CSSE 220

Coupling and Cohesion
Static variables

Please turn in your assignment at the back

Review of Design Problems

- Turn in your homework before we go over solution

Today’s topic

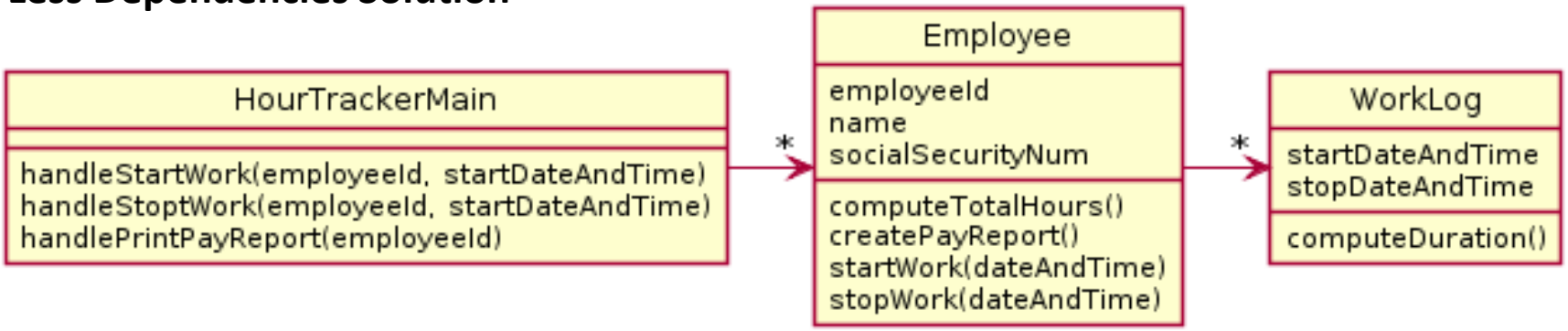
- **Minimize dependencies** between objects when it does not disrupt usability or extendability
  - Tell don't ask
  - Don't have message chains

Principles of Design (for CSSE220)

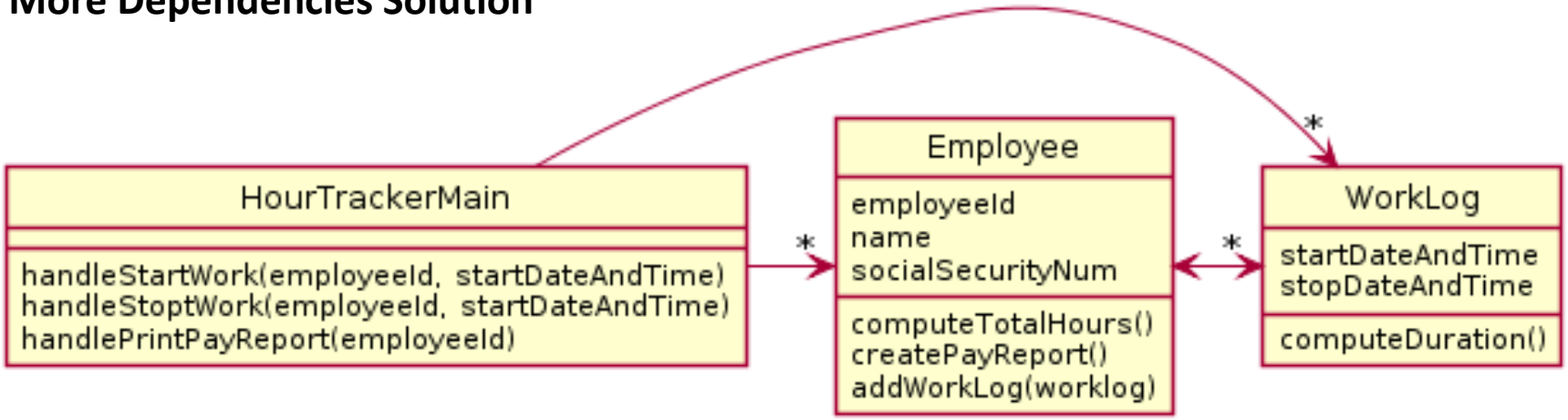
- Make sure your design **allows proper functionality**
  - Must be able to **store required information** (one/many to one/many relationships)
  - Must be able to **access the required information** to accomplish tasks
  - Data should **not be duplicated** (id/identifiers are OK!)
- Structure design **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 that includes total hours, the employee's name, social security number, and employee id.

