

Lecture Notes  
November 13, 2018

Many of you failed to submit the actual bug reports for the first pair [Bug Hunt assignment](#). Please submit them now as an upload for [this assignment](#).

Static analyzers are tools that automatically detect style violations, code smells, and generic bugs. In "code review", one or more *humans* read and analyze the code

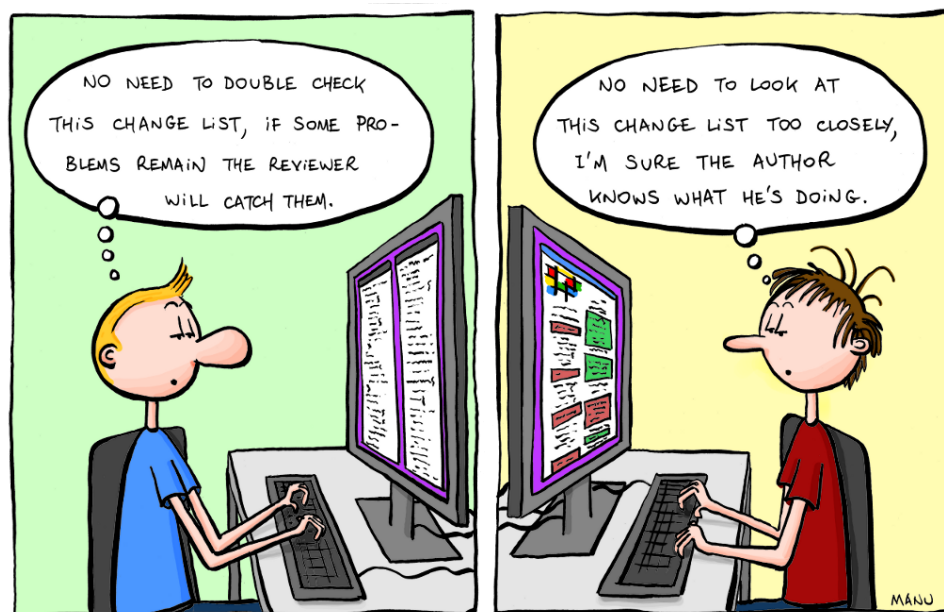
Like automated static analysis, human code review can potentially consider *all* inputs whereas testing can only consider *some* inputs - probably a very small subset of all inputs. Humans can spot application-specific bugs, as well as generic bugs that are beyond the state of the art in static analysis

Instant code review happens on the fly during pair programming, where the navigator reviews while the driver writes/edits

