http://www.dilbert.com/strips/comic/2009-09-21/

A missed opportunity... I didn't read Monday's paper before our last session!

You and I are contractors who just won a bid to design a custom video rental vending machine for customers of a local grocery store. Arnold owns the store and, like many software consumers, eschews standard solutions. He wants his own custom design. He is also, however, a cheapskate.

Arnold tells us he wants a simple machine. All he wants is a machine that rents videos for $1.00 each. That's all. He expects us to be able to put this little machine together quickly and for little cost. We decide that the machine will consist of:

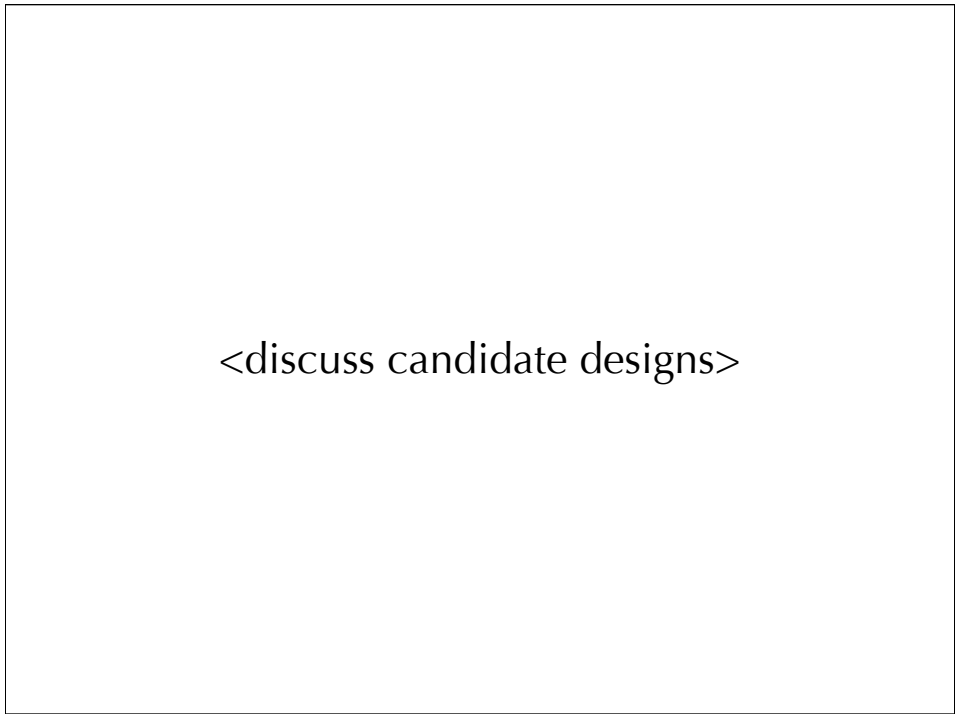
- a currency slot,
- a drop slot for videos and change, and
- three buttons, one for each video in stock.

The set-up for our exercise...

Design the program that runs the video rental machine.

Work in groups of 2 or 3. Your design should list the program's components, what each does, and how they interact.

Choose your favorite language and style (procedural, OO, functional).



Are there any open questions?

Which design is best?

How can we know?
What are the criteria?

