the resultant "engine" look like? Would you be able to reuse that engine in the context of a different application?

Exercises

28. Which of the following things would be better represented as code within a program, and which externally as metadata?

1. 1. Communication port assignments
2. 2. An editor's support for highlighting the syntax of various languages
3. 3. An editor's support for different graphic devices
4. 4. A state machine for a parser or scanner
5. 5. Sample values and results for use in unit testing

Temporal Coupling

What is *temporal coupling* all about, you may ask. It's about time.

Time is an often ignored aspect of software architectures. The only time that preoccupies us is the time on the schedule, the time left until we ship—but this is not what we're talking about here. Instead, we are talking about the role of time as a design element of the software itself. There are two aspects of time that are important to us: concurrency (things happening at the same time) and ordering (the relative positions of things in time).

We don't usually approach programming with either of these aspects in mind. When people first sit down to design an architecture or write a program, things tend to be linear. That's the way most people think—*do this* and then always *do that.* But thinking this way leads to *temporal coupling:* coupling in time. Method A must always be called before method B; only one report can be run at a time; you must wait for the screen to redraw before the button click is received. Tick must happen before tock.

This approach is not very flexible, and not very realistic.

We need to allow for concurrency[3] and to think about decoupling any time or order dependencies. In doing so, we can gain flexibility and reduce any time-based dependencies in many areas of development: workflow analysis, architecture, design, and deployment.

[3] We won't go into the details of concurrent or parallel programming here; a good computer science textbook should cover the basics, including scheduling, deadlock, star-vation, mutual exclusion/semaphores, and so on.

Workflow

On many projects, we need to model and analyze the users' workflows as part of requirements analysis. We'd like to find out what *can* happen at the same time, and what must happen in a strict order. One way to do this is to capture their description of workflow using a notation such as the *UML activity diagram.*[4]

[4] For more information on all of the UML diagram types, see [FS97].

An activity diagram consists of a set of actions drawn as rounded boxes. The arrow leaving an action leads to either another action (which can start once the first action completes) or to a thick line called a *synchronization bar.* Once *all* the actions leading into a synchronization bar are complete, you can then proceed along any arrows leaving the bar. An action with no arrows leading into it can be started at any time.

You can use activity diagrams to maximize parallelism by identifying activities that *could be* performed in parallel, but aren't.

Tip 39

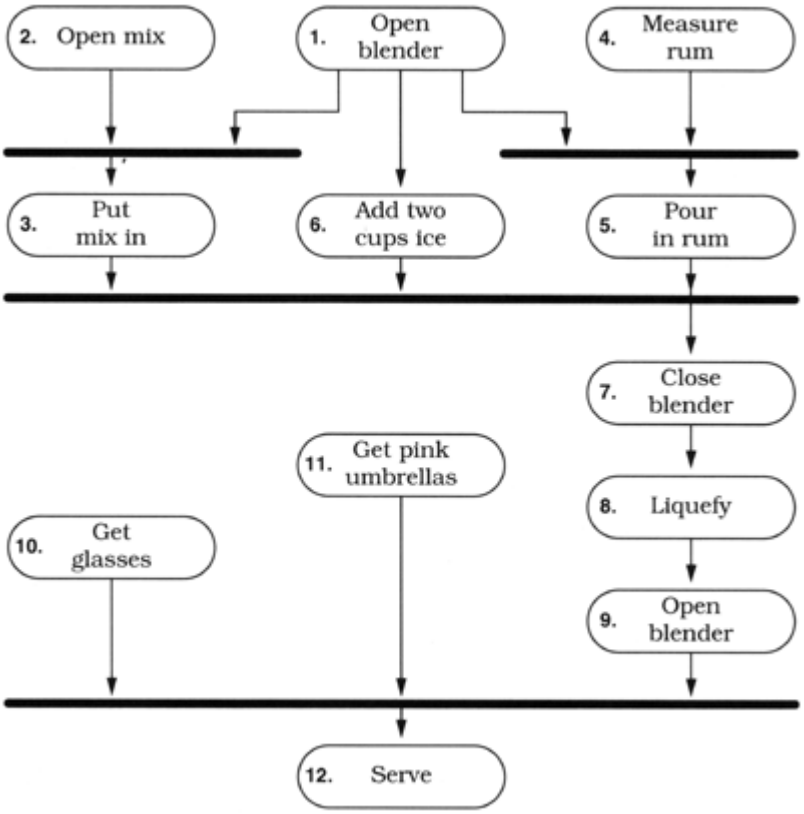
Analyze Workflow to Improve Concurrency

For instance, in our blender project (Exercise 17, page 119), users may initially describe their current workflow as follows.