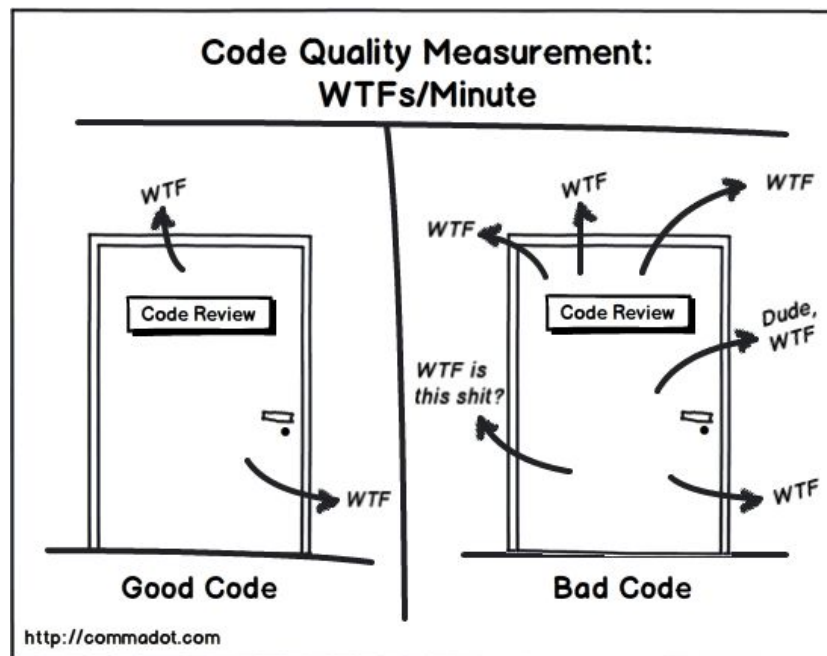


Many organizations require developers to submit their code for review by a peer or a designated reader prior to committing to the shared code repository - may be synchronous (reviewer sits with coder immediately after coder "done") or asynchronous (separately, often using code review tools)

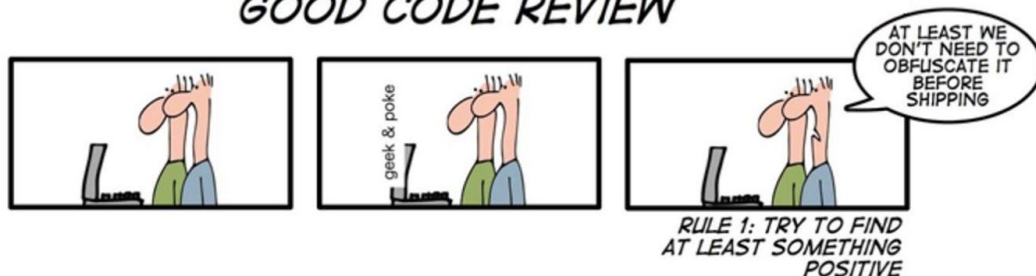
In theory, the developer and the reviewer make independent mistakes



Code review can find problems that dynamic testing and static analyzers cannot, such as code that passes style checking and other static analysis, and passes all test cases, but is still unreadable



### *HOW TO MAKE A GOOD CODE REVIEW*



What does this code do?

[Obfuscated C](#)

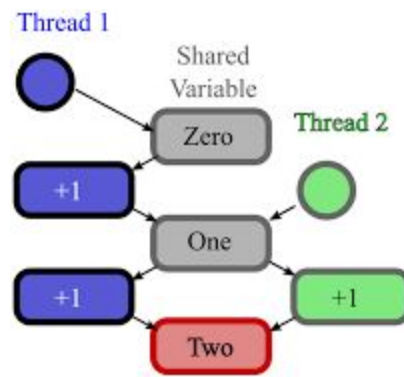
[Spoiler](#)

Code might reflect subtle assumptions that might not be violated in testing environment but will eventually occur in production

What is wrong with this code?

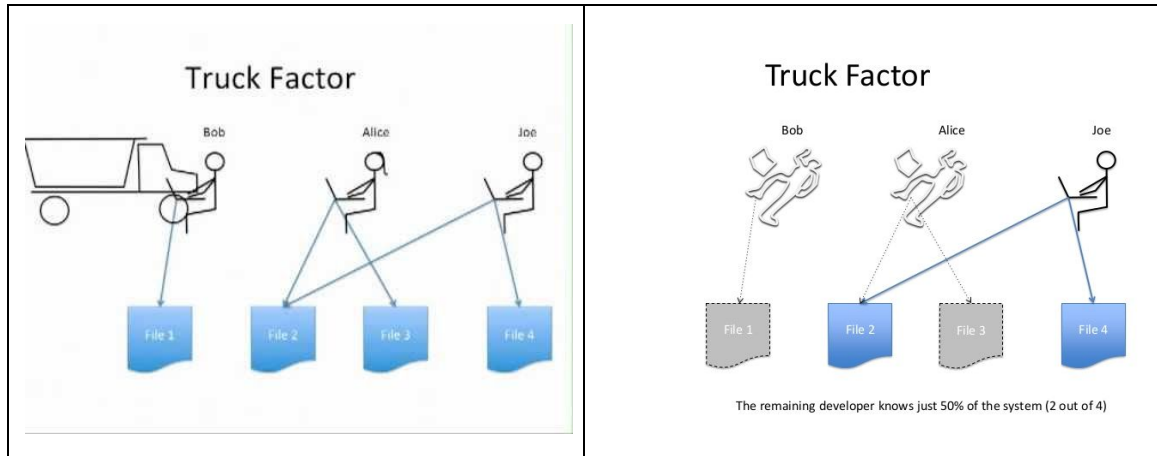
```
public boolean depositAmount(double amount) {  
    if (amount < 0) {  
        return false;  
    } else {  
        AccountBalance = AccountBalance + amount;  
        return true;  
    }  
}
```

```
public boolean withdrawAmount(double amount) {  
    if (amount > AccountBalance) {  
        return false;  
    } else {  
        AccountBalance = AccountBalance - amount;  
        return true;  
    }  
}
```



Race Condition!

Helps cross-train developers on code written by other people - minimize truck factor



Helps train junior developers - over-the-shoulder synchronous review by senior developer





*Lightweight* forms of code review involve one or two readers besides the coder

*Formal* code review (or [Fagan inspection](#)) is heavyweight - involves a *team* review process

Mandated for safety-critical software, or any [software that must work right the first time and every time](#), very rarely (never?) used for conventional business and consumer software aside from training purposes

Predefined roles:

- Moderator - runs meeting, often from outside development team such as another team or a separate quality assurance group
- Recorder - takes notes
- Reader - sometimes author, sometimes intentionally not the author (who may not find code intent as obvious as the author). Reads the code aloud, walkthrough line by line, explain along the way
- Everyone else is called an Inspector - attendees might include customer representatives, end-users, testers, customer/technical support, etc.

Multiple steps: Planning, Overview, Preparation, Inspection Meeting, Rework and Follow-up. Define output requirements for each step upfront and when running through the process. Check the output of each step and compare it to the desired outcome. Decide whether to move on to the next step or still have work to do in the current step

Schedule meeting and distribute specific code to be reviewed in advance - code should already compile, pass static analysis and unit tests (don't waste human time for work that can be automated)

Inspectors are supposed to prepare before the meeting by reading code on their own

Write report afterwards summarizing problems found, in addition to bug reports and change requests - helps identify problem code region or common problems across different parts of codebase

Problems classified as Minor - author is trusted to fix on own; Moderate - moderator checks; Severe - need another review meeting

Static analysis tools and human code review can detect code smells



*Refactoring* is a systematic approach to removing code smells = Modify code structure in very small steps that do not change any functionality (semantics-preserving)

Example refactoring patterns: see <https://sourcemaking.com/refactoring>

Refactoring tools sometimes detect code smells and suggest code changes - or even automate the edits, often implemented as an [IDE plugin](#)

Need to re-run affected test cases (or entire test suite) after every small change to verify that there were no functional changes

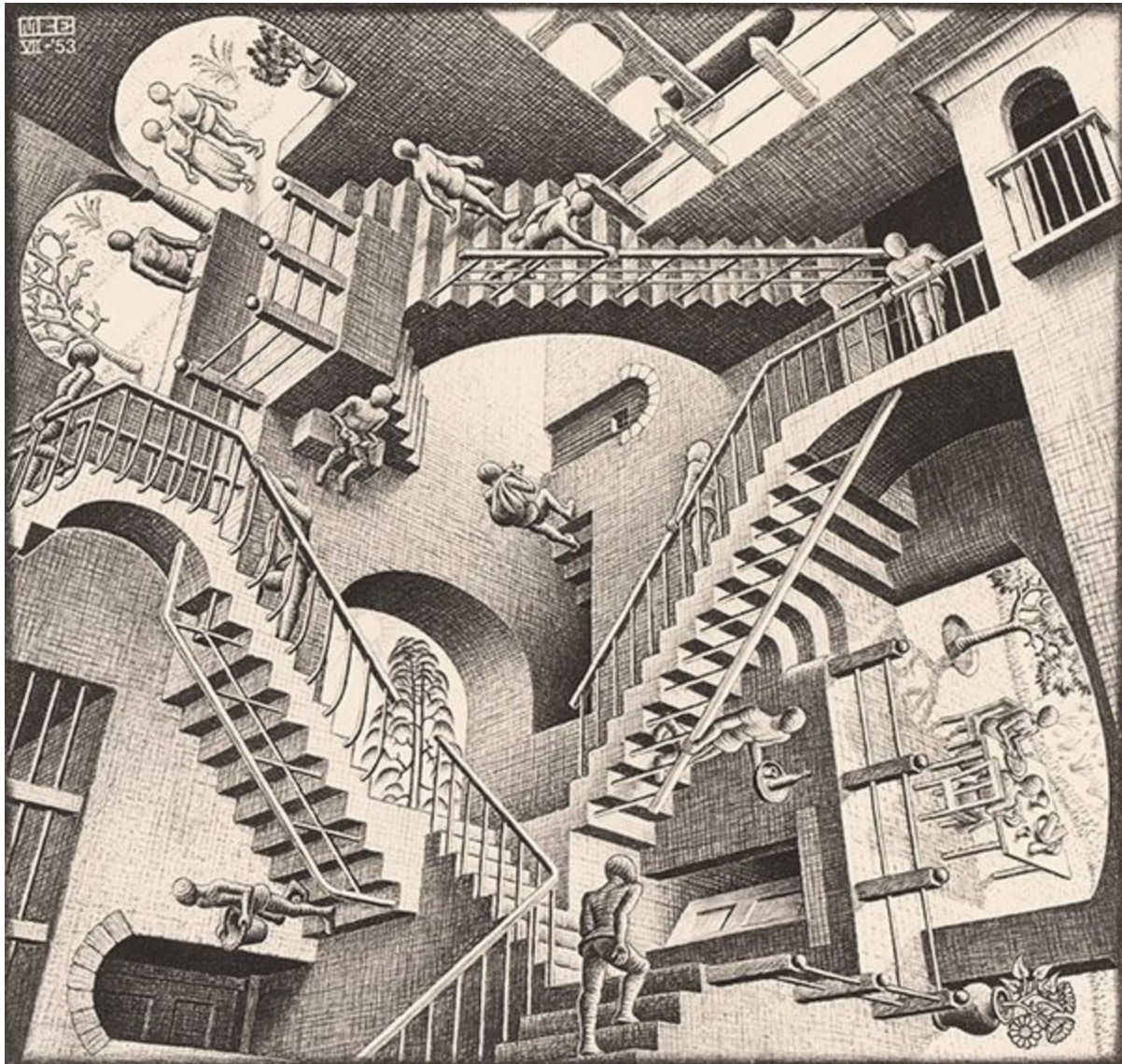
But if any of the refactoring changes cross unit boundaries, or introduce new units/remove old units, then we cannot expect the exact same unit tests to pass - need new unit and integration tests (i.e., refactor the test suite)

How do we know which test cases might be affected by refactoring?

Sometimes the term "refactoring" is used to refer to *any* re-engineering of the code base even if behavior is not strictly preserved, e.g., replacing a data structure or algorithm, reorganizing a monolithic architecture into microservices. Refactoring in the large is more of a design process than a coding process




Here is one approach to design:



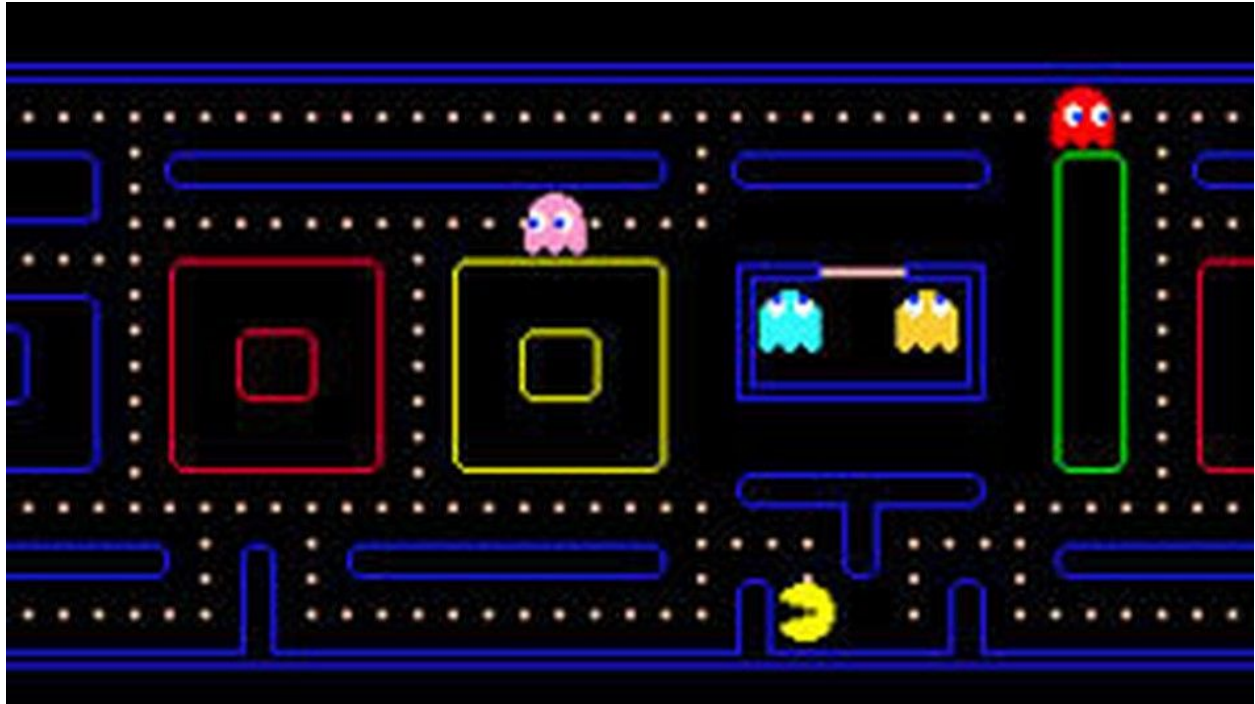
CRC Cards is an easier, more practical approach to design. CRC cards can be developed before writing the code, as part of planning, or after writing the code, as part of refactoring or documenting

Class Name	
Responsibilities	Collaborators

 pac_sprite.Ghost	
Responsibilities	Collaboration
<ul style="list-style-type: none"><li>• Draws a ghost.</li><li>• Moves the ghost along a random path through the maze.</li></ul>	<ul style="list-style-type: none"><li>• Sprite</li><li>• TileFactory</li><li>• TiledMap</li></ul>

Each class is a thing or entity (noun), each responsibility is some action it needs to do (verb), collaborators are other components it interacts with, communicates with, contains, knows about (noun)

The *Ghost* class displays a ghost and moves the ghost on a random path. The *Sprite* class gives a graphical entity the ability to display itself. The *TileFactory* loads graphics. The *TiledMap* generates the tiles that a ghost moves upon



Develop a stack of CRC cards for the current set of user stories. Classes and Collaborators do not need to correspond to code classes or types, can be modules, packages, libraries, even files, databases, devices

Then debug the CRC cards

- Classes are nouns that "do" operations, not the actors who initiate operations
- Split classes with too many responsibilities
- Combine classes with too few responsibilities
- Remove redundancy
- Avoid classes with only CRUD operations (create, read, update, delete)

