Topics in SW Design

Serialization

The first rule of Serialization (java mechanism) is DON'T USE SERIALIZATION. There are better ways to persist and transfer data. But it is widely used and the underlying concepts are worth understanding.

Serialization is to write an object to a stream, and typically a filesystem object. It allows us to save the current graph and state of objects to permanent storage. If the object changes its definition, then unserializing the objects will fail (e.g., change a field named `age` to `pAge`).

Compare to Python's `pickle` mechanism.

Examples

Save collection of objects to stream (`marshal`)

```
Employee[] staff = new Employee[2];
staff.add(new Employee(...));
staff.add(new Employee(...));
```

Construct ObjectOutputStream:

```
ObjectOutputStream out
   = new ObjectOutputStream(
      new FileOutputStream("staff.dat"));
```

Save the array and close the stream

```
out.writeObject(staff);
out.close();
```

And reverse this by unserializing or unmarshalling.

Serializes the object and its dependencies. No extra work needed. Use with `serializable` interface.

Not every object is serializable. (check with something like `if(someObj instanceof Serializable)`). In this case we can write our own code. See the example of Ellipse in the Car.java code. Since this class does not implement `Serializable` it will cause problems unless we mark it `transient` and implement our own logic (in this case, storing the primitive measures of what makes a wheel).

Serialization using Java is pretty simplistic and has ~~some~~ many problems.

- serialize everything, when that may not be necessary
- representation depends on class definition, not domain
- not human readable (vital during development of web services)
- can mess with inheritance (since the serialized object expects a certain format that child classes may violate).
- exposes internals, which can now be messed with. Mark sensitive data with `transient`.
- essentially `readObject` loads executable code from a random place (like `exec`). Never unserialize untrusted data (e.g., data that has been out of your control, or from another system).
- not a long-term solution.

Persistence is a longer-term concept. The main distinction is that we will think carefully about the objects being stored, and map the things we need to store into a representation format. In the car example, rather than serializing the `Car` object, we instead need to think "what do I need to store about cars".

JSON: We have already covered/worked with JSON serialization. JSON is derived from Javascript Object Notation, and is basically serialization for JS objects. Since those concepts are so useful (lists, dicts) it has taken off. Most Web apis now expect content-type `text/json`.

- Disadvantage: textual format, executable (exploitable), exposes internals.

Compare to roll-your-own text files, ASN.1 (binary), CSV, XML, YAML, netCDF, and Protobuffers.

Protocol buffers

Google internal data serialization, with schema for checking conformance (unlike JSON, which has no schema). Comes with toolset for fast data transfer and several language specific libraries.

Begin by declaring the data structures (messages) you wish to serialize

- addressbook.proto

Then compile the protobuffer to Java code. This generates a data object for you with getters, setters, and a Builder (the Builder pattern) for creating new objects.

- the biggest benefit (and why Google invented them) is that a single .proto file can be cross-compiled into C, Java, Python, Go, and ensure message interoperability. Data interoperability is a major headache (think endianness and work up from there).
- protocol buffers do not store complex logic, just data serialization.

Data storage

Ultimately long-term storage uses a system like Hibernate to map objects (`Car`) into relational tables (`cars`) (object-relational mapping ORM). Be aware that the conceptual model of an Object is fundamentally different to that of `Relation`. This leaves open the possibility that your ORM process is leaky: for sizable applications, something like 10% of your storage needs will require in depth analysis of relational database logic (e.g. to prevent expensive joins).

An alternative is to store data in a non-relational structure, such as key:value stores (Redis), graphs (Neo4J), or documents (Mongo).

Refactoring

Refactoring is the art of improving code quality **supported by tests**, in small steps (distinct from ***rearchitecting, modernizing, reworking***). Strictly speaking refactoring should NOT change system behavior (wrt functional behavior). Good OO design and code should be constantly thinking about how to improve readability, encapsulation, and other maintainability concerns. Code reviews might catch this, but often it is similar to the mason who takes that extra bit of time to ensure the concrete is perfectly smooth.

Code smells

Code smells, also from the Refactoring book, are heuristics that help us identify problems in our codebase. Here are a couple:

- *shotgun surgery*: one change (in domain) causes updates to many, scattered files.
- *long method*: the longer a method is, the harder it is to understand. Can look at comments to find pieces of the function that belong together and `extract method`.
- *long parameter list* : indicates highly coupled code, may have value if the caller has to assemble the dependencies. But perhaps we can replace the dependencies with method calls, or create a new object with the old parameters.
- *primitive obsession*: instead of using an object or type, we instead force everything into a primitive type like `int` or `string`. E.g., phone numbers.

Many others exist, and there are decent tools for checking many of them automatically. We should combine the code smell heuristics with our design knowledge---encapsulation, information hiding, SOLID, etc.---to figure out a good way to refactor the code.

Refactoring Support and Examples

Good IDEs have built-in refactoring support. The simplest refactoring is `Rename` (as shown in IntelliJ). Change "rental" to "aRental" for example. Note that renaming is surprisingly complex, as we want to change all instances of the name, even in comments.

Approaches

1. Draw a simple 'model'. Most of us cannot think abstractly without cognitive aids. Maybe UML, maybe CRC cards, maybe state machines...
2. Think high level. Remember implementation details don't matter (yet!)
3. Think static, dynamic, behavioral views (class, object, sequence/statechart)
4. Consider maintenance scenarios. If X changes, what else happens?
5. Look at cohesion: what responsibilities does this class have?
6. QA focus: what important qualities do we need to consider?
    - performance? scalability? (common interview qs!)

Model

- what are the three classes?
- how do they connect to each other?
- how *should* they connect to each other?

Take 1

movie-rentalv1.puml

Observations

- the program works!
- what if we want to print a statement in HTML?
  - rewrite entire statement method!
  - clone and change, not too hard
- what if we now change the calculations? E.g. a frequent renter gets a discount?
- The code is pretty hard to change given new requirements. Note that the new requirements are new functions and not the subject of refactoring.

Another common refactoring is `extract method`. (As I look at `amountFor`, I can see that it uses information from the rental, but does not use information from the customer.) We move amountFor to `Rental` and renamed it. The tests still passed.

Step 1: write good tests. Make sure we know it works. Step 2: what violates our aesthetic? (can also use metrics!) Step 3: make small changes and re-test (important tests are fast!) Step 4: does each method "belong"/change with with that object?

Step 5: is it worth it/ are we going to need it?

movierental-v3.puml

Resources

1. http://www.cs.mun.ca/jdk1.6/platform/serialization/spec/security.html
2. https://developers.google.com/protocol-buffers/docs/javatutorial
3. https://martinfowler.com/bliki/OrmHate.html
4. [NoSQL Distilled]
5. https://developers.google.com/protocol-buffers/docs/javatutorial
6. Refactoring Book