1. 1. Open blender
2. 2. Open piña colada mix
3. 3. Put mix in blender
4. 4. Measure 1/2 cup white rum
5. 5. Pour in rum
6. 6. Add 2 cups of ice
7. 7. Close blender
8. 8. Liquefy for 2 minutes
9. 9. Open blender
10. 10. Get glasses
11. 11. Get pink umbrellas
12. 12. Serve

Even though they describe these actions serially, and may even perform them serially, we notice that many of them could be performed in parallel, as we show in the activity diagram in Figure 5.2 on the next page.

Figure 5.2. UML activity diagram: making a piña colada



It can be eye-opening to see where the dependencies really exist. In this instance, the top-level tasks (1, 2, 4, 10, and 11) can all happen concurrently, up front. Tasks 3, 5, and 6 can happen in parallel later.

If you were in a piña colada-making contest, these optimizations may make all the difference.

Architecture

We wrote an On-Line Transaction Processing (OLTP) system a few years ago. At its simplest, all the system had to do was read a request and process the transaction against the database. But we wrote a three-tier, multiprocessing distributed application: each component was an independent entity that ran concurrently with all other components. While this sounds like more work, it wasn't: taking advantage of temporal decoupling made it *easier* to write. Let's take a closer look at this project.

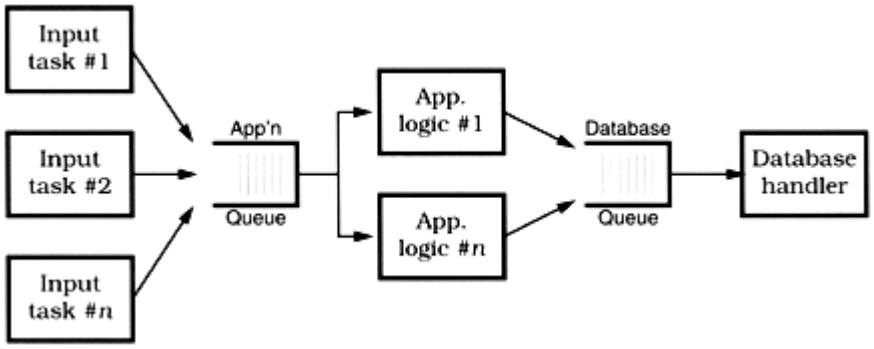
The system takes in requests from a large number of data communication lines and processes transactions against a back-end database.

The design addresses the following constraints:

- Database operations take a relatively long time to complete.
- For each transaction, we must not block communication services while a database transaction is being processed.
- Database performance suffers with too many concurrent sessions.
- Multiple transactions are in progress concurrently on each data line.

The solution that gave us the best performance and cleanest architecture looked something like Figure 5.3.

Figure 5.3. OLTP architecture overview



Each box represents a separate process; processes communicate via work queues. Each input process monitors one incoming communication line, and makes requests to the application server. All requests are asynchronous: as soon as the input process makes its current request, it goes back to monitoring the line for more traffic. Similarly, the application server makes requests of the database process,[5] and is notified when the individual transaction is complete.

[5] Even though we show the database as a single, monolithic entity, it is not. The database software is partitioned into several processes and client threads, but this is handled internally by the database software and isn't part of our example.

This example also shows a way to get quick and dirty load balancing among multiple consumer processes: the *hungry consumer* model.

In a hungry consumer model, you replace the central scheduler with a number of independent consumer tasks and a centralized work queue. Each

consumer task grabs a piece from the work queue and goes on about the business of processing it. As each task finishes its work, it goes back to the queue for some more. This way, if any particular task gets bogged down, the others can pick up the slack, and each individual component can proceed at its own pace. Each component is temporally decoupled from the others.

