Some principles to take away from the course:

Effective design maximizes cohesion and minimizes coupling, thus minimizing communication between autonomous programmers. Design can exploit duality between coupled entities, including data structure versus control structure, client versus server (in .NET).

Software complexity can be estimated and -- through that -- cost of development can be estimated. Software complexity is conserved under a variety of transformations.

Patterns ease common implementation tasks. A pattern -- unlike a ritual -- includes declared scope and limits. Literate programming eases the task of synchronizing code with documentation (principle of locality).

Perfect testing is intractible -- not undecidable(!) -- and one must settle for testing that is "good enough".

The reliability of software is based not only upon effective testing, but also, upon usage footprint and social pressure. It is possible to trade execution time for some kinds of language-supported software reliability.

Abstractions -- including SOA -- can apply patterns to handle certain kinds of complexity automatically outside of programming.

At the highest level, abstractions can represent a program without writing a line of code.
Q&A

Q: What relationship does business programming have to scientific R&D?
A: Science is behind.
Brooks: there is no "silver bullet" that will solve all software development problems.
Couch: there is no such thing as eliminating software development problems. One can, however, with some energy, move them around, and even assign responsibility to others.
My last words:

Complexity is conserved, while we travel on a path where no one thought.
We're swimming in an information sea, wherein concepts of software quality and software value are rapidly changing.
There is a new shamanism that shakes the foundations of software engineering and fundamentally challenges some of the values that we hold dear.
This generation of programmers looks at the problem of software quality very differently than does my generation.
Ginsberg's law

Wednesday, December 09, 2009
12:58 PM

Ginsberg's version of the three laws of thermodynamics
You can't win
You can't break even
You can't get out of the game
Ginsberg's corollary

Every major political philosophy violates some aspect of the laws of thermodynamics:

- **Capitalism** is based upon the idea that one can **win**.
- **Socialism** is based upon the idea that one can **break even**.
- **Mysticism** is based upon the idea that one can **get out of the game**.
Ginsberg's theory applied to software engineering

You can't win: complexity always increases.
You can't break even: moving complexity around costs effort.
You can't get out of the game: complexity is preserved in any closed system.
Example: web frameworks

It seems that you have to write much less code. In fact, you swap one problem for another:

Easier to write **common applications** using provided libraries

Much harder to modify existing libraries toward **new purposes**; one ends up **debugging the framework**.

This is a **reusability/customization tradeoff**: the more reusable something is, the harder it is to customize for what it does not do already!
Example: outsourcing

You don't have to write the code, and
The code **costs much less**, but
Your documentation has to be **much better** than before, and readable by non-native speakers of English.
Your contractor can **steal your secrets**!
Governments are starting to look at **code sweatshop regulation**!
Example: razor scooters

Some of the knockoff scooters with other brand names than razor are actually made in the same factory, for competitors!
In other words, what you outsource can become a competitive advantage for others!
Example: cloud-sourcing

Should you move your databases to the cloud?

It's much easier to assure data integrity.
- Backups are done for you.
- Backup strategy exploits economy of scale.

It's much more difficult to assure data privacy!
- Your competitors are storing their data there.
- The law is unclear on your right to privacy when data spans international boundaries!
Steven Barley's law:

When you make a major political or technological change, **you never get what you expect.** What you get is always "something else".
Example: twitter

Wednesday, December 09, 2009
1:18 PM

Example: twitter

Expectation: "I'm on the subway now." or "Hey, let's go out for pizza!"
Reality: "5 people were just killed in cold blood in front of my eyes." or "Hey, let's overthrow this government!"

I was there -- and noticed -- when Twitter changed its main prompt from "What are you doing now?" to "What's happening?"
Example: Star Wars (Strategic Defense Initiative)
The advertised plan: to protect ourselves from nuclear war via technological deterrents.
The reality: protected us from nuclear war via economic deterrents.
As Carl Sagan said, this was "a path where no one thought."
Where's my flying car?

SciFi novels of the 50s had us flying to work in 2000 in atomic-powered helicopters.

Where's my flying car, now?

