Introduction to Software Design

CSCI 5828: Foundations of Software Engineering
Lecture 19 — 10/28/2014

Goals

- Introduce the notion of Software Design
  - Present many different examples of design and design thinking
    - Design Guidelines
    - Design Patterns
    - The use of themes
    - Successful designs
    - Examples

What is Design?

- In software engineering
  - design is typically thought of as “the solution” to a problem defined by a customer or user
  - traditionally, it is the work that generates a solution **AFTER** the problem is understood (to some extent) but **BEFORE** implementation begins
    - (As we will see, successful solutions are called Design Patterns)
- “I hacked up a solution” — typically means the developers started coding before they had a design
  - In these situations, people will say “I needed to code it up once before I understood the problem enough to implement it correctly”
- “I designed a solution” — the developers spent time talking about the characteristics of a solution—the data structures needed, the algorithms, the system components, etc.—reached and agreement and **THEN** started coding

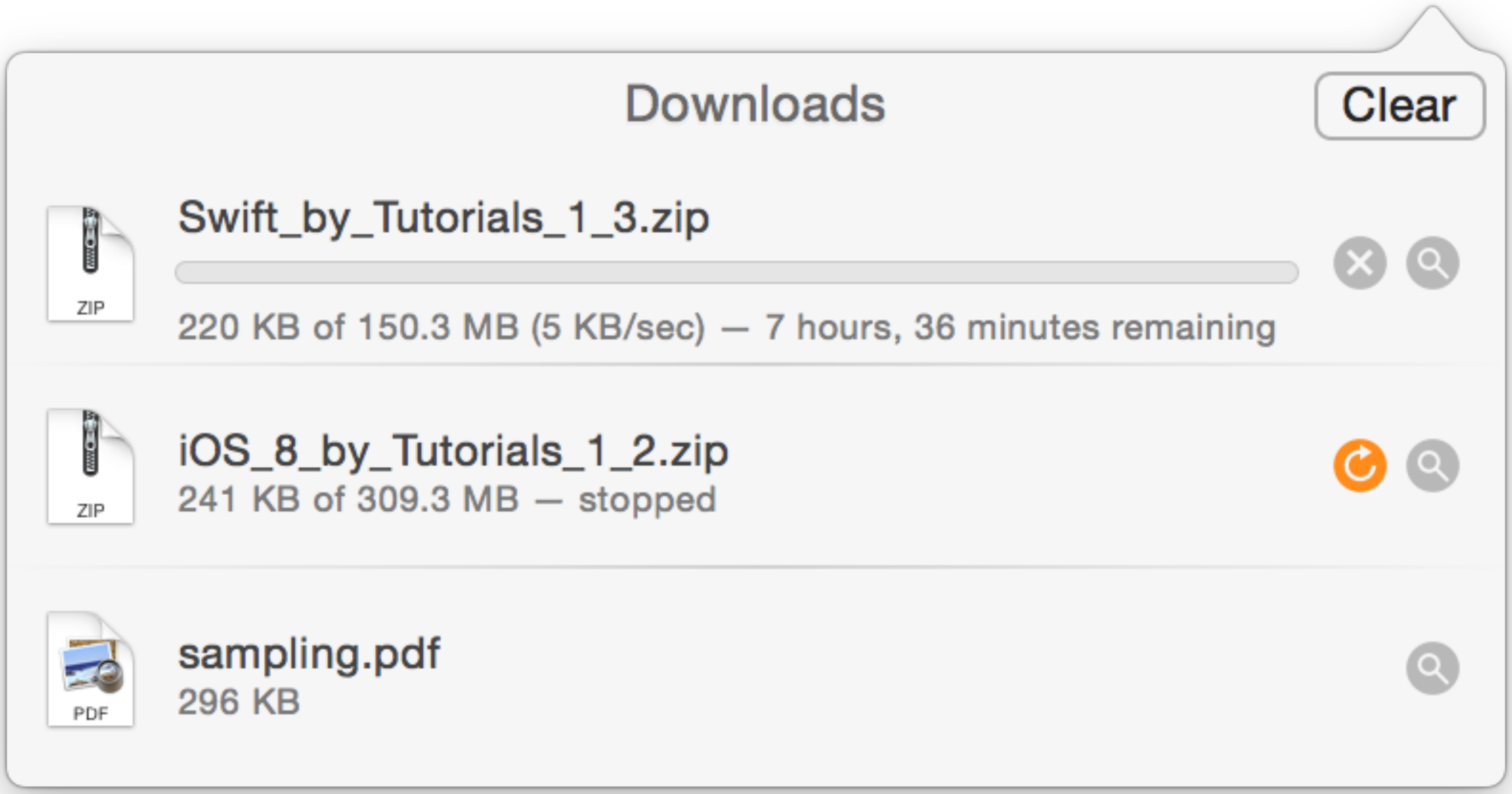
Dictionary Definitions

- design (noun)
  - a plan or drawing produced to show the look and function or workings of a building, garment, or other object before it is built or made
  - the art or action of conceiving of and producing a plan or drawing
  - an arrangement of lines or shapes created to form a pattern or decoration
  - purpose, planning, or intention that exists or is thought to exist behind an action, fact, or material object
- design (verb)
  - decide upon the look and functioning of a thing, typically by making a detailed drawing of it
  - do or plan something with a specific purpose or intention in mind

Design is Ancient

- Humans have been engaged in design in many fields for thousands of years
  - The result?
  - Look around you!
    - Excluding nature (plants, animals, chemicals, etc.), can you point to one object that hasn't been designed?
    - Everything around us was designed by a human at some point
      - In our lecture room, EVERYTHING was designed by humans
        - That means that everything around us SOLVES A PROBLEM!
          - A problem we would have if the object wasn't there
- This is actually quite stunning if you spend time thinking about it!

Design is **NOT** a *feature*



It is also **NOT** a *specific implementation*; it is a *set of ideas/techniques* about HOW to create the implementation of a feature; **the approach**

Design in Other Fields

- **Product Design**
  - How do you create a product that “fits” into its intended niche?
- **Architecture**
  - How do you design buildings so they are functional and serve a purpose?
