Designing for Integration

At FooSoft, your former coworker Paul has decided to leave the software enterprise for good and start his life anew as a contestant on “Deadliest Catch”. Because you’re not particularly adept at crab fishing, and prefer the hot summer weather of Chicago, IL, you and your team will need to pick up where Paul left off. Paul liked to work alone, and more or less kept to himself, so the details of his projects tend to be a bit sketchy. However, after digging through some old design documents, your team figured out the basic outline of what needs to be done.

The Task

Paul was working on several components for a payroll system, and had made some progress on a periodic report of employee hours. The sketch below gives a rough idea of what the final report should look like:

Employee	Week 1	Week 2	Total
Joe Frasier	13	27	40
John Adams	20	21	41
Sarah Miller	14	28	42
Ralph Jackson	18	39	57
Albert Frost	49	10	59

While not very fancy, it gives you a sense of the overall structure of the report and what data it needs to contain. FooSoft (amazingly) tends not to care about the particular look and feel of their reports, so long as they are provided in one of the following file formats: HTML, PDF, CSV, or plain text. That means your team is free to pick whatever format you’re most comfortable with.

The Data

Before his departure, Paul had just finished up the first stages of this project, an exporter that takes data from the current payroll system and converts it to YAML. While you don't happen to have the source for this utility handy, you do have a data file produced by it (payroll_data.yaml). Unfortunately, it is without documentation, so that means you're team will need to do a bit of exploration before they can begin coding.

Design Notes

Typically, things are prone to change rapidly at FooSoft. For this reason, it is best to decompose the problem into 2 separate components. The first is the DataProcessor, which extracts and transforms the YAML data into something closer to what is needed for the report. The second is the ReportGenerator, which takes the processed data and renders it in a particular format. The two should be wrapped together under a common interface, allowing the report to be generated using an interface similar to the one below:

```
TimeReport.new(data, period_start_date).render(filename)
```

The main idea is that the business logic should be separated from the formatting, so that a change to one component doesn't necessarily require a change in the other.

Division of Work

Because this task is easy to split up, it's a good idea to work on it in parallel. Both parts of your team need to do what they can to make integration easier, and some communication is necessary to agree on what the connections between the two modules will look like. Ideally speaking, your call to `TimeReport` should be doing little more than delegation, so be sure to discuss what APIs both sides will need to get the job done.

Requirements Discovery and Technical Assistance

For some strange reason, the trio that hosts *The Compleat Rubyist* have been hanging around FooSoft looking over developers shoulders as they work. And they also somehow have magic insight into Paul's project that I am having trouble bridging into this themed narrative.

If you have questions about the requirements or need some help with your code, just ask, and we will materialize out of thin air and come to assist you.