CMPE 276
Software Engineering

Instructor: Luca de Alfaro
luca @ soe
T Th, 4-5:45
Baskin 165

Lectures

- Course taught mostly from notes and from articles
- There is an optional book:
- Office hours: Mondays 1-2pm (ok?), Baskin 317A (my office) or by appointment.
- How to get hold of me: email (luca @ soe)

Course material

- All material will be on the web site (will be up after the weekend).
- No plagiarism: cite all sources used.
  This course was inspired by Pfleeger, "Software Engineering", and by Aiken’s course at UCB.
- Send me your email address, so that I can build a class list.

Two Tracks

Project track:
- You do a large software project (can be anything of your interest)
- Done in teams of 2-4 people (not alone)
- Apply the software engineering techniques we will present to manage the project
- Do two presentations in class (project design, project presentation)

Tools track:
- You apply tools for the analysis of software, and do readings and homeworks based on the tools
- Tools for static analysis, software verification (various kinds), interface specification
- Homeworks: both theoretical (read papers, do exercises) and practical.
- Presentation on tool experience.

Two Tracks

- It depends what you want to learn...
- Project track:
  - Do you want to learn how to manage software projects, and the classical "software engineering"?
- Tools track:
  - Are you interested in learning (and in perspective, contributing) to cutting-edge techniques for software analysis and production?
Two Tracks

- What are your interests?
- What do you expect from this course?

Project Timeline

- Requirements and specification
- Project design & plan
- Design review
  - Done by other team(s) - presentation
- Revised design & plan
- Quality Assessment
  - Done by other team(s) - presentation

Grading

- Approximately: 25% final, 15% midterm, 60% project/homeworks.
- You will help assessing other team's work (project reviews) and testing approaches.
- How many of you want a letter grade?

Course Outline (provisional)

- 9/19 Overview
- 9/24 Software Process
- 9/26 Requirements and spec
- 10/1 UML
- 10/3 Design patterns
- 10/6 Config management
- 10/10 Testing
- 10/15 Project present.
- 10/17 Testing: Varscof
- 10/22 Sw model checking
- 10/24 Midterm
- 10/29 Sw model checking
- 10/31 Interface specs
- 11/6 Dynamic Analysis
- 11/7 Static Analysis
- 11/12 Embedded Sw
- 11/14 Case study
- 11/19 Formal specs
- 11/21 Project present.
- 11/26 Adv syst models

Marielle Shellinga  Jim Whitehead  Guest lecturer

What is Software Engineering?

Your thoughts here...

What is Software Engineering?

- Many points of view:
  - Discovering what the customer wants
  - Project management (cost, people, time, ...)
  - Systems engineering (global view)
  - Software specification
  - Software production process
  - Software principles (how to write good software)
  - Testing problem (if only we could find the errors)
  - ...
What is Software Engineering?

- Many points of view:
  - Discovering what the customer wants
  - Project management (cost, people, time, ...)
  - Systems engineering (global view)
  - Software specification
  - Software production process
  - Software principles (how to write good software)
  - Testing problem (if only we could find the errors)
- Our focus:
  - Tools and principles that help programmers and teams.
  - Not tape ($$, people, ...) management problems.

Why is software difficult?

- Producing software is a huge problem:
  - Cost overruns
  - Missed deadlines
  - Cancelled projects
  - Buggy software
- Many other systems can be designed fairly reliably.
- Why is this??

The first problem: understanding what to build

- Focus on the problem to be solved, not on the tools available
  - Avoid "if all you have a hammer, everything looks like a nail"
- First decision: what part of the problem should be solved by software?
  - Cost/benefit analysis

My research

- Interface theory
  - How can we specify component interfaces, and:
    - Check compatibility in a design
    - Check that they are implemented correctly
    - Use interfaces to show that software components are correct
  - Many interface types (real-time, embedded, software, protocols, ...)
- Multi-component systems as games
  - Lots of game theory
- Anyone interested? RAship available.

Why is software difficult?

- Many opinions:
  - It's the poor languages we have
  - It's the "brittleness" of software
  - It's a newer engineering problem
  - ... 
- My view:
  - It's all about complexity
  - Similar engineering efforts are similarly buggy (civil engineering, home construction, ...) when sufficiently complex.
The first problem: understanding what to build

- Are customer and user the same entity?
- If not, who really knows what should be built?
- How to manage testing and complaints?

