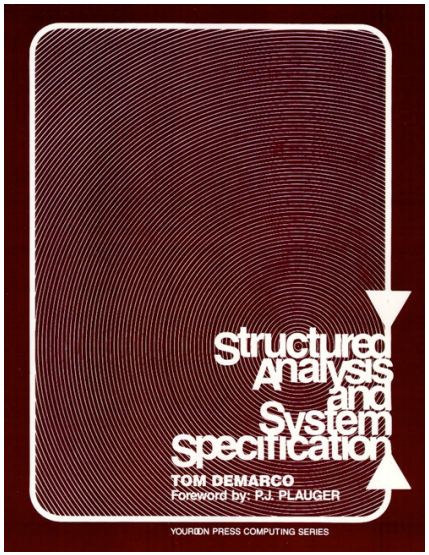


?

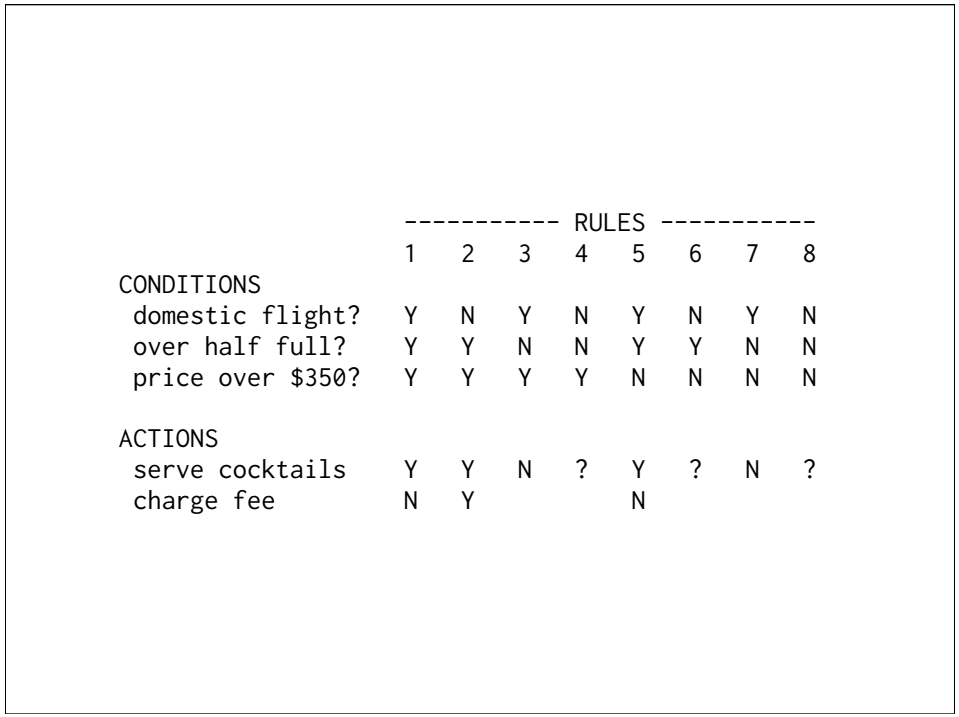
The source of the tools we have talked about this week.
The source of your reading assignment and homework.

```
4.3 Match buy and sell orders

for each Buy Order in Accumulated Orders Stack
   a. add Time of Day to the order
   b. enter into Buy Book by Security Name,
      Bid Price, and Time of Day

for each Sell Order in Sell Book
   a. find first Buy Order with matching Security Name
   b. if found
      1. if Quantity Bid equals Quantity Asked and
            Bid Price is within 1/8 of Asked Price,
         then combine Buy Order and Sell Order
              into a Trade
```

This is a Structured English description for a hypothetical stock trading system.

Pseudocode is nice because it does not commit to the details of a programming language.

But at this level it looks a lot like code.
If we can find a suitable scripting language, that might be preferred
— except for talking to customers.



	RULES							
	1	2	3	4	5	6	7	8
CONDITIONS								
domestic flight?	Y	N	Y	N	Y	N	Y	N
over half full?	Y	Y	N	N	Y	Y	N	N
price over $350?	Y	Y	Y	Y	N	N	N	N
ACTIONS								
serve cocktails	Y	Y	N	?	Y	?	N	?
charge fee	N	Y			N			

When the logic is a tangle of if-then decisions over several factors, pseudocode becomes clunky.

