



WE NEED TO MAKE 500 HOLES IN THAT WALL,
SO I'VE BUILT THIS AUTOMATIC DRILL. IT USES
ELEGANT PRECISION GEARS TO CONTINUALLY
ADJUST ITS TORQUE AND SPEED AS NEEDED.

GREAT, IT'S THE PERFECT WEIGHT!
WE'LL LOAD 500 OF THEM INTO
THE CANNON WE MADE AND
SHOOT THEM AT THE WALL.

HOW SOFTWARE DEVELOPMENT WORKS

Update: It turns out the cannon has a motorized base and can make holes just fine using the cannon's barrel as a battering ram. But due to design constraints it won't work without a projectile loaded in, so we still need those drills.

Class-Level Design

Discovering and Refining Classes

CSCI 2134: Software Development

Agenda

- Lecture Contents
 - Gathering Requirements
 - Identifying Classes
 - The CRC Method
 - Class refinement
- Brightspace Quiz

Readings:

- This Lecture: Chapter 6
- Next Lecture:

Scenario: Create a Replacement for Brightspace

- Dalhousie has decided to create their own system to replace Brightspace
- You have been asked to create it!
- After the initial shock has worn off, you ask the first question
- Where do I begin?

Requirements Gathering

- The first step of any software project is to figure out what it is that you are building
 - Functional requirements: What should it do?
 - Nonfunctional requirements: Other
- Requirements are gathered from:
 - RFP: Request for Proposal
 - Existing user base
 - Prospective user base
 - Management
 - Etc.
- A Software Requirements Specification (SRS) is a document that specifies the software to be built: all functional and non-functional requirements



The Ask

Students take courses at Dalhousie. In each course, students are given a variety of assessments, such as tests and assignments. These assessments have a weight associated with them as well as a score, reflecting how well the student did. The grade record for each student in the course is a list of all the assessments and the associated grades. Students should be able to view these grades.

Five Steps to Class Level Design

Task	Applicable SOLID Principles	Methodology
1. Identify classes/objects and their attributes	Single Responsibility Principle Open/Close Principle	Noun-Verb Mind Map (not discussed)
2. Identify operations on classes/objects	Single Responsibility Principle Open/Close Principle	Noun-Verb Mind Map (not discussed)
3. Identify interactions between classes/objects	Liskov Substitution Principle	CRC Method
4. Identify parts of classes/objects that should be public	Dependency Inversion Principle	Refinement (post CRC method)
5. Specify public interface (methods) of each class and interfaces that it implements	Interface Segregation Principle	Iterative refinement (next lecture)

Discovering Classes

We will need to create classes and objects that collaborate to meet the requirements

- What classes (and objects) we need?
- What do these classes and objects do?
- How do these classes and objects collaborate?

Goals:

- Identify what classes / objects we can reuse
- Identify what new classes / objects we need
- Identify what methods the classes / objects need
- Identify how the classes / objects interact

The Noun-Verb Approach

- Idea:
 - Use nouns from the problem domain to identify classes
 - Use verbs associated with the nouns to identify methods for the classes
- Example: The TimBot Simulation:
 - Nouns: DohNat (planet), District, TimBot, Plant, Danger
 - Verbs:
 - DohNat: **do** a phase, **add** a timbot, **add** a plant
 - District: **store** a timbot or plant, **do** a phase,
 - TimBot: **sense**, **move**, **shoot**, **shield**, **harvest**
 - Plant: **grow**, **yield** jolts
 - Danger: **do** damage



The Ask, with Nouns and Verbs Annotated

Students take courses at Dalhousie. In each course, students are given a variety of assessments, such as tests and assignments. These assessments have a weight associated with them as well as a score, reflecting how well the student did. The grade record for each student in the course is a list of all the assessments and the associated grades. Students should be able to view these grades.

Problem 1

Suppose we wanted to design the “View my grades” function for our Brightspace replacement:

- Propose 5 classes (Nouns) that would be involved in implementing this feature
- Propose verbs for each of the classes (as many as you believe are necessary)

Problem 1: Solution

- Suppose we wanted to design the “View my grades” function for our Brightspace replacement:
 - Propose 5 classes (names) that would be involved in implementing this feature
 - Propose verbs for each of the classes (as many as you believe are necessary)
- Possible Nouns:
 - Student, Course
 - Grade Record, Grade
 - Assignment, Test, Assessment
- Possible Verbs:
 - Student: **View** ...
 - Grade: **View, Assign**, ...
 - GradeRecord: **Add, Remove, View**
 - Assignment: **Score**, ...

Do's and Don't of the Noun-Verb Method

- Nouns should be concrete or conceptual from the problem domain:
 - E.g.,
 - Good classes: Student, Course, Grade, Grade Summary
 - Bad classes: Grade Sorter, Grade Summarizer
- Existing Classes may be named differently
 - E.g., Roster vs ClassList
- Avoid turning actions into classes
 - E.g., SortGrades
- Don't over-do. Decide when various data items in your program can be represented using basic types

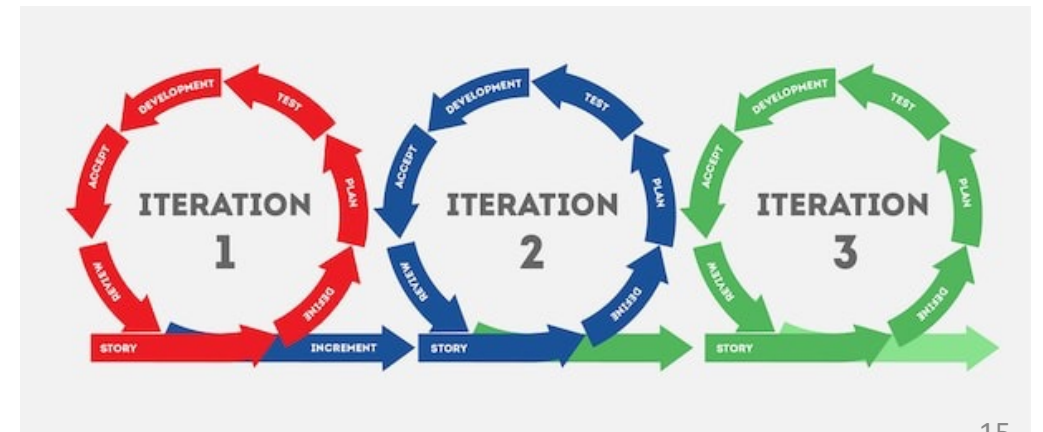


Keep Things Abstract

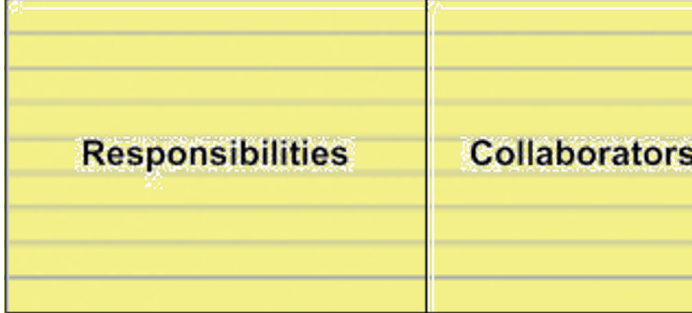
- **Key Idea:** Think of all classes as abstract
- Keep in mind:
 - Do NOT think about implementation
 - Most of these classes and methods will be refined
 - Most of these classes are general:
E.g., a *Student* class may have either *UndergradStudent* or *GradStudent* as concrete (or abstract) subclasses
- The purpose of Noun-Verb is to identify what classes we may need.
- Figuring out what classes do and how they interact is the next step.

What's Next?

