Thoughts on Dennis Lee’s talk
MidQuarter Feedback Survey Results
What is working for you?

General Themes

- **Projects**
  - Freedom of direction, planning, creation*, design
  - Ability to set own team structure, schedule
- **Industry examples** - Cray, guest speakers
- **Class interaction**, class exercises
- **Grading based on the process**, not just the project
- **Reasonable workload**, breadth of topics
- **Relating essay questions to readings**

*Converse also noted but to a lesser degree*
What could be improved?

General Themes

- High amount of reading material
- Lecture material doesn’t always relate to project

Also

- Scope and mystery of final exam
- Time in class to work with group
High amount of reading material

- Is it the *amount* of material, or the *relevance* to your class project? Or both?

- **On amount:**
  - I’ll try to specify important sections vs whole chapters, and be clear on what is required vs applicable/interesting
  - No additional paper reading for the second essay

- **On relevance:**
  - A goal for 403 is to not only to well engineer a class project, but to be prepared for projects in industry, of different shapes and sizes and teams. The readings (and lectures) may at times, serve the latter goal.
Lecture material doesn’t always relate to class project

- I’ll try to better motivate the material in terms of your project
- Caveat, some things you won’t experience in a quarter project, that are still important for us to cover

I’d like us to work together on this one
The Final

- Final will be inclass, open textbook (no electronic access, no notes)

- Questions will largely be short answer,
  - create an example to illustrate something
  - justify a statement
  - reflect on your project experience
  - recognize a scenario

Question study tips:
  - Assignment essay questions
  - Questions in yellow during lectures
  - Old 403 finals (Summer 2005)
  - Guest lectures

Would you like a class to discuss potential finals questions – you bring the questions, we discuss as a class how they could be answered
Class time for group meetings

- I pledge to stop by \textit{at least} 11:10 to allow you to connect with your group in class

- I need you to cut me off at 11:10 if class is going late!
Thanks for the feedback (42%)!

PERFORMANCE REVIEW

YOU DID TWO JOBS FOR A YEAR AND DID THEM WELL.

I HAVE NO BUDGET FOR RAISES, SO ALL I CAN OFFER IS AN ATTABOY.

THE PROBLEM IS: I DON'T WANT TO CHEAPEN THE WHOLE ATTABOY SYSTEM.
Pragmatic programmer tip: Eliminate Effects Between Unrelated Things – design components that are self-contained, independent, and have a single, well-defined purpose
Outline

- Where does design fit?
- What are desirable characteristics of a design?
- How do we attack the design process?
How do we attack the design process?

Treat design as a wicked, sloppy, heuristic process. Don’t settle for the first design that occurs to you.

Collaborate.
Strive for simplicity.
Prototype when you need to.
Iterate, iterate, and iterate again.

-- Steve McConnell, *Code Complete*
Heuristics help provide ideas

- There is no “right answer” with design
- Applying effective heuristics can provide insights and lead to a good design

- We’ll look at 9 heuristics – there are many!
  1. Identify objects
  2. Form consistent abstractions
  3. Encapsulate implementation details
  4. Favor composition over inheritance
  5. Hide information
  6. Keep coupling loose and cohesion strong
  7. Identify areas likely to change
  8. Use design patterns
  9. Consider testability
H1: Identify objects

Object oriented design

- Identify objects and their attributes
- Determine what can be done to each object
- Determine what each object is allowed to do to others
- Determine what parts are visible and hidden
- Define the public interface

Iterate
H2: Form consistent abstractions

Abstractions allow you to simplify the problem and ignore irrelevant details.
Abstractions in code

• Interfaces: a good class interface allows you to focus on the interface without needing to know the internal workings

• Base classes: Base classes abstract the common attributes of the derived classes – leaving the specifics to later
### Some examples of what not to do

```cpp
class Square {
    double lower_left_x_coord;
    double lower_left_y_coord;
    double lower_right_x_coord;
    double lower_right_y_coord;
    ...
};
```

**Why?**

```cpp
BaseContext::myname() {
    return name;
}
```

**Why?**

```cpp
GenericName::myname() {
    return basename->myname();
}
```

**Why?**

```cpp
SpecificName::myname() {
    return genericname->myname();
}
```
H3: Encapsulate

Abstraction = look at the object with a high level of detail

Encapsulation = don’t look at the object at any other level

When decomposing a system (creating abstractions, encapsulating knowledge) begin with a list of difficult design decisions or design decisions which are likely to change

Parnas
H4: Favor composition over inheritance

- **Inheritance** promotes abstraction and reuse but breaks encapsulation
- **Composition** allows flexibility, decouples concepts, binds decisions late (at runtime), fosters reuse

Can you think of a cost of composition vs inheritance?
H5: Hide information

- More motherhood and apple pie…
- Share the interfaces, hide the construction details – allowing changes to affect a class but not beyond it!

- Moreover – design to interfaces (not details)!

![Diagram showing abstraction and details]
H6: Keep coupling loose

- **coupling** assesses the kind and quantity of interconnections among modules

- Modules that are loosely coupled (or uncoupled) are better than those that are tightly coupled

Why?

The more tightly coupled are two modules, the harder it is to work with them separately, and thus the benefits become more limited.
And cohesion, strong!

- *cohesion* refers to how closely the operations in a module are related

- Tight relationships improve clarity and understanding

- No schizophrenic classes!
Breakout session!

With your team

Come up with a code example of:
- Good encapsulation vs bad
- Inheritance vs composition
- Good coupling/cohesion vs bad

Suggestion: Split your team into 3 groups, each attacking one example
H7: Identify areas likely to change

A study of great designers found that one attribute they had in common was their ability to anticipate change (Glass 1995).

Common theme – consider change from the start:
- Identify items likely to change
- Encapsulate them
- Isolate them - design the interface to be insensitive to the potential changes
H8: Use design patterns

Learn and use design patterns
  o Represent distilled knowledge about good designs
  o Define a language to more effectively describe designs (e.g., façade, visitor, bridge, strategy, etc.)

Example:

Factory => encapsulate “new” with a routine

Allows change of “new” behavior without changing all call sites

- “Design Patterns”, Gamma Helm, Johnson, Vlissides
- “Design Patterns Explained – A New Perspective on Object-Oriented Design”, by Alan Shalloway and James Trott
H9: Consider testability

- Loose coupling and strong cohesion favor testability
- If it’s too hard to test, something could be improved
- Writing tests first can help define the interfaces!

Pragmatic Programmer Tip: Design to Test
Start thinking about testing before you write a line of code.
The list goes on…

We could spend a whole quarter talking about design

- **Single responsibility principle** (a class should have only 1 reason to change)
- **Open-closed principle** (a class should be open for extension, closed for modification)
- **No redundancy principle** …

Follow good practices, read good code, read good books, get feedback on your designs, iterate, iterate, iterate!