Less Dependencies Solution



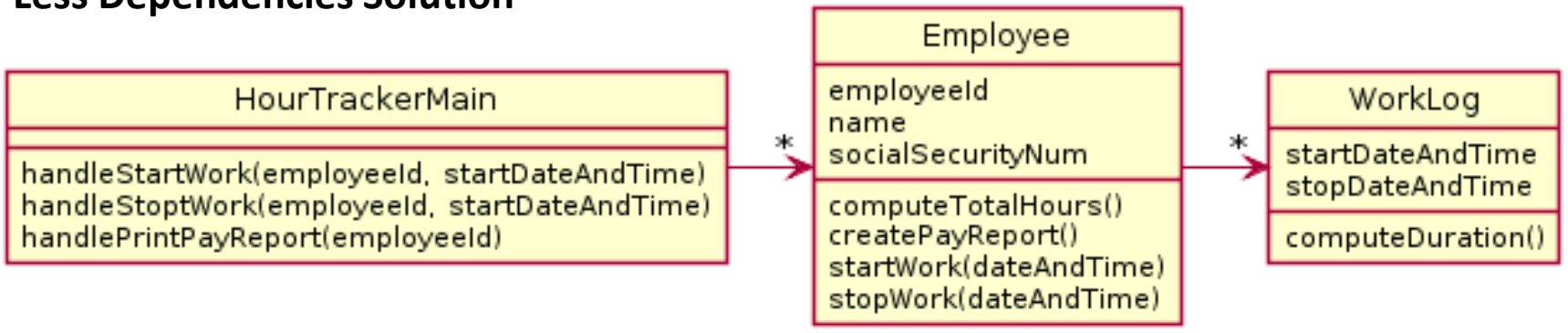
More Dependencies Solution



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.

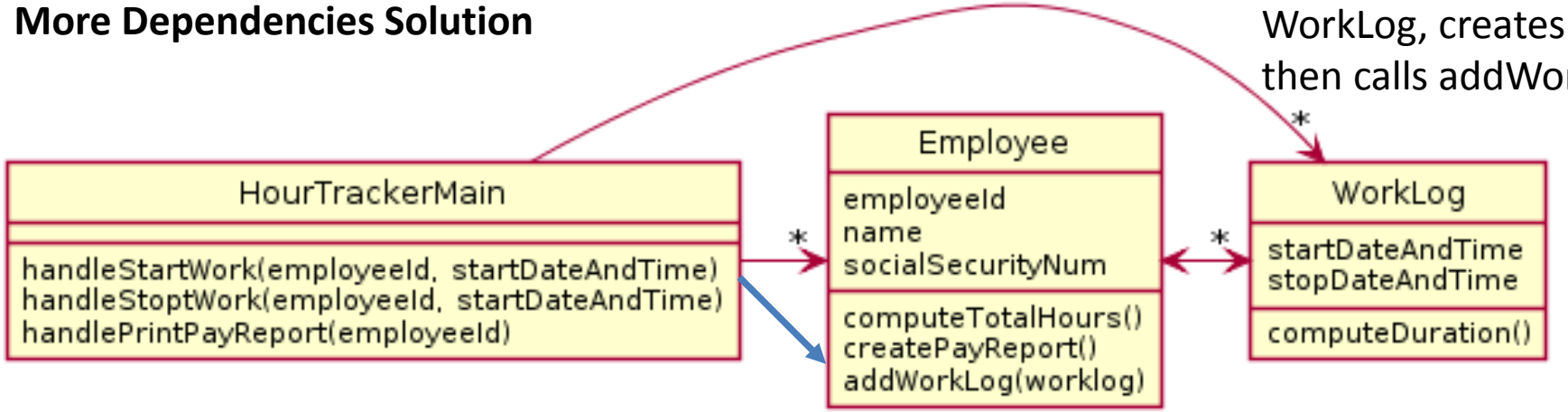
The less dependencies solution is also simpler. Employee fully “owns” all it’s own data. In more dependencies, the worklog is edited without employee’s knowledge.