Tip 40

Design Using Services

Instead of components, we have really created *services*—independent, concurrent objects behind well-defined, consistent interfaces.

Design for Concurrency

The rising acceptance of Java as a platform has exposed more developers to multithreaded programming. But programming with threads imposes some design constraints—and that's a good thing. Those constraints are actually so helpful that we want to abide by them whenever we program. It will help us decouple our code and fight programming by coincidence.

With linear code, it's easy to make assumptions that lead to sloppy programming. But concurrency forces you to think through things a bit more carefully—you're not alone at the party anymore. Because things can now happen at the "same time," you may suddenly see some time-based dependencies.

To begin with, any global or static variables must be protected from concurrent access. Now may be a good time to ask yourself *why* you need a global variable in the first place. In addition, you need to make sure that you present consistent state information, regardless of the order of calls. For example, when is it valid to query the state of your object? If your object is in an invalid state between certain calls, you may be relying on a coincidence that no one can call your object at that point in time.

Suppose you have a windowing subsystem where the widgets are first created and then shown on the display in two separate steps. You aren't allowed to set state in the widget until it is shown. Depending on how the code is set up, you may be relying on the fact that no other object can use the created widget until you've shown it on the screen.

But this may not be true in a concurrent system. Objects must always be in a valid state when called, and they can be called at the most awkward times. You must ensure that an object is in a valid state *any time* it could possibly be called. Often this problem shows up with classes that define separate constructor and initialization routines (where the constructor doesn't leave the object in an initialized state). Using class invariants, discussed in Design by Contract, will help you avoid this trap.

Cleaner Interfaces

Thinking about concurrency and time-ordered dependencies can lead you to design cleaner interfaces as well. Consider the C library routine `strtok`, which breaks a string into tokens.

The design of `strtok` isn't thread safe,[6] but that isn't the worst part: look at the time dependency. You must make the first call to `strtok` with the variable you want to parse, and all successive calls with a `NULL` instead. If you pass in a non-`NULL` value, it restarts the parse on that buffer instead. Without even considering threads, suppose you wanted to use `strtok` to parse two separate strings at the same time:

[6] It uses static data to maintain the current position in the buffer. The static data isn't protected against concurrent access, so it isn't thread safe. In addition, it clobbers the first argument you pass in, which can lead to some nasty surprises.

```
char buf1[BUFSIZ];
char buf2[BUFSIZ];
char *p, *q;

strcpy(buf1, "this is a test");
strcpy(buf2, "this ain't gonna work");

p = strtok(buf1, " ");
q = strtok(buf2, " ");
while (p && q) {
  printf("%s %s\n", p, q);
  p = strtok(NULL, " ");
  q = strtok(NULL, " ");
}
```

The code as shown will not work: there is implicit state retained in `strtok` between calls. You have to use `strtok` on just one buffer at a time.

Now in Java, the design of a string parser has to be different. It must be thread safe and present a consistent state.

```
StringTokenizer st1 = new StringTokenizer("this is a test");
StringTokenizer st2 = new StringTokenizer("this test will work");

while (st1.hasMoreTokens() && st2 hasMoreTokens()) {
  System.out.println(st1.nextToken());
  System.out.println(st2.nextToken());
}
```

`StringTokenizer` is a much cleaner, more maintainable, interface. It contains no surprises, and won't cause mysterious bugs in the future, as `strtok` might.

Tip 41

Always Design for Concurrency

Deployment

Once you've designed an architecture with an element of concurrency, it becomes easier to think about handling *many* concurrent services: the model becomes pervasive.

Now you can be flexible as to how the application is deployed: standalone, client-server, or *n*-tier. By architecting your system as independent services, you can make the configuration dynamic as well. By planning for concurrency, and decoupling operations in time, you have all these options—including the stand-alone option, where you can choose *not* to be concurrent.

Going the other way (trying to add concurrency to a nonconcurrent application) is *much* harder. If we design to allow for concurrency, we can more easily meet scalability or performance requirements when the time comes—and if the time never comes, we still have the benefit of a cleaner design.

Isn't it about time?

Related sections include: