CSSE 220

Coupling and Cohesion
Scoping

Today’s topic

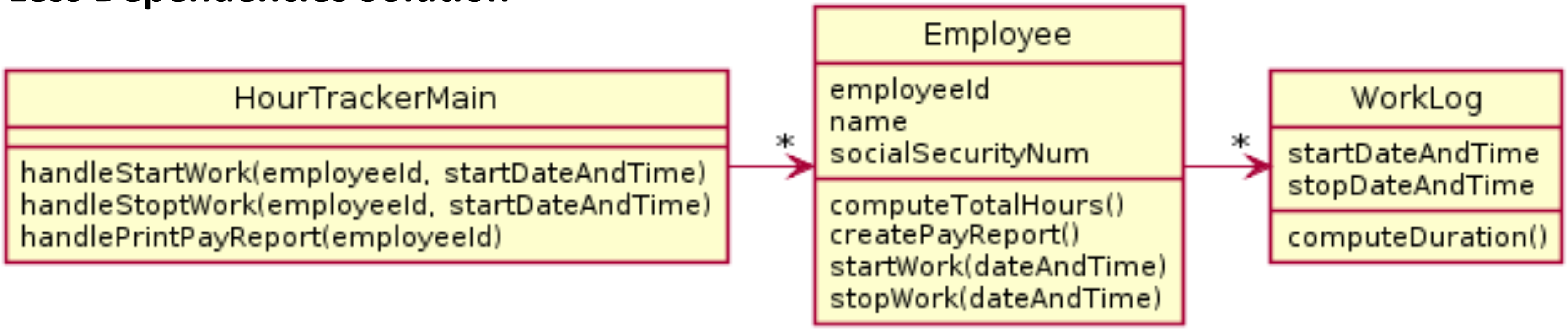
- **Minimize dependencies** between objects when it does not disrupt usability or extendability
  - Tell don't ask
  - Don't have message chains

Principles of Design (for CSSE220)

- Make sure your design **allows proper functionality**
  - Must be able to **store required information** (one/many to one/many relationships)
  - Must be able to **access the required information** to accomplish tasks
  - Data should **not be duplicated** (id/identifiers are OK!)
- Structure design **around the data** to be stored
  - **Nouns should become classes**
  - **Classes should have intelligent behaviors** (methods) **that may operate on their data**
- Functionality should be **distributed efficiently**
  - **No class/part should get too large**
  - **Each class should have a single responsibility** it accomplishes
- **Minimize dependencies** between objects when it does not disrupt usability or extendability
  - Tell don't ask
  - Don't have message chains
- **Don't duplicate** code
  - Similar "chunks" of code should be **unified into functions**
  - Classes with similar features should be given **common interfaces**
  - Classes with similar internals should be simplified using **inheritance**

A system tracks employee hours at a particular company. Every time any employee starts work and stops work, the system must log it so the employee can be paid correctly and so management knows who was working when. The system must also print out a weekly pay report for each employee which includes total hours, the employee's name, social security number, and employee id.

Less Dependencies Solution



HourTrackerMain

handleStartWork(employeeId, startDateAndTime)
handleStoptWork(employeeId, startDateAndTime)
handlePrintPayReport(employeeId)

*

Employee

employeeId
name
socialSecurityNum

computeTotalHours()
createPayReport()
startWork(dateAndTime)
stopWork(dateAndTime)

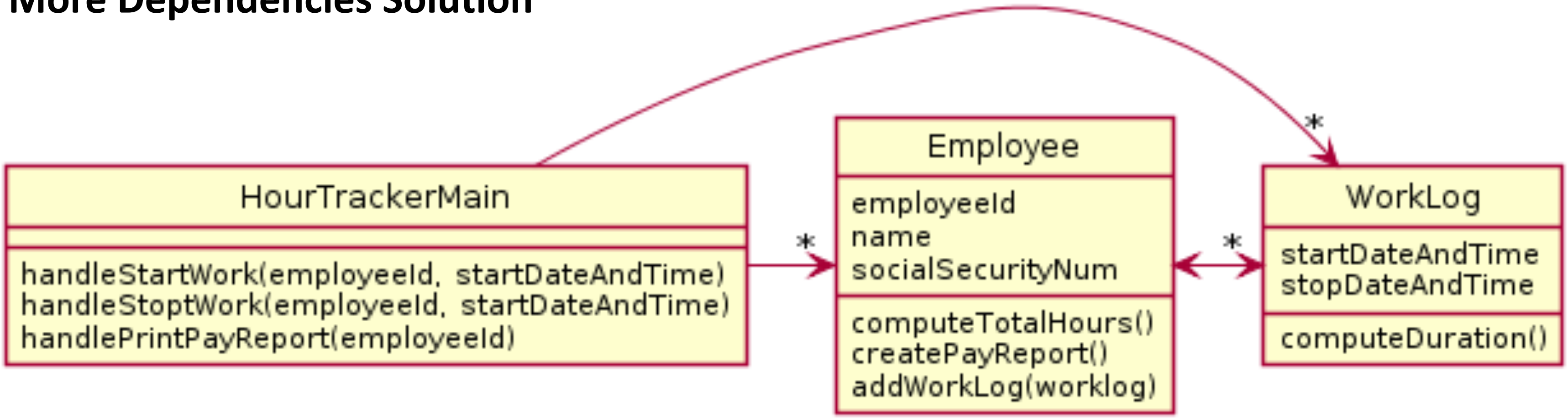
*

WorkLog

startDateAndTime
stopDateAndTime

computeDuration()

More Dependencies Solution



HourTrackerMain

handleStartWork(employeeId, startDateAndTime)
handleStoptWork(employeeId, startDateAndTime)
handlePrintPayReport(employeeId)

*

Employee

employeeId
name
socialSecurityNum

computeTotalHours()
createPayReport()
addWorkLog(worklog)

*

*

WorkLog

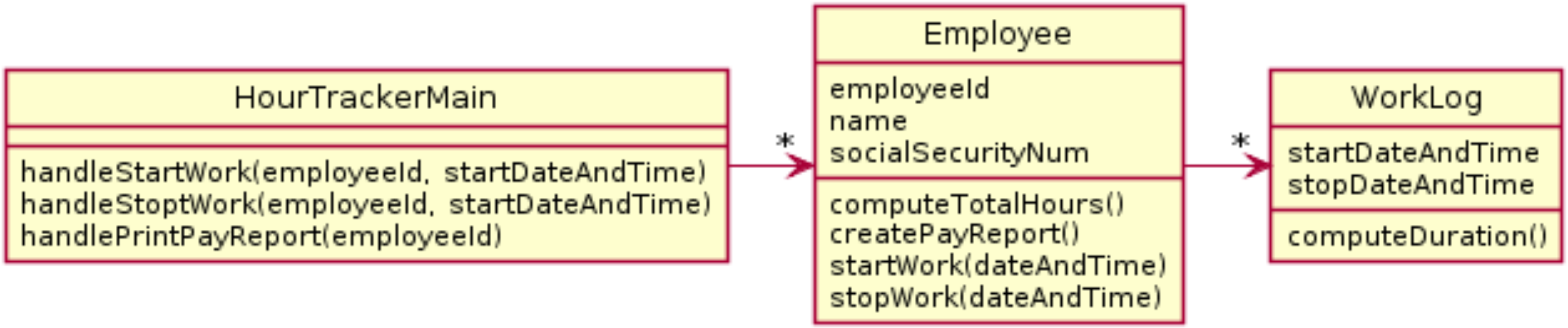
startDateAndTime
stopDateAndTime

computeDuration()

In less dependencies, Employee “insulates” HourTrackerMain from the existence of the WorkLog class. This means changes in the way WorkLog works cannot affect Employee. Similarly, changes in Employee cannot affect WorkLog.

The less dependencies solution is also simpler. Employee fully “owns” all it’s own data. In more dependencies, the worklog is edited without employee’s knowledge.

Less Dependencies Solution



HourTrackerMain

handleStartWork(employeeId, startDateAndTime)
handleStoptWork(employeeId, startDateAndTime)
handlePrintPayReport(employeeId)

Employee

employeeId
name
socialSecurityNum

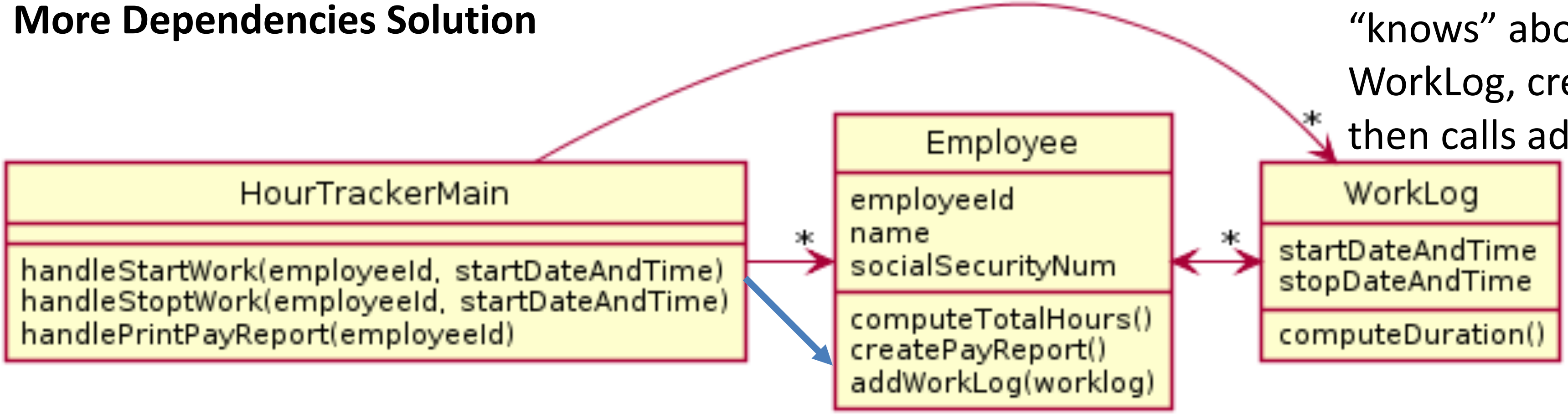
computeTotalHours()
createPayReport()
startWork(dateAndTime)
stopWork(dateAndTime)

WorkLog

startDateAndTime
stopDateAndTime

computeDuration()

More Dependencies Solution



HourTrackerMain

handleStartWork(employeeId, startDateAndTime)
handleStoptWork(employeeId, startDateAndTime)
handlePrintPayReport(employeeId)

Employee

employeeId
name
socialSecurityNum

computeTotalHours()
createPayReport()
addWorkLog(worklog)

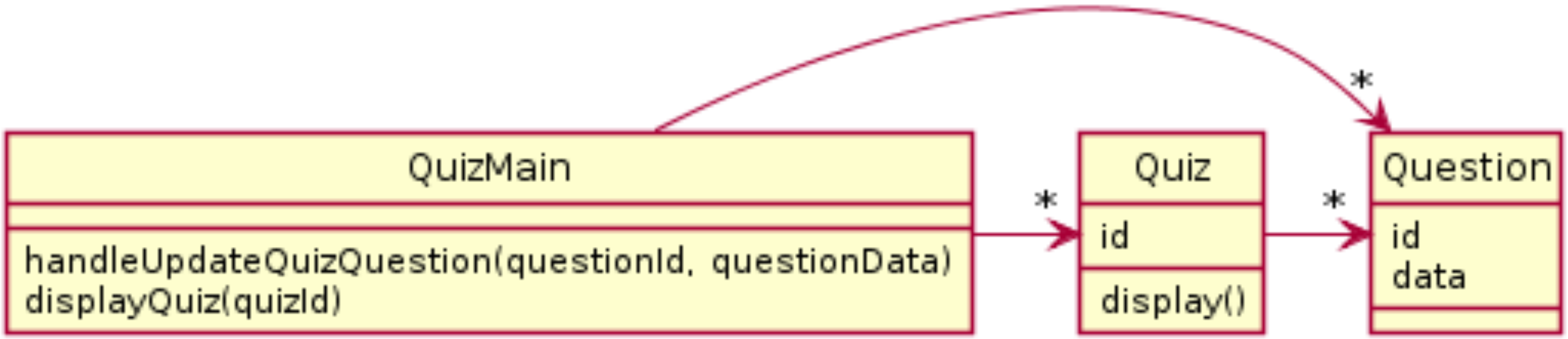
WorkLog

startDateAndTime
stopDateAndTime

computeDuration()

HourTrackerMain “knows” about WorkLog, creates or then calls addWorkL

Oftentimes you cannot remove dependencies without breaking functionality though.



Today’s topic - #1

- **Minimize dependencies** between objects when it does not disrupt usability or extendability
  - If you can see a simpler design that works use it
  - But if you can’t see a simpler design than the one that you have, at least ensure that you:
    - Tell don't ask
    - Don't have message chains

Tell Don’t Ask – getter methods

```
// Client program of region
Point2D center1 = region1.getPosition();
Point2D center2 = region2.getPosition();
double dist = center1.distance(center2);
if(dist > region1.getRadius()) {
   region1.setIsOverlapping(true);
}
// This code is determining if two regions intersect
```

Many ASKS

Sometimes you’ll have code that calls a lot of *getters* on some other object. In essence, this code is **Asking** for a lot of information from the region object.

Note how much this code “knows” about the Region class. It knows about many of its fields. It has a very strong dependency on the Region class.

Tell Don’t Ask
Use Procedural Abstraction

```
region1.flagOverlappingWith(region2);
```

TELL

When client uses a collection of *getters* to do some computation, then that computation is a good candidate to become a new method in the called-upon class

In this code, we’ve moved the center point and distance calculations into the Region class. Now rather than **asking** the Region for all sorts of data we simply **tell** the region to handle the problem itself and rely on it to do it.

Now, because we rely on the Region object to handle its own data, we have a weaker dependence on the region object.

Tell Don’t Ask – Bad Design

```
public LogFramework getLogFramework() {
   return this.framework;
}
```

ASK

Asking is especially a *bad design* when you return some internal class that the caller would otherwise not know exists. Why does the caller want the framework? Maybe that should be a tell?

Violates “separation of concerns” – Client now knows “how” called on code works – this increases “coupling” between client and called-on class/code, high coupling is usually a poorer design choice

```
public void activateVerboseLogging() {
   this.framework.setLevel(5);
}
```

TELL

If the caller only needs to do one thing, just add a method to do that thing and insulate the caller from dependence on LogFramework.

Tell Don’t Ask

- Be wary getter methods
- Prefer methods that command (tell) a class to do something that the called-upon class is controls for its own state and responsibilities
- If client code accesses a lot of internal data of another class, consider if a tell method in that called-upon class might improve the design

A simple example of Tell Don't Ask

In your TeamGradebook classes, you need to calculate a student's average grade. This could be accomplished by:

1) Adding a getAverage() method to the Student class which calculates the average
2) Adding a getGrades() method to the student class, which the TeamGradebook class could call, and then use to compute the average

A simple example of Tell Don’t Ask

In your TeamGradebook classes, you need to calculate a student’s average grade. This could be accomplished by:

1) Adding a getAverage() method to the Student class which calculates the average

This approach engineers Student class so that it “knows” more about what goes on with Students, and TeamGradeBook “knows” less

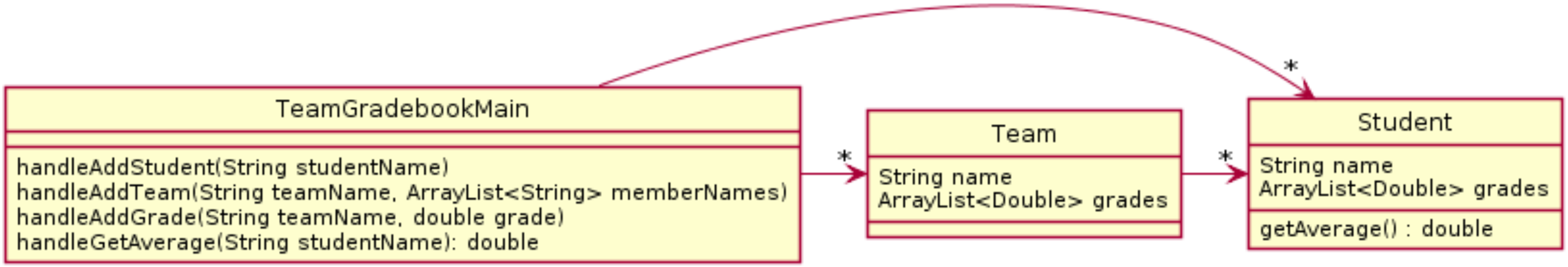
A simple example of Tell Don’t Ask

In your TeamGradebook classes, you need to calculate a student’s average grade. This could be accomplished by:

Second approach increases coupling between TeamGradeBook and Student class, i.e., TeamGradBook “knows” more about Student

2) Adding a getGrades() method to the student class, which the TeamGradebook class could call, and then use to compute the average

Diagrams look similar!


TeamGradebookMain

handleAddStudent(String studentName)
handleAddTeam(String teamName, ArrayList<String> memberNames)
handleAddGrade(String teamName, double grade)
handleGetAverage(String studentName): double

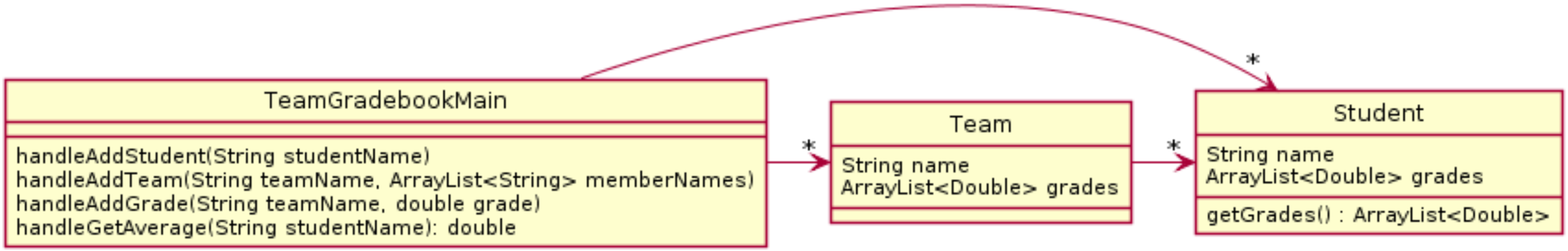
Team

String name
ArrayList<Double> grades

Student

String name
ArrayList<Double> grades

getAverage() : double


TeamGradebookMain

handleAddStudent(String studentName)
handleAddTeam(String teamName, ArrayList<String> memberNames)
handleAddGrade(String teamName, double grade)
handleGetAverage(String studentName): double

Team

String name
ArrayList<Double> grades

Student

String name
ArrayList<Double> grades

getGrades() : ArrayList<Double>

Diagrams look similar!

How would the actual code compare when performing the stated task “calculate a student’s average grade”?

getGrades()

```
public class TeamGradebook {
…
private String handleGetAverage(String studentName) {
        Student student = getStudentByName(studentName);
        if (student == null) {
                return "student " + student + " not found";
        }
        double total = 0;
        for (double d: student.getGrades() ) {
                total += d;
        }
        double average = total / student.getGrades().size();
        return Long.toString(Math.round(average));
}
…
}
```

Calculation happening in TeamGradebook!

getAverage()

```
public class TeamGradebook {
…
private String handleGetAverage(String studentName) {
       Student student = getStudentByName(studentName);
       if (student == null) {
              return "student " + student + " not found";
       }
       return Long.toString(Math.round(student.getAverage()));
}
…
}
```

Calculation happening in Student!

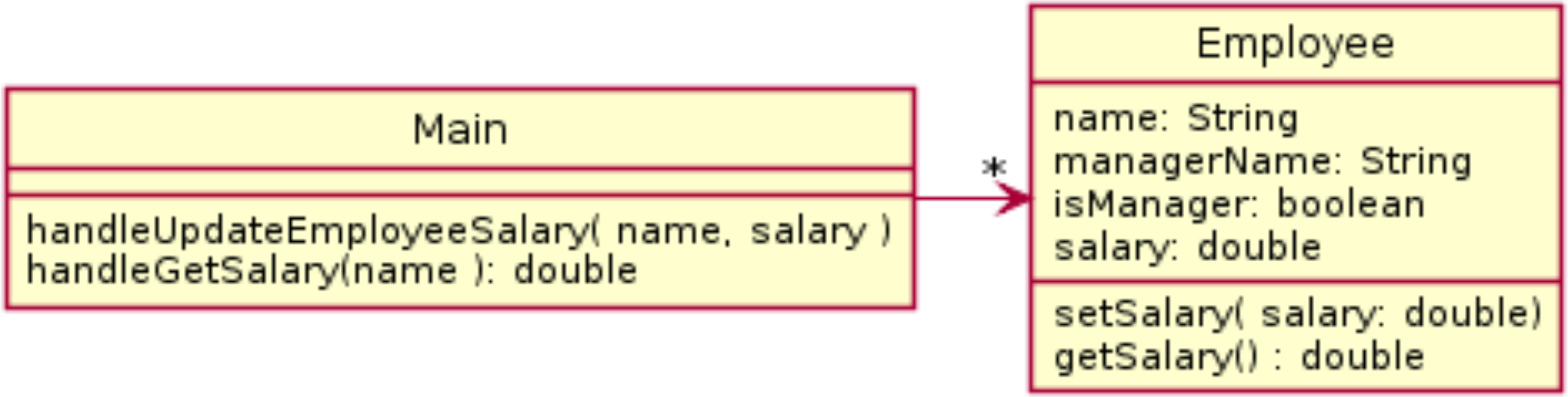
Why does this improve the design?

Reduces coupling between two classes:

- It makes the Student object more featureful, and puts the code in an expected place
- Reduces the code in TeamGradebook which is already quite long
- Allows you to change how the grades are represented in TeamGradebook, should you wish to drop lowest score, for example

Employee Salary Problem
In-Class Quiz Questions #1 & #2

There is a company which has employees, each of which has a salary. There are managers which oversee other employees. Employees have salaries which can be updated from time to time. Unlike employees, a manager’s salary is always 10% more than the salary of their top paid employee.



Main

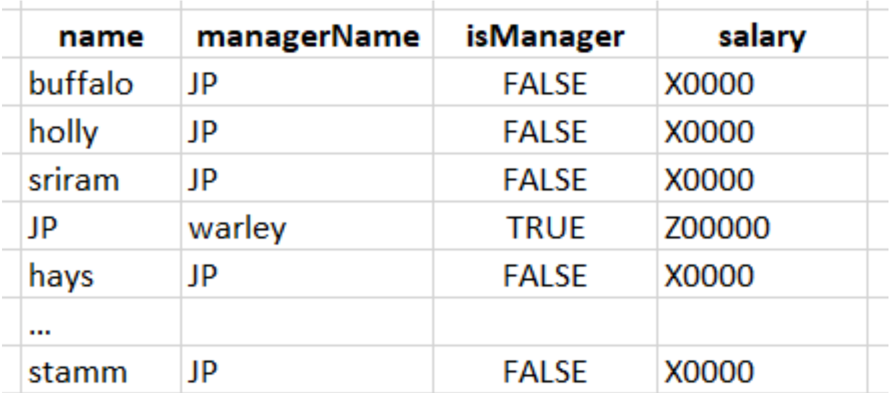
handleUpdateEmployeeSalary(name, salary)
handleGetSalary(name): double

Employee

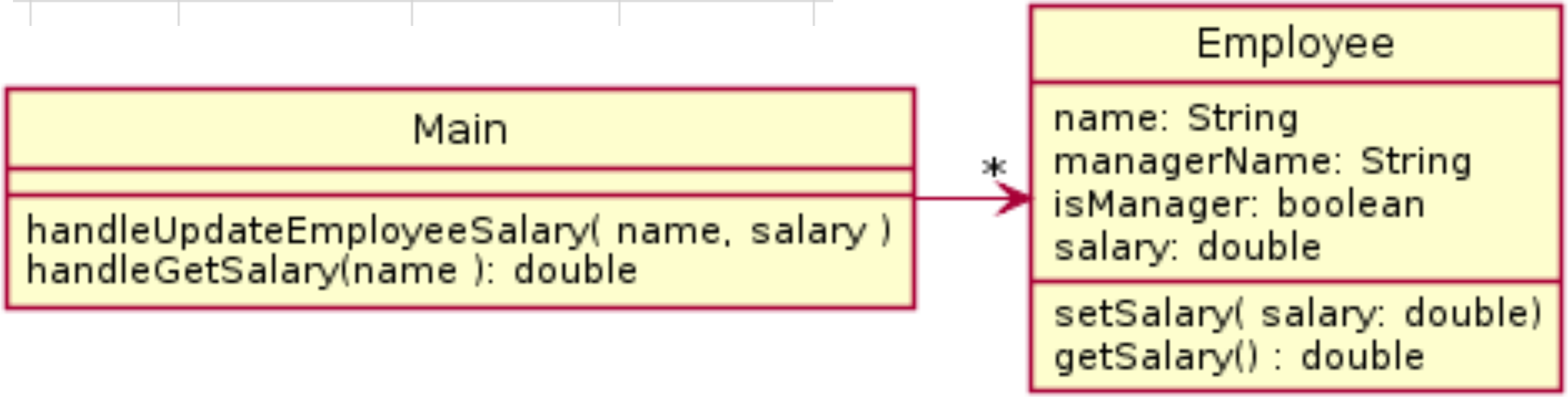
name: String
managerName: String
isManager: boolean
salary: double

setSalary(salary: double)
getSalary() : double

Employee Salary Problem
In-Class Quiz Questions #1 & #2



name	managerName	isManager	salary
buffalo	JP	FALSE	X0000
holly	JP	FALSE	X0000
sriram	JP	FALSE	X0000
JP	warley	TRUE	Z00000
hays	JP	FALSE	X0000
…			
stamm	JP	FALSE	X0000



Main

handleUpdateEmployeeSalary(name, salary)
handleGetSalary(name): double

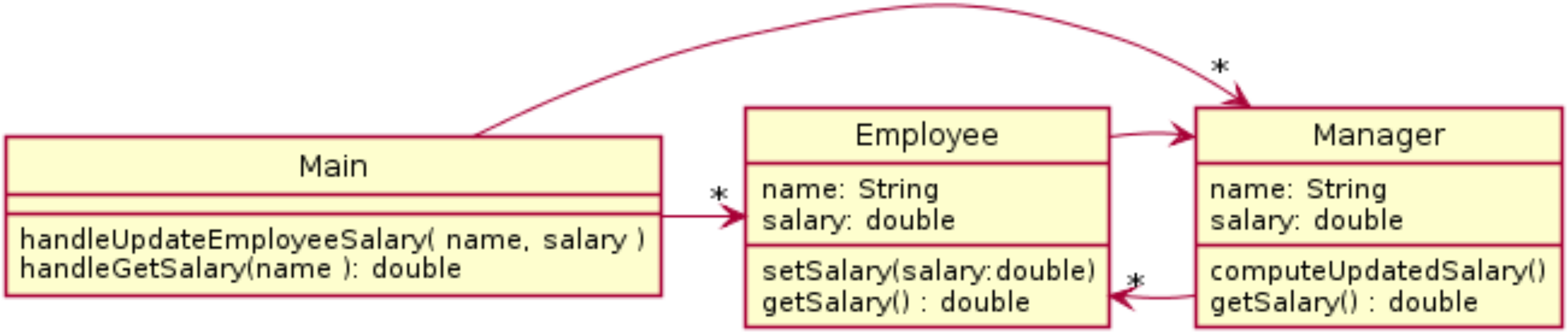
*

Employee

name: String
managerName: String
isManager: boolean
salary: double

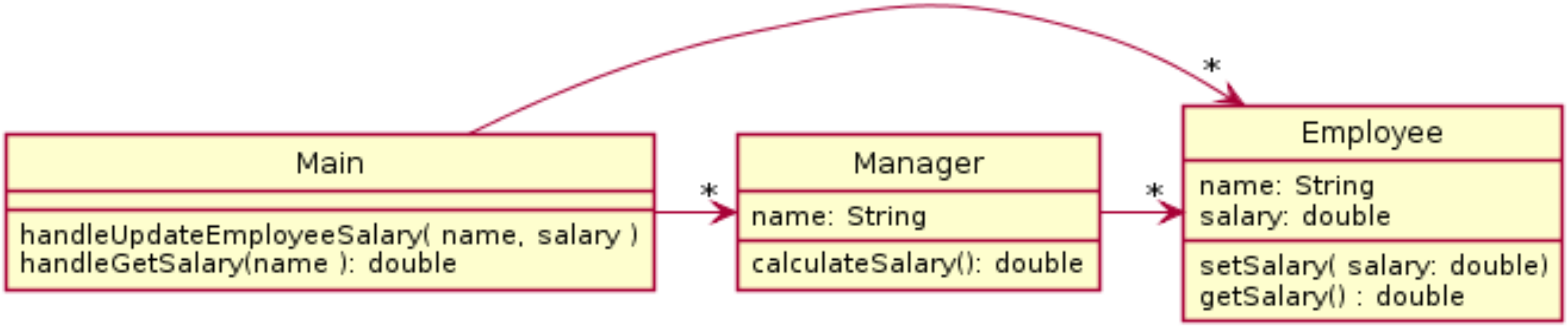
setSalary(salary: double)
getSalary() : double

Better Solution



- Anything wrong?
- Room to improve?

Eliminate manager salary field!


Main
handleUpdateEmployeeSalary(name, salary)
handleGetSalary(name): double
*
Manager
name: String
calculateSalary(): double
*
*
Employee
name: String
salary: double
setSalary(salary: double)
getSalary() : double

Data is technically duplicated if manager contains its own salary field.
What if the two pieces of data were out of sync?
Works well to calculate the salary as needed since it depends upon other data.

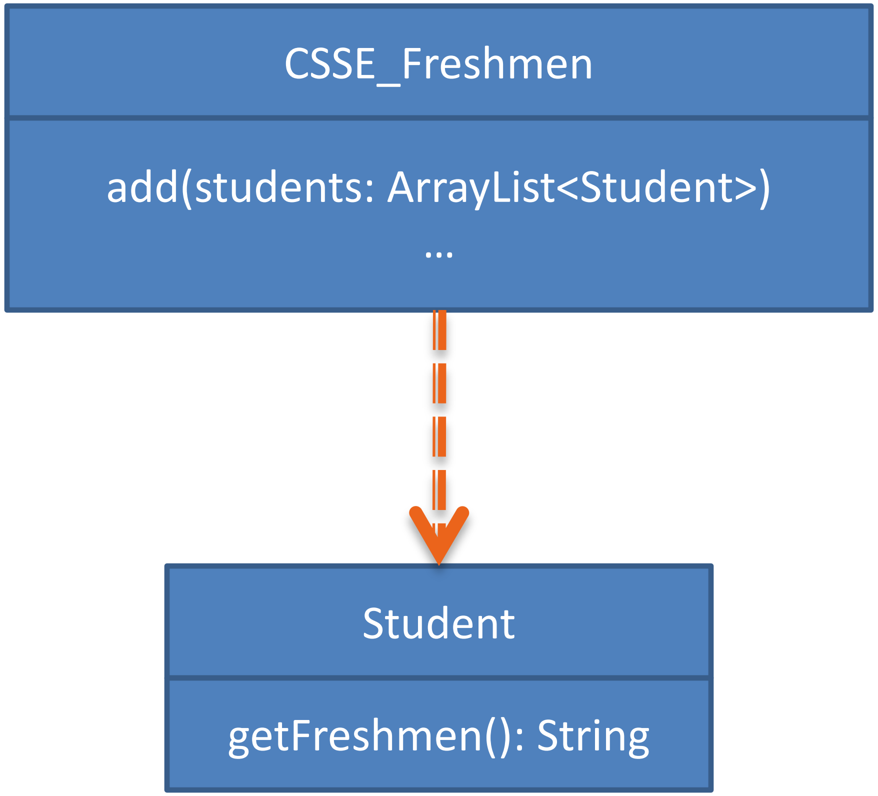
Today’s topic - #2

- **Minimize dependencies** between objects when it does not disrupt usability or extendability
  - If you can see a simpler design that works use it
  - But if you can’t see a simpler design than the one that you have, at least ensure that you:
    - Tell don't ask
    - Don't have message chains

UML Interlude: Dependency Relationship

- When one class requires another class to do its job, the first class depends on the second

- Shown on UML diagrams as:
  - dashed line
  - with open arrowhead



Message Chain – Don’t Have Them

A message chain is code in the form:

```
someObject.someMethod().otherMethod().stillOtherMethod();
```

For example

```
myFrame.getBufferStrategy().getCapabilities().getFlip().wait(17);
```

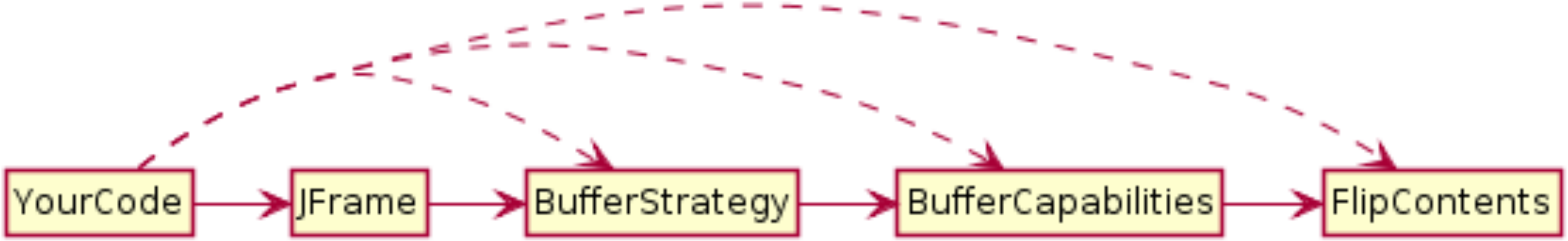
This is generally considered to a warning sign of excessive dependency and problems.

Message Chain
Rewritten using variables

Message chains are not better if you space them across multiple lines, but it does make it more obvious what the problem is.

```
BufferStrategy strategy = myFrame.getBufferStrategy();
BufferCapabilities capabilities = strategy.getCapabilities();
FlipContents flip = capabilities.getFlipContents();
flip.wait(17);
```

You are depending on internal classes deep within some other object's data



Message Chain
Rewritten using variables

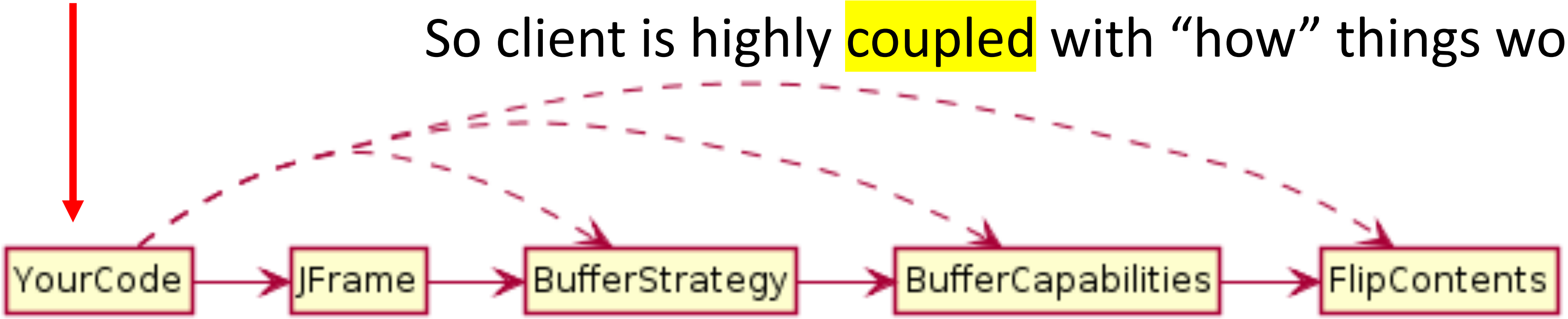
Message chains are not better if you space them across multiple lines, but it does make it more obvious what the problem is.

```
BufferStrategy strategy = myFrame.getBufferStrategy();
BufferCapabilities capabilities = strategy.getCapabilities();
FlipContents flip = capabilities.getFlipContents();
flip.wait(17);
```

You are depending on internal classes deep within some other object’s data

Client program – “knows” details 4 levels deep in called ops

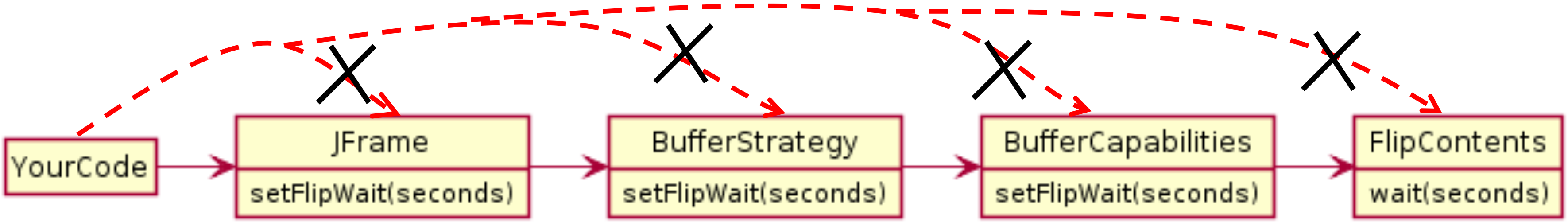
So client is highly coupled with “how” things work



Message Chain: Solution

The solution is usually to embed the required feature in the first class in the chain. This insulates the caller from the inner classes. Then the first class might implement the feature itself OR if it still needs to rely on its internals repeat the message chain removal.

```
myFrame.setFlipWait(17);
```

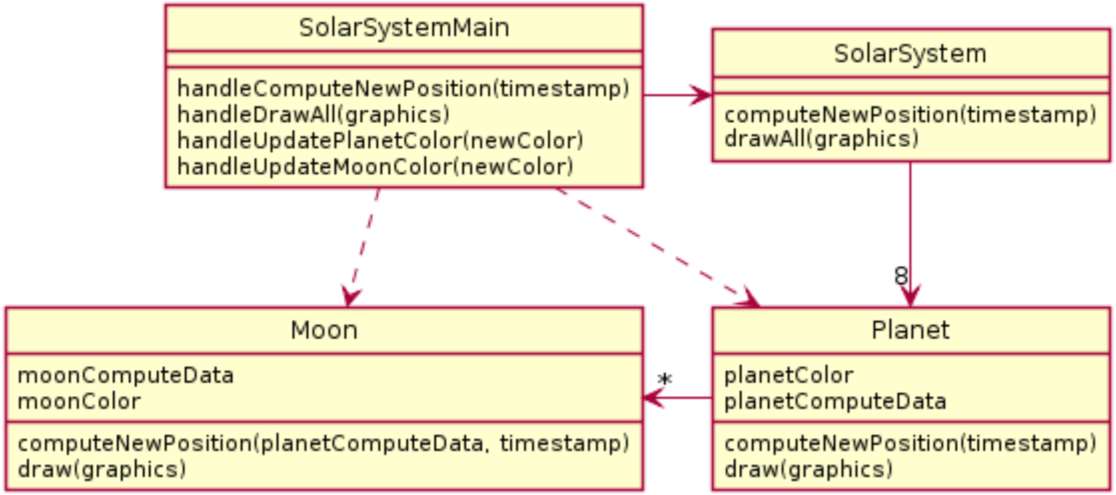


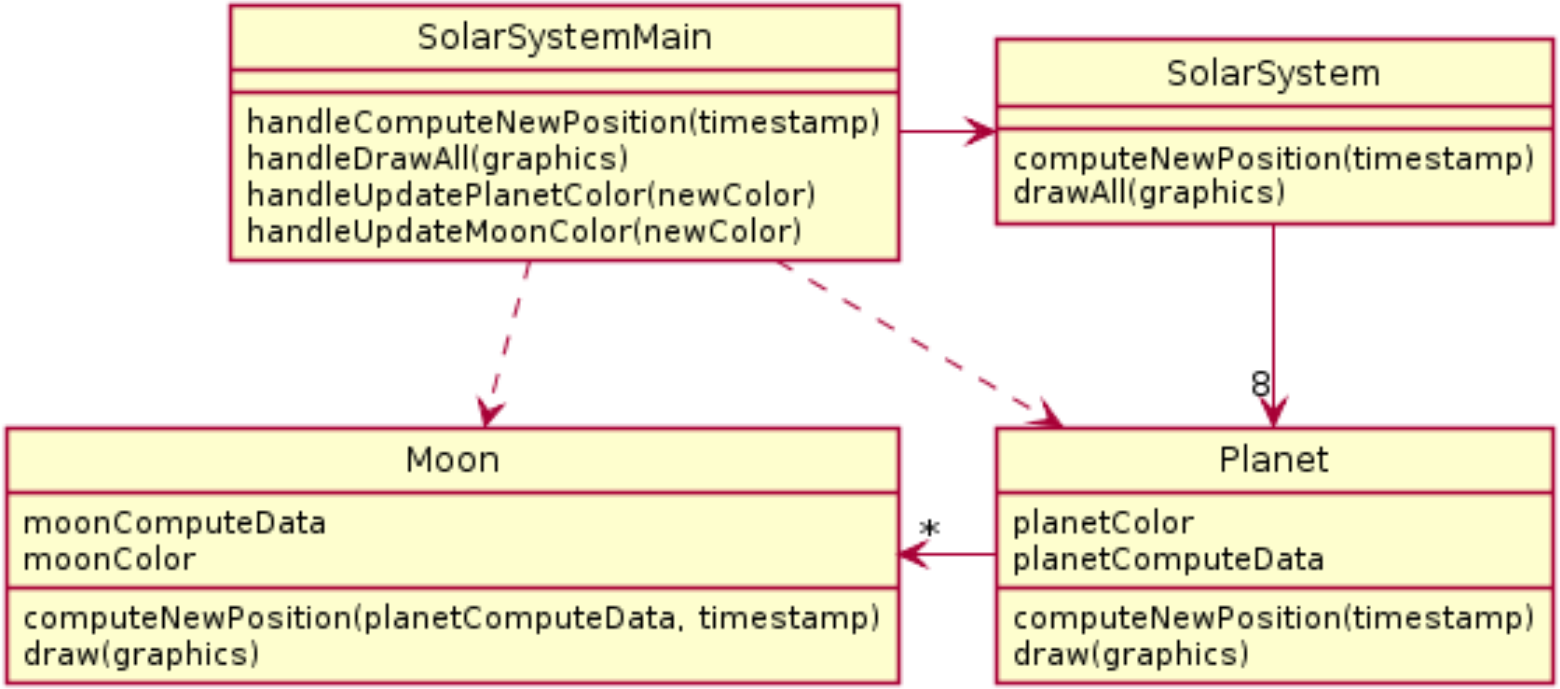
This approach also actually decouples:

1. BufferCapabilities from FlipContents
2. BufferStrategy from BufferCapabilities
3. Jframe from BufferStrategy

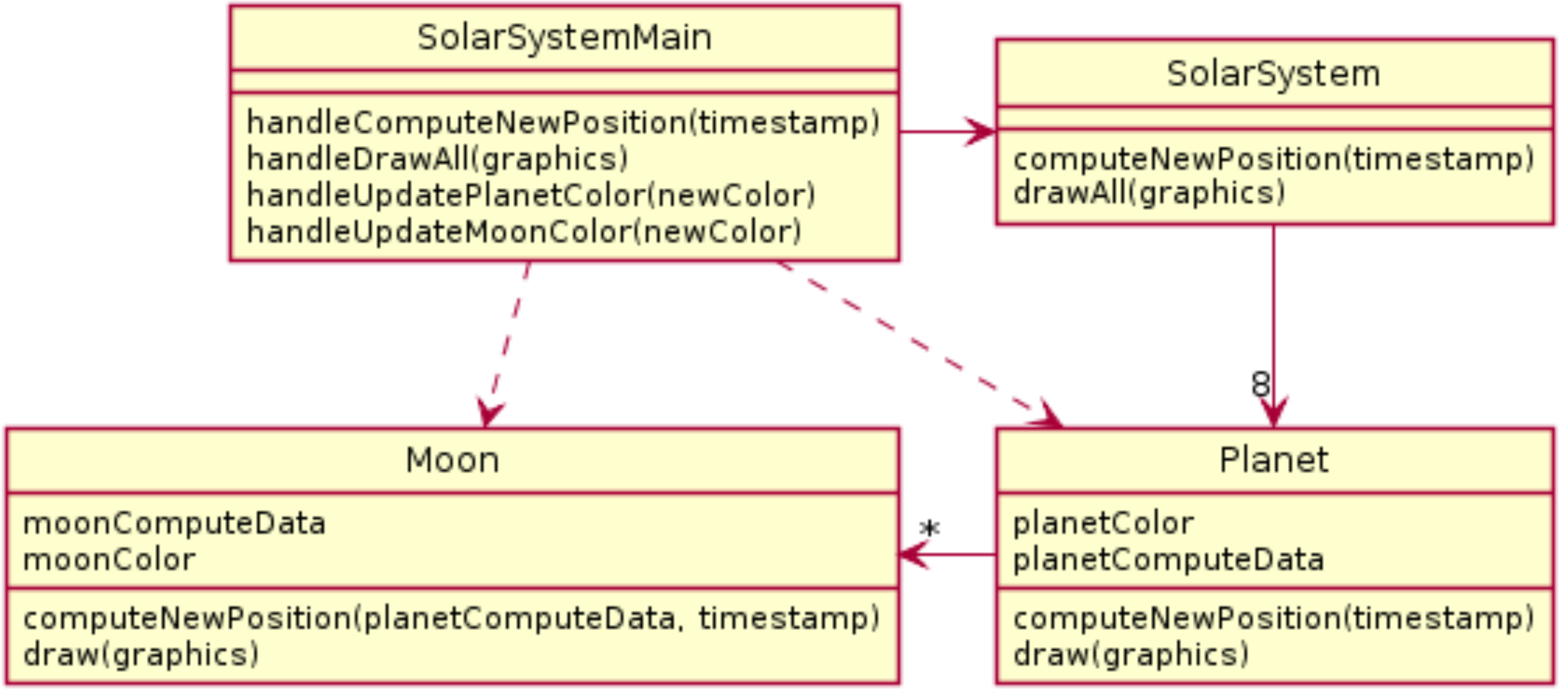
Solar System Problem
In-Class Quiz Questions #4 & #5

A java program draws a minute by minute updated diagram of the solar system including all planets and moons. To update the moon's position, the moon's calculations must have the updated position of the planet it is orbiting. The diagram is colored - all planets are drawn the same color and all moons are drawn the same color. However, it needs to be possible to reset the planet color or the moon color and the diagram should reflect that.


