Computer Science 520/620
Software Engineering: Synthesis and Process
Spring 2012
Instructor: Leon Osterweil

I. An Introduction to Software Engineering

CS 520 and 620 Together
• The difference is the expected level of student understanding
  – As evidenced by project submissions
    – Showing stronger grasp (e.g. of formalism)
• Lectures are the same
• 620 students will be expected to do more reading for deeper understanding
• 620 students will probably be required to do an additional term paper.
• Final exam may have different questions for 520 and 620 students

Course Logistics
Spring 2012
Instructor: Leon J. Osterweil (ljo@cs.umass.edu)
Office: CS 302

Class meeting times: T, Th 9:30-10:45
Meeting Room: CS 140
TA: Wenzhe Wu (wruhe@cs.umass.edu)
Course Web Site: http://lesor.cs.umass.edu/courses/cs520-620.Spring12

Text: None Required, but Ghezzi, Jazayeri, and Mandrioli, Fundamentals of Software Engineering has helpful supplementary reading.
Some material on the web Some Readings will also be suggested.
Instructor slides: Will be on course web site (hopefully, usually) before lecture
Instructor Office Hours: T, Th 11:00-12:00, and by appointment

*20 vs. *21
• Synthesis vs. Analysis: Which comes first?
• The goal is software that is well enough understood to be trusted, relied upon
  – Requires analysis and testing (*21)
  – But, of what? (*20)
• The need to do what is taught in *21 shapes the software artifacts we teach in *20
  – And conversely
• Can’t exclude analysis and testing ideas from *20
  – Review for some students

Course Educational Objectives
• Acquaint students with various approaches to developing complex software systems.
• Help students develop skills in analyzing and evaluating different approaches to software development
• Provide framework and tools for making prudent choices for how to select development approaches that are most appropriate for different software projects.

Evaluation Approach
• Evaluation:
  – The primary basis will be 3-4 major projects.
    – 620 Students will be expected to show greater depth of understanding (e.g. more use of mathematical formalism)
    – 620 students will be required to do a research paper.
  – There will be no midterm examination
  – There will be a final examination
    – It will probably be a take-home exam
    – It will have a modest effect on the final course grade.
• Course Projects
  – 3-4 project assignments.
    – NOT be group efforts: individual effort.
Course Outline

1. Introduction to the course (why is this so important, why is it so hard?)
2. Artifact representations (how to model software artifacts)
3. Anatomy and physiology of a software product (how to pick the right models)
4. Analysis, constraints, and relations (how they should be connected)
5. Processes and their models (how to build and connect artifacts to form software products)
6. Case studies of specific software processes and how they are used to build software products
   – Which processes seem to be good for what
7. Additional topics (as time allows)

Engineering Software?

• The elusive nature of software
• How to make it
  – Faster, better, cheaper
• Requires being able to measure it
• Requires being able to materialize it
• Materialize by modeling
  – Software is models (?)

Software Products and their models

• Software Models
  – DFGs, FSAs, Petri Nets, CFGs, Object diagrams, Architecture diagrams, etc.
• Anatomy and Physiology of a software product
  – Requirements, Architecture, Code, Testcases, etc.
• Analysis, constraints, and relations among product components
• Evolution, Reengineering, Reverse Engineering leads to new versions of the above.

Processes and their models

• Why and how to model processes
• Microprocess vs. macroprocess
• Some process modeling approaches
  – Funsoftnets, Marvel, Slang, etc.
  – Little-JIL process definition language
• Process are products too
  – Both are models

Faster, Better, Cheaper

• Rests on measures
  – That come from analyses
• What to analyze, how, when, and why
• Approaches to doing this
  – E.g. static vs. dynamic
• Much of the way we synthesize software is driven by our need to analyze it
  – Synthesizing for analyzability

Specific software processes and products

• Requirements definition
• Design
• Architecture
• Coding
• Extreme Programming
• Testing and Analysis
• Formal Verification
• Rework
• Evolution
• Configuration Management
  – Reengineering, Reverse Engineering
Some Possible Additional Topics

- Software Project Management
- Software Cost Estimation
- Software Economics
- Offshore Software Development
- Global Software Development

Why Study Software Engineering?