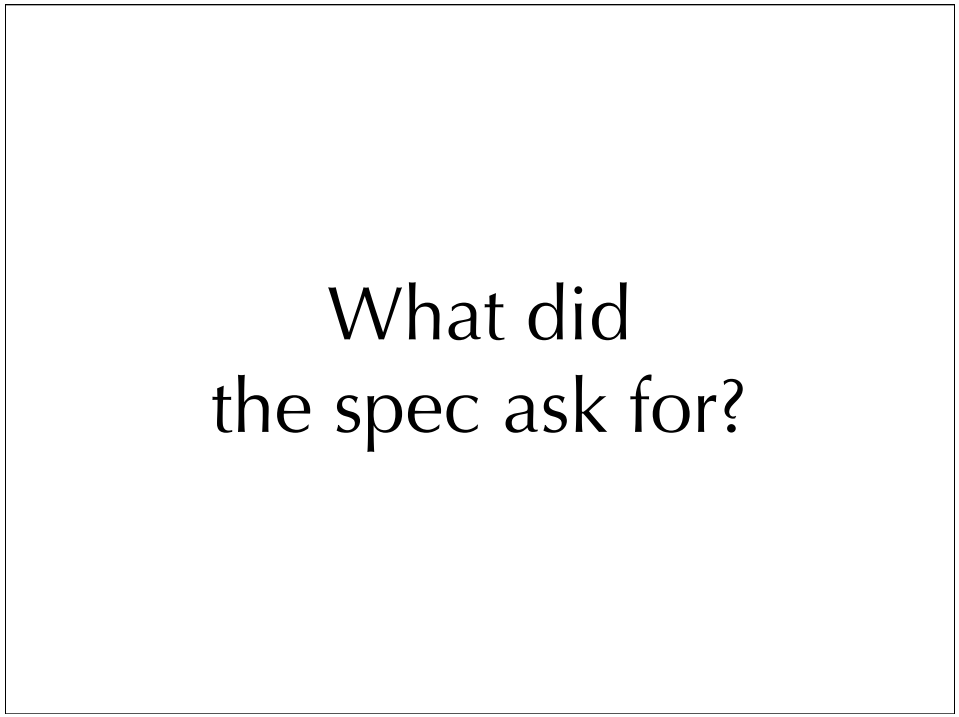
correctness
simplicity

These are important criteria for almost any problem.

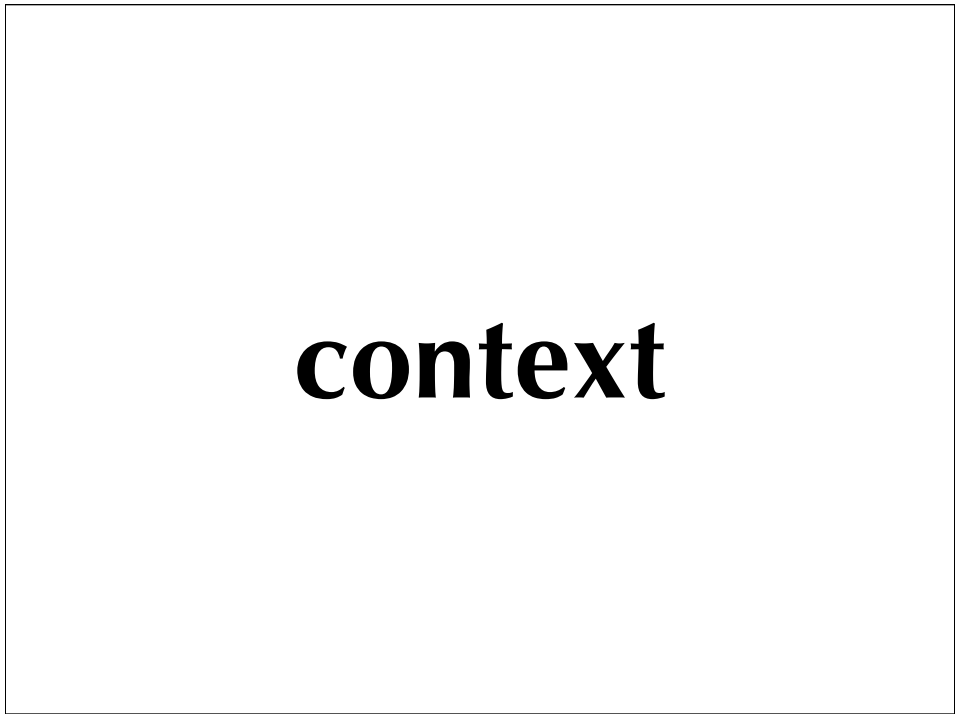
Why is simple good? Easier to understand. Easier to implement, and so cheaper to client?

flexibility
modifiability

Are these important criteria for almost any problem?
If not, why not?