SolarSystemMain
handleComputeNewPosition(timestamp)
handleDrawAll(graphics)
handleUpdatePlanetColor(newColor)
handleUpdateMoonColor(newColor)
SolarSystem
computeNewPosition(timestamp)
drawAll(graphics)
8
Moon
moonComputeData
moonColor
computeNewPosition(planetComputeData, timestamp)
draw(graphics)
*
Planet
planetColor
planetComputeData
computeNewPosition(timestamp)
draw(graphics)


SolarSystemMain
handleComputeNewPosition(timestamp)
handleDrawAll(graphics)
handleUpdatePlanetColor(newColor)
handleUpdateMoonColor(newColor)
SolarSystem
computeNewPosition(timestamp)
drawAll(graphics)
8
Moon
moonComputeData
moonColor
computeNewPosition(planetComputeData, timestamp)
draw(graphics)
*
Planet
planetColor
planetComputeData
computeNewPosition(timestamp)
draw(graphics)

- What is wrong here?

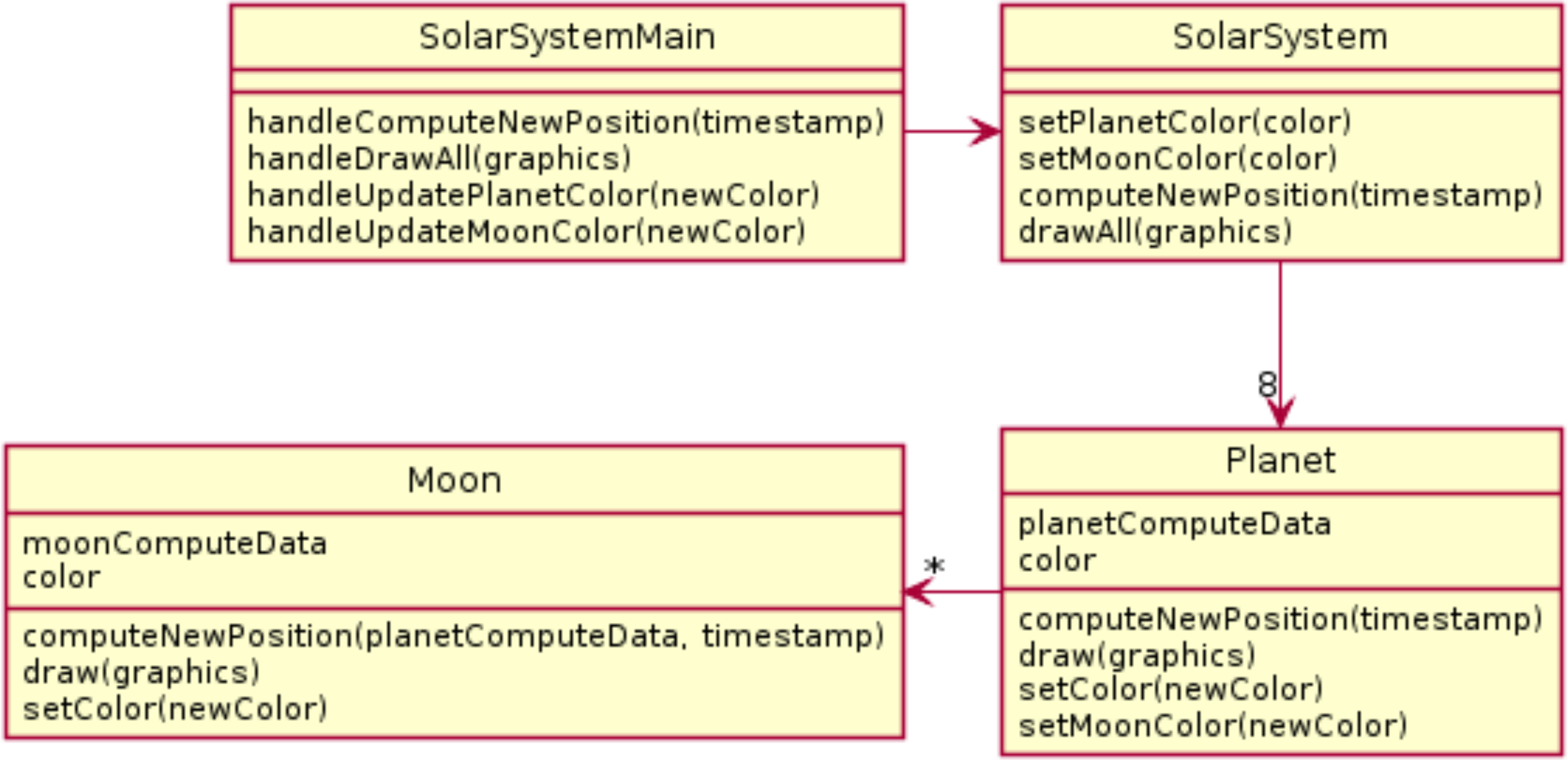

SolarSystemMain
handleComputeNewPosition(timestamp)
handleDrawAll(graphics)
handleUpdatePlanetColor(newColor)
handleUpdateMoonColor(newColor)
SolarSystem
computeNewPosition(timestamp)
drawAll(graphics)
8
Moon
moonComputeData
moonColor
computeNewPosition(planetComputeData, timestamp)
draw(graphics)
*
Planet
planetColor
planetComputeData
computeNewPosition(timestamp)
draw(graphics)

- What is wrong here?

4b. methodChain to update moon

```
ss1.getPlanets().get(0).getMoons().get(0).setColor(color);
```

Partial Solution



SolarSystemMain

handleComputeNewPosition(timestamp)
handleDrawAll(graphics)
handleUpdatePlanetColor(newColor)
handleUpdateMoonColor(newColor)

SolarSystem

setPlanetColor(color)
setMoonColor(color)
computeNewPosition(timestamp)
drawAll(graphics)

Planet

planetComputeData
color

computeNewPosition(timestamp)
draw(graphics)
setColor(newColor)
setMoonColor(newColor)

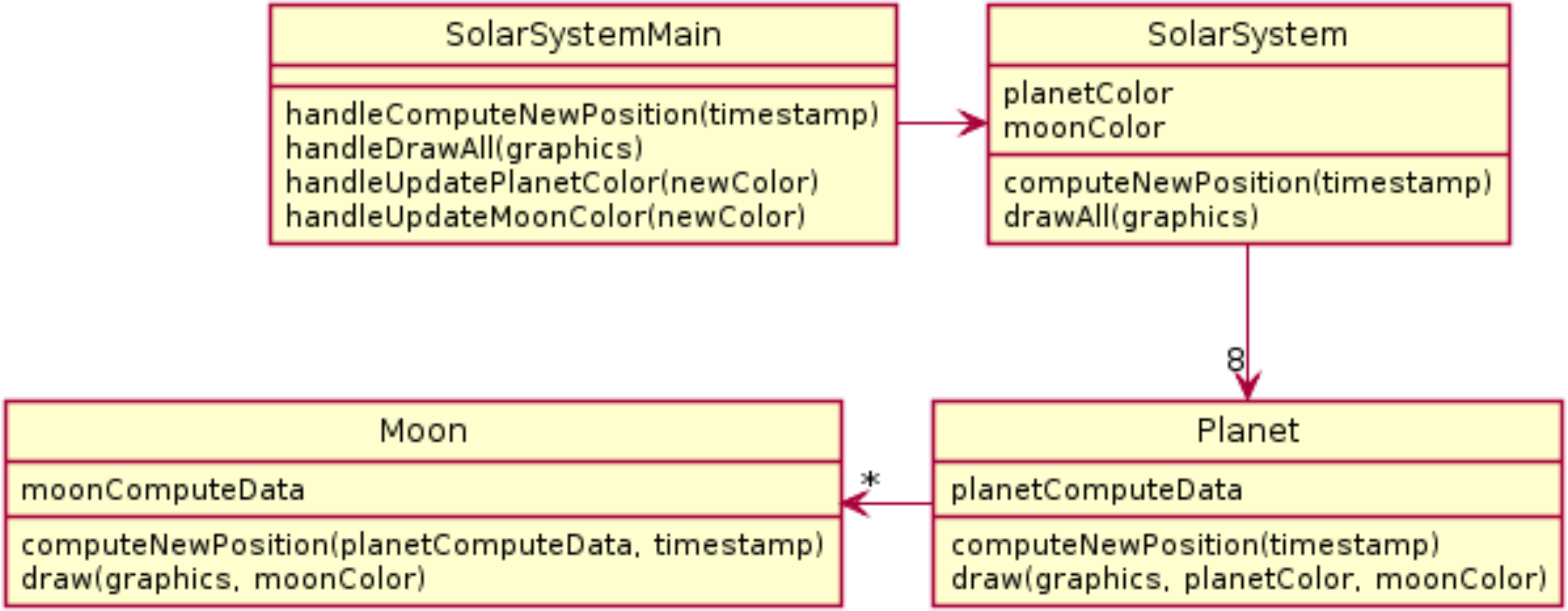
Moon

moonComputeData
color

computeNewPosition(planetComputeData, timestamp)
draw(graphics)
setColor(newColor)

Better Solution

Eliminate Data Duplication


SolarSystemMain
handleComputeNewPosition(timestamp)
handleDrawAll(graphics)
handleUpdatePlanetColor(newColor)
handleUpdateMoonColor(newColor)
SolarSystem
planetColor
moonColor
computeNewPosition(timestamp)
drawAll(graphics)
8
Moon
moonComputeData
computeNewPosition(planetComputeData, timestamp)
draw(graphics, moonColor)
*
Planet
planetComputeData
computeNewPosition(timestamp)
draw(graphics, planetColor, moonColor)

Today's topic

- **Minimize dependencies** between objects when it does not disrupt usability or extendability
  - If you can see a simpler design that works use it
  - But if you can't see a simpler design than the one that you have, at least ensure that you:
    - Tell don't ask
    - Don't have message chains
- Now two related terms:
  - coupling
  - cohesion

The plan

- Learn 3 essential object oriented design terms:
  - Encapsulation (done- yesterday)
  - Coupling
  - Cohesion
- Scope (if we have time)

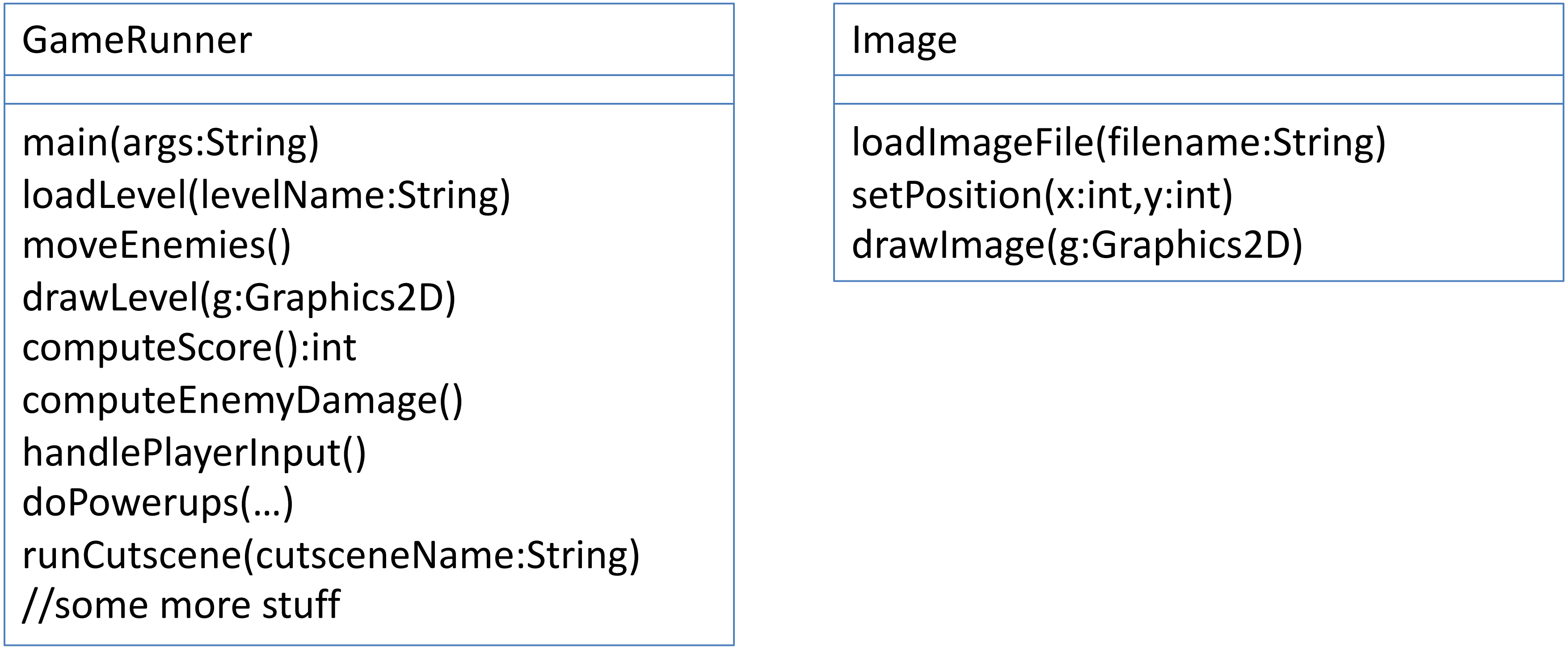
Coupling and Cohesion

- Two terms you need to memorize
- Good designs have:
  - High coHesion
  - Low coupLing

Consider the opposite:

- Low cohesion means that you have a small number of really large classes that do too much stuff (i.e., do more than one thing)
- High coupling means you have many classes that depend ("know") too much on each other

Imagine I want to make a Video Game
Here are two classes in my design
Which one has low cohesion?



