Comp 411
Principles of Programming Languages
Lecture 30
Retrospective On Program Design

Corky Cartwright

April 19, 2021

Common Issues

- Python is pragmatic rather than a principled basis for teaching program design. The biggest weakness of Python is that it does not obey lexical scoping rules or any coherent alternative. Even dynamic scoping (a broken variant of lexical scoping introduced to simplify the implementation of passing functions as arguments) has an easily described if ugly semantics. Python is the only mainstream high-level language since Fortran with that does not use either lexical scoping or its crippled sibling, dynamic scoping.
- C, C++, and Java all obey lexical scoping rules but restrict lexical scope by limiting procedure and method definitions to the top-level (except via the complex backdoor of inner classes in the case of Java and recent versions of C++).
- In Java and C++, the absence of nested methods (which can be achieved at the cost of additional semantic complexity by using inner classes) is typically addressed by passing objects as parameters. In designing OO programs, it is easy to bundle the requisite values in a small number of objects that serve as the receiver and parameters in method calls.

Common Issues

- A good case can be made for defining a Java or C++ class as an inner class when class instances nced access to fields or method variables (only available when the inner class is nested inside the method!) in the enclosing class. (So-called static inner classes should not be called “inner classes” because the are simply top-level classes with names that are qualified by the name of the enclosing class.)
- The trade-off in using an inner class versus using a top level class and passing the fields and method variables to be accessed is based on whether the class should really be hidden from exposure in the API and from direct testing. Recall that every instance of an inner class must have an enclosing instance and that this enclosing instance is a “hidden” parameter in the constructor for the inner class. In most cases, the code that creates an inner class is located within the enclosing class. In this case, the compiler automatically passes `this` as the enclosing instance. If the inner class instance is created outside the enclosing class, then the enclosing instance must be passed explicitly as the first argument to the constructor.

Java vs Algol Runtimes

- The Java runtime is simpler than the classic Algol 60 runtime which was designed to support lexical scoping in nearly full generality.
- The missing generality in the Algol 60 run-time is the lack of support for procedures/functions as “first-class” values that can be bound to variables, returned as results of procedures/methods, assigned to fields of objects/structures/records.
- Guy Steele completely solved this problem by
  - Assuming the presence of a heap (for dynamic allocation of objects/structures/records).
  - Using lightweight closures that copy the invariant bindings [values and immutable cell addresses] corresponding to the specific free variables in the procedure/method code instead linking to the entire closing environment).
  - Representing the invariant binding of a closed over mutable variable in its activation record using a level of indirection placing the actual mutable cell in the heap, so it is never deallocated as long as it is accessible.
  - Guy Steele’s generalized Algol 60 runtime is now the standard runtime for languages that support nested procedures and functions (like Swift).
- Since methods cannot be nested in Java (except via inner classes), there is no static link in the activation record for a method invocation. C and C++ share this “simplification”. (Not that this simplification does not simplify the well-written code!)

Commenting Guidelines

Like good writing, good commenting is difficult and requires judgement. A few observations:

- Over-commenting in the form of stating the obvious content of a few lines of code is perhaps the most common mistake in writing comments. It insults the reader's intelligence and distracts attention from comments that actually do have content.
- The most important comments are:
  - Descriptive class names and compatible variable names (which can be short). Descriptive class names are critical because they effectively name program datatypes. The class solutions to Assignments leverage this form of commenting.
  - Class descriptions: What invariant on the class fields is maintained except during the execution of class methods?
  - Method contracts. The contract should be written in terms of the program state a the point where the method is called and specify what value is returned. It also must mention any side effects to class fields.
  - Descriptions of the meanings of class fields and local variables (other than obvious temporaries) are also helpful. Note that the meanings of method parameters are typically already provided with implicit descriptions by the corresponding method contracts. The standard Javadoc notation for documenting method is lame compared to genuine contracts (which need to mention any side effects to class fields in addition to specifying the returned values).
  - Any non-obvious invariant or program property on which the meaning or corrections of a chunk of code depends.
  - Concise description of the application level comments in the ReadMe file.
  - In a real application, the API is typically the signatures and descriptions of the public methods.

Stylistic Suggestions

1. Program in a mostly functional style. Pretend that you are programming an elegant functional language and map the abstractions that you generated into Java design patterns, e.g. pattern-matching -> visitor pattern. (Haskell-like code is perhaps "a bridge too far" but scheme/racket abstractions are well suited ot Java. Use imperativity only when it is mandated by a method contract or yields major performance gains.
2. Avoid wasteful algorithmic approaches (*e.g.*, linear searches when binary searches or equivalent are easy to express) but do not try to optimize your initial solution.
3. Once you have a working program, consider using the following optimizations to make it run faster:
   a. Tail recursion (when it shrinks asymptotic complexity)
   b. Memoization
   c. Rewriting tail recursive code using explicit loops (Java)

Program Testing

When I use the term unit testing in the context of Java programming, I am referring to comprehensively testing every non-trivial program method. Methods that are part of the "published" API must be tested directly. Other methods can be tested indirectly, but you need to be sure that all lines of code that are intended for execution are covered by a test. The notion of a line of code is actually a bit larger than ideal. Every node in the program AST that is intended to executed in normal program operation should be tested. Unfortunately, code coverage tools typically do not support such a fine level of granularity in coverage logging. (Some program coordinates in real applications are presumed unreachable by their creators, but humility leads these creations to mark unreachable points of control with aborting error messages and error codes to cope with mistakes in their control analysis.)

In principle test case generation should be driven more by method contracts than method code. Every non-trivial program method typically has a primary argument. In most cases the type of that argument has a simple inductive definition. Your tests for the method need to cover all of the cases in that inductive definition plus a few larger cases that combine different forms of data (if applicable). For exampled, for the type of simple arithmetic expressions built from natural numbers, and the binary operators {+, -, *, /}, you need to test a few small numbers (including 0), applications of each of the binary operators, putting small numbers including 0 in each possible position, and a few larger composite expressions. Writing comprehensive tests can be time consuming.

Coping with Concurrency

Concurrency makes everything harder. When possible, rely on embarrassing parallelism: subdivide your problem into completely independent subproblems where the only synchronization mechanisms are producer/consumer (Unix pipes) and futures (where no consumption takes place).

My advice: take a course from John Mellor-Crummey.