Less Dependencies Solution

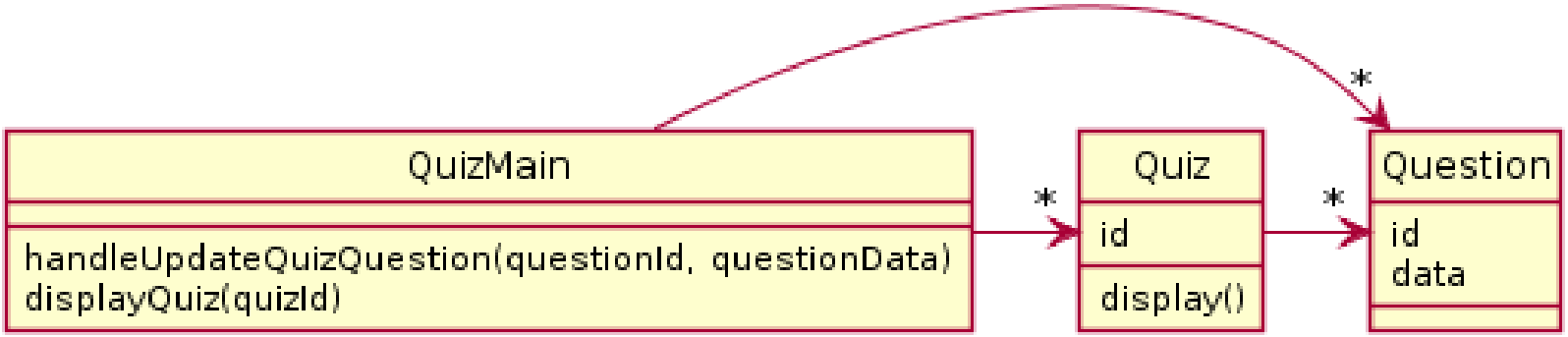


More Dependencies Solution

HourTrackerMain “knows” about WorkLog, creates one, then calls addWorkLog



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
```

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);
```

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 EmailClient getEmailClient() {
   return this.emailClient;
}
```

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

Violates "separation of concerns" – Client now knows "how" the called on code works. This increases "coupling" between client and called-on class/code, high coupling is usually poorer design choice

