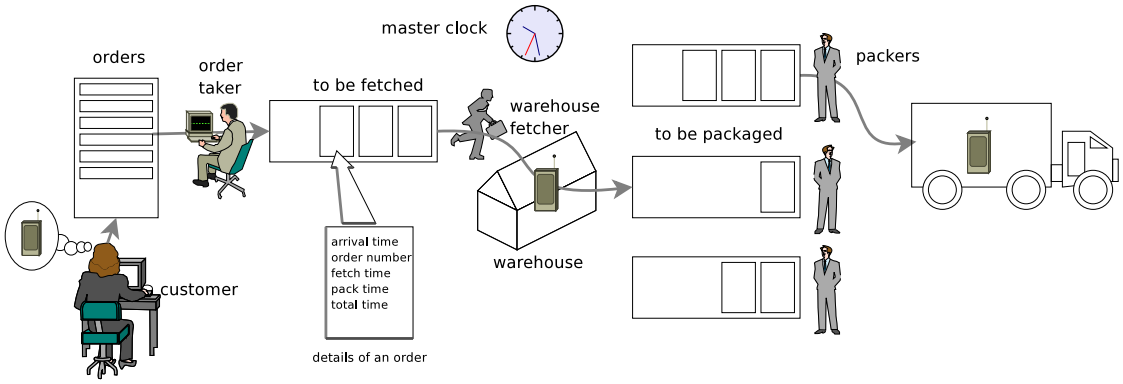
``Ask not what you can do with your data; ask what your data can do for itself´´



For example: an order might arrive at t = 20 min, take 3 minutes to fetch from the warehouse, and take 4 minutes to pack in a box for shipping.

But the "total time" might be 20 minutes if there are other orders ahead of it.

Project Planning: Comp15

For the first project in COMP15 you will write a program to simulate the order processing system for an on-line vendor. The diagram above shows the main components and operation of the flow of data. There is a lot going on.

How do you design and plan a project like this?

In COMP11, you drew call trees to plan projects. A call tree* displays the functions in your program and the connections between those functions.

Functions act on data. You pick a data structure to represent the things in your program and then you decide on functions to do things with that data.

This technique of [a] *data structure*, [b] *functions*, and [c] *call tree* works well for small to medium projects. It does not work as well for larger projects. There are too many parts in the diagram above to model with a single data structure and a tree of functions.

``*How do I design larger projects?*´´ you ask.

For bigger projects, like this order processing system, you need a more powerful approach, You need to think in terms of *objects*. Objects are instances of *classes*. This handout explains and shows how.

*A call tree is not technically a tree in the CS sense. Because multiple functions can call common sub-functions, the diagram is more accurately called a directed acyclic graph.

Seating Chart: The Old Days

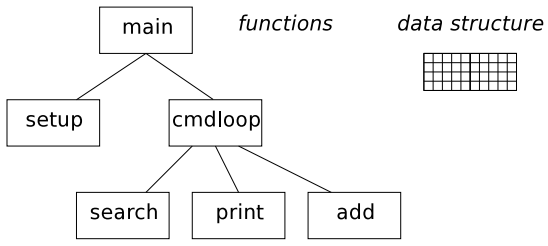
Imagine writing a program to manage seat reservations for a concert hall. How do you design this program using the data structure, functions, and call tree?

[a] Select a data structure. A 2-D array is a good choice. Each element in the array represents one seat. The element contains a value representing the person reserving that seat.

[b] Decide on functions: actions the program performs on the seating chart. One action is to print the chart. Another action is to search the chart. Another action is to add a reservation... Each function is passed an array as an argument and acts on that array.

[c] Organize those actions into a tree, factor out common actions as sub functions, split big actions into separate boxes.

Then write code to see if your design works.



Software Grows Up

When a child is very young, others have to do things for the infant. Initially, a parent *feeds* an infant, *dresses* an infant, and *puts to bed* an infant.

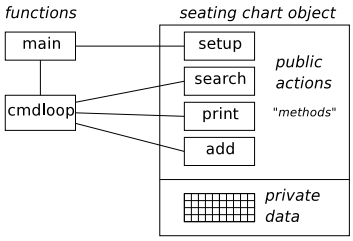
Later, as a child develops skills of its own, the verbs change. Then, we say the child *eats*, the child *gets dressed*, the child *goes to bed*. As a person grows up, he or she develops internal capabilities to act without needing help from outside agents.

Data Structures Grow Up

When you first began using data structures (like arrays, structs, linked lists, trees, etc) you had to write functions to do things for the data structure. In those days, a function *prints* an array, a function *searches* an array, a function *adds data* to an array. Each function is passed the data, and the function acts on the data. This is like a parent doing things for a young child.

One major idea of *object-oriented programming* is that the data structure has skills of its own and does not require an outside function to do things for it. For example, in an object-oriented design, the seating chart array is able to print its contents. The seating chart is able to add reservations, and the seating chart is able to search its contents for a name.

Programming with classes looks like:



In code:

Old way:

```
string chart[ROWS][COLS];
setup(chart);
print(chart);
```

New way:

```
ConcertHall chart;
chart.setup();  // move to constructor
chart.print();
```

How do we plan/design with classes?

What Do You Store? What Can You Do?

When planning a project using classes, we no longer ask what are the data structures and what functions do we apply to them. Instead, we ask "*what are the objects*". That is, we ask what are the players in the program, what are the things that make up the system we are simulating. For each of those things, each of those objects, we ask:

- "*What does the object store?*"
- "*How do I initialize the object?*"
- "*What can the object do for itself?*"
- "*How do I de-allocate the storage?*"

Objects in the Order Processing System

Look at the diagram at the top of this handout. What are the components? There are queues, four of them to be precise. What does a queue store? What can a queue do? We discussed this in class and in lab -- a queue has an add operation, a remove operation, an isEmpty operation. Internally, a queue might have an expand operation, or it might be a linked list, able to grow item by item.

There are orders. What data does an order store? What operations can an order perform? An order can be created, initialized, updated, destroyed when completed. You might define an order class.

What about the workers? These workers manage queues. What does each worker do? What are his/her actions? What does each worker act on?

Notice how this approach differs from the data structure and call-tree approach. Rather than picking a single data structure and a set of functions, view the system as a collection of data structures, each with its own set of functions.

How to Design Big Programs

Seeing the entire system as several connected objects, you can now think in terms such as "*the warehouse fetcher takes an order from the order queue, spends some time to get the item, then adds the order to one of the packing/shipping queues.*"

Do not throw away the data structure and call-tree model. Use that earlier approach to model each subsystem of the bigger system. View the larger problem as inter-related sub-systems. Each subsystem has a data structure and a set of actions. Model each of those subsystems as a class with private data and public actions. Then use these components/objects to build a solution to the big problem.

☞ **Your Planning Document** is a set of class definitions. Each class definition lists and *describes* [a] what the class represents, [b] the data in the class, [c] the functions (public and private) in the class.

In addition to listing class definitions, describe how the classes interact. For example, does one class contain other classes as members? Does a class function take another object as an argument? What kinds of connections can components have? Instead of a data structure and a call tree, design a set of classes and their connections.