go over midterm

4156 Thursday 10/31/13 page 1

test-driven development (TDD)
not required for this class, but
it you did it that's great

write test cases from user stories or
use cases FIRST, before writing code
fine-grained
low-level unit tests, so start with
some very small part of user story

test case may not even compile
since refers to classes, objects,
methods, etc. that do not exist
so write simplest skeleton code to
get it to compile - but still fail

measure of success
then add simplest code that will
allow test case to pass
in worst case making up design as
you go along - "evolutionary design"

one test case at a time,
not set of test cases
so maybe just a test case to
create an empty object
then more, one at a time, for
setters + getters (in book they do several in one test)

build class diagram as you go

write simplest code to get each
test case to pass
resist urge to add other functionality
you might need in future

YAGNI principle
"you ain't gonna need it"

cycle -
red - test fails
green - test passes w/ simplest code
refactor - clean up duplication,
ugliness, old code, etc.

covered later in course

good habits
- each test should verify one thing
- avoid duplicate test code
- keep test cases in mirror
directory to source code
(use version control)

book mentions pulling out common
code into setup/teardown methods

what is another reason we need
setup/teardown? (in general,
not just TDD)

rely on setup state Not result of a previous test

setup - need proper context for running unit test

teardown - need to clean up before running next test

sometimes had to do - may need clear JVM or reboot the OS

dependencies on other parts of system ~~as~~ (which might not exist yet)
or on external system(s)

need to simulate ~~emulate~~ those other parts as simply as possible & fill in later - ~~[illegible]~~
test independent of dependencies
too tightly coupled

Use mock objects & multiple implementations of same interface

→ ~~coupling & cohesion (book doesn't talk about much)~~

~~discuss~~ ~~cover~~ later in context of ~~refactorings~~

after test cases showing that basic functionality works
next need test cases to handle edge cases - when the basic functionality shouldn't work, error case, or needs to do something special

4156 Thursday 10/31/13 page 4

book talks only a little bit about
coupling & cohesion
- so read other materials

want to achieve
- keep things that have to change together as close as possible in code
- allow unrelated parts of code to change independently
- minimize duplication (don't repeat yourself)

sort of single responsibility principle

how do I know where new code should go? higher cohesion

how do I know when I've put code in the wrong place? lower coupling

coupling relationship among classes

code is hard to understand when different concerns are mixed
"separation of concerns"
any changes will ripple through the system
limits reuse in other contexts

"code smells" signs that ~~th~~ something is wrong

divergent changes
feature envy
shot gun surgery
too tightly coupled

4156 Thursday 10/31/13 page 5

cohesion
relationship w/in class

how closely related are the responsibilities, data & methods of a single class
put code where you expect to find it - compare roles to class name

~~eliminate inappropriate intimacy~~

~~driver~~ "code smells"
signs of low cohesion or tight coupling

- divergent changes
- feature envy
- Shotgun surgery
~~Eliminate~~ inappropriate intimacy

Law of Demeter violated - only talk to ~~immediate~~ immediate friends (not their friends)
Tell, Don't Ask
Say it once & only once

show draft assignment from last year