GameRunner

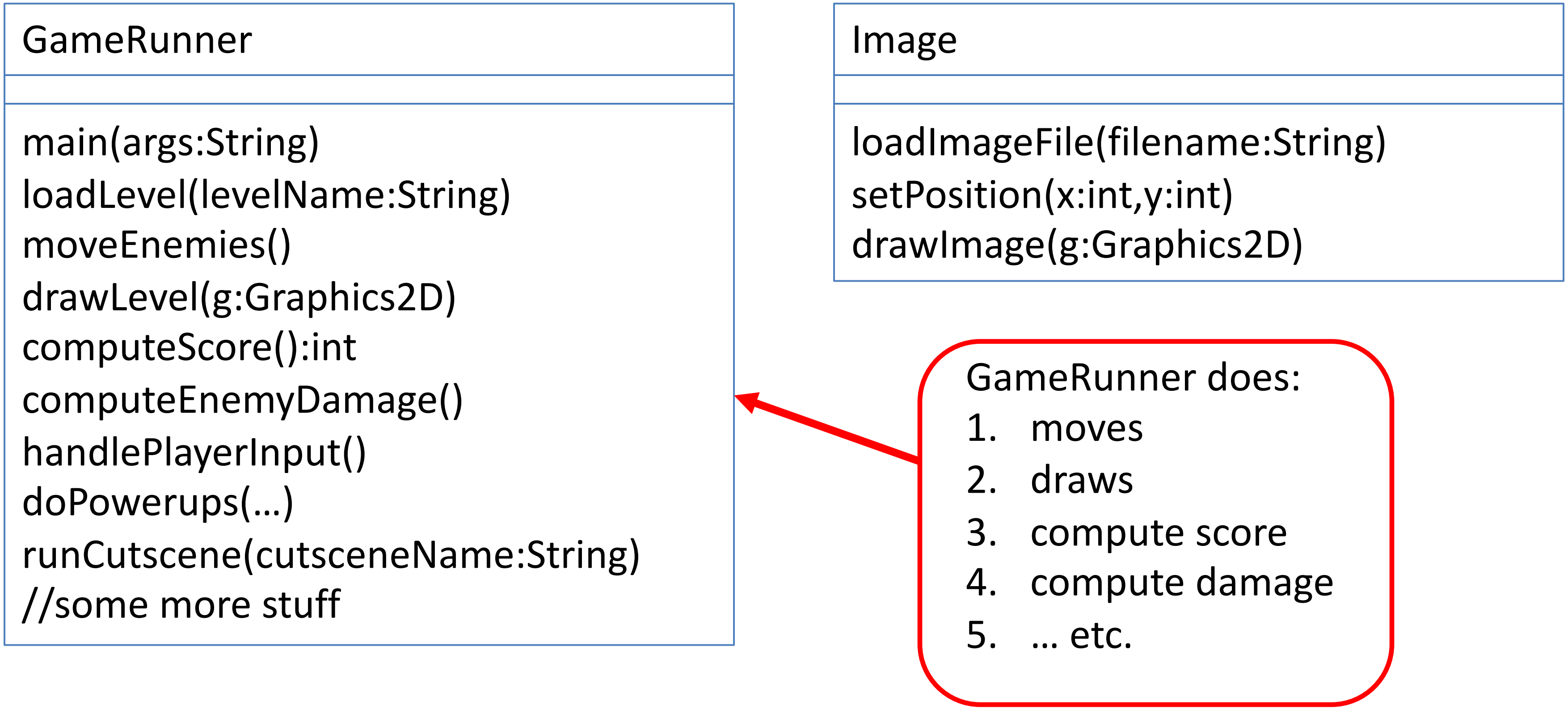
main(args:String)
loadLevel(levelName:String)
moveEnemies()
drawLevel(g:Graphics2D)
computeScore():int
computeEnemyDamage()
handlePlayerInput()
doPowerups(…)
runCutscene(cutsceneName:String)
//some more stuff

Image

loadImageFile(filename:String)
setPosition(x:int,y:int)
drawImage(g:Graphics2D)

*Note that in both these classes I've omitted the fields for clarity

Imagine I want to make a Video Game
Here are two classes in my design
Which is one has low cohesion?



GameRunner

main(args:String)
loadLevel(levelName:String)
moveEnemies()
drawLevel(g:Graphics2D)
computeScore():int
computeEnemyDamage()
handlePlayerInput()
doPowerups(…)
runCutscene(cutsceneName:String)
//some more stuff

Image

loadImageFile(filename:String)
setPosition(x:int,y:int)
drawImage(g:Graphics2D)

GameRunner does:

1. moves
2. draws
3. compute score
4. compute damage
5. … etc.

*Note that in both these classes I've omitted the fields for clarity

Cohesion – From Textbook

- A class should represent a single concept. All interface features should be closely related to the single concept that the class represents. Such a class is said to be cohesive.

  - Your textbook

On to coupling...

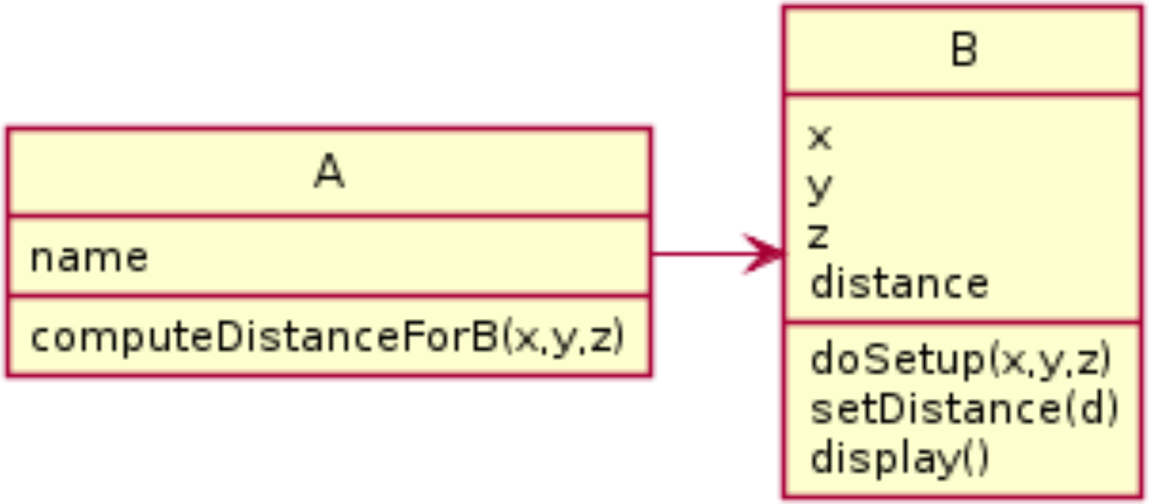
Coupling

- Coupling is when one object depends strongly on another

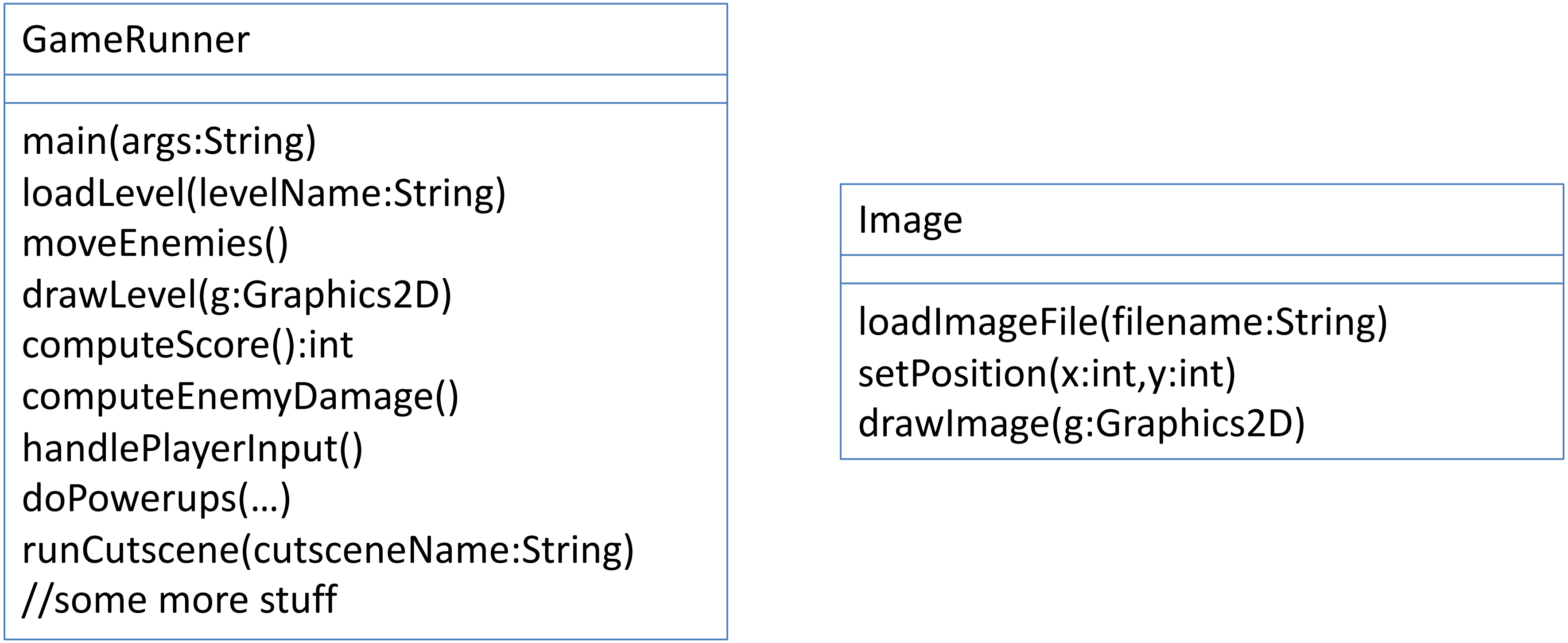
```
//do setup must be called first
this.myB.doSetup(1, 2, 3);

//now we compute the parameter
double distance = computeDistanceForB(0,0,0);
this.myB.setDistance( distance );

//finally we display
this.myB.display();
```



Note that in this design, GameRunner probably had many objects of the image class, but Image does not know the GameRunner class even exists. That’s a sign of low coupling between Image and GameRunner.