- After identifying classes, we need to figure out
 - What they do: Responsibilities
 - How they interact: Collaborate
- This is an iterative process
 - We consider each class and ask the above two questions
 - We then revisit the classes we have looked at earlier and refine our answers
- We use the CRC method



The CRC Method

- CRC = **C**lass, **R**esponsibilities, **C**ollaborators
 - Idea:
 - Use an index card for each identified class
 - Divide card into three section:
 - Class name
 - Responsibilities : The verbs / methods that the class is responsible for implementing
 - Collaborators : Other classes / objects that will be used to implement the responsibility
 - Iterate through all the verbs / methods and add them to the responsibilities section of the respective class
 - Identify the collaborating classes
 - Ensure that collaborators provide the necessary methods in their responsibilities
- 
- | Responsibilities | Collaborators |
|------------------|---------------|
| | |
| | |
| | |
| | |
| | |

Class Name	
Responsibilities	Collaborators

Keep the Single Responsibility Principle in Mind

- If a class has unrelated responsibilities, split it up
 - If a class has no responsibilities, get rid of it
 - If a class has specialized responsibilities of another class this may indicate inheritance
 - A class can have multiple specific responsibilities, but they should all be related to one primary responsibility
 - Example: *TimBot* represents a timbot in the simulation
 - Perform a phase
 - Keep track of its energy level
 - Decide what to do in each phase
 - This is an exploration stage so don't worry about making a bad decision
- | Class Name | |
|------------------|---------------|
| | |
| | |
| | |
| Responsibilities | Collaborators |
| | |
| | |

Class Name	
Responsibilities	Collaborators

Problem 2:

Use the CRC method for the following classes from Problem 1

Student	
<u>Responsibilities</u>	<u>Collaborat.</u>

Grade	
<u>Responsibilities</u>	<u>Collaborat.</u>

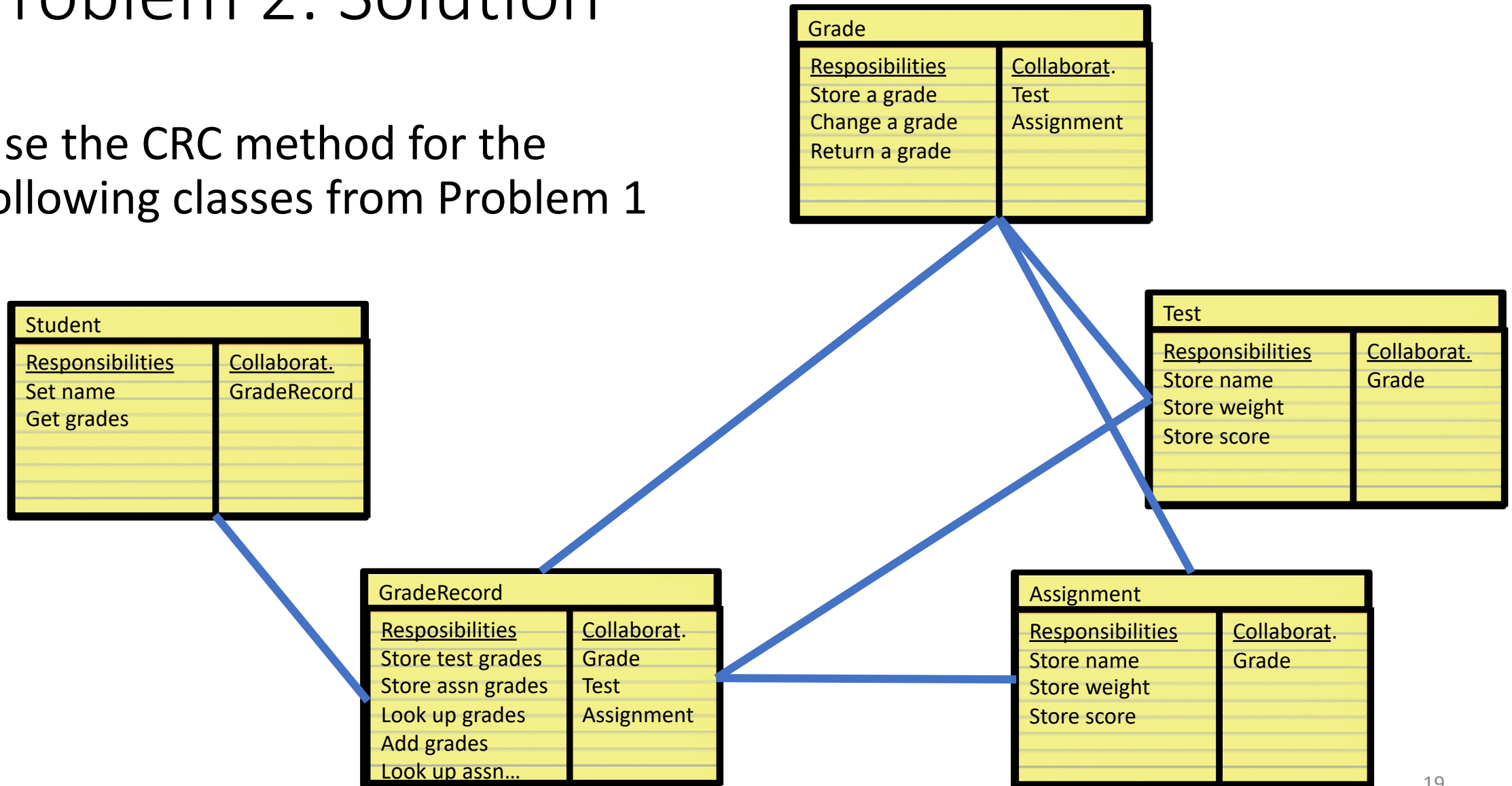
Test	
<u>Responsibilities</u>	<u>Collaborat.</u>