This relates to correctness and simplicity.
But it is also the first step to a more general criterion...



Designs must be considered and evaluated in context.
This is also true of the programs that implement them.

Designs are not good or bad
so much as better or worse.

... in a given context.

When the context changes, the evaluation of quality changes.
(Perhaps we should think in terms of suitability or fitness.)

Keep this in mind whenever you encounter any program, design, or system.

Good design
comes from experience.

Experience
comes from bad design.

Reading and studying designs helps.
Doing design is essential.

Unlike analysis, we have **techniques** for creating designs and **heuristics** for evaluating their quality.

Designing any system requires:

- naming the key components in the system
- identifying the main responsibilities of each component
- identifying the main communication paths among the components

This is true of all design.

In structured design, our components take the form of procedures or modules.
In object-oriented design, our components take the form of objects.
In functional design, our components take the form of pure functions (no state).

high-level design

architecture

Components and their interactions are design at the high level.
They determine the shape of the system and the interaction patterns.
They do not determine the implementation of the system.

low-level design

As we move away from the top-level shape of the system, we approach implementation. This is design, too: selecting algorithms and data structures, parameter-passing modes, databases, etc.

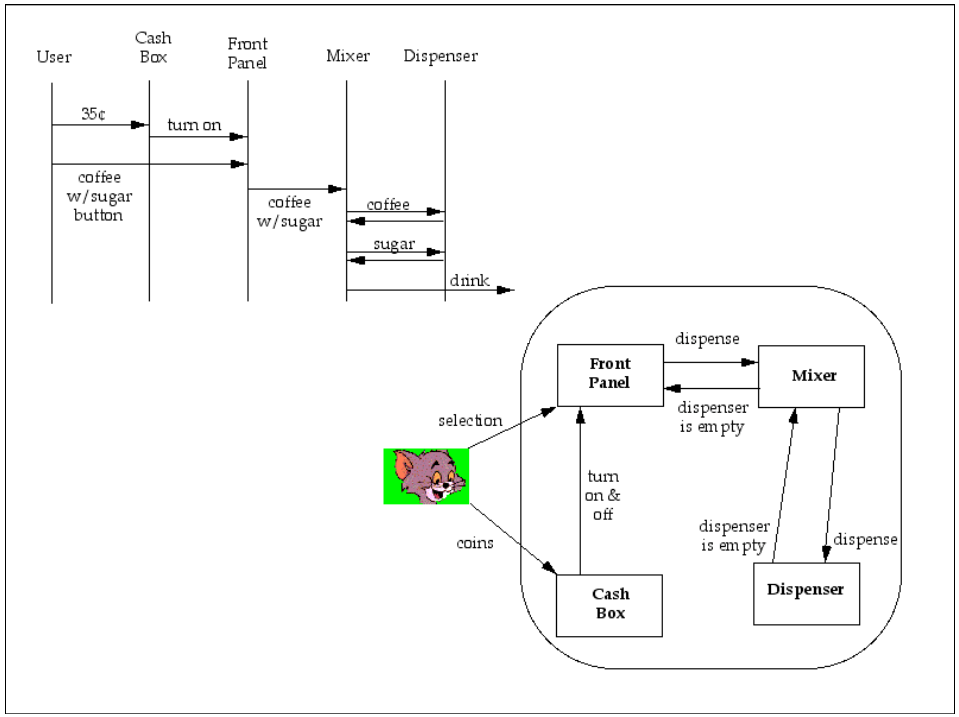
Can we take design all the way down to the code level, say, the choice between an `for` statement and a `while` statement? Yes, indeed. Those are design decision, too — local and perhaps easier to change, but design.

We will talk about software architecture next time and lower-level design after that.

How can we communicate our design?

We have to communicate design to our teammates, our bosses, our clients, their accrediting agencies — not to mention programmers who will maintain the code in months and years.