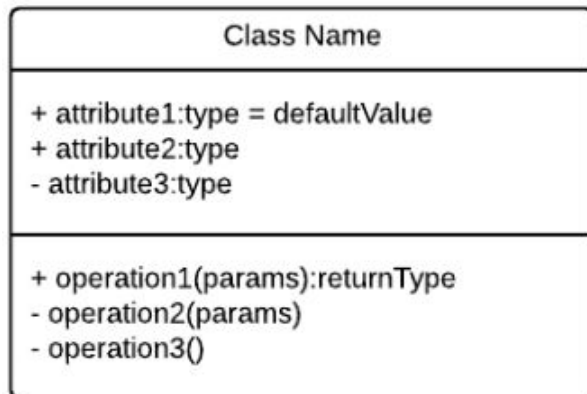
Walk through the scenarios of each use case and *animate* the CRC cards: move index cards on a table or sticky notes on a board. Make sure everything is handled by some class

**YAGNI** (you aren't going to need it) - only design for the *current* iteration, not classes, responsibilities, collaborators that may or may not come up later

After CRC cards cover every user story for the current iteration, next step is Class Diagrams - or just write the code



Class Diagrams convey information about *static structure* of your application domain - there are other kinds of [UML](#) diagrams for dynamic behavior



A class in a class diagram can be a type (that can be instantiated many times) or a singleton (no more than one). Applies to non-OO languages, however the language defines data types and singleton data entities

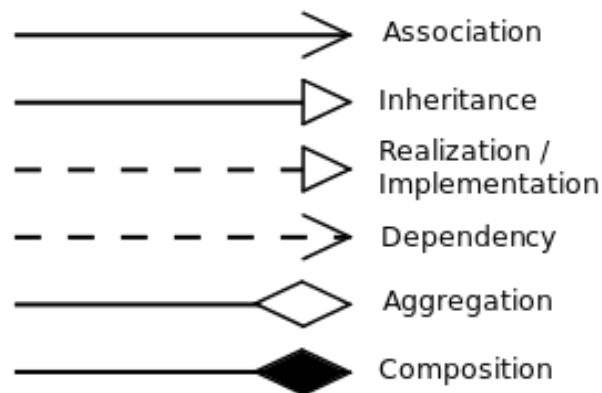
The attributes are properties of the object - may be primitive values or references to collaborators

The operations are responsibilities or behaviors

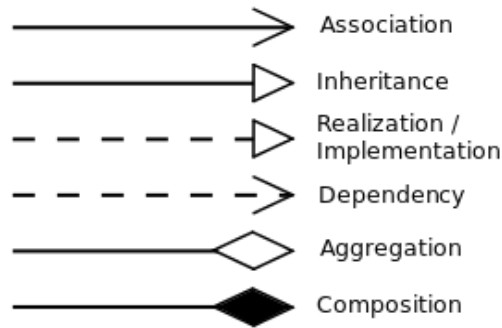
Usually omit CRUD

Some class diagrams show the structure of a single class, with its attributes and operations, while others focus on the *relationships* among classes

On CRC cards, *any* kind of relationship is fine for “collaborators”, but class diagrams refine the kind of relationship



**Association** is most general, just means instances of one class know about or communicate with instances of another class (or the same class). Unidirectional represented as line with arrow, bidirectional represented as just plain line

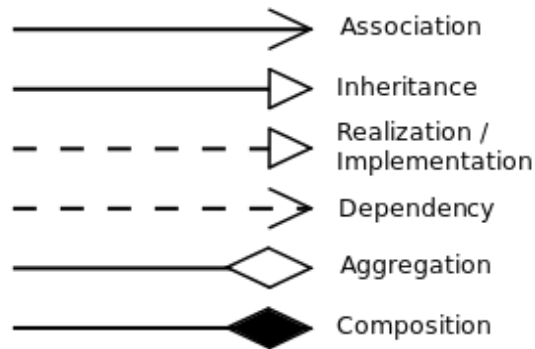


Aggregation is special case of association, corresponds to HAS-A, no lifecycle relationship. Children can have arbitrary number of parents (and be parent of their own parent). Line with open diamond at container class