Grade Record	
<u>Responsibilities</u>	<u>Collaborat.</u>

Assignment	
<u>Responsibilities</u>	<u>Collaborat.</u>

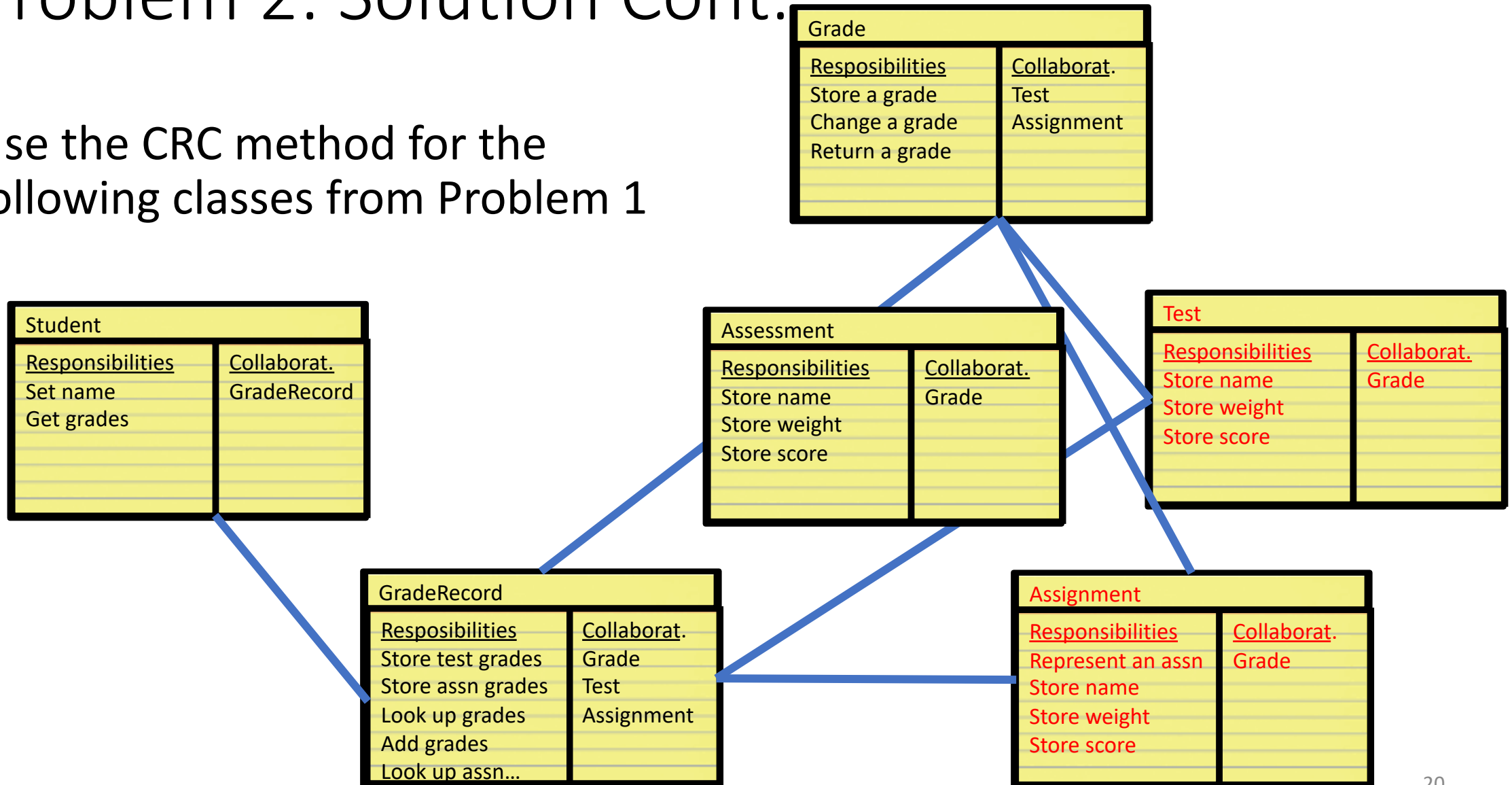
Problem 2: Solution

Use the CRC method for the following classes from Problem 1



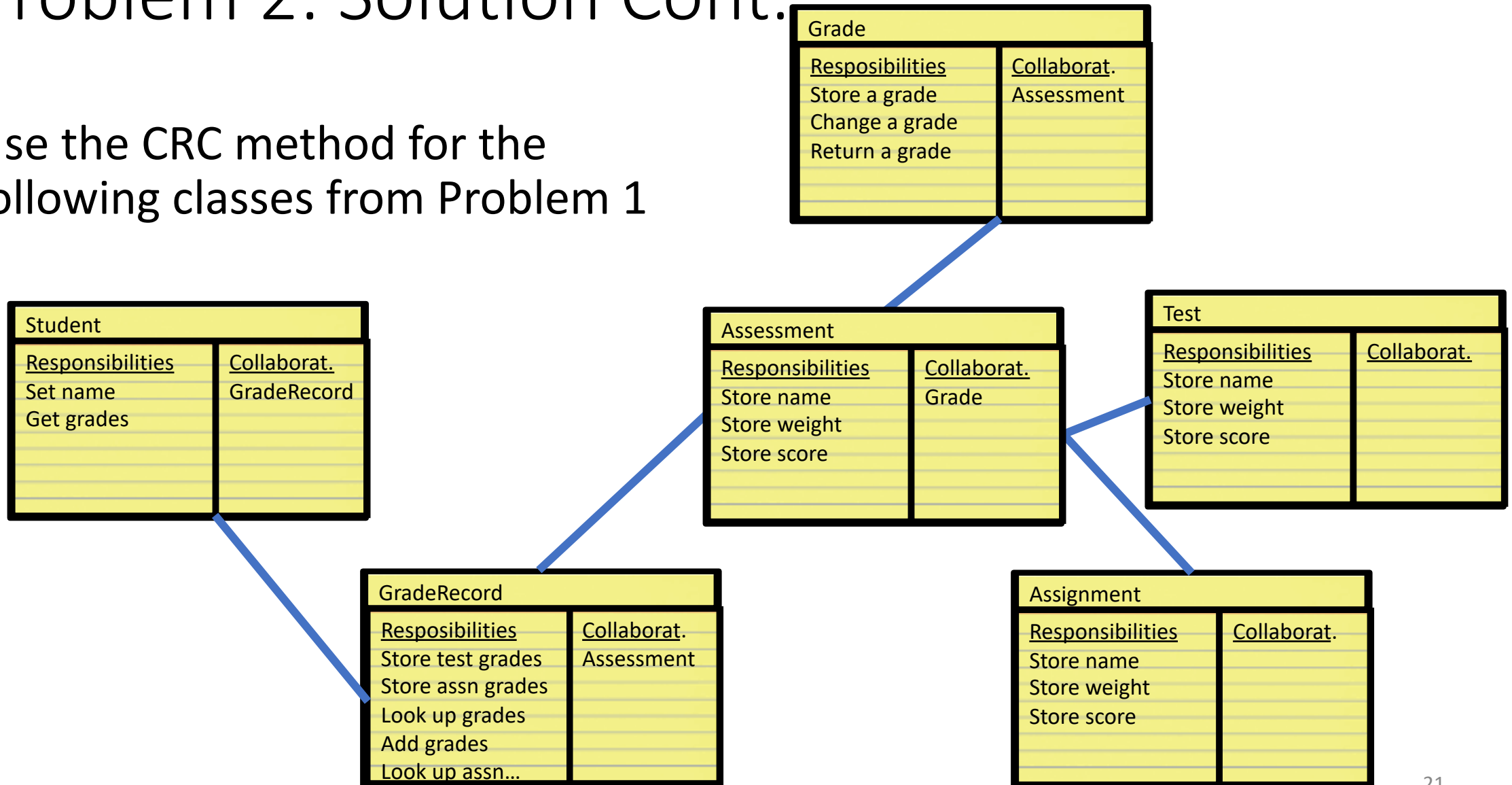
Problem 2: Solution Cont.

Use the CRC method for the following classes from Problem 1



Problem 2: Solution Cont.

Use the CRC method for the following classes from Problem 1

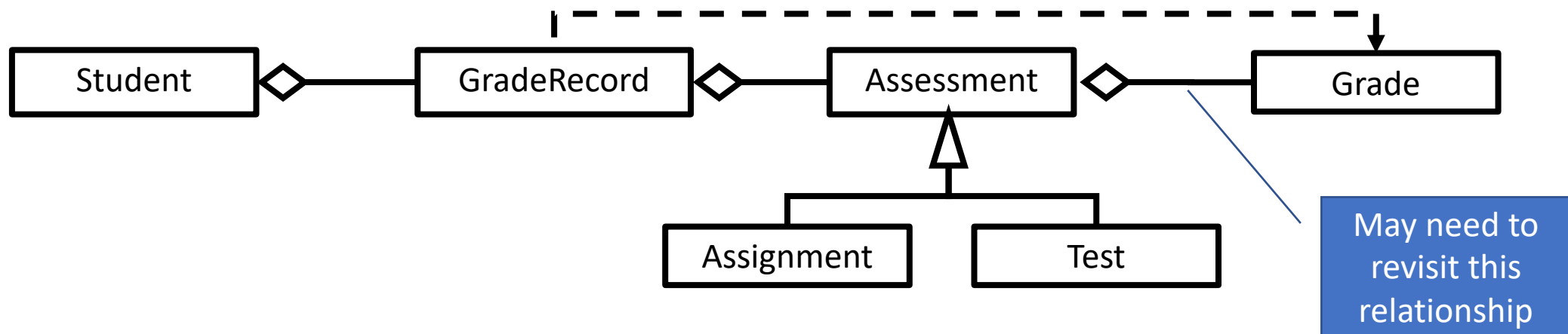


Problem 3

Draw a UML diagram with all the relationships between the classes from Problem 2

Problem 3: Solution

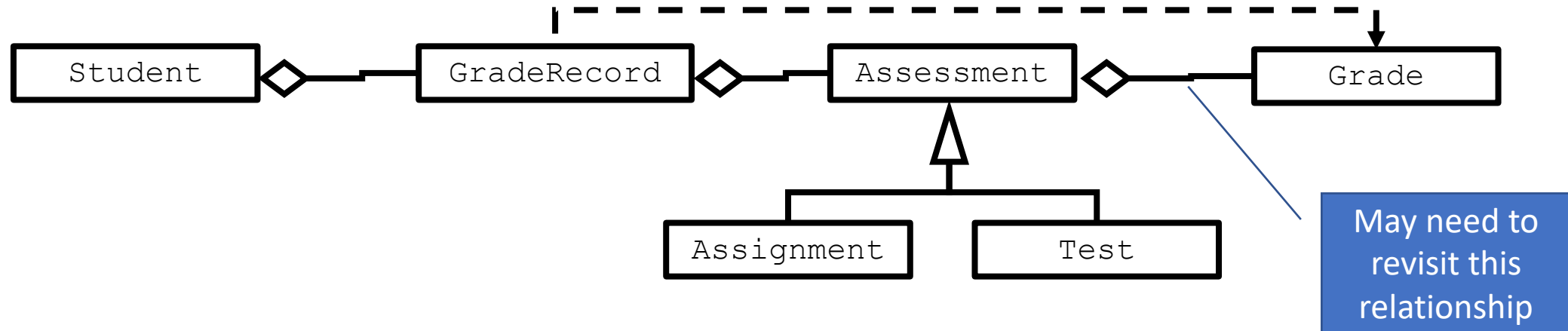
Draw a UML diagram with all the relationships between the classes from Problem 2



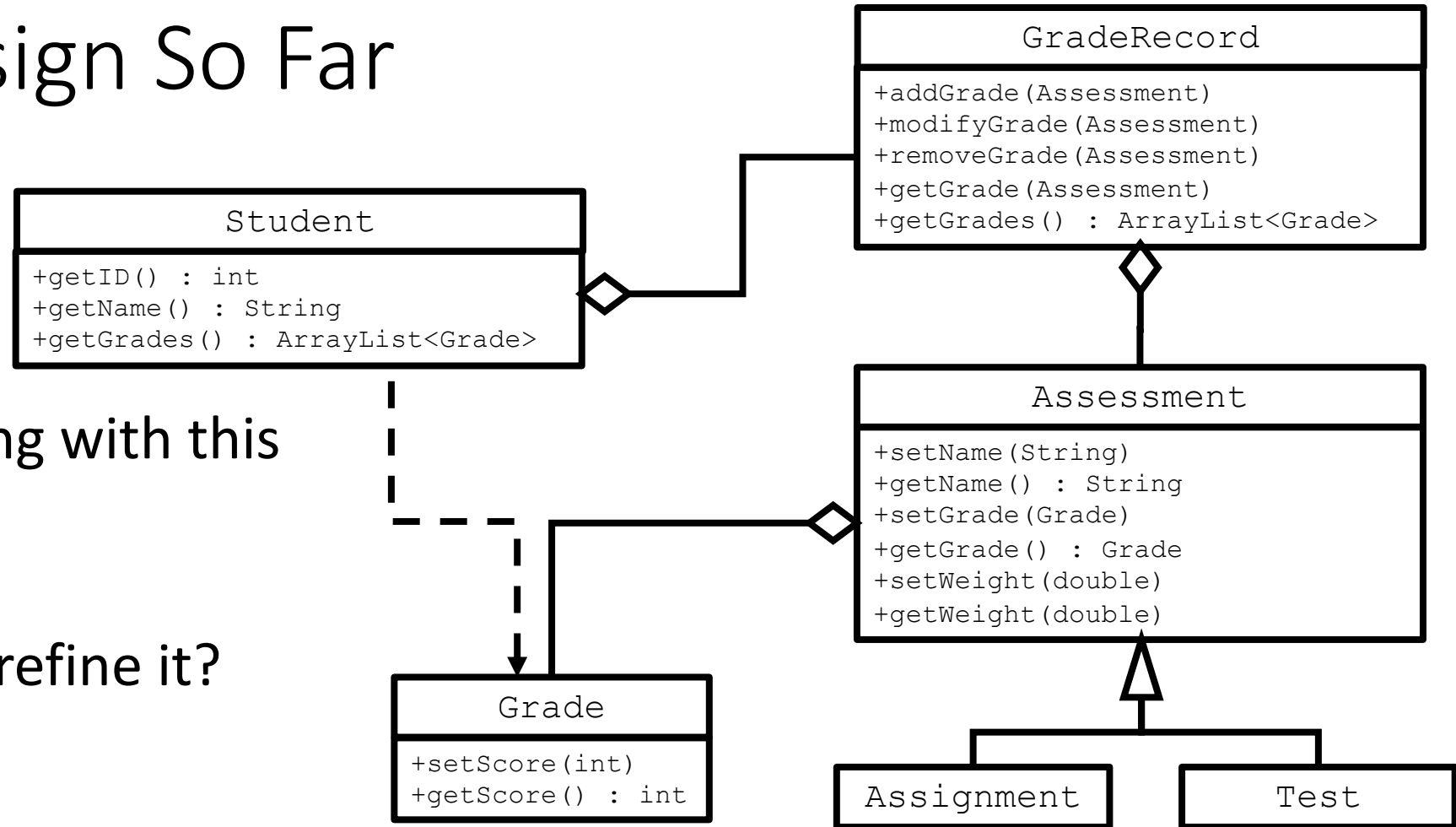
Five Steps to Class Level Design

Task	Applicable SOLID Principles	Methodology
1. Identify classes/objects and their attributes	Single Responsibility Principle Open/Close Principle	Noun-Verb Mind Map (not discussed)
2. Identify operations on classes/objects	Single Responsibility Principle Open/Close Principle	Noun-Verb Mind Map (not discussed)
3. Identify interactions between classes/objects	Liskov Substitution Principle	CRC Method
4. Identify parts of classes/objects that should be public	Dependency Inversion Principle	Refinement (post CRC method)
5. Specify public interface (methods) of each class and interfaces that it implements	Interface Segregation Principle	Iterative refinement (next lecture)

Our Design So Far



Our Design So Far



What's wrong with this design?

Plenty!

How do we refine it?

We'll see!

How do we refine our initial set of classes?

- Apply SOLID properties
- Create abstractions where possible
- Apply design heuristics

Design Heuristics

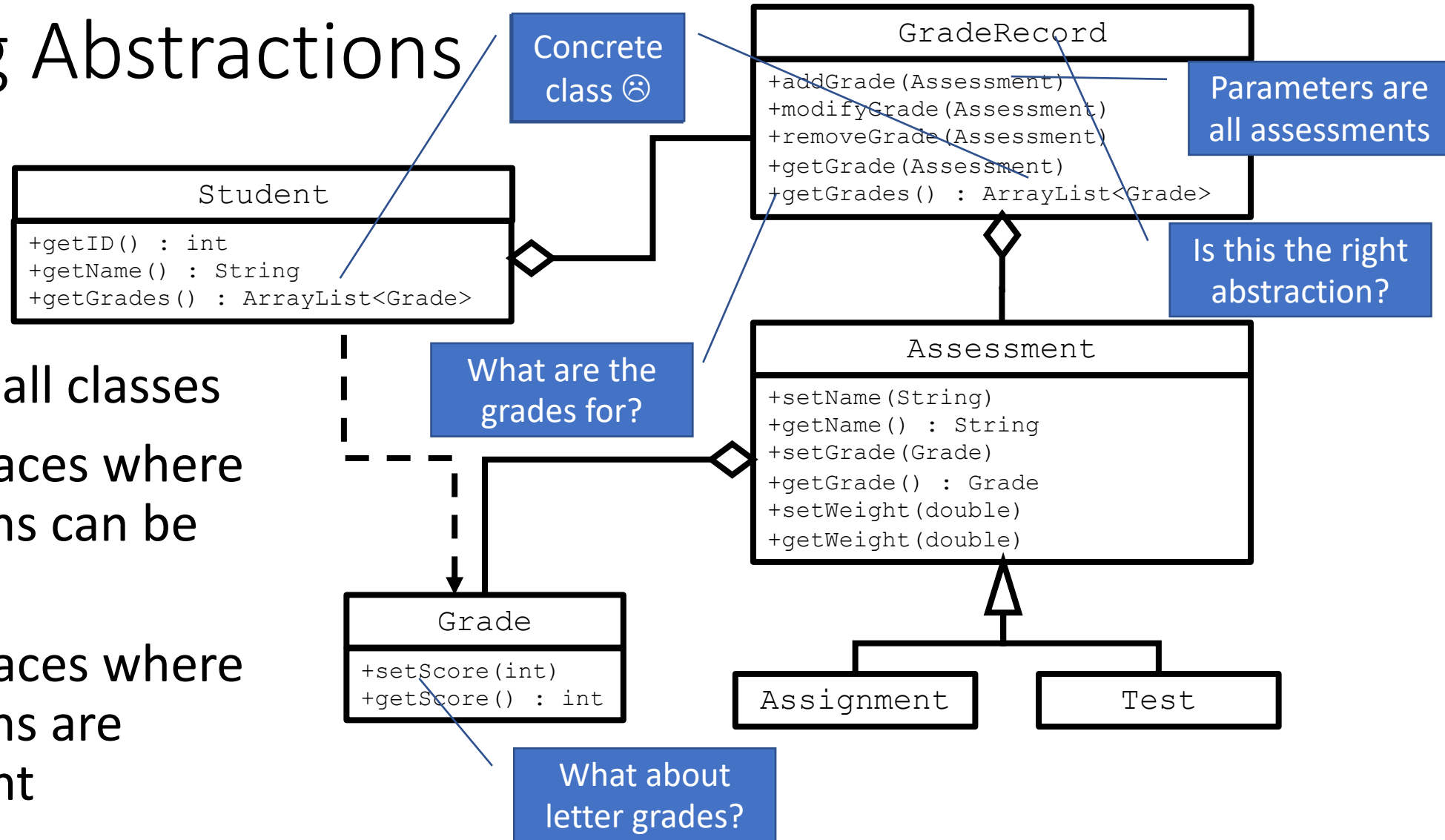
McConnell, “CC2”, 2004

- Look for common design patterns
- Identify areas likely to change
- Keep coupling loose
- Form consistent **abstractions**
- **Encapsulate** implementation details
- Use **inheritance** – where appropriate
- **Hide** secrets

Form Consistent Abstractions

- **Idea: Abstractions are intended to reduce complexity by hiding details and minimizing coupling**
- Abstractions should be at the right level of detail:
 - A *GasPedal* class should not collaborate with a *SparkPlug* class, but may collaborate with the *Engine* class
 - A *Course* class should (probably) not collaborate with a *Grade* class, but may collaborate with an *Assessments* class
- Abstractions should be consistent with their responsibilities and collaborator
 - An *Assignment* class should not collaborate with a *Lecture* class but should collaborate with a *Dropbox* class.
 - A *BrakePedal* class should not collaborate with a *SteeringWheel* class but should collaborate with the *BrakeLight* class
- Use interfaces and abstract classes whenever possible
 - Use *List* instead of *ArrayList*
 - Use *AbstractDanger* class instead of *Tarpit* or *CoffeeAddict* classes (Lecture 19).
 - This is the Dependency Inversion Principle!
- The goal of abstractions is to make our design simpler
 - If an abstraction does not fit, or is not used properly, we need to refine our design

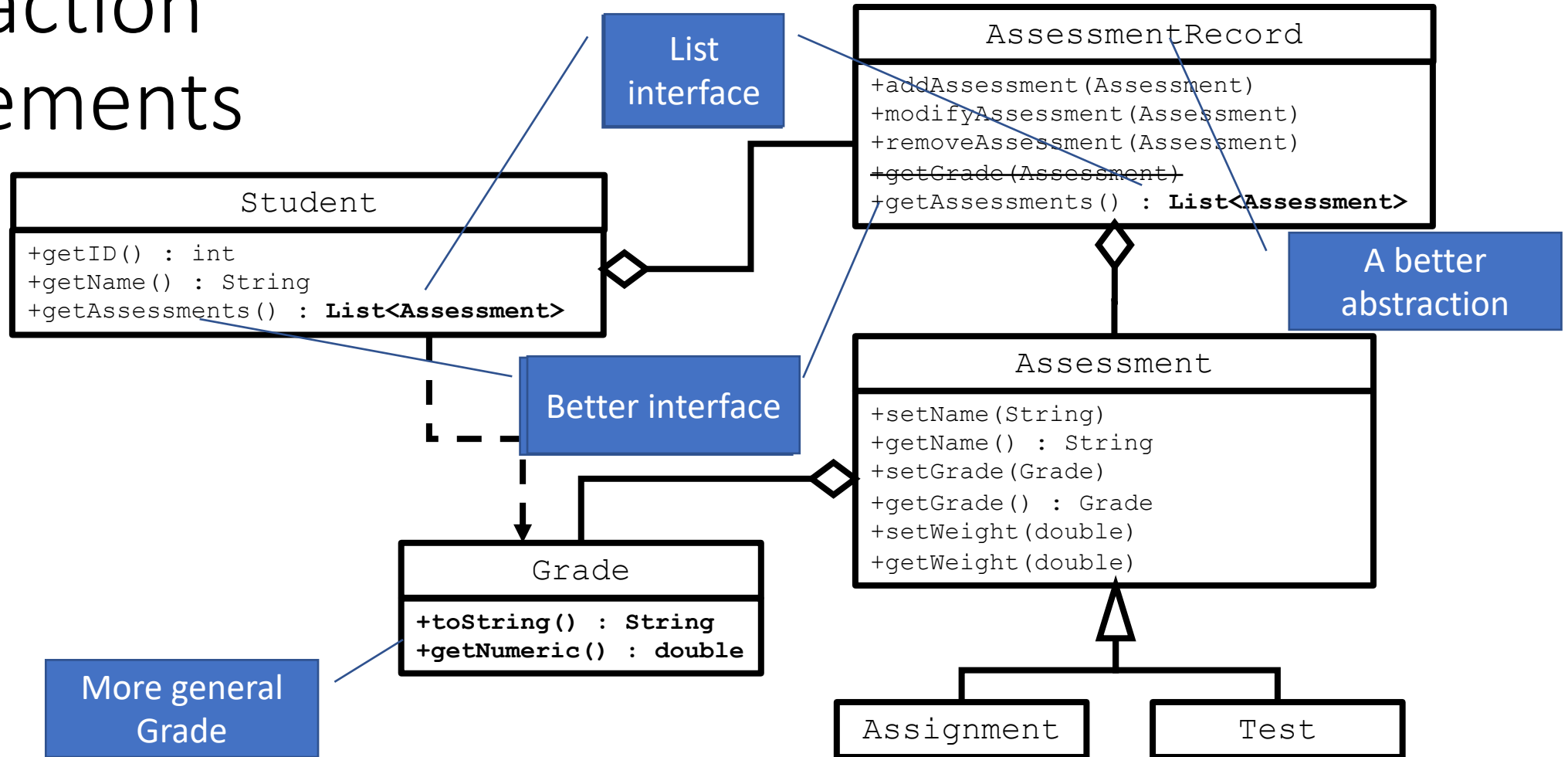
Refining Abstractions



Iterate over all classes

- Identify places where abstractions can be used
- Identify places where abstractions are inconsistent

Abstraction Refinements



Encapsulate Implementation Details

Abstraction is about looking at things at a high-level (not worrying about the details)

- I.e., You do not **need/want** to look at the implementation/details

Encapsulation is about preventing access to the details

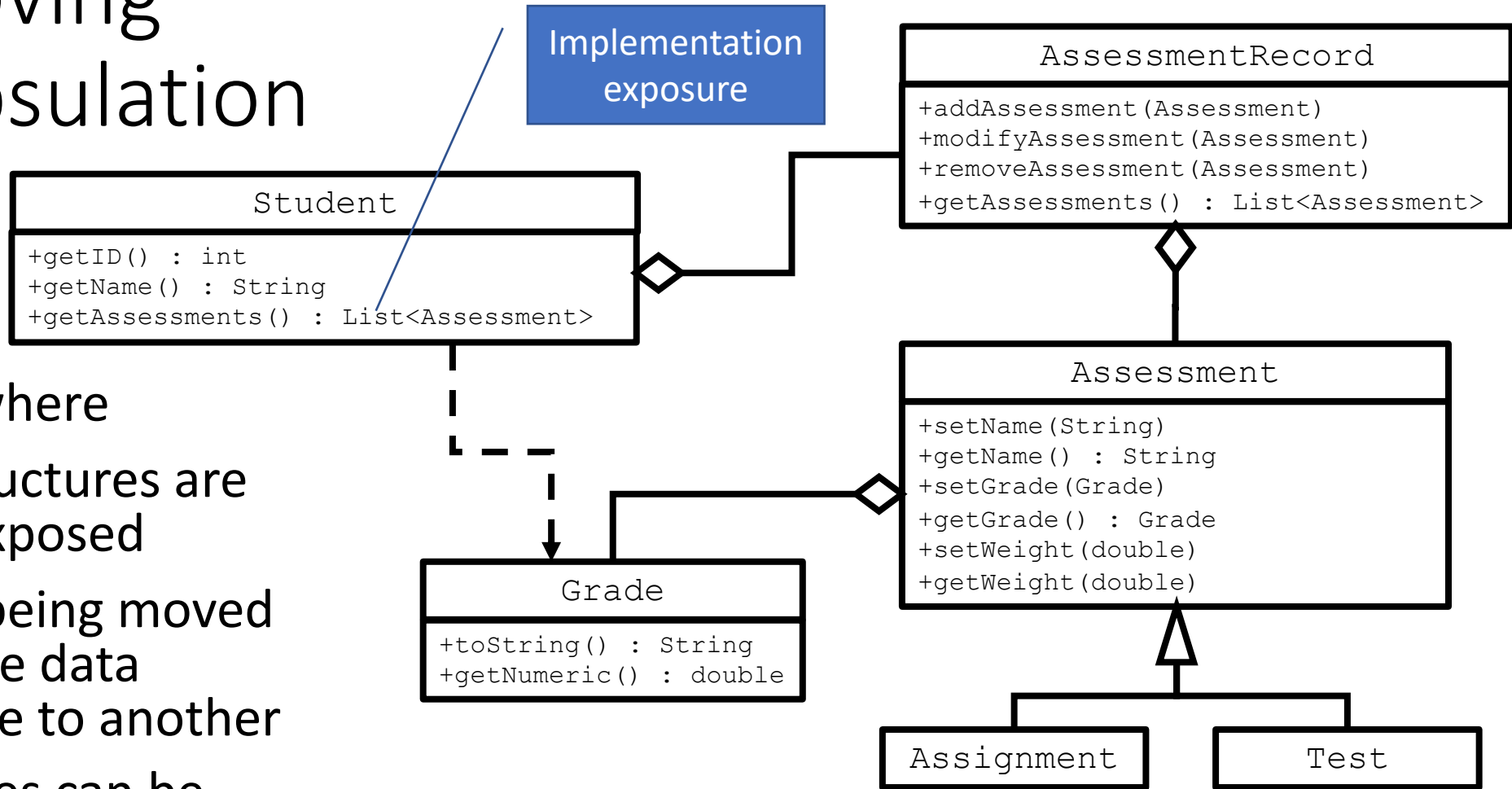
- I.e., You are **not allowed** to look at the implementation/details

- **Idea:** Encapsulation complements (enforces) abstraction
 - **Assume everything is private and make it public only when needed**
- **SOLID Principles:** ISP, DIP
- **Heuristics:**
 - Public methods should not have parameters or return types that are concrete classes
 - Think carefully about returning lists of items instead of an aggregation class
 - Avoid having public or protected instance variables
 - Make the public methods consistent in what parameters they take and their return types

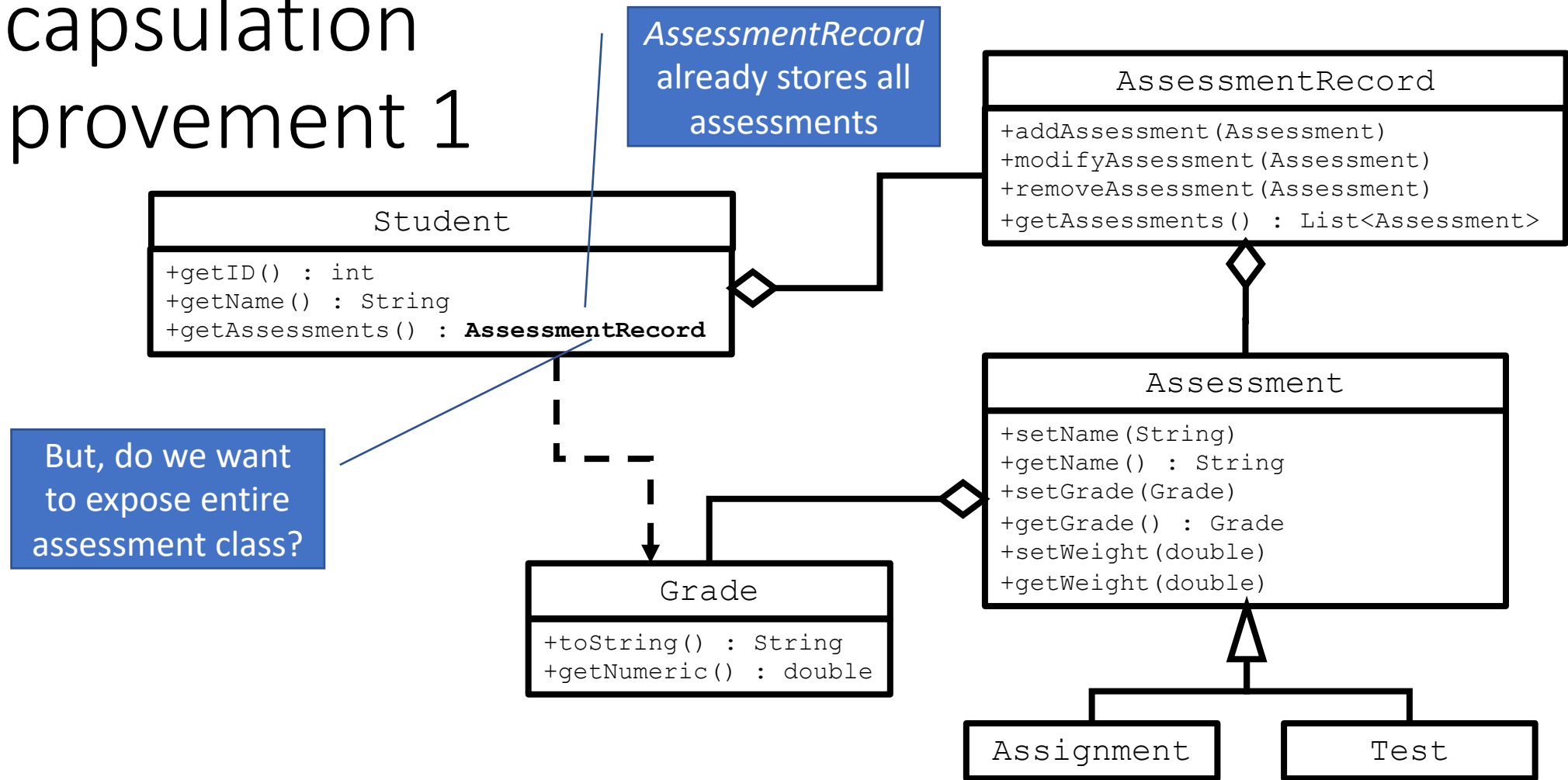
Improving Encapsulation

Identify where

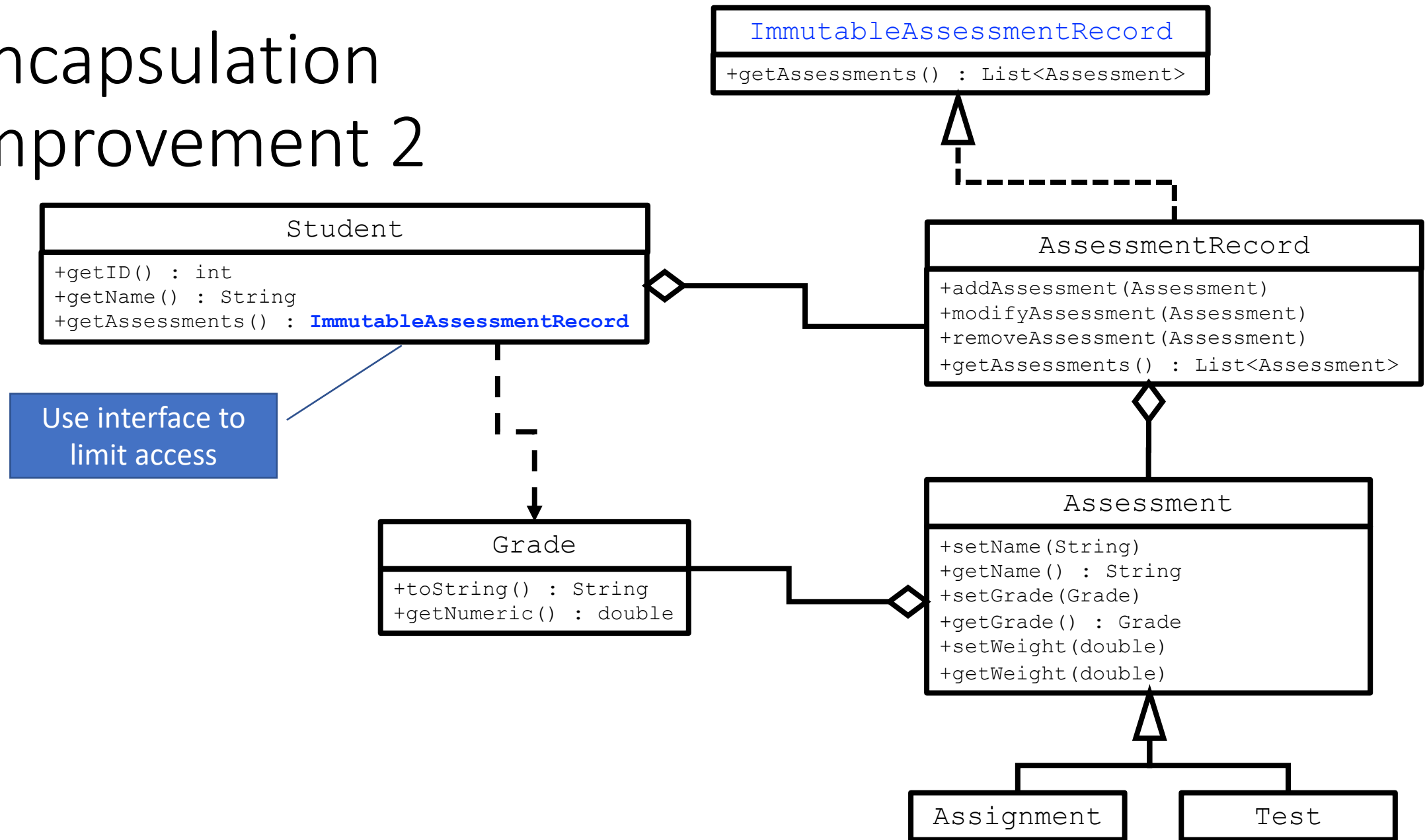
- data structures are being exposed
- data is being moved from one data structure to another
- Interfaces can be narrowed



