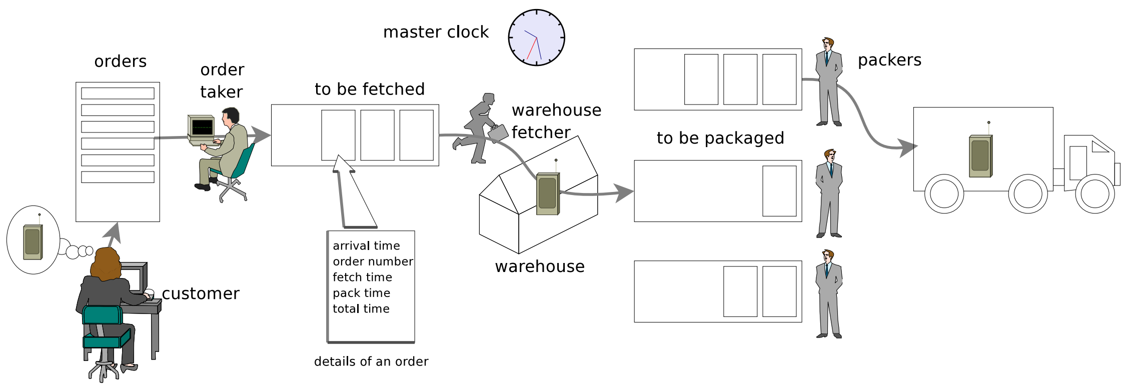
``Ask not what you can do with your data; ask what your data can do for itself´´



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?

The answer is, as always, use the four big ideas:

• abstraction
• modularity
• divide-conquer-glue
• modeling

In short, we look at the big picture above and look at it as a collection of interacting things. Look at the picture now. There is a stream of orders arriving at the site, the orders are stored in a list of orders to be processed, then someone takes orders from that list and gets items from the warehouse and puts them onto belts to be put in boxes and given labels and postage and sent on their way.

What are the things? How can we find common structures and how can we represent those structures in a computer language? The C++ answer is to use *classes*. To recap:

``*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.

Big Idea Four: Modeling

Comp11 used a lot of the first three big ideas. This project is large enough for us to focus on big idea four: modeling. In the previous section, we identified the main components of this system.

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 are you trying to represent?*"
• "*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 above. 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, we view the system as a collection of objects, each with its own internal data and its own set of functions.

Gee, is there a name for this approach?
Yes: Abstraction, Data Abstraction[*]

Tell Me More about this *Data Abstraction*

Right! Here's an example: we need to represent queues of orders to be fetched and queues of orders to be packed and shipped. We abstract the details of how these are stored and moved within the company and build a class that has the right functions.

This abstraction allows us to build a program that has order queues rather than focus on the details of moving specific structs into arrays or to linked lists or things you will learn about later in the course.

To summarize: data abstraction allows us to program as if the language has order queues as basic types rather than having to think in terms of structs and ints and strings, etc. We now think and program in terms of the problem not in terms imposed by the language.

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.*"

And that sentence can correspond closely to code we can write by using our abstract types.

Do not throw away the data structure and call-tree model. Use that earlier approach to model each subsystem of the bigger system. To put this in terms of the big ideas:

Divide View the larger problem as inter-related sub-systems.

Conquer 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.

Glue Then use these components/objects to build a solution to the big problem.

[*] cue (queue?) the James Bond music here

☞ **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.

What Did We Do Here?

This document helps you plan for project1. But it does much more. It shows you an approach to planning for *any project*.

In particular:

[a] Sketch a diagram of the system

[b] Identify the major components

[c] Decide which ones you want to represent as classes

[d] Look for similar components,
Ask if one class can represent all of them

[e] Focus on what each thing stores/does

[f] Looked at how they interact

[g] Write a code outline that uses that the new vocabulary your object provide