  - How do you design buildings and the spaces between so they work well with each other
    - i.e. urban planning
  - See for instance “The social life of small urban spaces, a film by William H. Whyte”
    - discussed in Palen’s Social Computing Class
- **Fashion Design**
  - How do you pull materials together so they serve a purpose while they also “make a statement”?
- **Cooking, Music, Film, Art: any creative endeavor requires design!**

Design begets Design Thinking

- At <https://www.vitsoe.com/gb/about/good-design>, Dieter Rams, a famous product designer, reflects on what makes “good design”?
- **Good design is innovative**
  - Taking advantage of new techniques; using existing techniques in unexpected ways
- **Good design make a product useful**
  - Emphasize utility while removing anything that detracts from that
- **Good design is aesthetic**
  - Aesthetics are integral to a product’s usefulness; we use these items every day
- **Good design makes a product understandable**
  - “You don’t have to read the manual”
- **Good design is unobtrusive**
  - “Products fulfilling a purpose are like tools.” “It gets out of your way.”

Dieter Rams, continued

- **Good design is honest**
  - The design does not attempt to fool the user that it can do something that it cannot
- **Good design is long-lasting**
  - It solves the problem so well that it avoids being “fashionable”
- **Good design is thorough down to the last detail**
  - “Nothing must be arbitrary or left to chance. Care and accuracy in the design process show respect towards the user”
- **Good design is environmentally-friendly**
  - “It conserves resources and minimizes physical and visual pollution throughout the life cycle of the product.”
- **Good design is as little as possible**
  - “Less, but better—because [the design] concentrates on the essential aspects [of the problem], and the product is not burdened with non-essentials.”

Thinking About Design is also Ancient

- “A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.”
  - — Antoine de Saint Exupéry
- More quotes on design in general, located here
  - <http://www.designwashere.com/80-inspiring-quotes-about-design/>

Design is Hard

- One of my favorite type of reading is
  - "Developers blogging about design problems"
- Examples
  - Brent Simmons on synching mobile app data via a web service
  - Marco Arment on how to do tilt scrolling on a mobile device
  - ridiculous fish on how he tried to beat grep
    - The article starts "*Old age and treachery will beat youth and skill every time.*"
  - Brian Lovin on the visual design of Paper by Facebook
  - Jesse Squires on adaptive user interfaces in iOS 8
- Please share similar examples that you find on the web or, better yet, that you write yourself!