Encapsulation Improvement 1



Encapsulation Improvement 2



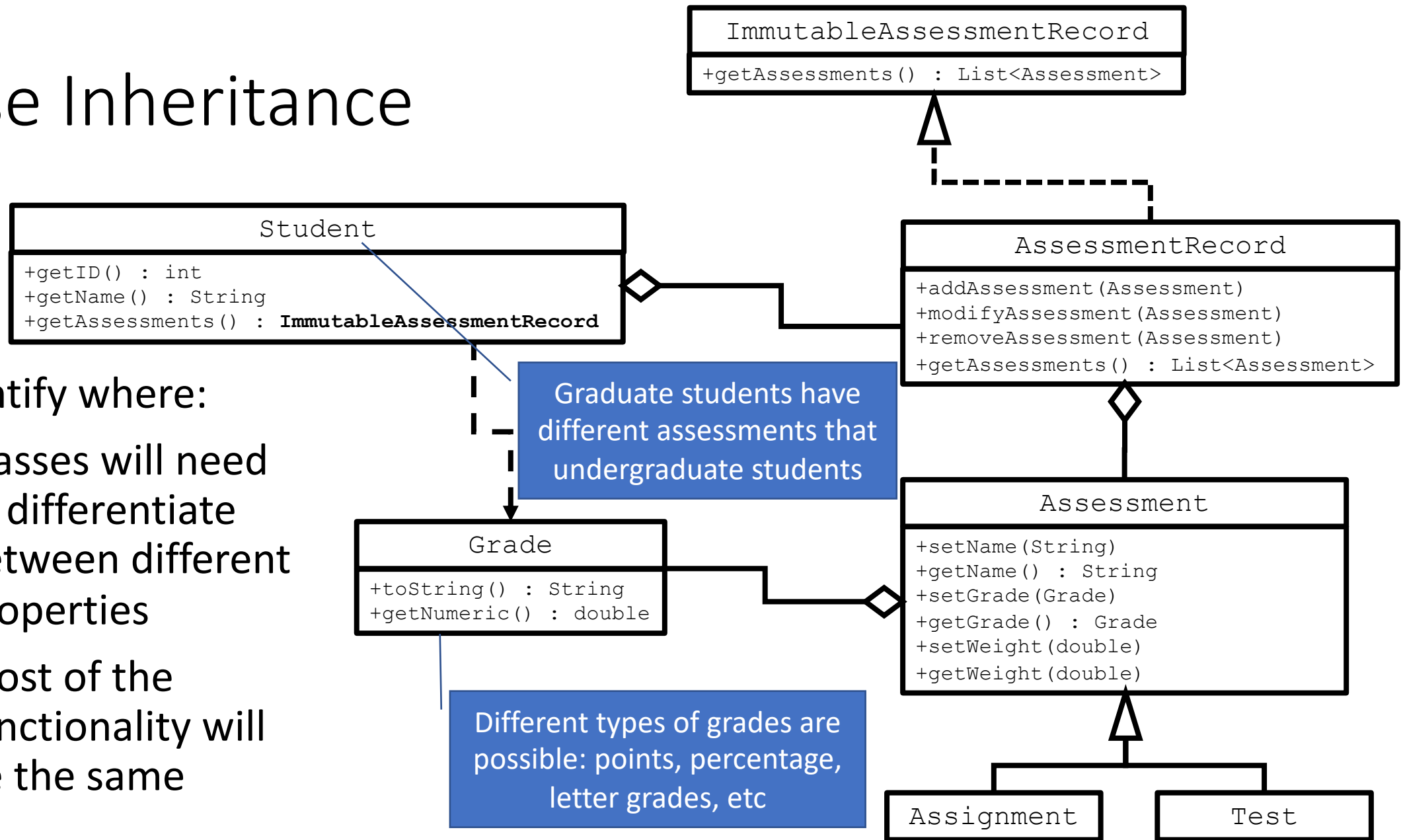
Use Inheritance – Where Appropriate

- Seek common function across classes with **common responsibilities**
 - Gather the common function into a base class
 - Encode the common code and attributes as the base class
 - Typically base class should be abstract

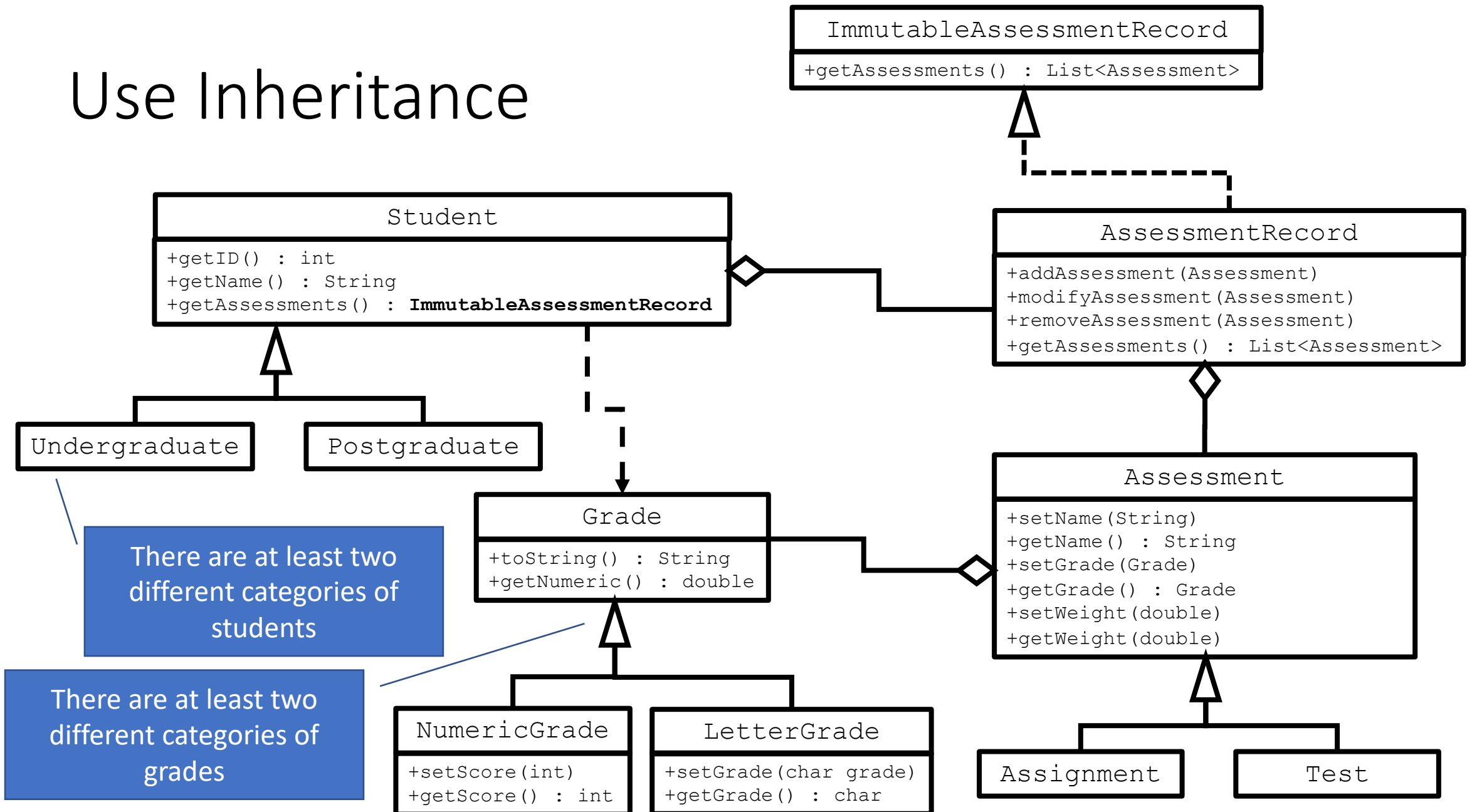
Use Inheritance

Identify where:

- Classes will need to differentiate between different properties
- Most of the functionality will be the same