Text descriptions are almost always value. They allow us to explain things and discuss nuances.



OO design: object diagrams, interaction diagrams

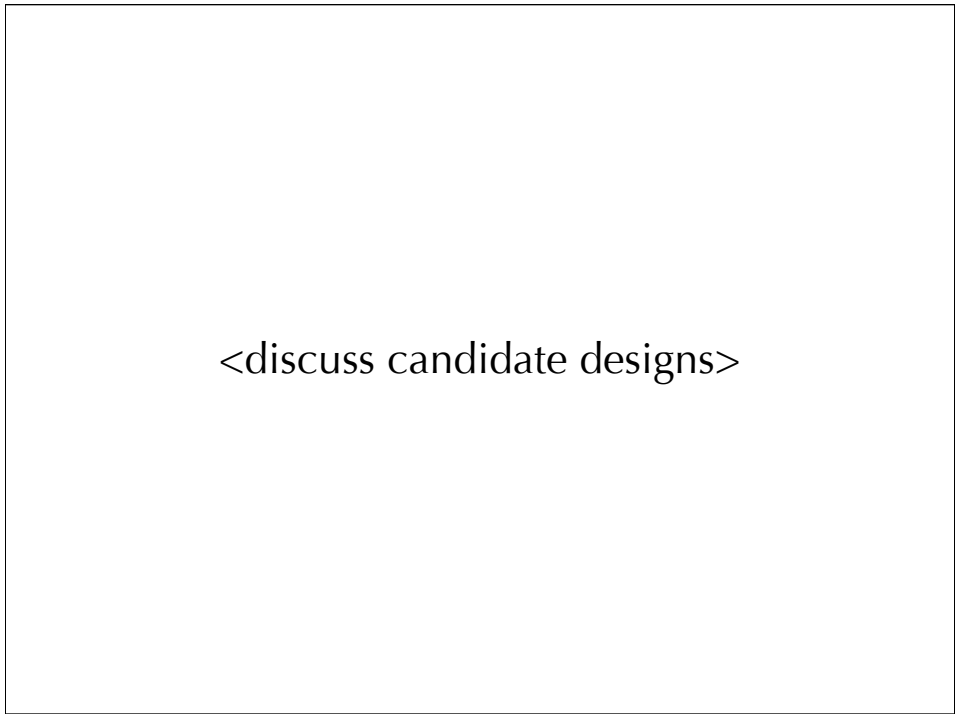
The Return of Arnold

After five machines are installed and in operation for a while, Arnold comes along and says, "I would like to add Blue-Ray disks, at a price of $2.00 each, to every machine. Change the software."

He gives us a machine with three more buttons.

How do you change your software design?

Work with your original group of 2 or 3.



Who knew the price of the DVDs in your original design?
Who knows the price now?

What was **simplest** before can become **incorrect** in a new context!

procedural decomposition

stepwise refinement

modules and submodules

Structured design is what most students learn in CS1-2, especially in an Ada or C formulation. This approach follows directly from the work of DeMarco and Yourdon on structured analysis and design in the 1970s. It has been modernized over time to take into account concerns more common now, such as concurrency, but the techniques are fundamentally the same.

Implement the inner modules with simple algorithms that call submodules and combine their answers.

Implement leaf modules with simple algorithms using base data types and language constructs.

We strive for simplicity in all modules, and decompose when a simple, direct implementation is impossible.

Change is unpredictable, but we can try to predict change.

The key to design systems in advance.

Arnold changed the system's requirements after we implemented our system. That's not unusual. In fact, it's the way things usually go. Tongue-in-cheek aphorism: "Clients don't know what they want until you don't give it to them." Or until they see what you do give them. And their business needs change, too, because their clients and environment change.

By considering different scenarios, we guide the evolution of a design that is more flexible. Most designers find it easier to grow a design than to create a complete design from scratch. This is true whether we use a traditional approach or an agile one.

Many agile proponents encourage **not** trying to predict the future. Build the system and let the future happen.