- Practical engineering challenges: SOFTWARE IS:
  - A critically important infrastructure component
  - A key enabler
  - Militarily
  - Economically
  - Scientifically
  - Societally
  - Culturally
- But it is generally
  - Expensive
  - Of poor quality
- The wealth of companies and nations depends upon it
- It raises important engineering challenges
- It raises hard and engaging scientific questions

Software as Key Infrastructure

- Software is key in the development of roads, bridges, schools, banks, hospitals......
- A key component of all other scientific disciplines
- And most of the arts, humanities, and social sciences too
  - Economic forecasts and models
  - Repositories of historical, anthropological, info
  - Worldwide access to art of all kinds
- Essential scientific "grand challenges"
  - Human genome mapping
  - Smart cars, houses, buildings
  - Better health care

Software Economics

- Trillions-of-dollars-a-year industry
- A “make or break” key competency for
  - Most of the largest companies
  - Entire national economies
- Software costs hundreds (sometimes thousands) of dollars per line
- Lifetime maintenance costs are higher still
- Software can represent major (the majority of) costs of
  - Aircraft and spacecraft
  - Communications systems
  - Medical devices and care systems
  - A variety of surprising products
- Greatest costs may be due to software that is late or software that fails

Software Quality

- There have been many (in)famous software failures:
  - National long-distance telephone outage
  - London Ambulance Service disaster
  - Mars rover that died (subsequent brought back to life)
  - Airbus that skidded off the runway
  - American Airlines crash in Colombia
- Other manifestations of poor quality:
  - Optimizing software that is suboptimal
  - Software that is too slow/costs time and money
  - Software that is incomprehensible
  - Software that is more trouble to use than it is worth

Past Approaches to These Problems:

- Use more people
  -- creates chaos, more problems
- Create "better" programming languages
  --bad programs can be written in any language
- Design before writing code
  --how do you get the design right?
  --are you are designing the right program?
- Start by "baselineing" requirements
  --but they inevitably change, what then?
- Test programs longer
  -- find more bugs, but is that good?
More Approaches

- Train managers better
  -- but how do you manage this kind of product?
- Software tools to help people write programs better
  -- software tools are often bad software
  -- people don’t know how to use them
- Use superior software processes
  -- How do you know when a process is “good”
- Train people better
  -- what to teach them?

Struggling with Hard Technological Problems Can Lead to Better Science

- Need to create more accurate calendars
  -- Led to laws of planetary motion around the sun
- Desire to cure colds and cancer
  -- Led to Microbiology and molecular biology
- Need to keep bridges from falling down
  -- Led to Statics, materials science
- Need to capture and store electricity
  -- Led to development of Electromagnetism

Superficial Questions in Software

- How can we prevent errors in software?
- What development process should we use?
- What tools should I buy?
- How much testing should we do?
- How can we deliver on time?
- How can we keep our customers happy?

Deep Questions in Software

- What constitutes quality in software?
- What is an error?
- Can errors be proven to be absent?
- What do people do when they create software?
- How can using more people make things worse?
- What is software anyway?
  - Is it like anything else
  - Is software based on any science? governed by any laws?

Our Strategy

- Start with engineering problems
- Look for deeper scientific questions
- Hypothesize conceptual foundations
- Base engineering solutions on conceptual foundations
- Use experience with engineering solutions to advance the science:
  -- validate hypotheses
  -- new subhypotheses
  -- suggest new hypotheses
  -- suggest new issues and questions

What is the Nature of Software?

- It is insensible, nonphysical
- It is a component of a larger system that integrates the actions of hardware, people, mechanical devices, software systems, communications
- It transforms data using computers
- It changes constantly both proactively and reactively
- It has a complex structure
- It is usually very large, takes a lot of time and money to build
What's Different about Software?

