Can Great Programmers Be Taught?

John Ousterhout
Stanford University

PLATFORMLAB

Q: What is the most important idea in Computer Science?

A: Problem decomposition

... no-one teaches it

Elite programmers are >10x more productive

... no-one teaches elite skills

Teaching Great Programmers

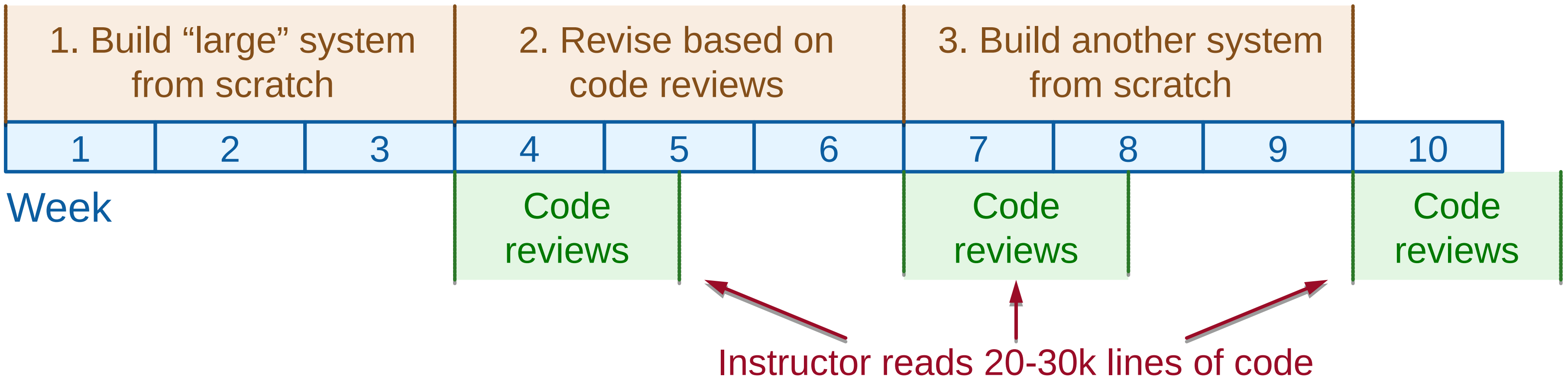
Is it possible?

By whom?

How?

CS 190: Software Design Studio

- **Iterative approach, like English writing class:**
  - Write
  - Get feedback
  - Rewrite
- **Small class: ≤ 20 students**