Time to Market

- Huge motivation
  - HP profits: 80% from products less than 2 years old
  - For large systems (air control), "soon" may mean "ten years" - still difficult to achieve
- How to achieve short time to market?
  - As in hardware: parallelism.
  - Divide the design in components, and have them implemented separately.

Complexity and Communication

- There is an upper bound to the complexity (and work) that can be handled by a single person (in a given time, but life, and patience, is finite).
- Dividing the work requires communication: specifying what the parts have to do, and make them fit together.

Parallelizing Software Engineers

- Divide the design in components,
- Document the interface of each component
  - Requires specification language/methods
- Have each team put the results together
- Validate the result,
- Pitfalls?
  - Your opinion here.
Pitfalls of Parallel Development

- As in distributed computing.
- Tradeoff between development and communication:
  - Assigning large components to individuals/teams limits overhead.
  - Dividing the project into small components creates large overhead in communication and interface specification.

Pitfalls of Parallel Development

Communication is a huge problem:
- Ambiguity (you say one thing, I understand another, and we both believe we understand each other)
  - "Flag as exception all income tax returns where the gross income charged by more than 30%"
  - \( \text{new-old} / \text{old} > 0.3 \) or \( \text{new-old} / \text{new} > 0.3 \)?

Pitfalls of Parallel Development

Communication is a huge problem:
- Ambiguity (you say one thing, I understand another, and we both believe we understand each other)
  - "Flag as exception all income tax returns where the gross income charged by more than 30%"
  - \( \text{new-old} / \text{old} > 0.3 \) or \( \text{new-old} / \text{new} > 0.3 \)?
  - \( \text{foo} \left( \text{amount, n_of_installments} \right) \)
  - is the amount the total amount, or the amount per installment?

Pitfalls of Parallel Development

Communication is a huge problem:
- Ambiguity (you say one thing, I understand another, and we both believe we understand each other)
  - "Flag as exception all income tax returns where the gross income charged by more than 30%"
  - \( \text{new-old} / \text{old} > 0.3 \) or \( \text{new-old} / \text{new} > 0.3 \)?
  - \( \text{foo} \left( \text{amount, n_of_installments} \right) \)
  - is the amount the total amount, or the amount per installment?
- Incompleteness
- Different background assumptions

Pitfalls of Parallel Development

Ambiguity or incompleteness in interfaces leads to incompatibility:
- The work is divided, with specification on what the components should do, and when put together - it just doesn't work!
- This on top of the normal bugs, of course.
Interfaces

Deciding where to place the interfaces is hard:
- If the interfaces change too often, bad division the development is not parallel
- Each component needs to be implementable by someone (not too heterogeneous in competence required)

How to Decompose a system?

Your suggestions...

How to decompose a system?

Many choices:
- Competency: let teams do what they know how to do.
- Functionality: decompose according to function
- Data: along datalines.
- Information flow: follow the data.

How to decompose a system?

Often:
- What it does
- How we build it
- Who builds it

[Aiken, 2002]

What it does

- The application itself usually dictates natural divisions
- A compiler has a
  - Lexer
  - Parser
  - Type checker
  - Optimizer
  - Etc

[Aiken, 2002]
How we build it

- Buildings need scaffolding during construction
- So does software!
- Two areas in particular:
  - Lots of extra code that is not really part of the final product
  - Influence of third-party subsystems
- Test harnesses, stubs, ways of building and running partial systems
  - Examples? [Aiken, 2002]

Who builds it

- Software architecture reflects the structure of the organization that builds it
- Often, 5 developers = 5 components

Summary: designing a system

- Understand what must be built
- Decompose the design
- Interfaces must be well-chosen
- Build the components
- Integrate them

Specifications are essential

Specification of the complete system:
- To know what you are doing
- To support system testing
Specification of the interfaces:
- To divide the work
- To support unit testing

This course

Software process:
- Know what you do
- Develop a plan for doing it
  - Project specification, decomposition, implementation, testing,...
Software tools:
- What tools are available to help us?
  - Specification methods, dynamic and static analysis, software verification, programming patterns,...