- Software is extremely malleable – we can modify the product all too easily
- Software construction is human-intensive, there are no real costs of materials
- Software is intangible: no laws of physics are applicable
- Software is not detectable by any of the five human senses
- Software application horizons expand too fast—with human demands/imaginations
- Software problems are unprecedentedly complex (a machine with millions of moving parts)
- Software has a discontinuous operational nature

Software Is Not Completely Unique

There are products that share some of its characteristics

Studying such analogs can be useful:

- Help us learn about computer software
- Find points of similarity
- Suggest successful approaches to be emulated
- Avoid known mistakes

Analogy 1: Custom Home Building

Problem: Create a product that provides shelter, sanitation, food preparation facilities, recreation

Solution Medium: DWELLING UNIY

- Product Components
  - Customer statement of needs—size, location, style, number of rooms
  - Architect sketches
  - Engineer blueprints
  - Instructions, manuals
  - Additions/remodelling as needed

Strengths of Analogy 1

- Problem solving to meet real-world need
- Single solution medium
- House is not whole solution
  - only a component
  - must "fit in" with: utilities, neighbors, customer lifestyle

Weaknesses of Analogy 1

- Customer familiarity with
  - Solution medium
  - Descriptive terms
- Tangibility of the product
- Visibility of
  - Final product
  - Intermediate products
- Too specific: one need, one house

Analogy 2: Legislation

Problem: Create a product to meet needs of citizens or solve problems, such as Common defense, domestic tranquility, establish justice

Solution Medium: Agencies, procedures, bureaucracies

- Product Components
  - Statement of needs
  - Analysis of needs/possible solutions
  - Legislative plan or blueprint
  - Bill/Law
  - Implementing bureaucracy
  - Amendments
  - Court decisions

Strengths of Analogy 2

- Needs are from the "real world"
- Single solution medium
- Laws/agencies are not the solution—they must "fit in"
- Solution medium is often inadequate or inappropriate
- Customer unfamiliarity with solution product
- Behavior of product often unexpected
- Adjustments, improvements must be planned in
- Overhaul and abolition often needed
Other Similar(?) Types of Products

- Plays and Movies
- Recipes
- Kitbuilding instructions
- Driving instructions (e.g. rallies)

Some Key Features of These Products

- Problem enunciation, understanding
  - What is the problem to be solved?
- Solution formulation
  - How might the problem be solved?
- Solution reduction to practice
  - How will the problem actually be solved?
- Solution implementation
  - The actual solution to the problem
  - Interconnections among all of these
  - Evidence of consistency
  - Intuition (?) about what makes them “good”
  - Need to evolve them over time

“Classical” Software Development

- requirements
- architecture
- design
- coding
- analysis/testing
- evolution/maintenance

Traditional Waterfall Model

Leaves out a lot
Some Things To Address

- Are all of these things always needed?
- What is contained in each under what circumstances?
- In what order should they be developed, integrated?
- What order makes sense when?
- What about iteration?
- How to decide all of the above?
- Is there help with some of this: tools, techniques, ...??
  - What kind?
  - What to use to do what when?

Approach: A Stakeholder-centered view of Software Engineering

- Software must meet the needs of all of its stakeholders
- Software Quality is relative to stakeholder expectations
- Different software products may have different parts
- Built in different ways, with different approaches
- ........
- Accounts for why different people have different views of the quality of software
- Underscores a core difficulty in building software
  - A software product is good if it meets the needs and expectations of all stakeholders

Some Types of Stakeholders

- Users
- Customers
- Testers
- Service recipients/end users/innocent bystanders
- Product integrators (imbedders)
- Investors
- Regulators/monitors
- Etc.

What Are the Stakes?