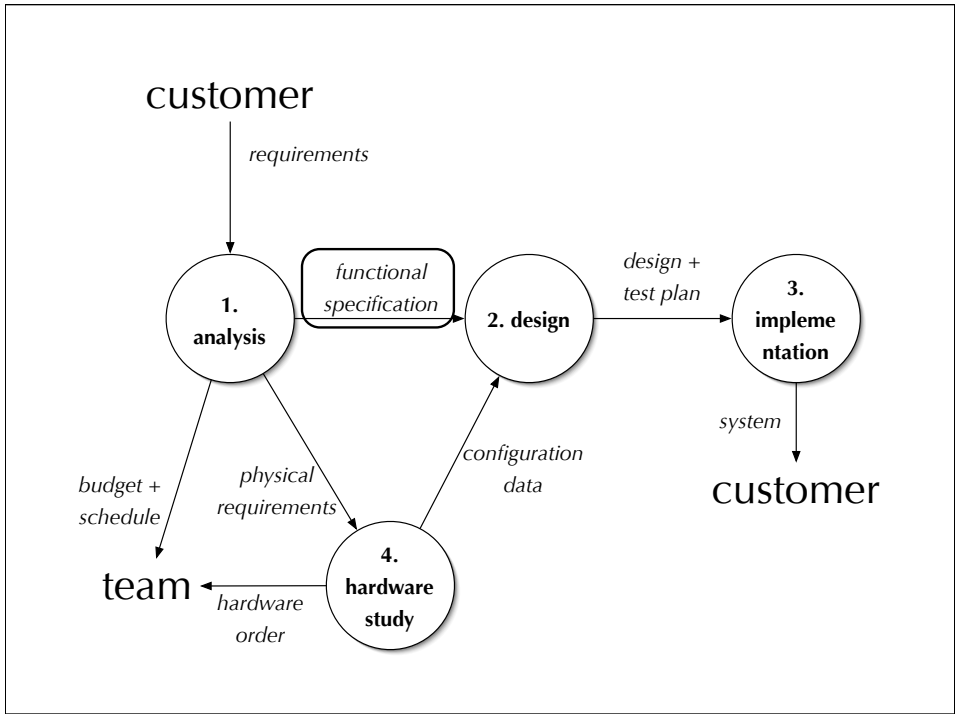
A **decision table** can capture a lot of knowledge more clearly and concisely.
It also exposes holes in our knowledge quickly.

In the 1970s and 1980s, a popular approach to expert system was if-then rules of the sort captured in a decision table. Non-programmers could begin to codify domain knowledge with a programmer's help. We could even build expert system shells -- little programming environments -- for non-programmers to use.

Cub Scout Subsidy Policy

rank	year	subsidy
	1	25
none	2	35
	3	50
	1	35
wolf	2	45
	3	60
	1	40
bear	2	50
	3	65
	1	90
lion	2	55
	3	70

A **decision tree** is a decision table drawn graphically.
Some customers prefer to see pictures over a logic table.



The primary product of analysis is a functional specification.
Why?

Failing to write a spec is
the
single biggest
unnecessary risk
you take in a
software project.

A quote from Joel Spolsky:
http://www.joelonsoftware.com/articles/fog0000000036.html

What is the consequence of not writing a spec?

"... on any non-trivial project
(more than about 1 week of coding
or more than 1 programmer),
if you don't have a spec,
you will *always* spend more time
and create lower quality code."

How can this be?

The spec serves at least three roles...

The act of writing a spec
forces you
to design your program.

The claim: Programmers are inclined to dive in.
Someone needs to think about design before coding begins.
Writing a spec ensures that will happen.

I don't believe that about programmers.
Good programmers love to design their code, at all levels.

More important to me:

A spec saves time communicating.

You only have to communicate what the program is supposed to do once.

A *functional spec*
describes
what a system does
from the user's
perspective.

It doesn't care how anything is implemented.
It talks about features.

