Software Engineering   
Midterm Study Guide

The midterm will focus on the basic terminology we've learned so far this semester   
(Expect that the Final will include more SE 'problems' and implementations)

You will have the entire class period to finish it, but there will be lots of questions:

If you rely on your notes or the text too much, you may not have time to complete it

It would be wise to make a Table of Contents for your notes!

Please bring red and black pencils and your BU ID to the exam

Bringing a straightedge would be a good idea also  
Please do NOT bring pens!

Background and activity topics (these topics are either not found in text OR are an elaboration of topics found in text):

* BASH Shell commands – W-1Review/Refresher Material (pwd, cd, ls, cat, mkdir, rm, rmdir, cp, mv, redirection, etc)
* Software Crisis and birth of Software Engineering – W0.2.1M (1968 NATO conference on Software Engineering, Garmisch, Germany, 10/7-11/1968)
* Software Life Cycle – W0.2.1W (Hosier-1961)
* Software Engineering Process Models – W0.2.1W (Waterfall, V, Throwaway & Rapid Prototyping, Phased Incremental, Phased Iterative, Spiral, RAD, Agile)
* Agile Manifesto – W0.2.1W (2/11-13/2001, 4 main values, 12 Principles)  
  Team Organization and Workstyles – W0.2.2W (Lines of Communication, Extroverts, Introverts, Intuitives, Rationals and combos, Democratic Decentralized, Controlled Decentralized, Controlled Centralized, structure vs. creativity)
* Ruby Language – W1.2.1M (History, description)  
  Regular Expressions –W1.2.2M & W1.3.1M (be prepared to give the meaning of various RE symbols such as \*, (), \d and their combinations)
* Git – W2.2.1M, W2.2.2M & W2.3.1M (git init, git add, git status, git commit, git checkout, git diff, git pull, git push)
* HTML – W2.2.3M & W2.3.2M (basic page structure, links, paragraphs, tables, containers for separating content)
* CSS – W2.2.2W & W2.3.2W (four types of selectors, id vs. class, purpose: separating visual appearance from logical structure) (Check out text Ch. 2.3)
* HAML – W2.2.3W & W2.3.3.W (use of indentation, use of %, use of – and =)
* Software Architecture - W2.2.1W (pipes-and-filters, client/server, peer-to-peer, layered)
* SaaS Stack – W2.3.1W (arch. diagram of local development stack, and how various layers are mapped to it)
* Pivotal Tracker – W2.3.4W (Requester, Owner, status sequence)
* Rails – W3.2.1-2M (Framework vs. Toolkit vs. Architecture, history of Rails)
* Metrics and Quality Assurance – W4.2.1M (measure, metric, indicator, fault, failure, MTTF, MTTR, MTBF, SQA responsibilities)
* Effort estimation – W4.2.2M (experiential - such as WAG or Delphi, discussion of algorithmic/formulaic – such as Walston-Felix, Bailey-Basili & COCOMO in text Chap. 7.10)
* Risk Management W4.2.1W (Risk Prob, Risk Impact, Risk Exposure, RMMM Plan: Mitigation, Monitoring & Management)
* Scheduling and Tracking – W4.2.2W (SRS, WBS, PERT, Gantt, Activity, Milestone, Predecessor, Successor, Duration, Due date, Critical Path, Real Time, Available Time, Slack Time, Event or Node, Job/Activity or Edge, Earliest/Latest Start Time, Earliest/Latest Finish Time)
* Planning Poker vs. 3-pt system – W4.3.1W (Planning Poker uses scale based on Fibonacci Series)
* TDD – W5.2M (Who 'rediscovered' TDD and championed it: Kent Beck, the creator of Extreme Programming)
* Cyclomatic Complexity – W5.2.2W (Know basic Flow Graph Notation, know the 3 ways of computing Cyclomatic complexity)
* Overall – W5.2.1W: make sure you understand the synopsis of our process

Text/Lecture Topics:

Chapter 1

* 1.4: SOA
* 1.5: SaaS
* 1.6: Cloud Computing
* 1.7: Legacy code
* 1.8: Testing terms: Verification, validation, unit testing, integration testing, acceptance testing
* 1.9: Productivity: Moore's Law, DRY

Chapter 2

* The important take-away information from Chapter 2 is the overall architecture for SaaS apps, and the many embedded architectural layers that make up its implementation as summarized in your W.2.3.1W activity (expect to replicate your diagram, label it, and explain each layer)

Other details:

* 2.2: Protocols, URI structure, DNS, localhost, stateless protocol, cookies
* 2.3: HTML, CSS (as above)
* 2.4: 3-Tier shared-nothing
* 2.5: Model-View-Controller
* 2.6: Active Record for models, CRUD
* 2.7: REST, Route
* 2.8: Template View, HAML

Chapter 3

* 3.1: Three Pillars of Ruby
* 3.2: Remember that every method must have an object as its receiver, objects can ask about themselves (reflection)
* 3.3: Understand that all operations are arguments of the send method (be able to show), class vs. instance methods, ramifications of dynamic typing, poetry mode
* 3.4: definition of class vs. instance methods, class vs. instance variables, #{} interpolation syntax
* 3.5: attr\_accessor, method\_missing
* 3.6: block, block syntax, iterators (each)
* 3.7: mix-in (duck typing), modules, mix-in implementation syntax, Enumerable
* 3.8: yield

Chapter 4

* 4.1: Gemfile, bundle install (automation for repeatability), rake routes, root
* 4.2: migration, rails environments
* 4.3: ActiveRecord, convention over configuration, CRUD
* 4.4: ApplicationController, ActionController::Base, ActionView::Base, controller methods, helpers (movie\_path) & helper methods (link\_to), HAML
* 4.5: instrumentation, inspect
* 4.6: Form tag helpers
* 4.7: redirection, flash
* Don't forget that Create and Update each require two interactions (Ch. 2.7, Screencast 2.7.1)
* Be aware of the basic operations performed on models (e.g., create, find, update\_attributes, destroy) and whether they are class or instance methods

Chapter 7

* 7.1: BDD, user stories, Connextra format
* 7.2: Points, velocity, Pivotal Tracker, Spike, Epic
* 7.3: SMART User Stories
* 7.4: Lo-Fi UI sketch vs. Storyboard
* 7.5: Agile Cost Estimation (Time and materials for given period of time with team sized for max efficiency)
* 7.6: Cucumber/Capybara, feature, scenario, steps, mapping of steps to step definitions w/ reg. exps.
* 7.7: integration and acceptance testing w/Cucumber and Capybara
* 7.8: Using Background to DRY out common steps
* 7.9: Explicit vs. Implicit requirements, Imperative vs. Declarative Scenarios
* 7.10: Generation of User Stories, Point estimation and Velocity in Agile corresponds to Req. Elicitation, generation of Req. Spec, cost estimation, generation of schedule and monitoring thereof, Change Management for Req, Cost & Schedule (Requirements Creep), Implementation Validation (traceability), and Risk Analysis and Management
* 7.11: Add more people to project if behind schedule (wrong!), throwaway vs. rapid prototyping (customers tend to confuse mock-ups with completed features)  
  (Note: Having to write Cucumber code will NOT be on the midterm)

Chapter 8

* 8.2: TDD, Red, Green, Refactor, FIRST unit tests, RSpec (domain-specific language)
* 8.3: Seam, double or mock, stub (what are they, what do they do)
* 8.5: Fixture vs. factory
* 8.6: context and before blocks (what do they do)
* 8.7: code coverage, C0 vs. C1 vs. C2 coverage
* 8.8: basic blocks, control flow coverage, DU-coverage, Black box vs. white box, mutation, fuzz
* 8.9: Plan and Document Integration Testing (why this is in Ch. 8, I don't know) – Top-down, Bottom-up, Sandwich, stubs vs. drivers, Dijkstra: testing can show presence of bugs, but can never show their absence
* 8.10: 100% test coverage means no bugs (wrong!), code-to-test ratio, have to isolate unit tests but make sure that stubbed out calls are tested fully in integration tests, be careful not to create dependencies on particular order of tests
* (Note: Having to write RSpec code will NOT be on the midterm)

Chapter 9

* 9.5: control flow graph, basic block, cyclomatic complexity, code smells (shotgun surgery, data clump, inappropriate intimacy, repetitive boilerplate), SOFA

Chapter 10

* 10.1: Two-Pizza Team, Fred Books: You cannot win if your team loses, and you cannot fail if your team wins, Scrum, scrum meeting 3 questions, sprint or iteration, Team, ScrumMaster, Product Owner, don't forget Intel Disagree and Commit
* 10.2: Pair programming, driver or pilot, navigator, promiscuous pairing

Overall:

* What are some of the reasons that we need to practice Software Engineering?
* Why can't we just write code the way we want to and deliver it?
* **WHAT IS OUR PROCESS MODEL THIS SEMESTER?**