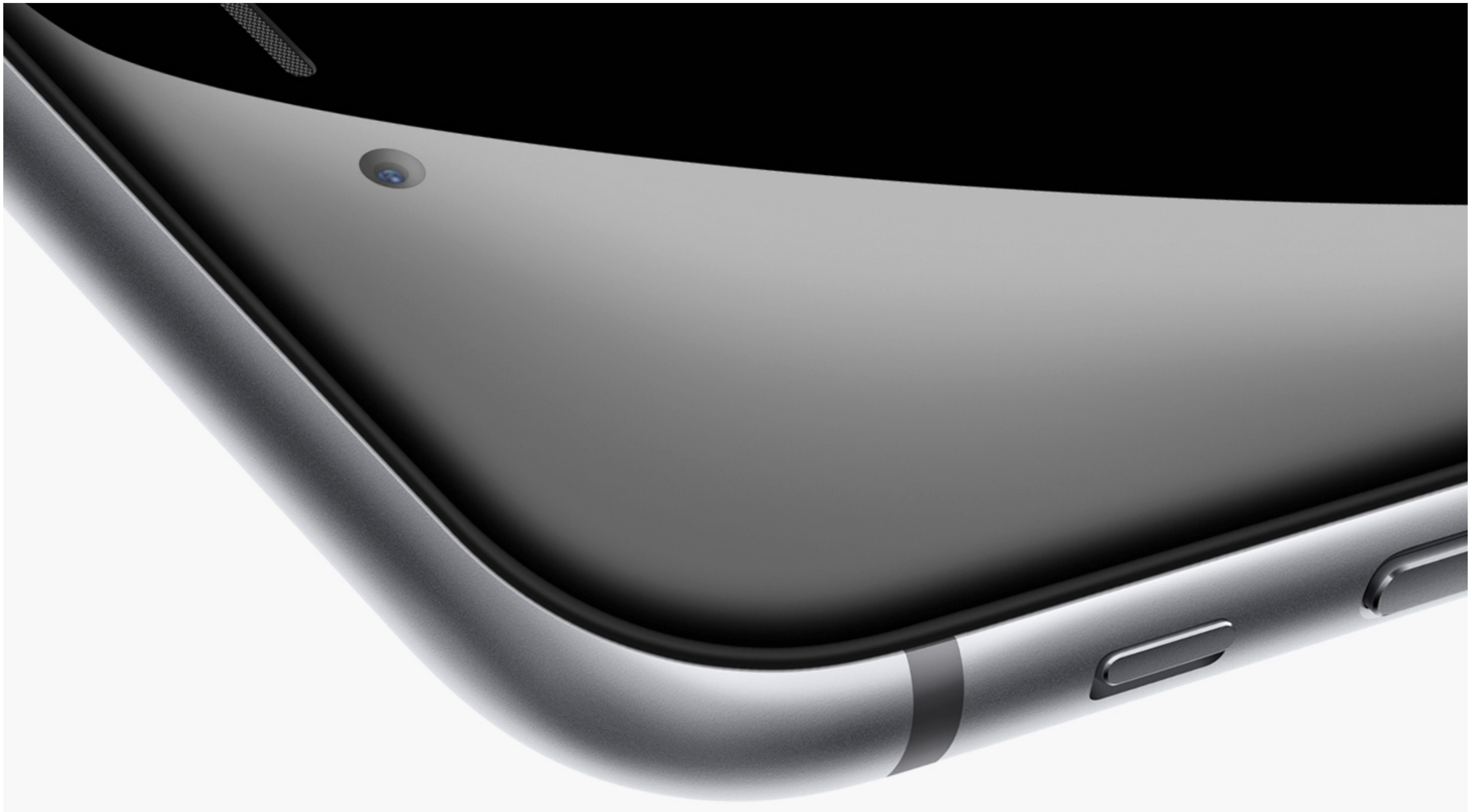
Design is Transformative

- There are a lot of choices one can make in any particular design space
    - Once a good set of choices has been made, it influences everything that comes after