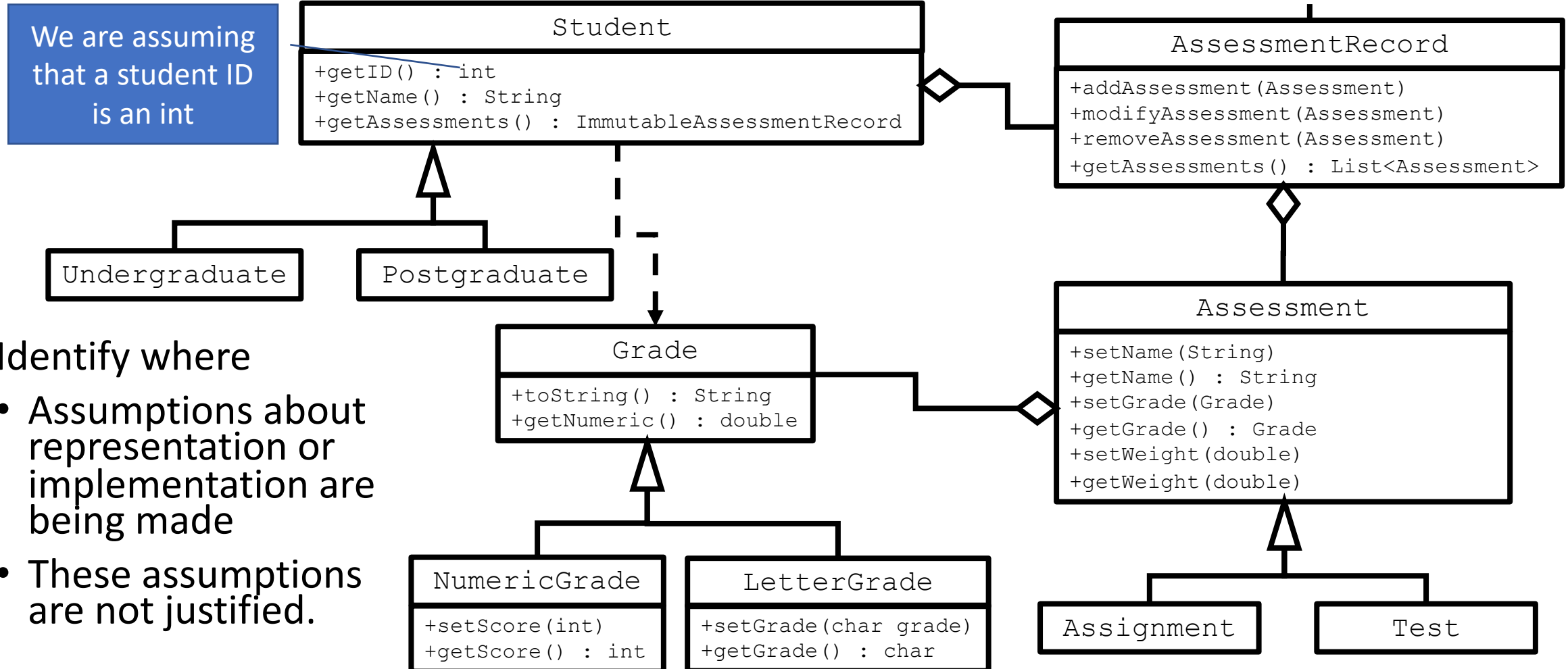
Use Inheritance



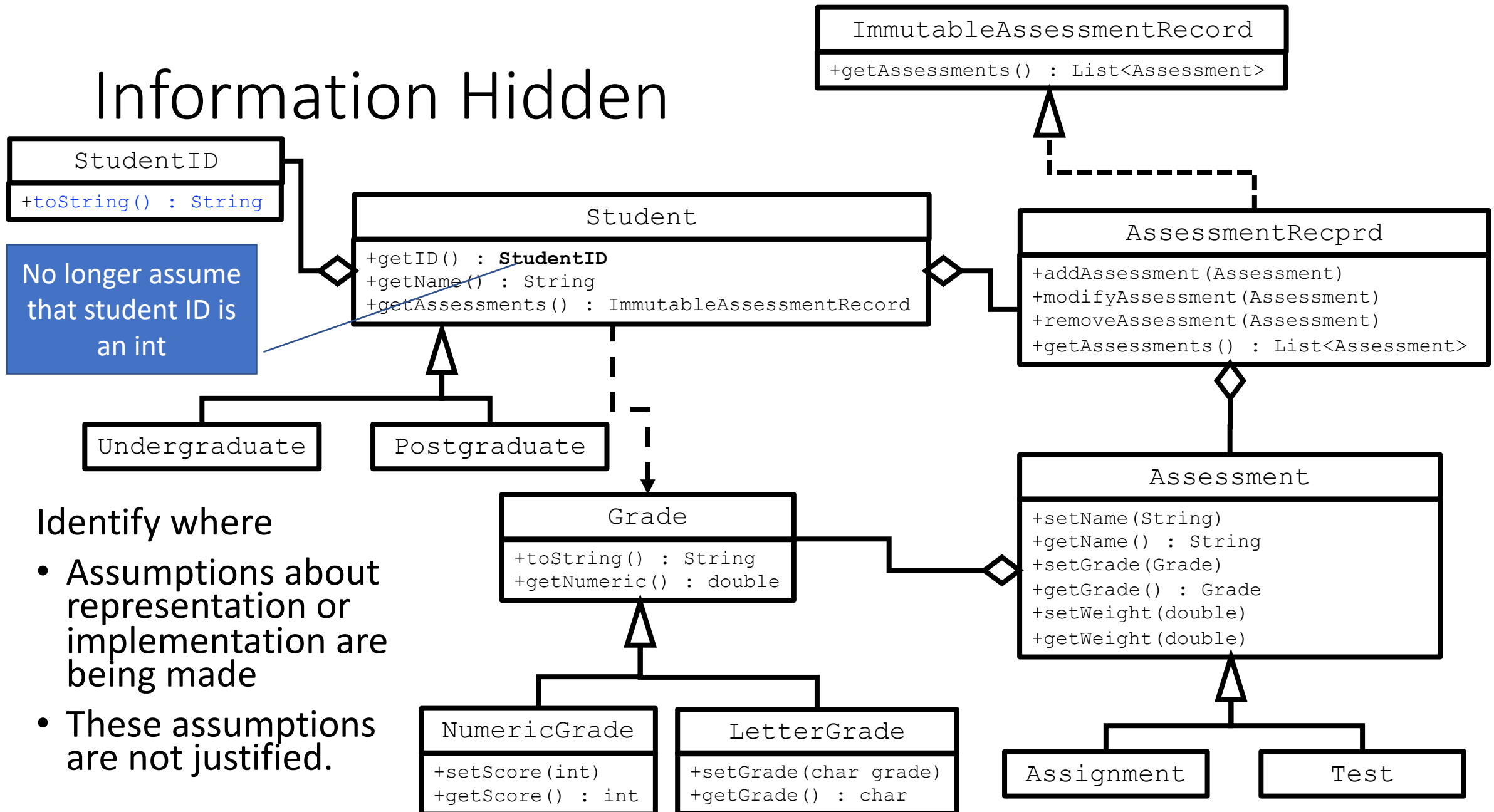
Hide Secrets – Information Hiding

- This is similar to encapsulation
 - **Encapsulation** prevents other classes from making assumptions about the implementation of a class
 - **Information hiding** limits the information available about an implementation
- Like encapsulation, it prevents classes from making assumptions about implementation or representation of other classes
- Hide information about implementation by:
 - Using interfaces (DIP)
 - Wrapping data representation inside classes
- Both encapsulation and information hiding
 - Hide the complexity of the task or the solution
 - Hide or isolate areas that are more likely to change

Hide Information



Information Hidden

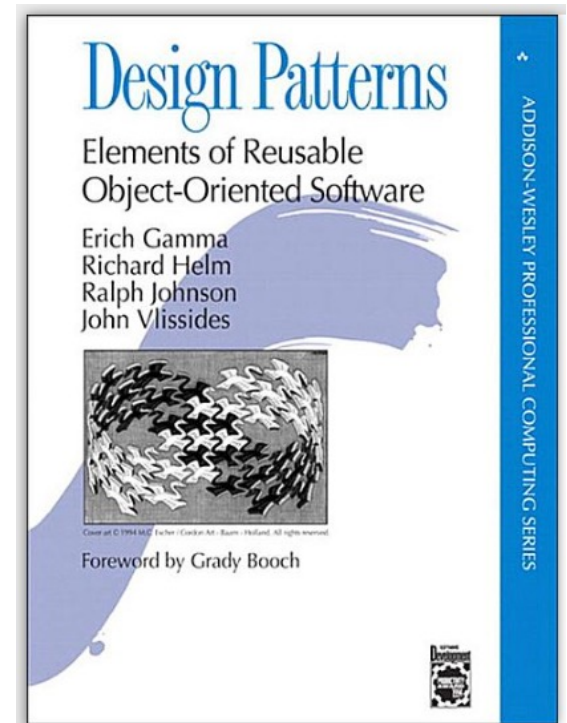


Identify where

- Assumptions about representation or implementation are being made
- These assumptions are not justified.

Other Design Heuristics

- Identify areas likely to change
 - As the design is evolving, keep a list of all decisions that have not been made.
 - E.g., Type of students, all types of assessments, different grade scales
 - Anything that is liable to change should be isolated using encapsulation and information hiding, so that later changes will not affect all the code
- Look for common design patterns
 - Design patterns are a well studied set of class patterns used to solve common problems in object-oriented design
 - **Problem:** suppose we want to notify students whenever their grades were updated
 - **Answer:** We can use the *Observer* design pattern to solve this problem
- Keep coupling loose
 - All the SOLID principles and most of the heuristics are all about minimizing coupling between software components.



Key Points

- The Noun-verb method is a good way to identify classes that you may need for your program
- The CRC (Class-Responsibility-Collaborator) is a method for determining what each class does
- When using these methods, focus on the abstract classes, don't worry about the implementation
- Good design evolves through an iterative process
- The refinement process focuses on: abstractions, SOLID principles, applying design heuristics
- The primary tools in our design toolbox are:
 - Creating consistent abstractions
 - Encapsulation and information hiding
 - Inheritance (where appropriate)
 - Design patterns

Image References

- <http://blog.nuvmconsulting.com/interviewing-tips-for-software-requirements-gathering/>
- <https://stevenwilliamalexander.wordpress.com/2015/07/31/non-functional-requirements-cart-before-horse/>
- <https://www.teacherspayteachers.com/Product/Parts-of-Speech-Printable-Posters-Noun-Verb-Adjective-Adverb-218930>
- <https://www.travel-palawan.com/palawan-dos-dont/>
- <http://pengetouristboard.co.uk/vote-best-takeaway-se20/>

Retrieved December 2, 2020

- <https://xkcd.com/2021/>

Image References

- <http://pengetouristboard.co.uk/vote-best-takeaway-se20/>

Retrieved December 4, 2020

- <https://toggl.com/blog/heroes-and-villains-of-software-development>