```
public void pruneContactList() {
   this.emailClient.removeDupContacts();
}
```

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

Tell Don’t Ask

- Be wary of getter methods
- Prefer methods that command (tell) a class to do something and be responsible for its own state and responsibilities
- If client code in class A accesses a lot of internal data of another class, B, consider if a tell method in that other class, B, 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 that 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 that calculates the average

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

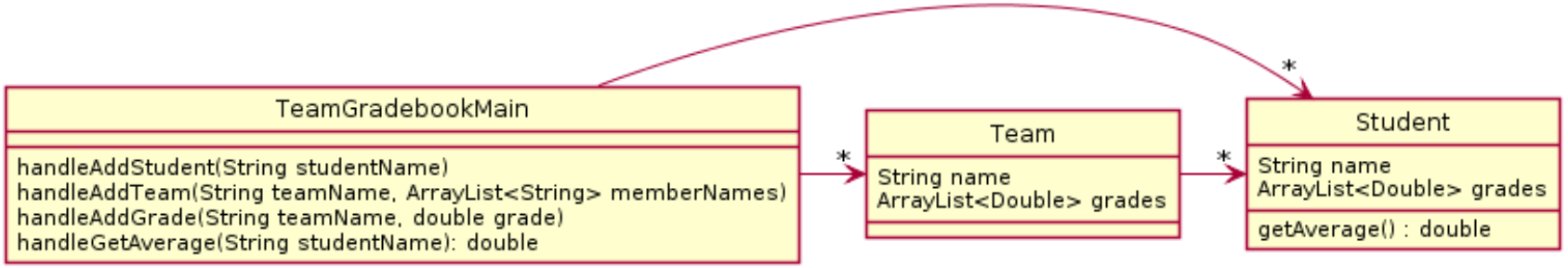
A simple example of Tell Don't Ask

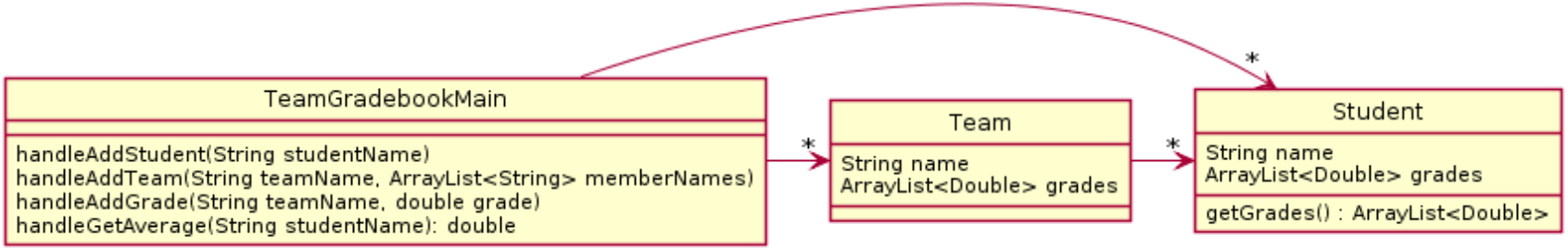