Transformation In Detail

Note: images found here: http://random.andrewwarner.com/what-googles-android-looked-like-before-and-after-the-launch-of-iphone/

Latest Iteration: Takes Initial Trend to Logical End



Note: Image comes from http://www.apple.com/iphone-6/

The Structure of Design

- One interesting thing about design is that it often has a structure that is "tangible"—sometimes physically—but sometimes in just the way it influences our thinking
  - Consider music and the structure of songs
    - Thousands of songs exists that have this basic structure
      - Verse 1; Refrain; Verse 2; Refrain; …
    - Another common structure
      - Intro; Verse 1; Refrain; Verse 2; Bridge; Verse 3; Refrain (repeat til fade)
  - Creativity can then come in the form of playing with that structure
    - "Unusual and interesting songs" often are ones that have rearranged the basic structure, thus playing with our expectations

Structure in Software Design: Design Patterns

- In 1995, a book was published by the “Gang of Four” called Design Patterns
  - It applied the concept of patterns to software design and described 23 of them
    - The authors did not invent these patterns
      - Instead, they included patterns they found in at least 3 “real” software systems.

Cultural Anthropology

- Design Patterns have their intellectual roots in the discipline of cultural anthropology
  - Within a culture, individuals will agree on what is considered good design
    - “Cultures make judgements on good design that transcend individual beliefs”
  - Patterns (structures and relationships that appear over and over again in many different well designed objects) provide an objective basis for judging design

Christopher Alexander (I)

- Design patterns in software design traces its intellectual roots to work performed in the 1970s by an architect named Christopher Alexander
  - His 1979 book called “The Timeless Way of Building” that asks the question “Is quality objective?”
    - in particular, “What makes us know when an architectural design is good? Is there an objective basis for such a judgement?”
  - His answer was “yes” that it was possible to objectively define “high quality” or “beautiful” buildings

Christopher Alexander (II)

- He studied the problem of identifying what makes a good architectural design by observing all sorts of built structures
  - buildings, towns, streets, homes, community centers, etc.
- When he found an example of a high quality design, he would compare that object to other objects of high quality and look for commonalties
  - especially if both objects were used to solve the same type of problem

Christopher Alexander (III)

- By studying high quality structures that solve similar problems, he could discover similarities between the designs and these similarities where what he called patterns
  - “Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”
  - The pattern provides an approach that can be used to achieve a high quality solution to its problem

Four Elements of a Pattern

- Alexander identified four elements to describe a pattern
  - The name of the pattern
  - The purpose of the pattern: what problem it solves
  - How to solve the problem
  - The constraints we have to consider in our solution
- He also felt that multiple patterns applied together can help to solve complex architectural problems

Design Patterns and Software (I)

- Work on design patterns got started when people asked
  - Are there problems in software that occur all the time that can be solved in somewhat the same manner?
  - Was it possible to design software in terms of patterns?
- Many people felt the answer to these questions was “yes” and this initial work influenced the creation of the Design Patterns book by the Gang of Four
  - It catalogued 23 patterns: successful solutions to common problems that occur in software design

