithm can handle any ratio.

```
class Money...
 public Money[] allocate(long[] ratios) {
  long total = 0;
  for (int i = 0; i < ratios.length; i++)
    total += ratios[i];
  long remainder = amount;
  Money[] results = new Money[ratios.length];
  for (int i = 0; i < results.length; i++) {
    results[i] = newMoney(amount * ratios[i] / total);
    remainder - = results[i].amount;
  }
  for (int i = 0; i < remainder; i++) {
    results[i].amount++;
  }
  return results;
}
```

Money-Example

- You can use this to solve Foemmel's Conundrum.

```
class Money...
 public void testAllocate2() {
  long[] allocation = {3,7};
  Money[] result = Money.dollars(0.05).allocate(allocation);
  assertEquals(Money.dollars(0.02), result[0]);
 assertEquals(Money.dollars(0.03), result[1]);
}
```

References

- Martin Fowler's Patterns of Enterprise Application Architecture

Thank You!