CSE 331: Design Patterns

Defensive Programming in ADTs

Recall defensive programming: checking properties you "know" to be true at runtime.
For example, it is generally a good idea to check your preconditions, unless they are expensive to check.

In an ADT implementation, the representation invariant should always be true (when not in the middle of a method).
It is a good idea to check this. Where should we check it?
- At the end of the constructor. (The constructor should establish the RI.)
- At the end of every mutator. (The mutator changed the state, and the new state must still satisfy the RI.)
- At the beginning of every method. (Rep exposure.)

It is CSE 331 best practice to define a private method in every ADT called `checkRep()` to check that ADT's RI.
`checkRep()` should throw an exception if the RI is false, and otherwise do nothing (return void).
`checkRep()` should be called in all the places mentioned above.

Programming in this way increases the effectiveness of testing. (Why?)

Design Patterns

A *design pattern* is a known solution to a common programming problem.
Popularized by "Gang of Four" book: *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma *et al.*
Each pattern tends to work around lack of programming language features.

Example pattern: **Exceptions**
Problem: Errors in one part of the code should be handled elsewhere, without returning error codes (clutter)
Solution: Use language structures for throwing and catching exceptions.
Disadvantages: Exceptional control flow can be hard to trace. Implementation can be inefficient.

Categories of design patterns:
Creational: having to do with creating objects
Structural: controlling the way objects refer to each other in the heap
Behavioral: affecting object semantics

Java constructors are inflexible:
They do not have names.
Cannot return a subtype of the class they belong to instead.
Cannot decide, in the constructor, not to allocate the object.

Creational patterns work around these inflexibilities.
Factories: factory method/object, dependency injection
Sharing: singleton, interning

Factories
Problem: client code wants to control object creation (select between different implementations, ...)
Solutions:
Factory method: method on the client called by the library whenever it wants to create an object
Factory object: separate object to hold factory method(s)
Dependency injection: Factory that is configurable using runtime data, not code.

```
interface Matrix { ... }
class DenseMatrix implements Matrix { ... }
class SparseMatrix implements Matrix { ... }
```

```
class MatrixFactory {
  public static Matrix createMatrix() {
    return new SparseMatrix();   } }
```

Sharing
Problem: constructors always return a *new* object, never a pre-existing one
Solutions:
Singleton: only one object of the class exists at runtime
Interning: only one object per value exists at runtime

```
class DatabaseConnection {
  private static DatabaseConnection theConn;
  private DatabaseConnection() { ... }
  public static getConnection() {
    if (theConn == null)
      theConn = new DatabaseConnection();
    return theConn; } }
```

```
class String {
  private static HashMap<String, String> toCanonical;
  private static String getInstance(String s) {
    if (!toCanonical.containsKey(s))
      toCanonical.put(s, s)
    return toCanonical.get(s); } }
```