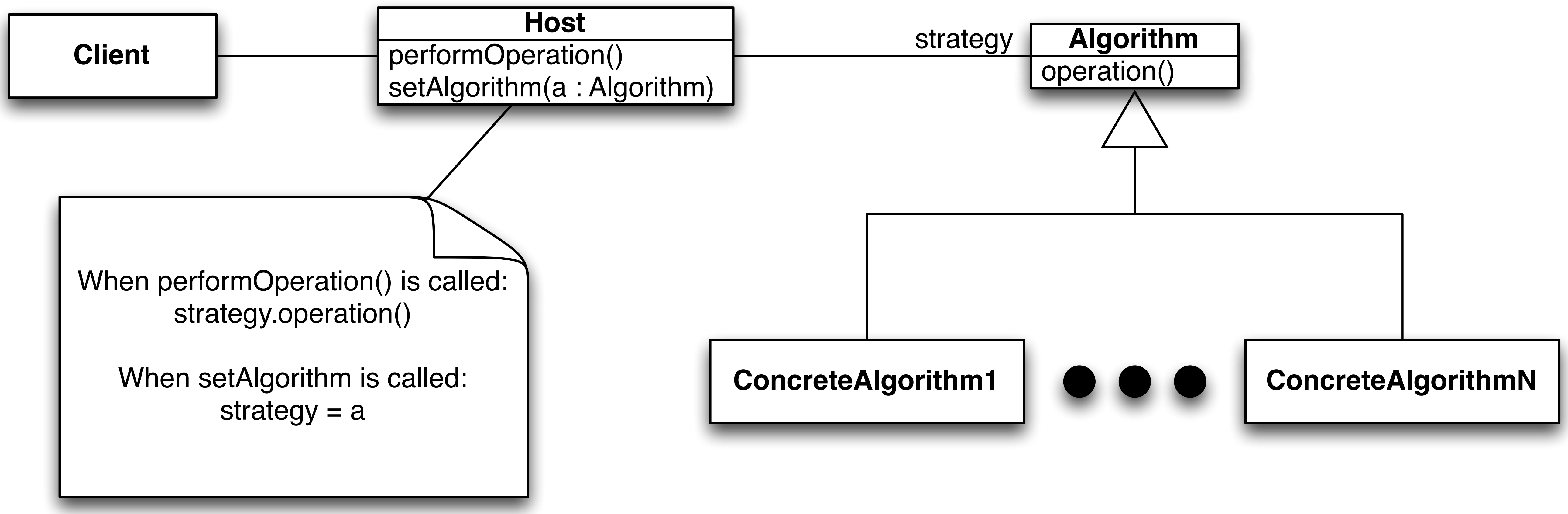
Design Patterns and Software (II)

- Design patterns, then, assert that the quality of software systems can be measured objectively
    - What is present in a good quality design (X's) that is not present in a poor quality design?
    - What is present in a poor quality design (Y's) that is not present in a good quality design?
- We would then want to maximize the X's while minimizing the Y's in our own designs

Key Features of a Pattern

- **Name**
- **Intent**: The purpose of the pattern
- **Problem**: What problem does it solve?
- **Solution**: The approach to take to solve the problem
- **Participants**: The entities involved in the pattern
- **Consequences**: The effect the pattern has on your system
- **Implementation**: Example ways to implement the pattern
- **Structure**: Class Diagram

Design Pattern Example: Strategy



Name: Strategy

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

Why Study/Develop Design Patterns?

- Patterns let us
  - reuse solutions that have worked in the past; why waste time reinventing the wheel?
  - have a shared vocabulary around software design
    - they allow you to tell a fellow software engineer “I used a Strategy pattern here to allow the algorithm used to compute this calculation to be customizable”
      - You don’t have to waste time explaining what you mean since you both know the Strategy pattern

Why Study Design Patterns? (II)

- Design patterns provide you **not with code reuse** but with **experience reuse**
  - Knowing concepts such as abstraction, inheritance and polymorphism will NOT make you a good designer, unless you use those concepts to create flexible designs that are maintainable and that can cope with change
- Design patterns can show you how to apply those concepts to achieve those goals

A Sense of Perspective

- Design Patterns give you a higher-level perspective on
  - the problems that come up in OO A&D work
  - the process of design itself
  - the use of object orientation to solve problems
- You'll be able to think more abstractly and not get bogged down in implementation details too early in the process

The Carpenter Analogy (I)

- An excellent example of what we mean by a “higher-level perspective”: Imagine two carpenters having a conversation
  - They can either say
    - Should I make the joint by cutting down into the wood and then going back up 45 degrees and…
  - or
    - Should we use a dovetail joint or a miter joint?

The Carpenter Analogy (II)

- The latter is at a high-level and enables a richer conversation about the problem at hand
  - The former gets bogged down in the details of cutting the wood such that you don’t know what problem is being solved
- The latter relies on the carpenter’s shared knowledge
  - They know that dovetail joints are higher quality than miter joints but with higher costs
  - Knowing that, they can debate whether the higher quality is needed in the situation they are in

The Carpenter Analogy in Software

- “I have this one object with some important information and these other objects over here need to know when its information changes. These other objects come and go. I’m thinking I should separate out the notification and client registration functionality from the functionality of the object and just let it focus on storing and manipulating its information. Do you agree?”
- vs.
- “I’m thinking of using the Observer pattern. Do you agree?”