Composition corresponds to CONSISTS-OF, strong lifecycle relationship = deleting parent automatically deletes all the children, or all the children must have already been deleted to delete the parent. Children can belong to only one parent, no cycles. Line with closed/filled diamond at container class

Both aggregation and composition may indicate multiplicity on corresponding end of line

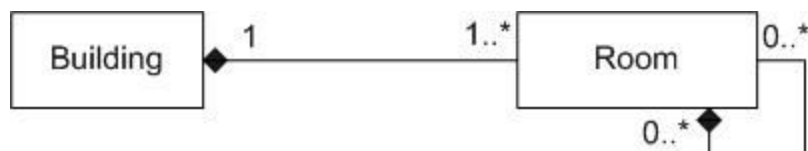
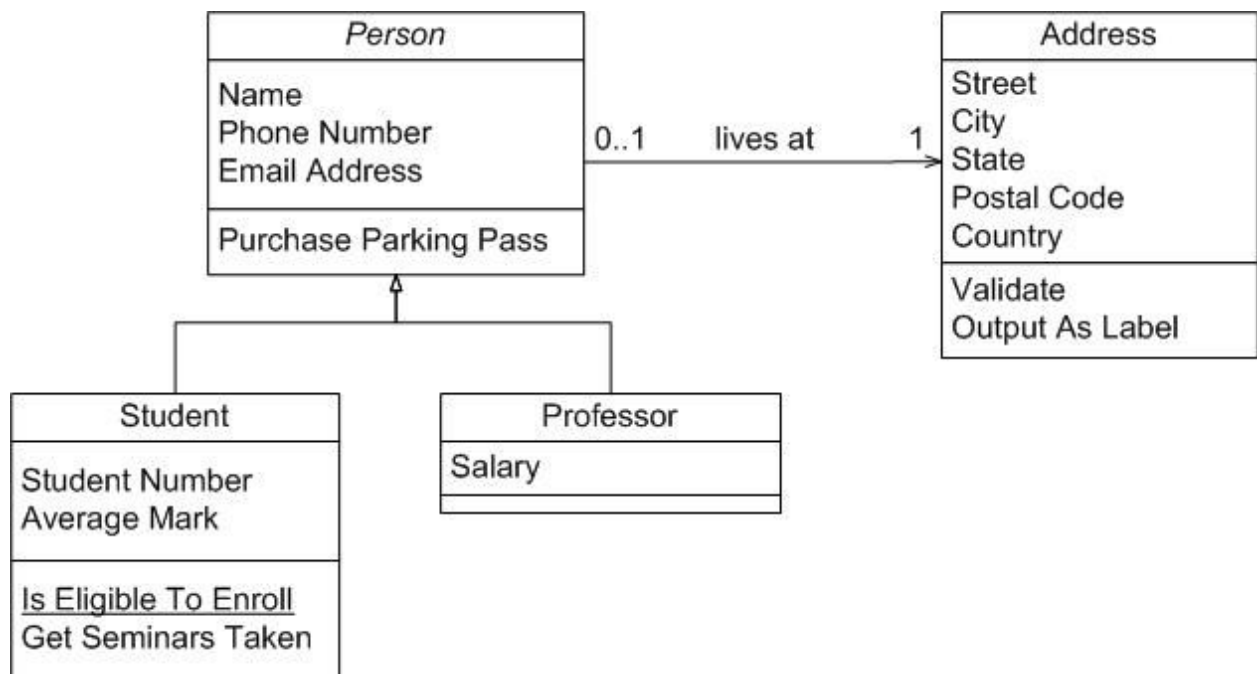
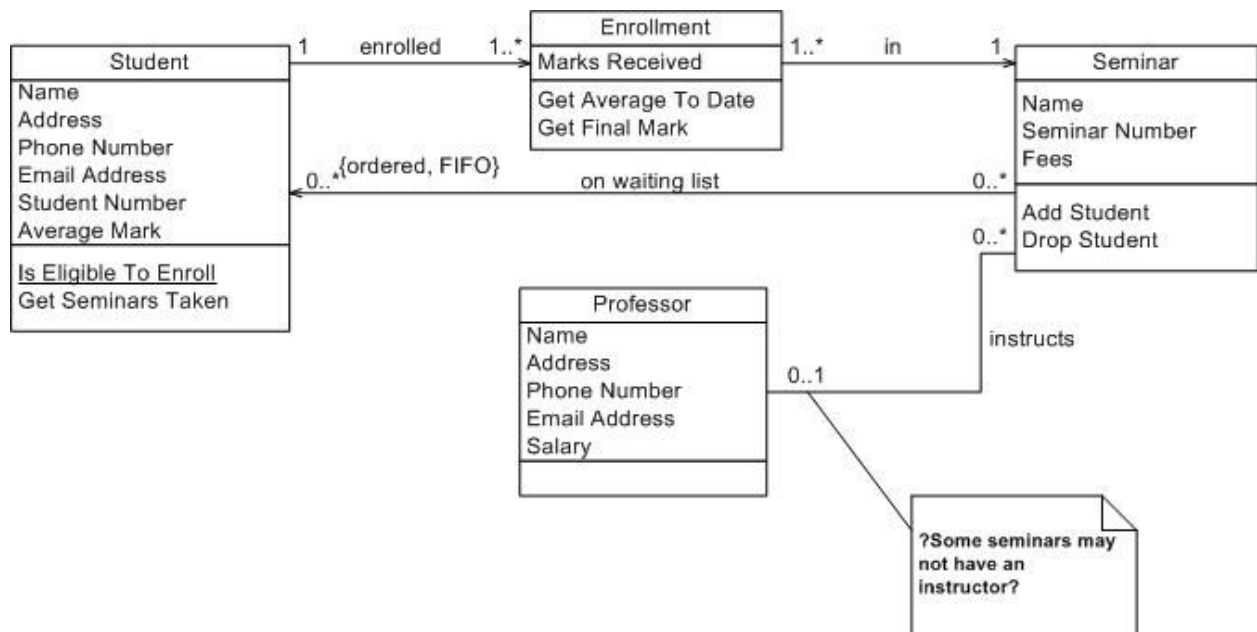
Indicator	Meaning
0..1	Zero or one
1	One only
0..*	0 or more
1..*   *   *	1 or more
<u>n</u>	Only <u>n</u> (where <u>n</u> > 1)
0.. <u>n</u>	Zero to <u>n</u> (where <u>n</u> > 1)
1.. <u>n</u>	One to <u>n</u> (where <u>n</u> > 1)



Generalization or Inheritance from superclass (abstraction, base class) to subclass (specialization, extension, derived class). Applies even to non-OO languages - generalized types and specialized types does not necessarily imply inheritance, polymorphism, etc. Solid line with open (unfilled) arrowhead from subclass to superclass

Realization or Implementation of an interface (abstract class). Dashed line with open arrowhead (triangle) from implementing class to interface

Various other kinds of associations ...



I do not expect teams to follow any particular approach to design for team projects this semester. If you submit CRC cards, class diagrams, or any other UML, for any pair or team assignment, you are doing it wrong



# DUAL SCREEN

You're Doing It Wrong

But they may appear on the second individual assessment...

Reading for Thursday:

- ★ [Martin Fowler's Yagni](#)
- ★ [Agile Design Principles](#)

Next pair assignment [Bug Hunt 2](#) due Tuesday,  
November 20, 10:10am

[Second Iteration](#) due Tuesday November 27, 11:59pm

[Second Iteration Demo](#) due Tuesday December 4,  
11:59pm

Second Individual Assessment will become available on  
December 4