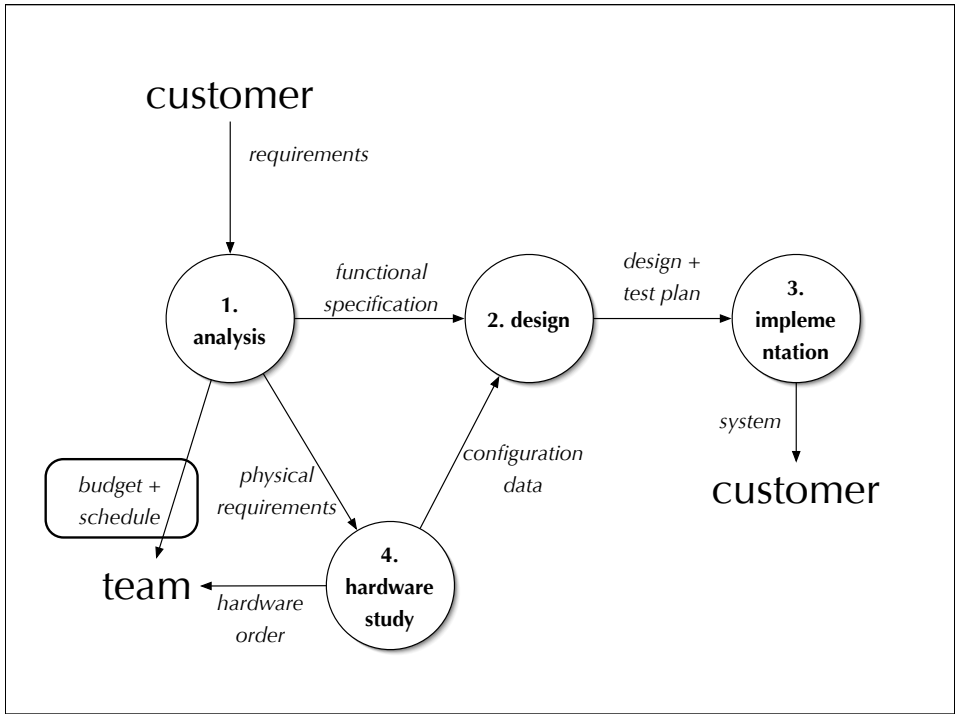
It does specify input screens, menus, dialogs, etc.
They are defined in the data dictionary.

A *technical spec*
describes
how a system
does what it does.

It specifies choice of programming languages and tools, data structures, relational database models, specific algorithms, etc.

A spec makes
it possible to
schedule.

In nearly any business, you have to have an idea of how long development will take. Developing a product costs money.



The primary product of analysis is a functional specification.
Why?

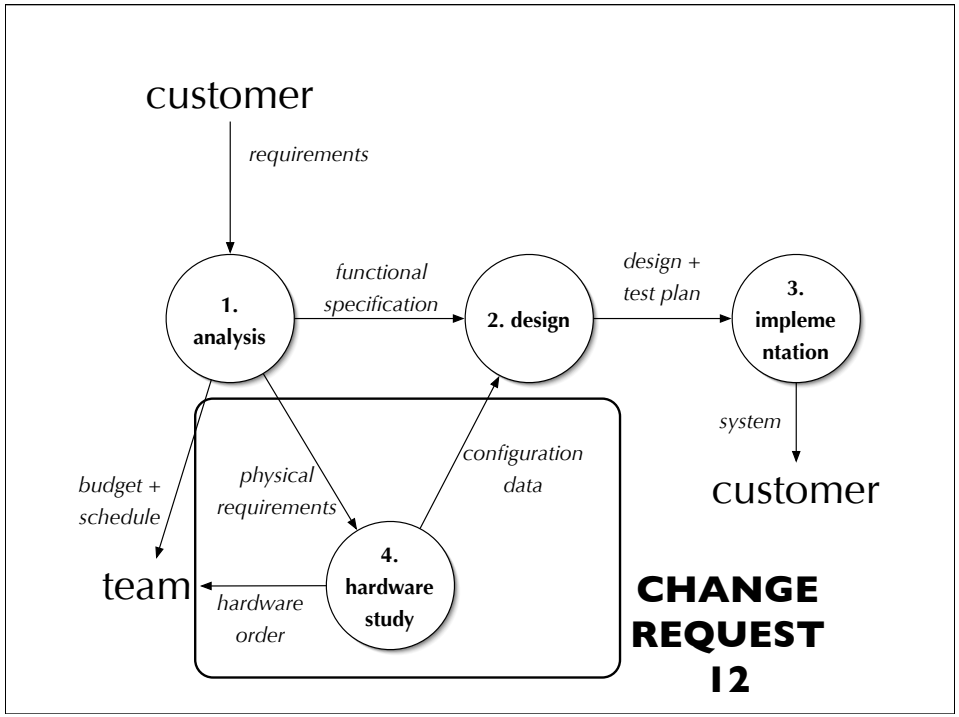
A spec makes
it easier to
change
requirements.