More about Design Patterns

- You can learn more about design patterns from the original book
  - <http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/>
  - You will find the examples referenced in this book to be outdated but the patterns themselves are pure gold
- I also found this book that looks to be a terrific resource and a more modern presentation of these ideas
  - <http://sourcemaking.com/design-patterns-book>

Design Themes; Where are these used?

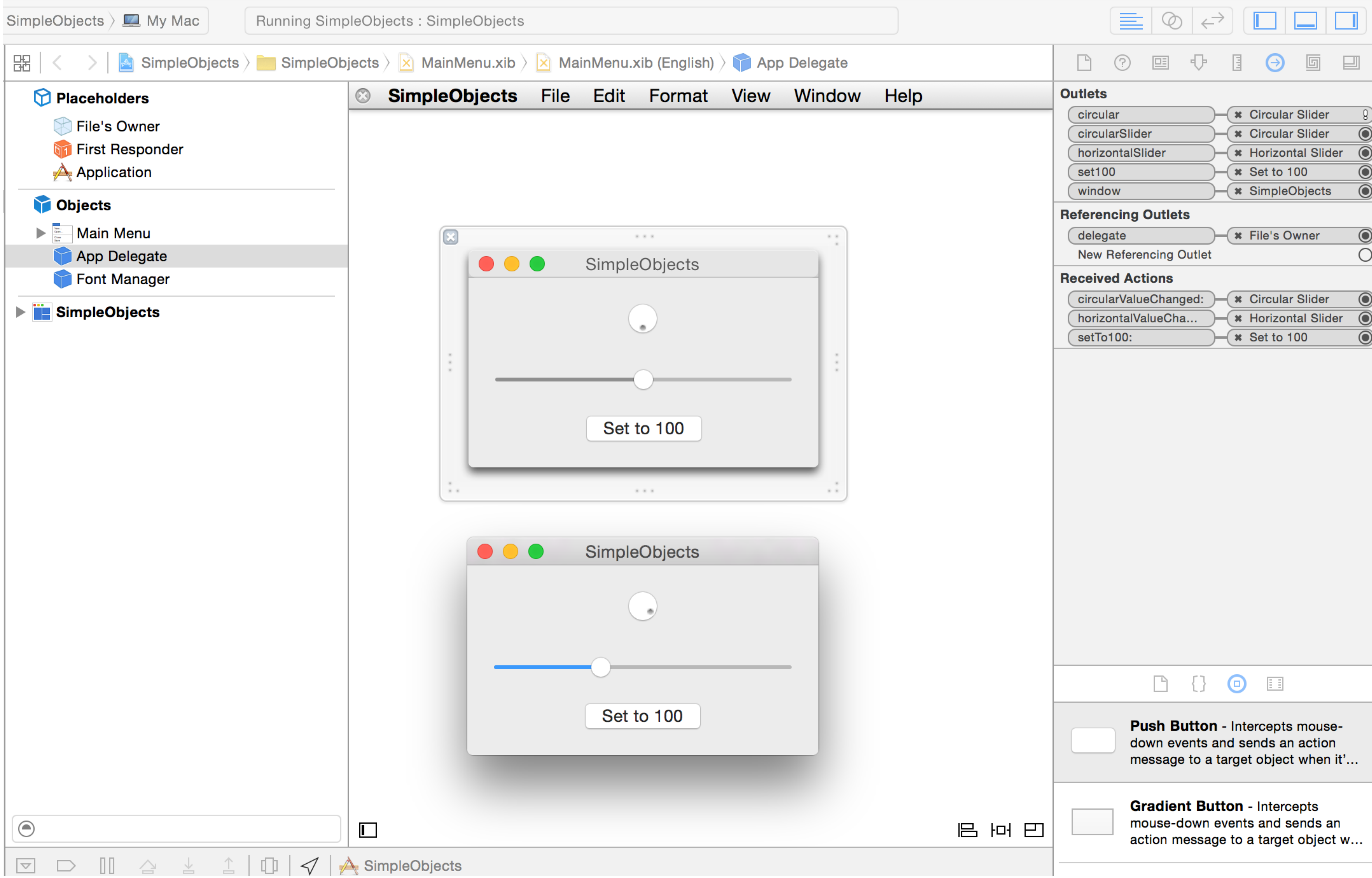
- “Everything is a file”
- “Everything is a resource”
- “Everything is an object”
- All data can be stored in tables with rows and columns
- The presentation details of information should be separated from its structure