Barley's law gives somewhat of an explanation: somewhere along the line, it turned into flying information.
Redefining software quality

We're all swimming in the information sea. More or less everyone has access to the same information. So if you have an idea, chances are that 1000 other people have it as well. This changes the whole game of software engineering: being first to market is everything, and traditional SQA becomes meaningless.
Redefining quality

Software quality is a moving target
Most abstract definition: software empowers people
Under this definition, the roughest hacker script has quality!
Sociology of software defects

We might think that bugs are "bad" in online games. According to a study by Jim Waldo (project DarkStar, Sun Microsystems),

Bugs in online role-playing games are expected and exploited, and

Users are disappointed when well-known bugs are fixed!
Not every bug has to die!

According to Jim Waldo, an **acceptable bug:**
- Affects only session state (i.e., current situation).
- Doesn't affect accumulated state (i.e., character development)

In other words, don't bother testing a new RPG at all, except for the "reset button" that restores accumulated state! :)

Defining acceptable defects

Doesn't get directly in the way of users.
Easy workarounds known and posted.
The beaten path is safe, the path less traveled is fraught with danger.
The new shamanism:

Users who can get defective programs to work anyway are the new shamans!
Effective use of a sufficiently defective program is indistinguishable from magic.
Workarounds are the new talismans!
If a service is valuable enough, people are willing to learn the incantations of power in order to use it.
The new shamanism
    Mediates bugs rather than mitigating them.
    Accepts bugs as a part of life.
    Commits brain-power not to understanding, but to remembering the **incantations that work**.
    Result: **software alchemy**, very similar to **hacking exploits**.
Some questionable assumptions we have studied
What is appropriate testing?
What is appropriate documentation?
Is any of what we learned really worth it?
99% of internet startups bomb, in which case software engineering wasn't worth it.
Can always re-engineer later after you have a customer base and revenue.
Time to market and marketing are everything.
So, Brooks is wrong, and the prototype is the product.
Test the paths most taken.
Forget about side paths.
Make sure the appropriate path is obvious.
Punish those who don't figure that out.
Some hard choices

...between adopting the old values for software engineering and adapting to the social forces in the current world...

...choosing between "software quality" and "supporting a culture of shamanism and magic"...

...between "huge success by ignoring quality" and "meager success by quality"...

"If you build it, they may not come!"
Modern software development
Leaves testing to customers.
First adopters are first victims and first shamans.
If value is high enough, people ignore the victims.
So the game is not to make quality software, but instead, **software whose lack of quality can be ignored.**
How will software engineers survive?
By occupying the dying niche that requires quality.
By learning the subtleties and dynamics of internet interaction.
By "fitting into" the new picture and learning the new rules.
Value proposition is knowledge, and not working code. Credit is given for papers, and not for software. A defect that doesn't affect results is relatively unimportant. What is important is enabling the scientific workflow, which results in papers.

Two choices:
High quality software and no tenure (or Ph.D.)
Low quality software, high quality papers.
Questions to ask about a research prototype

- Does it have a potentially long lifetime?
- Are multiple programmers necessary?
- Is performance important?

There is a place for "quick and dirty".
For a short-lived program without performance requirements, nothing we learned about software quality applies.

It is time to apply our principles when
- The program has a potentially long lifetime and/or
- The program is sufficiently complex to require more than one programmer and/or
- Performance matters.
The concept of a scientific workflow

A precedence graph
Starts with data sources.
Ends with academic papers.

Two basic activities regarding software:
Software development is one node.
Software evaluation is another!
What about evaluation?

... of the user interface (COMP171)
... of performance (COMP160, COMP111, COMP112, ...)

This is a little weird, because R&D is only successful if evaluation works.
Samples of scientific workflow

Monday, December 10, 2012
5:07 PM

Product Flowchart

SAT

RADAAR (LIDAR)

Geo Data

Model

Results

"the intellectual property"

Eval
What we're building

Monday, December 10, 2012
5:10 PM

Workflows for:
- Data interchangeability and interoperability.
- Verification of models.
- Validation of new data sources.
  (e.g., weather stations on top of high schools)

"Project EarthCube"
- Provide a comprehensive digital earth.
We might think that workflows are for business, but project EarthCube will only succeed if we create them for science.
Technical problems

choice of common language (MatLab, R, Excel, ...). A: we need a new one.

semantics: how can we make sure that data that someone says is something really is. (integrity)
middleware: what layer should we use to attack the problem?
We as computer scientists are being left in the dust.

There is a national crisis in computing systems.
   Ph.D's
   Tenure
   Promotion