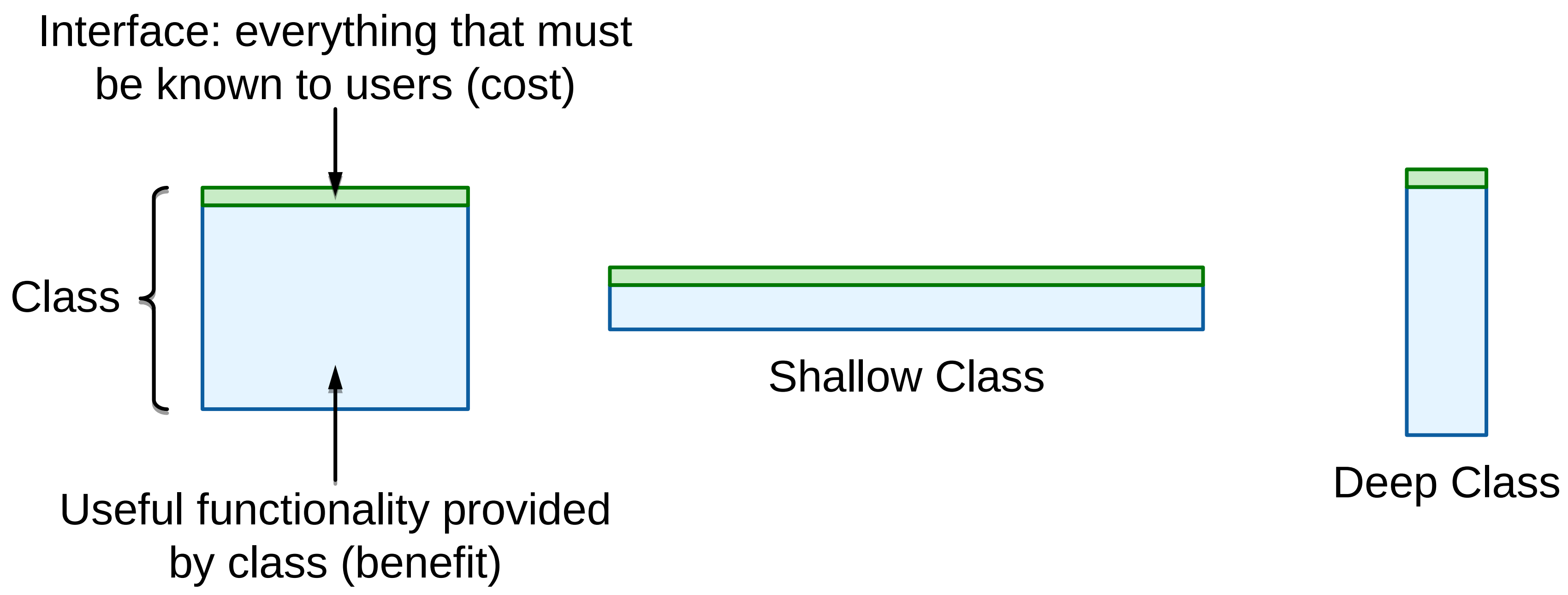


1. Build "large" system from scratch
2. Revise based on code reviews
3. Build another system from scratch
1
2
3
4
5
6
7
8
9
10
Week
Code reviews
Code reviews
Code reviews
Instructor reads 20-30k lines of code

What are the Secrets?

- **A few (somewhat vague) overall concepts:**
  - Working code isn’t enough: must minimize complexity
  - Complexity comes from dependencies and obscurity
  - Strategic vs. tactical programming
  - Classes should be deep
  - General-purpose classes are deeper
  - New layer, new abstraction
  - Comments should describe things that are not obvious from the code
  - Define errors out of existence
  - Pull complexity downwards
- **Red flags**
- **Most constructive in the context of code reviews**

Classes Should be Deep



Interface: everything that must be known to users (cost)
Class
Useful functionality provided by class (benefit)
Shallow Class
Deep Class

Reformulation of classic Parnas paper:
"On the Criteria to be Used in Decomposing Systems into Modules"

Typical Shallow Method

```
private void addNullValueForAttribute(String attribute) {
    data.put(attribute, null);
}
```

Classes Should be Deep, cont'd

- **Common wisdom: "classes and methods should be small"**
- **Result: classitis**
- **Rampant in Java world:**

```
FileInputStream fileStream =
        new FileInputStream(fileName);
BufferedInputStream bufferedStream =
        new BufferedInputStream(fileStream);
ObjectInputStream objectStream =
        new ObjectInputStream(bufferedStream);
```

- **Length isn't the big issue, it's abstraction**

A Deep Interface

- **Unix file I/O:**

```
int open(const char* path, int flags, mode_t permissions);
int close(int fd);
ssize_t read(int fd, void* buffer, size_t count);
ssize_t write(int fd, const void* buffer, size_t count);
off_t lseek(int fd, off_t offset, int referencePosition);
```

- **Hidden below the interface:**
  - On-disk representation, disk block allocation
  - Directory management, path lookup
  - Permission management
  - Disk scheduling
  - Block caching
  - Device independence

Define Errors Out of Existence

- **Exceptions: a huge source of complexity**
- **Common wisdom: detect and throw as many errors as possible**
- **Better approach: define semantics to eliminate exceptions**
- **Example mistakes:**
  - Tcl `unset` command
    (throws exception if variable doesn't exist)
  - Windows: can't delete file if open
  - Java substring range exceptions