(One Set of) Anwsers

- “Everything is a file” — **Unix**
- “Everything is a resource” — **Web**
- “Everything is an object” — **Ruby (and many other programming languages)**
- All data can be stored in tables with rows and columns
  - **Relational Databases**
- The presentation details of information should be separated from its structure
  - **CSS (presentation details) and HTML5 (structure)**

Everything is an Object

- Examples
  - 5.upto(10) { |i| puts i }
    - 5
    - 6
    - 7
    - 8
    - 9
    - 10
  - "Design is Cool!!”.upcase
    - "DESIGN IS COOL!!”
- etc.



Everything is an Object (more advanced)

Unix (I)

- “Everything is a file”
  - One API can be used to read/process
    - files, sockets, devices, and memory
  - One example of the latter
    - `tree <large directory>; tree <large directory>`
  - The first time this command runs, it will take a long time;
    - The second time runs almost instantly. Why?
    - The file system cache; the files are pulled into memory by the operating system. The second time around `tree` is reading from memory although it thinks it is reading from disk

Unix (II)

- “Everything is a file”
  - Another advantage: program input/output expectations
    - Every program can read from standard in
    - Every program can write to standard out
      - Standard In and Standard Out can point to “anything”
        - Memory, Files, Sockets, Devices, etc.
- This lets you do things like
  - `find . -type f -name \*.rb | grep -i "Tweet" | wc -l`
- In English: “How many ruby files in this directory tree have the word “tweet” in their filename?”

Unix (III)

- Even cooler, the commands in a pipe structure run in parallel
  - find . -type f | grep -i "CSCI" | ruby ~/Desktop/DesignIntro/uppercase.rb
- This invokes three programs, “find”, “grep” and a ruby program I wrote
  - In parallel
    - find looks for file names (ignoring directory names)
    - grep looks for file names containing “CSCI” in a case insensitive fashion
    - The ruby program converts all of its input to uppercase

Unix (IV)

- Speaking of Ruby
  - Command chaining in Unix (actually Unix shells) is so powerful that many programming languages optimize the creation of programs that can do this
- By default, ruby's `gets` and `puts` are set-up to read/write standard in/out
- My ruby program looks like this

```
while line = gets
  puts line.chomp.upcase
end
```

- That's all that's needed to get started in this type of programming

Unix (V)

- The ability to combine programs in this way, gives the user a language that allows them to solve problems
- Last night my daughter had a vocabulary exercise that said:
  - `Not vibrant but c_l___e__`
- And she needed to fill in the missing letters
  - we both thought about it and came up with nothing
- so I wrote this “program”
  - grep "^c.l...e..$" /usr/share/dict/words
- In English: “what nine-letter words begin with c and have an l and an e in them in positions 3 and 7?” => 17 choices: “colorless” jumped right out

Unix (VI)

- Likewise, she had the question
  - `Not unknown but f_____`
- `grep “^f.....$" /usr/share/dict/words | wc -l`
  - “How many six letter words start with the letter f?” => 568
- `grep "^f.....$" words | subl3 --`
  - “Show them to me…”
- After scrolling through the words, we found “famous”

Summary

- We introduced the concept of software design and design patterns
  - Design is **NOT** an individual feature or implementation
    - it is an **APPROACH** to *solving a problem*
  - We talked about design in general
    - Design is Ancient => Design is **EVERYWHERE**
    - Design gets to the essentials
    - Design is transformative
    - Design has structure
    - Design is **HARD**
- We talked about design themes and saw examples

Coming Up Next

- Lecture 20: The Design of Design, Part One
- Lecture 21: User Stories, Chapters 12-16