GameRunner

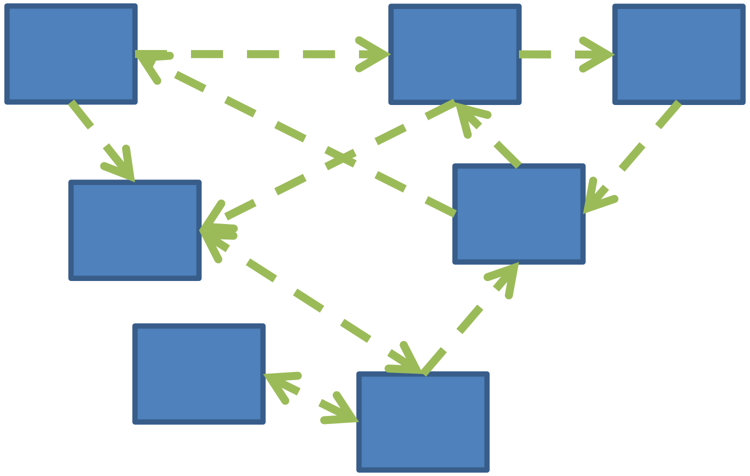
main(args:String)
loadLevel(levelName:String)
moveEnemies()
drawLevel(g:Graphics2D)
computeScore():int
computeEnemyDamage()
handlePlayerInput()
doPowerups(…)
runCutscene(cutsceneName:String)
//some more stuff

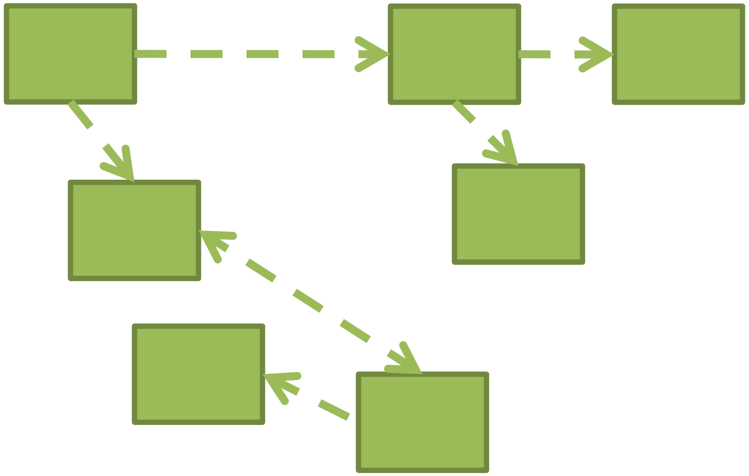
Image

loadImageFile(filename:String)
setPosition(x:int,y:int)
drawImage(g:Graphics2D)

Coupling – UML Diagrams

- Lot’s of dependencies → high coupling
- Few dependencies → low coupling



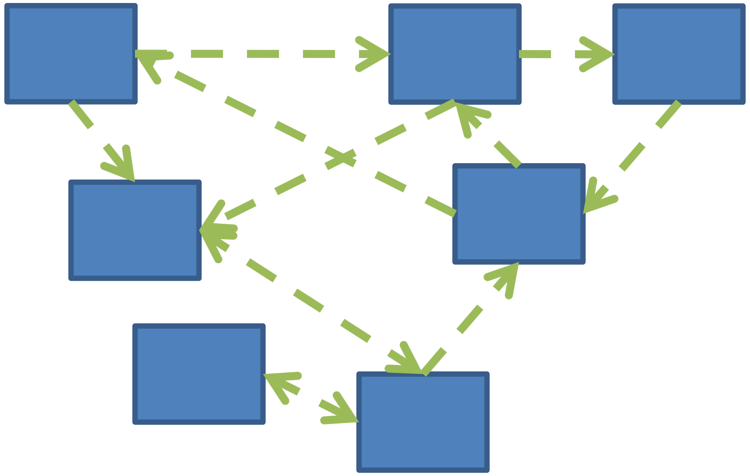


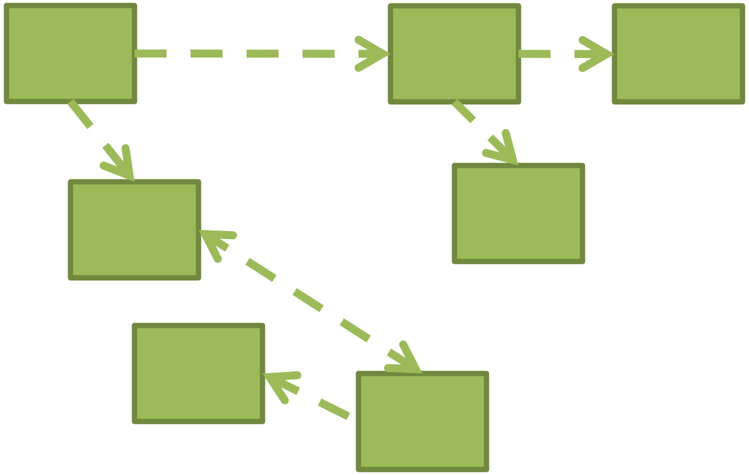
How hard will it be to change code with:

High coupling? Low coupling?

Coupling – UML Diagrams

- Note:
- “essential” dependencies cannot be eliminated
- if they are eliminated, then functionality fails





If we do our design job carefully

- Divide & Conquer - Break our larger problem into several classes
- Each of these classes will do one thing well (i.e. they will have *high cohesion*)
- Our classes will only need to depend on each other in specific, highly limited essential ways (i.e. they will have *low coupling*).
- Many classes won't even be "know" of most of the other classes in the system

Imagine that you’re writing code to manage a school’s students

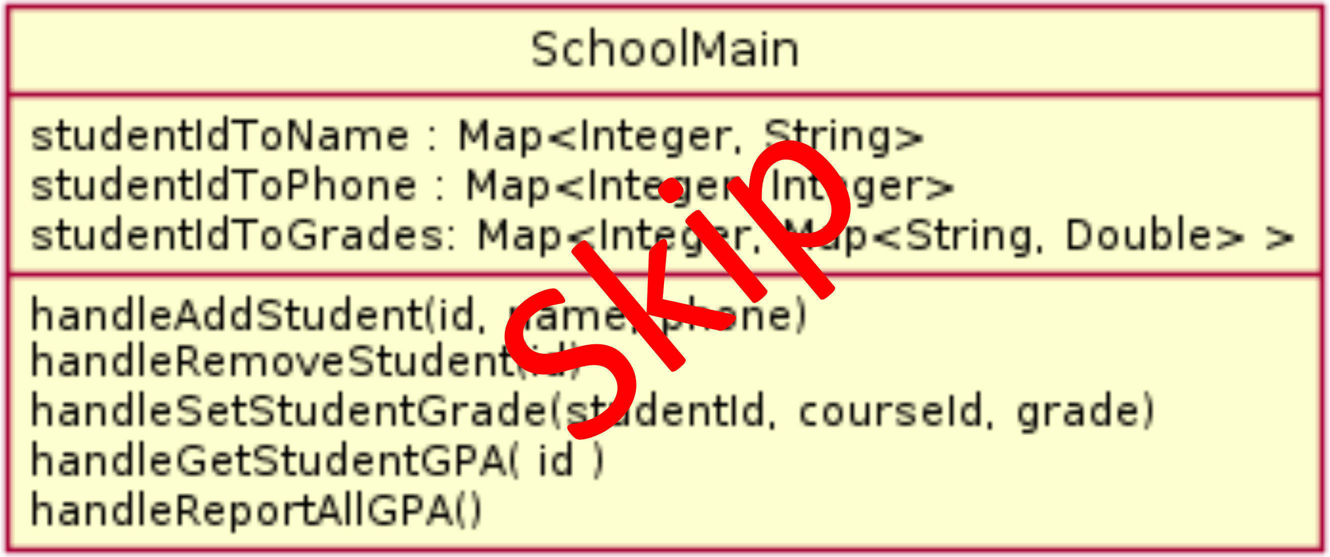
Things your design should accommodate:

- Handle adding or removing students from the school
- Students should have a name, phone number, and grades for specific courses (can use a courseId String)
- Setting the individual course grades for a particular student
- Compute the average GPA of all the students in the school
- Sort the students by last name to print out a report of students and GPA

Skip

Discuss and come up with a design with those near you. How many classes does your system need?

1 class solution



SchoolMain

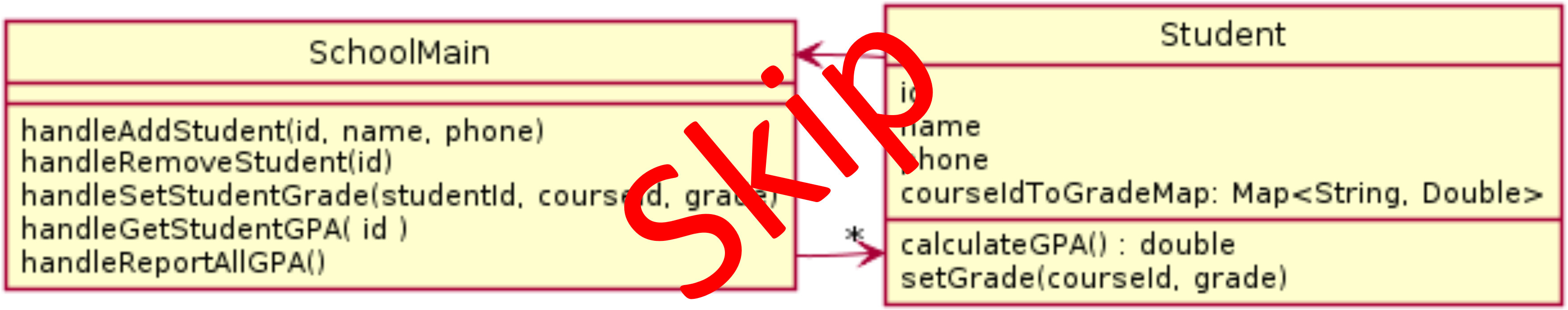
studentIdToName : Map<Integer, String>
studentIdToPhone : Map<Integer, Integer>
studentIdToGrades: Map<Integer, Map<String, Double> >

handleAddStudent(id, name, phone)
handleRemoveStudent(id)
handleSetStudentGrade(studentId, courseId, grade)
handleGetStudentGPA(id)
handleReportAllGPA()

Skip

- Coupling?
- Cohesion?

2 class solution



SchoolMain

handleAddStudent(id, name, phone)
handleRemoveStudent(id)
handleSetStudentGrade(studentId, courseId, grade)
handleGetStudentGPA(id)
handleReportAllGPA()

Student

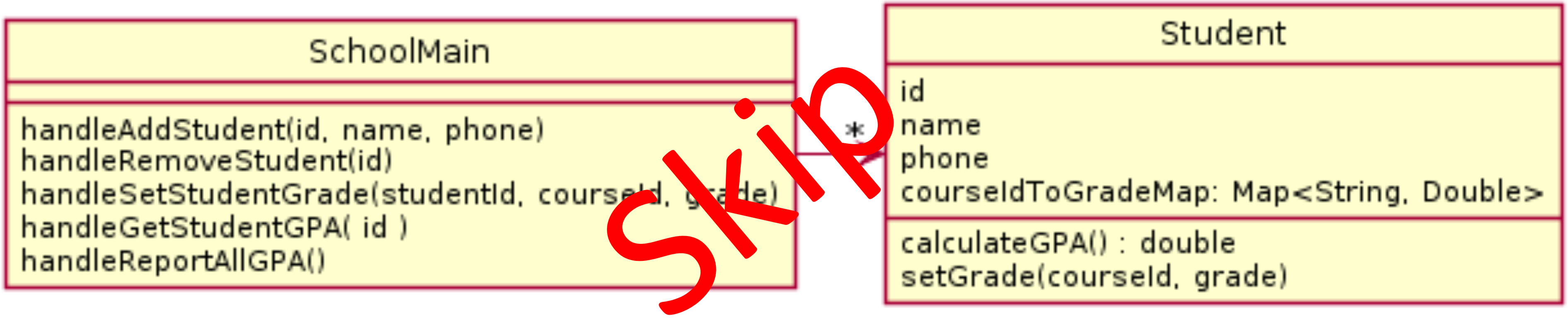
id
name
phone
courseIdToGradeMap: Map<String, Double>

calculateGPA() : double
setGrade(courseId, grade)

Skip

- Coupling?
- Cohesion?