Interesting!

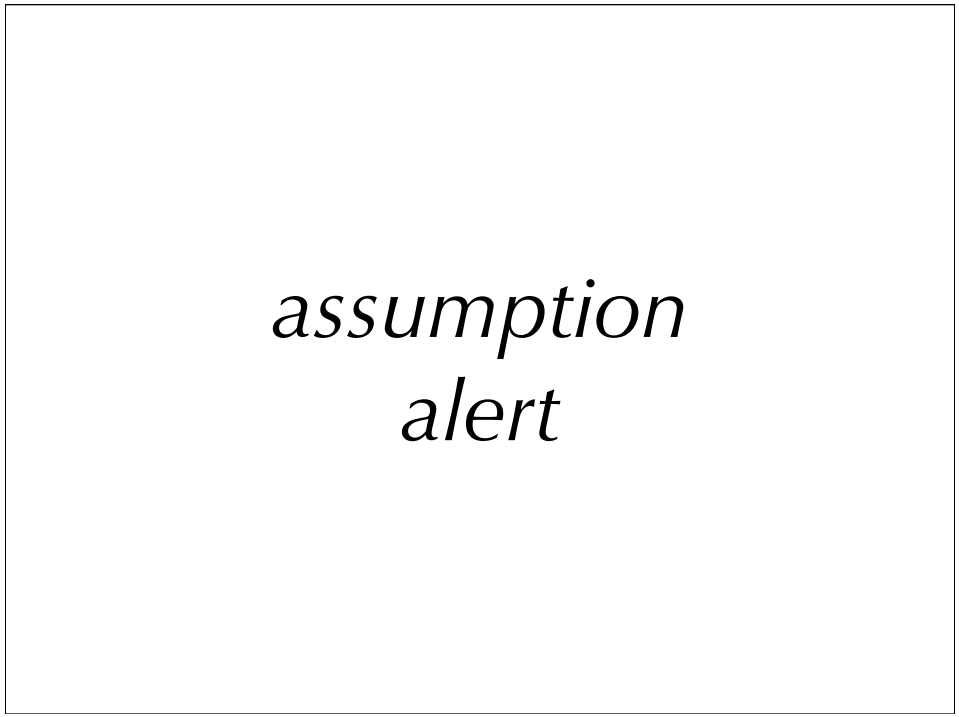
Most everyone recognizes that, in most systems,
the requirements will change over time.
We will learn more.
Users' needs and expectations will change.

Some conclude from this that there is no reason to understand requirements at all.
This is wrong!

Documenting requirements clearly lets you keep track of changes and
keep everyone — clients, analysts, developers, testers — on the same page.
(See "communication" above".)



The primary product of analysis is a functional specification.
Why?



Remember from last time:
Doubt what you know.
Question assumptions.

What does this assume??

Changing
a human-language spec
is easier than
changing
a computer program.

In what ways could this be true?
In what ways could this be false?

Whatever else is true, this is essential.
The spec is a living document.
It must change as the environment changes.
All parties must stay abreast of the changes.

(This is another direct quote from Spolsky.)

The spec defines
the functional
requirements.

This can be an informal document.
Your reading for next time includes a tongue-in-cheek example from Spolsky.

It can also be...

all data flow diagrams
+
the data dictionary
+
all process definitions

... the products of development.

But there will be work to do to “package” the spec...

call attention
to *key interfaces*

add *supplemental*
material for readers

The key is: give the reader enough **context** to understand the DFDs, DD, and processes.

Fill in details.

In the process of collecting requirements, we usually run into questions.
We try to answer them when we can. When we can't, we mark the spot.

The final spec should have all details filled in
— even if we know the spec will change.

On to the project...

Questions?
Ideas you would like a team to work on?