What exactly is software design?

There’s a surprising amount of debate about this

We do have a bunch of techniques that we use often

Algorithms to achieve some goal in an efficient way

Separation of interface from implementation

Design “patterns” that codify common practices

Programming language “features” that codify common practices
So far, we’ve designed “small” things

Beautiful, mathematically precise definitions, e.g., a tree is:
A “comparable” value
A left tree of values “less than” this value
A right tree of values “greater than” this value
Or:
An empty tree

```java
public class Tree<T extends Comparable<T>> implements ITree<T> {
    private final ITree<T> left, right;
    private final T value;

    // external tree users: don't use this; insert to an empty tree instead
    Tree(T value, ITree<T> left, ITree<T> right) {
        this.left = left;
        this.right = right;
        this.value = value;
    }
}
```
And we’ve talked about “small” features

<table>
<thead>
<tr>
<th></th>
<th>Really Old School</th>
<th>Old School</th>
<th>New School</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error handling</td>
<td>return null</td>
<td>throw Exception</td>
<td>return Optional</td>
</tr>
<tr>
<td>State management</td>
<td>global variables</td>
<td>mutating objects, o-o dispatch</td>
<td>immutable state, functional programming</td>
</tr>
</tbody>
</table>
| Data structures  | reimplement every time | generic implementations | ???
But what about designing “big” things?

Example: “Build me a web calendar application”

Q: How do you approach a “big” problem?
A: Break it down into smaller problems!

Today, we’ll advocate a “top-down” approach
Bottom-up with a web calendar
Bottom-up with a web calendar

First, I need an efficient, scalable data store
Wander off into the world of Hadoop vs. Cassandra vs. Apache Spark vs. ...
Bottom-up with a web calendar

First, I need an efficient, scalable data store
Wander off into the world of Hadoop vs. Cassandra vs. Apache Spark vs. ...

Oh, and don’t forget sophisticated user login / permission management
Wander off into the world of two-factor authentication, federated identity systems, access control rules on calendar entries, ...
Bottom-up with a web calendar

First, I need an efficient, scalable data store
Wander off into the world of Hadoop vs. Cassandra vs. Apache Spark vs. ... 

Oh, and don’t forget sophisticated user login / permission management
Wander off into the world of two-factor authentication, federated identity systems, access control rules on calendar entries, ...

Oh, and I guess we need a web front-end
More wandering...
Bottom-up with a web calendar

First, I need an efficient, scalable data store
Wander off into the world of Hadoop vs. Cassandra vs. Apache Spark vs. ...

Oh, and don’t forget sophisticated user login / permission management
Wander off into the world of two-factor authentication, federated identity systems, access control rules on calendar entries, ...

Oh, and I guess we need a web front-end
More wandering...

Oh, and how about mobile apps
Android? iOS? Other platforms?
Bottom-up with a web calendar

First, I need an efficient, scalable data store
Wander off into the world of Hadoop vs. Cassandra vs. Apache Spark vs. ...

Oh, and don’t forget sophisticated user login / permission management
Wander off into the world of two-factor authentication, federated identity systems, access control rules on calendar entries, ...

Oh, and I guess we need a web front-end
More wandering...

Oh, and how about mobile apps
Android? iOS? Other platforms?
Top-down with a web calendar

Mock it up with one user, one platform, one use-case

Does it actually serve their needs?
What are the required features?

“Plan to throw one away; you will, anyhow.”
Modularity is your friend
Modularity is your friend

Build your code with a “data store”, not a specific database design
Make it easy to remove one back-end and add another one later
Modularity is your friend

Build your code with a “data store”, not a specific database design
Make it easy to remove one back-end and add another one later

Build your user-login in a general-purpose way
Start off with something dumb like usernames & passwords, plan to replace later
Modularity is your friend

Build your code with a “data store”, not a specific database design
Make it easy to remove one back-end and add another one later

Build your user-login in a general-purpose way
Start off with something dumb like usernames & passwords, plan to replace later

Iterate quickly, discover what your users need and change course
Versus building and building for years, only to be stuck with the wrong design
Today’s exercise: a classic videogame

ASTEROIDS™
Asteroids: the smash success of its day
Atari Asteroids: released in 1979

Sold ~70,000 units (= $150 million then, or $489 million in today’s dollars)
Inspired by “Spacewar!” (DEC PDP-1, 1962)

Custom hardware then, runs in your browser today
http://www.freeasteroids.org/welcome/
Asteroids: top-down design

What major subsystems do we need?
Screen rendering
User input
Sound effects
Coin sensing (and free-play mode toggle)
Score tracking
Two-player vs. one-player mode
High score list (input and rendering)
Demo mode

Game state management?
Updating object positions
Collision detection
“AI” for the alien spaceships
Verb-first or noun-first?

Functions are like verbs
Thrust, turn, shoot, collide, split, “hyperspace”

Objects are like nouns
Spaceships, asteroids, aliens, shots

O-O designers start with nouns. F-P designers start with verbs.
Is mutation okay?

It's certainly how they did it in 1979:
8-bit 1MHz CPU, ?? kilobytes of RAM
dedicated sound chip, dedicated vector graphics chip

But they started over from scratch for Asteroids Deluxe (1981)
And again for Space Duel (1982)
And again for game console ports
And the design requirements evolved

Two player “cocktail” models
And the design requirements evolved

Two players (optionally) tied together, colors, weird behaviors, ...
And the design requirements evolved
Even less-capable home gaming consoles (Atari 2600, circa 1981)
How do you engineer for unknowns?

A1: You don’t.
A2: Be clever and you’ll be able to reuse code up to a point.
A3: But don’t be afraid to rearrange, “refactor,” rework as necessary.
A4: Anticipate future needs, but don’t solve future problems.
Consider the collision-detection problem

$n$ objects in the world. Potentially $n^2$ collisions.

Q1: should you use a sophisticated data structure to save time?
There’s a beautiful literature of spatial data structures (quad-trees, k-d trees, etc.)

Q2: how many collisions per second can you evaluate?
On a 1MHz 8-bit micro, with no floating point, doing math is expensive

Typical trick of the day: don’t do this: 
\[ d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \]

Instead, do this: 
\[ d^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2 \]

A1: Compare squared-distance; if small enough then worry about the geometry.
Is mutation okay in 2015?

We’ve got plenty of memory
Plenty of CPU
Plenty of everything

But we’re expected to crank out new features / products to huge audiences
The “big” games have multi-million dollar budgets and hard shipping deadlines

How can functional programming help us?
Functional programming makes your life easier

Fewer synchronization issues when you’re computing in parallel
Just wait until Comp322

Fewer dependency issues when updating old state
What if you’re evaluating collisions (with mutation) while updating locations (also with mutation)?

Easier to save the entire history of the game
Useful for post-facto debugging
Also “instant reply videos”

Easier to build unit tests
Start with a good state, test each transition function
After the midterm

You now know Java. We’re done teaching you the language. You know enough to figure out the libraries, if necessary.

**Time to start building real things!**

Less code that we give you. More code that you write yourself. And you’re going to start doing the *design*, not just filling in the blanks.
Watch Piazza over the weekend

By Sunday, we’ll post room assignments for the midterm
Three rooms! Here (BRK101) and Duncan Hall (DH1042 and 1064).

“Open Internet”, but no programming language tools of any kind.
No two-way communication of any kind.

Exam starts promptly at 9:00am, ends promptly at 9:50am.