2 class solution



SchoolMain

handleAddStudent(id, name, phone)
handleRemoveStudent(id)
handleSetStudentGrade(studentId, courseId, grade)
handleGetStudentGPA(id)
handleReportAllGPA()

*

Student

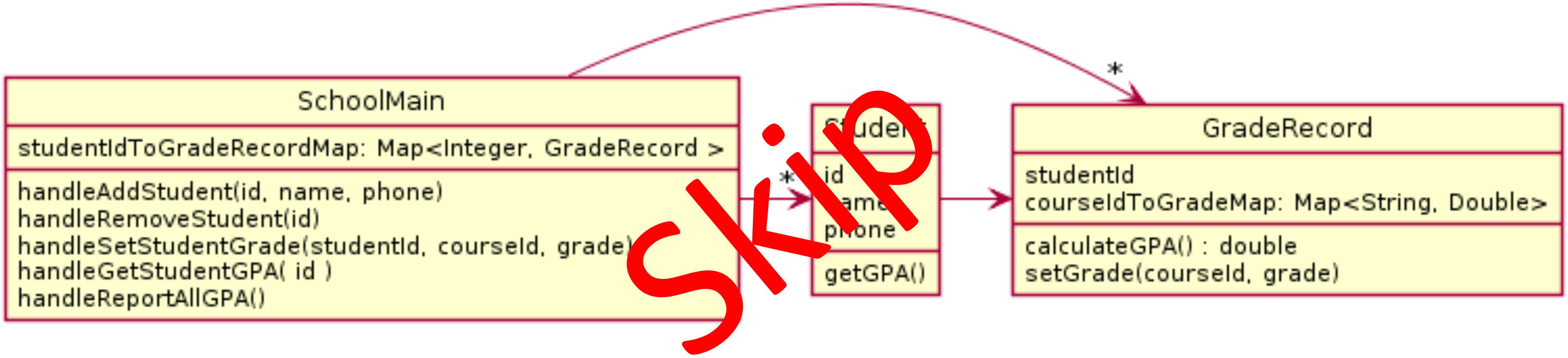
id
name
phone
courseIdToGradeMap: Map<String, Double>

calculateGPA() : double
setGrade(courseId, grade)

Skip

- Coupling?
- Cohesion?

3 classes



SchoolMain

studentIdToGradeRecordMap: Map<Integer, GradeRecord >

handleAddStudent(id, name, phone)
handleRemoveStudent(id)
handleSetStudentGrade(studentId, courseId, grade)
handleGetStudentGPA(id)
handleReportAllGPA()

Student

id
name
phone

getGPA()

GradeRecord

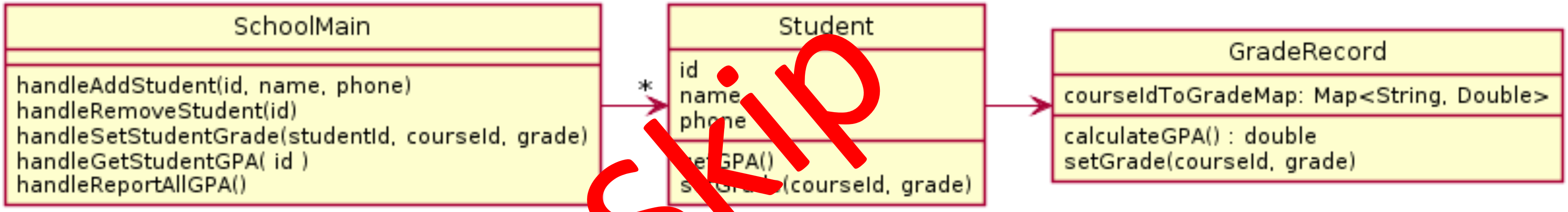
studentId
courseIdToGradeMap: Map<String, Double>

calculateGPA() : double
setGrade(courseId, grade)

Skip

- Coupling?
- Cohesion?

3 classes improved



SchoolMain

handleAddStudent(id, name, phone)
handleRemoveStudent(id)
handleSetStudentGrade(studentId, courseId, grade)
handleGetStudentGPA(id)
handleReportAllGPA()

*

Student

id
name
phone

getGPA()
setGrade(courseId, grade)

GradeRecord

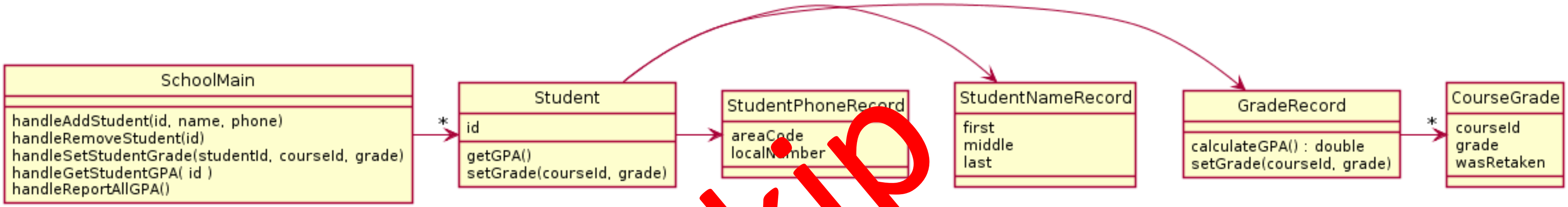
courseIdToGradeMap: Map<String, Double>

calculateGPA() : double
setGrade(courseId, grade)

Skip

- Coupling?
- Cohesion?

…6 classes



SchoolMain

handleAddStudent(id, name, phone)
handleRemoveStudent(id)
handleSetStudentGrade(studentId, courseId, grade)
handleGetStudentGPA(id)
handleReportAllGPA()

Student

id

getGPA()
setGrade(courseId, grade)

StudentPhoneRecord

areaCode
localNumber

StudentNameRecord

first
middle
last

GradeRecord

calculateGPA() : double
setGrade(courseId, grade)

CourseGrade

courseId
grade
wasRetaken

Skip

- Coupling?
- Cohesion?

Note that

- Cohesion makes us want:
  - Many smaller classes
  - Classes that do only one thing well
- If classes are too small
  - Tend to need to depend on each other
  - Coupling rises
- Want “Goldilocks” design

Rule of Thumb: No Global Variables

- Or static variables that are used like globals
- A static variable can be accessed/modified in any function at any time
- As a result many parts of the code can be coupled to a single class

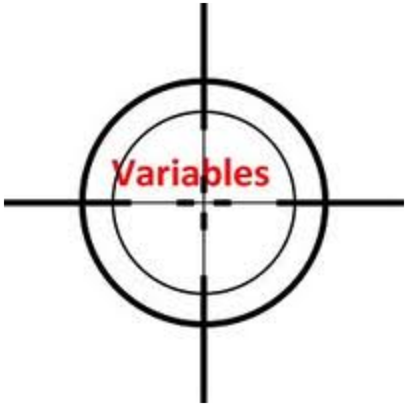
Rule of Thumb: No Global Variables

- Or static variables that are used like globals
- A static variable can be accessed/modified in any function at any time
- As a result many parts of the code can be coupled to a single class

- Why?
- Increases coupling among all the clients that get or change value of the global variable

Stop Here Today

Variable Scope



Scope **is the region of a program in which a variable can be accessed**

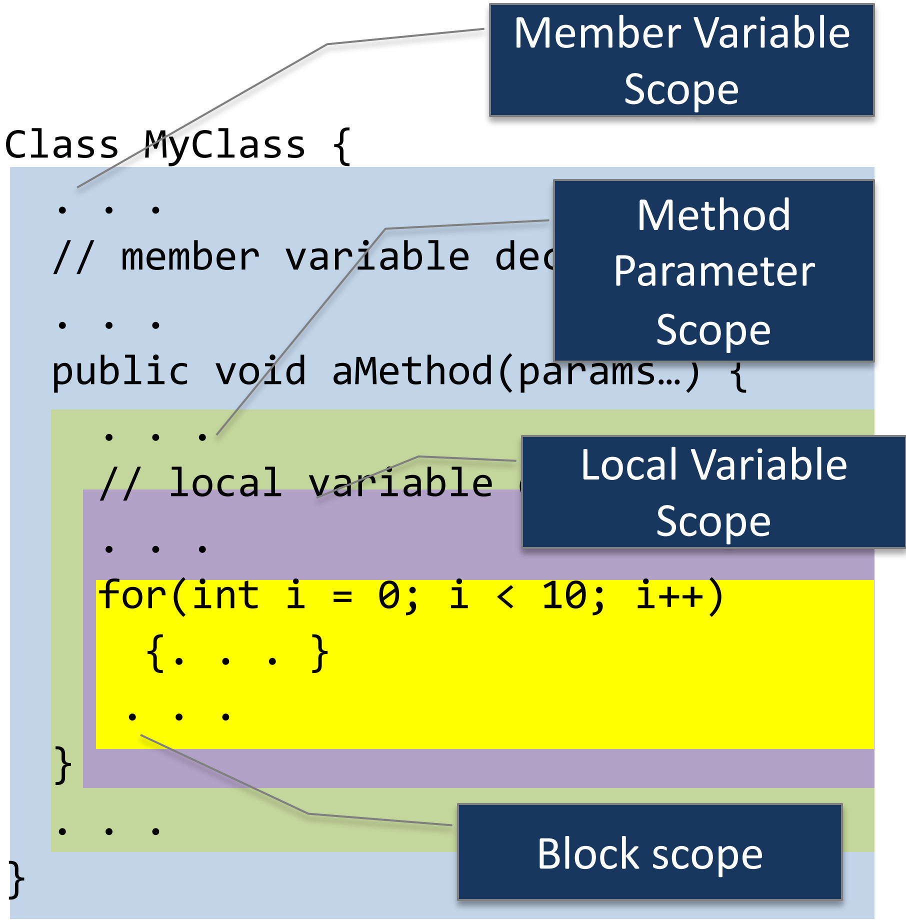
- *Parameter scope:* the whole method body
- *Local variable scope:* from declaration to block end

```
public double myMethod() {
    double sum = 0.0;
    Point2D prev = this.pts.get(this.pts.size() - 1);
    for (Point2D p : this.pts) {
      sum += prev.getX() * p.getY();
      sum -= prev.getY() * p.getX();
      prev = p;
    }
    return Math.abs(sum / 2.0);
}
```

Member Scope (Field or Method)

- *Member scope:* anywhere in the class, including *before* its declaration
  - Lets methods call other methods later in the class

- `public static` class members can be accessed from outside with “class qualified names”
  - `Math.sqrt()`
  - `System.in`



```
Class MyClass {
  . . .
  // member variable dec
  . . .
  public void aMethod(params…) {
    . . .
    // local variable 
    . . .
    for(int i = 0; i < 10; i++)
      {. . . }
     . . .
  }
  . . .
}
```

Overlapping Scope and Shadowing

```
public class TempReading {
     private double temp;

     public void setTemp(double temp) {
           this.temp = temp;

     }
     // …
}
```

What does this “temp” refer to?

Always qualify field references with `this`. It prevents accidental shadowing.

Work Time

- ImplementingDesign2 – see due date on schedule page
- ImplementingDesign1

ImplementingDesign2 Notes:

- You will be given a starter uml file for plantuml
- You must pass the unit tests, but don't approach this by trying to a pass one test at a time
- Instead test functionality as you go by running commands
  - Make a UML DESIGN BEFORE you code
  - It is required that you submit a first draft
  - (It does not have to be perfect, we expect you to have to change)