A few more things about Agile and SE

Could help in interviews, but don’t try to bluff your way through
Refactoring

How to do it, where it fits in

http://www.cse.ohio-state.edu/~crawfis/CSE3902/index.htm
Refactoring basics

- Systematically rewriting code to improve design
  - Preserve input-output behavior (loosely speaking)

- Conceptualized as “transformations” to code
  - Careful cut-and-paste editing by hand
  - Automated transformations like those found in Eclipse

- A very large catalog of transformations
  - Some generic (rename)
  - Some language specific (push down)

- Many common-sense refactorings are actually a composition of a few refactorings
1. Identify the problem
   - Violations of SRP, DRY, “smells”

2. Formulate a goal design
   - Sometimes get more ideas while refactoring

3. Construct a refactoring plan
   - **Incremental** – don’t attempt the whole thing at once
   - **Each major “step” should be meaning-preserving and testable**
   - **Reduces risk** of hard-to-find bug at the end

4. Apply refactoring transformations. **For each:**
   a. Verify preconditions (won’t break the program, legal)
   b. Apply transformation(s)
   c. Test (regression tests)
   d. Repair and retest as necessary
Refactoring is the last step in BDD
- Initial “design” is minimalist & simple
- Often easier to see best design after implementation

Basically, an on-going activity, as you learn about your code and design

Also more comprehensive refactorings between iterations or milestones (just like special testing iterations)
TDD: Test-Driven Development

- A predecessor of BDD (Behavior-Driven Development)

- Key ideas, captured by a process
  1. **“Test first”** – write tests before writing code
  2. Write the **simplest code possible** to pass tests
     - Go “from red to green” in JUnit
  3. **Refactor** as necessary to achieve good design

- Shortcomings (addressed by BDD)
  - Focus on class impl’s (the “unit” in “unit testing”)
  - Yet little guidance on how to design classes

- BDD
  - Expands Stories to include Scenarios
  - Scenarios clarify behavior and also fuel OOD (nouns/verbs)
  - Easier to test features (match the story) than classes (what part of story does class achieve???)

- If you get interview question on TDD, say “I learned and used BDD, a successor to TDD, so I’m more prepared to talk about BDD.”
Basic idea: Integrating your code into the build several times a way

1. Develop in small units (e.g., Stories/features or portions of stories that realize useful part of a feature)
2. Submit your code to repository through CI tool
3. Code is automatically compiled into “build” and tested
4. If tests pass, code is checked in (else not)

Typical tools are Bamboo and CruiseControl

Advantages - avoids broken builds
- Keeps bad code out of repository
- Keeps build current and working

It is basically automated “always runnable”
- “Always runnable” is ten-cent CI. ;)

CI: Continuous Integration
Software Architecture - Why

- Helps you identify the essence of your app
- Helps you locate code/features in your system
- Helps make your system extensible and says how to extend it
- Tells you what you can’t do

Pipe-and-Filter Architecture
Gary’s Game System Framework
Feature List

1. The framework supports different types of terrain.
2. The framework supports different time periods, including fictional periods like sci-fi and fantasy.
3. The framework supports multiple types of troops or units that are game-specific.
4. The framework supports add-on modules for additional campaigns or battle scenarios.
5. The framework provides a board made up of square tiles, and each tile has a terrain type.
6. The framework keeps up with whose turn it is.
7. The framework coordinates basic movement.

Risk-Oriented Look at Feature List

- Recall that a risk-oriented approach has you take on your biggest risks first

- Looking at the feature list, pick out if any of these resonate:

  1. It’s the **essence** of the system (if you get this wrong, you’re toast)
  2. I don’t know **what it means** (if you don’t understand it, you’re toast)
  3. I don’t know **how to achieve it** (if you don’t know how to build it, you’re toast)
I picked these just for ESSENCE

It’s not a game if it doesn’t have a board, pieces, and moves

What’s significant?
The board for the game
Game-specific units
The framework coordinates basic movement.

Why?
Q1
Q1, Q2
Q3 (and maybe Q2)

We decided that the board was core to the game... without a board, there really isn’t a game!

We thought that troops were essential to the game... and we’re not sure what “game-specific” might really mean. So two Qs applied here.

This seems a little vague, but it’s not something we’re sure about how to do. Definitely worth spending some time up front figuring out what this means, and what we need to do.
Abstract to a Diagram

Even though Board doesn’t have any variables of type Unit, it’s still associated to Unit because of its methods that take in Unit instances.

UML doesn’t have a good way to show multi-dimensional arrays, which is what the tiles variable really is. So we can just use an ordinary association.

There are only three classes in the project, but it’s still a lot more structure than what we had before.
Software Architecture is your design structure, and highlights the most important parts of your system, and the relationships between those parts.

- It’s about **structure**: components & relationships
  - Easily visualized with boxes and arrows

- It’s the **whole**: summarizes whole system

- It’s **macro**: a small number of boxes and edges
  - See & talk about key decisions in whole system

- Its **rules** govern evolution: way things are done
  - Not just a pretty picture, and the whole system
What are the architectural rules?

- Board has to be 2D (x,y)
  - Coordinates integer values
- Space in a coordinate shall be represented by a Tile
  - Units are held by Tiles on Board
- Unit can’t be on multiple Tiles (is it really a rule? I believe it)
- Ability add/remove Units from Tiles
- Interactions between Units on separates Tiles have to go through Board
- Board is a Façade for Tile (not precisely captured in this diagram)
The Software Architecture Paradox

- An architectural decision is a decision to make something shared: not modular, unmodular
  - Helps make other things modular
  - Helps make other things easy to do
  - Trick is to make the right thing shared/global
    - It shouldn’t change in the future
    - A wise trade-off

- What’s global is articulated in the rules
  - “Thou shalt X” means “you can freely do X repeatedly (violate DRY) and you’re not creating a change nightmare”
  - A lot of interfaces, which are supposed to be stable anyway, but now need to be really stable
Some army-related games might have tanks and soldiers...

...and flight simulators might use planes, jets, and rockets.

...fantasy games might have rangers, magicians, and swordsmen.

Chapter 7
Here are a few of the game-specific units, and their properties, mentioned on the last couple of pages.

**Tank**
- attack = 12
- experience = 22
- defense = 9.5

**Human Soldier**
- weapon = Bazooka
- name = "Simon"

**Airplane**
- speed = 110
- gun = Gatling
- model = "A-10 Thunderbolt II"

What is common among these different types of units? What basic things can we say that would apply to any game's units?
Architectural Styles and Patterns

- Design Patterns are “microarchitectures”
  - Not enforced system-wide

- Different arch. styles solve different problems
  - Can take a patterns approach (leverage experience)

**Pipe-and-Filter**

**Layered**

**Blackboard**
A journey comes to an end
Where we started

- We knew how to write code
- Knew how to use data structures
- Could write a small program for yourself
Where you are today

- Know the importance of risks, tradeoffs, and iteration
- Acquire requirements from a customer
  - Refine to remove assumptions
- Assemble an agile plan
  - Stories, scenarios, tasks
  - Iterations and Milestones
- Design your code for the long haul
  - So each iteration can build on the next
  - Deliver to your customer frequently
  - Adjust to changing conditions
- Build on frameworks to realize amazing functionality
- Schedule, test, and deliver with confidence
  - It works, and it's on-time
  - OK to cut lower priority features!
- Work in a team, not just alone