- User:
  - Easy to get it to do what it is supposed to do
  - Cost is consistent with benefits
- Developer:
  - It is always "under control", and easy to modify
  - Easy to see that it does what it is supposed to
- Customer:
  - It doesn’t cost too much
  - Development progress is visible and satisfactory
- Innocent Bystander:
  - It does no harm
  - It provides needed services
- Investor
  - It returns a profit

A Product

And its stakeholders
An Example: Air Traffic Control System (ATCS)

- Large
- Life-Critical
- Distributed
- Integrates hardware devices
  - E.g. radars
- Integrates software systems
  - E.g. Collision Avoidance Systems
- Human-in-the-loop
  - Pilots, controllers, etc.
- Time-critical
- Diverse stakeholders
- Need for change, evolution

ATCS Stakeholders

- Users
  - Pilots
  - Air traffic controllers
- Developers
- Customers
  - Airports
  - Airlines
- FAA
- Testers
- Hardware manufacturers (e.g. tracking radar manufacturers)
- Airplane manufacturers
- FAA safety examiners
- Passengers
- People who live near airports

All have different stakes in this software

ATCS Stakeholders

- Users
  - Pilots
  - Air traffic controllers
- Developers
- Customers
  - Airports
  - Airlines
- FAA
- Testers
- Hardware manufacturers (e.g. tracking radar manufacturers)
- Airplane manufacturers
- FAA safety examiners
- Passengers
- People who live near airports

All have different stakes in this software

Other Kinds of Products

- Systems designed for rapid evolution
  - Commercial web sites
  - Entertainment systems (e.g. games)
- Commercial software
  - What Microsoft sells
- Embedded software
  - In cars, medical devices, etc.
- Small, limited lifespan systems
  - Rapid prototypes
  - Demonstration software
- One-of-a-kind systems
  - Mars rover software

Have very different stakeholder group types and emphases

- Air traffic control system
  - Safety is key
  - Regulator stakeholders are very important
  - Very diverse user community to be satisfied
  - Web site in a highly competitive environment
  - Strong focus on user stakeholders and usability
  - Some brittleness may be OK
- Anti-lock braking system software
  - Safety, robustness are key
  - Speed matters a lot
- Mars rover software
  - Correctness is very important
  - Safety probably not as important

Each type needs to be built in a different way

Examples: What Stakeholders Want to Know

- Some Examples of understandings needed:
  - What does the product do and how do we know
  - What is the product supposed to do
  - How does it work
  - What would happen if I did ......
  - Suppose we change ......
- There are infinitely many such questions
- For each there are endless varieties of answers
- The answers themselves form key parts of the product
- Superior products are tightly interconnected bundles of:
  - Component artifacts, used as the basis for:
    - These questions
    - Their answers
  - Solid basis for believing the answers

Stakeholders and their questions
Key role of Relations

- Connections between artifacts
  - And types of artifacts
- They defined "well-formedness"
- Are the basis for evaluation
- Are used in evaluation

Some Examples of “Relations”

- Executing this code must meet this requirement
- This code must conform to that design element
- This compiled code came from this compiler
- This design element addresses those requirements
- These lower level requirements are elaborations of these higher level requirements
- This is the date by which that test must be passed
- Component invocations conform to component abstract interface specifications
- Documentation describes the actual system
- ETC.....

Key Focus of Software Engineering

- How to describe different product types and
  - Their component parts
  - Their relations/interconnections
- What processes should be used to build different kinds of products?
  - And assure that they are "good" at the end
- How to grow and evolve different kinds of products
  - At acceptable costs
  - And improving quality

Software Engineering = Products + Processes

Key Obstacles to Doing This

- How can you control it if you can’t see it?
- How can you tell if it is on target if you can’t see the target?
- What intuitions apply to something that does not obey any laws of Physics/Chemistry/Biology/Sociology....

First Problem we will attack:
REPRESENTATION
How to help people see and sense software so that it can be engineered?

Two Good, Frothy Reads

- Anguished Cries
  - Fred P. Brooks, Jr., The Mythical Man Month
    - Addison Wesley, 1975, then 1995
  - Fred P. Brooks, Jr., "No Silver Bullet….”
    - In IFIP 86 Proceedings, then in IEEE Computer