Overall goal: minimize the number of places where exceptions must be handled

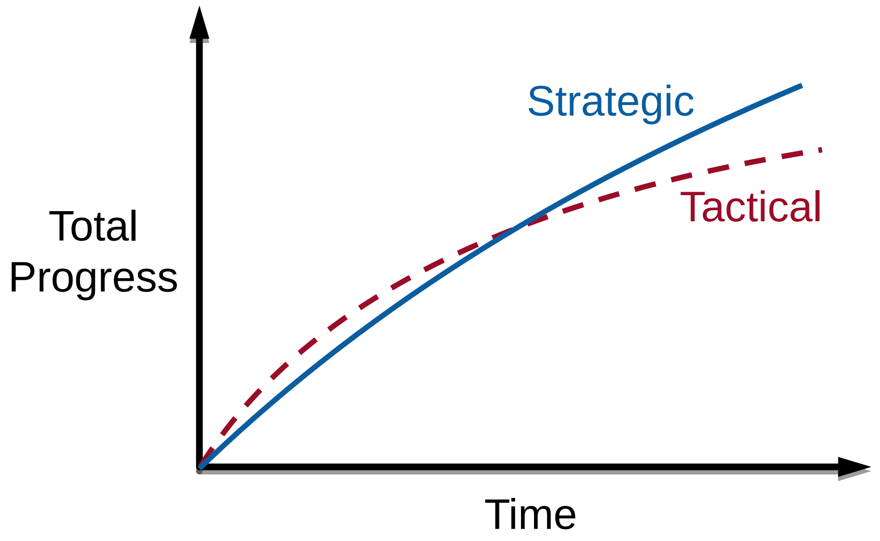
Tactical vs. Strategic Programming

- **Tactical programming**
  - Goal: get next feature/bug fix working ASAP
  - A few shortcuts and kludges are OK?
  - Result: bad design, high complexity
  - Extreme: tactical tornadoes
- **Complexity is incremental**

Working code isn’t enough

Tactical vs Strategic Programming, cont'd

- **Strategic programming**
  - Goal: produce a great design
  - Simplify future development
  - Minimize complexity
  - Must sweat the small stuff
- **Investment mindset**
  - Take extra time today
  - Pays back in the long run



How Much To Invest?

- **Most startups are totally tactical**
  - Pressure to get first products out quickly
    - “We can clean this up later”
  - Code base quickly turns to spaghetti
  - Extremely difficult/expensive to repair damage
- **Facebook: “Move quickly and break things”**
  - Empowered developers
  - Code base notoriously incomprehensible/unstable
  - Eventually changed to “Move quickly with solid infrastructure”
- **Can succeed with strong design culture: Google and VMware**
  - Attracted best engineers

How Much To Invest, cont'd

- **Make continual small investments: 10-20% overhead**
- **When writing new code**
  - Careful design
  - Good documentation
- **When changing existing code**
  - Always find something to improve
  - Don't settle for fewest modified lines of code
  - Goal: after change, system is the way it would have been if designed that way from the start

Ask yourself: "is this the most I can afford to invest right now?"

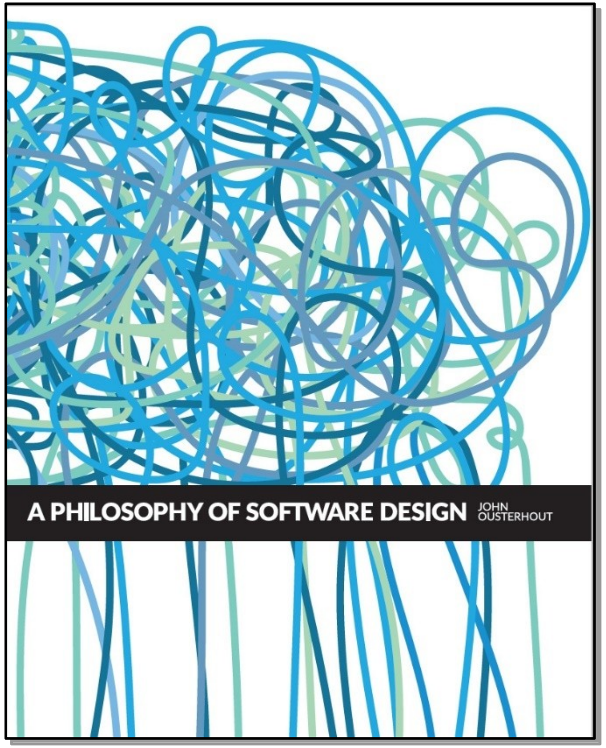
Is the Course Working?

- **Hard to know: ask students in 5-10 years?**
- **Just the first step towards becoming a great programmer**
- **Good energy in class:**
  - Tone of discussions changes halfway through
  - Students are thinking about their code in new ways
- **Interesting challenges for me:**
  - What causes complexity?
  - How to design simple code?
- **Discovering new ideas from reading students' code**
  - Specialized → complicated
  - General-purpose → simple, deep

Software Design Book

- **Goal: capture ideas from CS190**
  - Reach more people
  - Start a discussion
  - Define terminology
- **Short: 170 pages**
- **More philosophical than prescriptive**
- **Published on Amazon: April 2018**

Will the design ideas make sense standalone, without code reviews?



Conclusion

- **It is possible to teach software design**
  - But not currently scalable
- **Principles gradually emerging**
- **Long-term goal: increase design awareness in the software community**
  - Book as vehicle for discussion
  - Attract criticisms, new ideas, better examples
  - Mailing list: software-design-book@googlegroups.com
  - Incorporate new ideas in future versions of book

Can we agree on a set of software design principles?

Questions/Discussion

PLATFORMLAB

Discussion Questions

- How much are you willing to invest?
- How much does a poor code base slow you down?
- Do your code reviews consider design issues?