In your TeamGradebook classes, you need to calculate a student's average grade. This could be accomplished by:

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!





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!

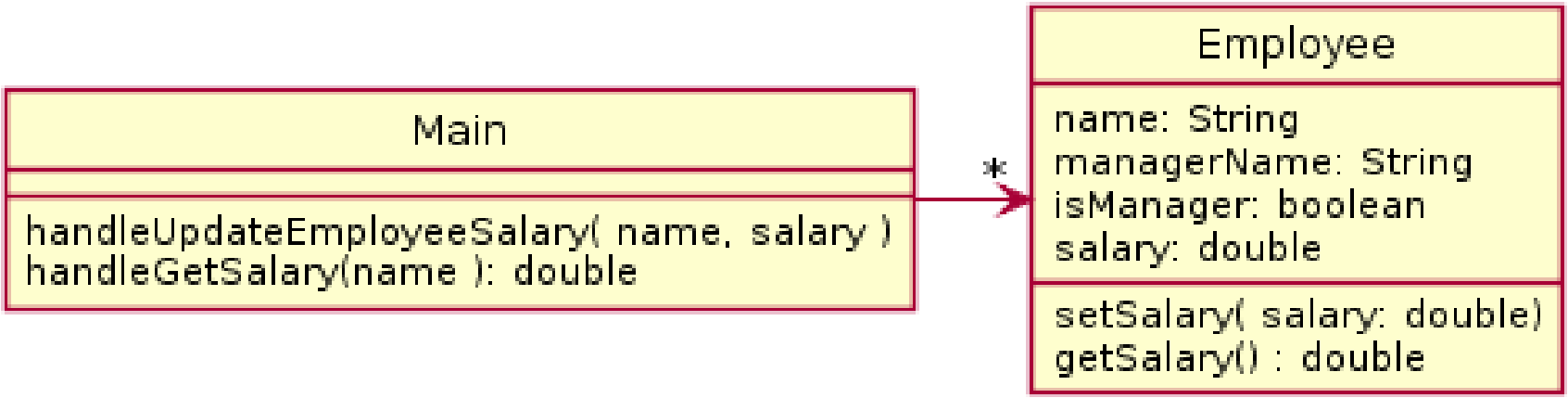
Why does this improve the design?

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 (i.e. drop lowest score)

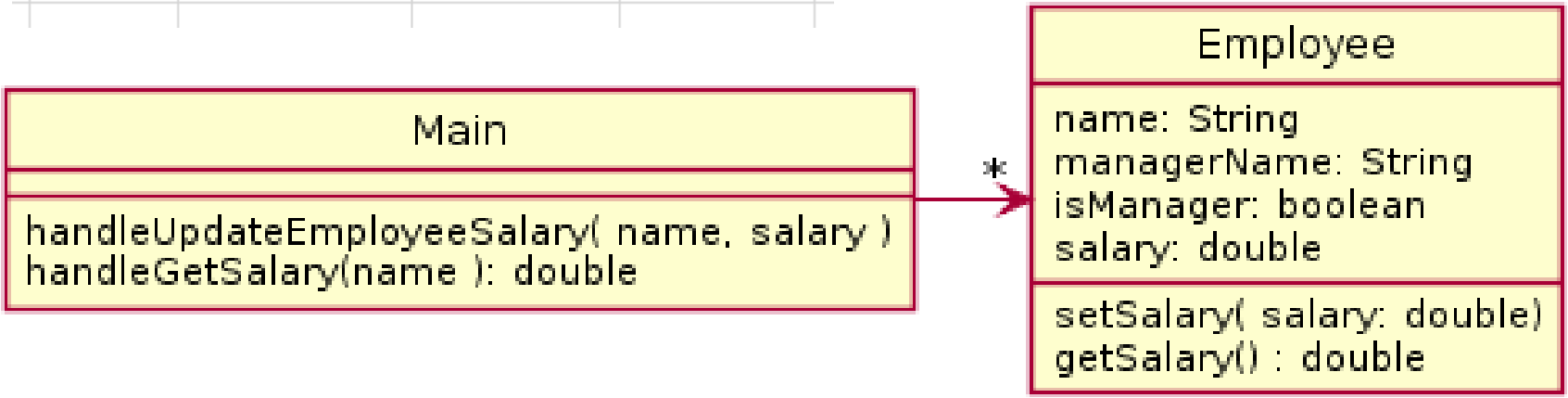
Employee Salary Problem
In-Class Quiz Questions #1 & #2

There is a company that has employees, each of which has a salary. There are managers that oversee other employees. Employees have salaries that 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.

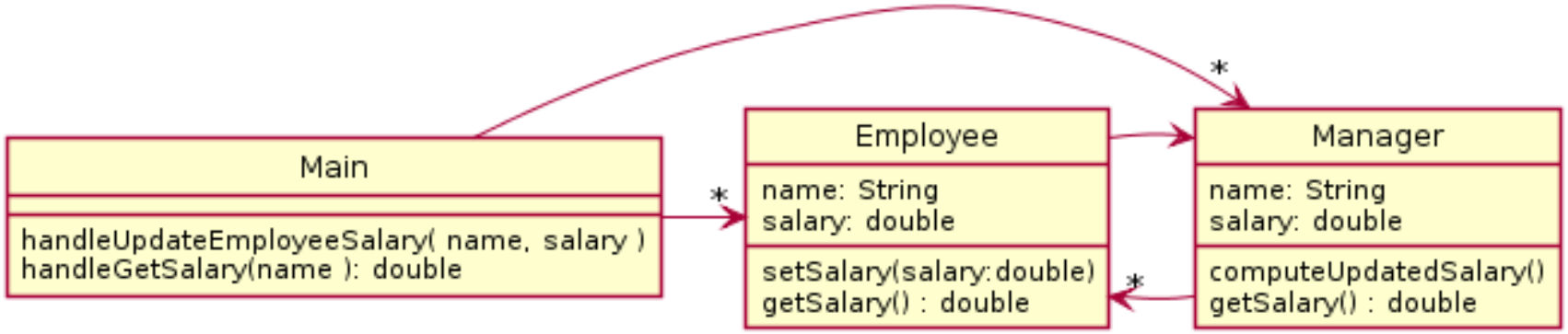


Employee Salary Problem
In-Class Quiz Questions #1 & #2

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

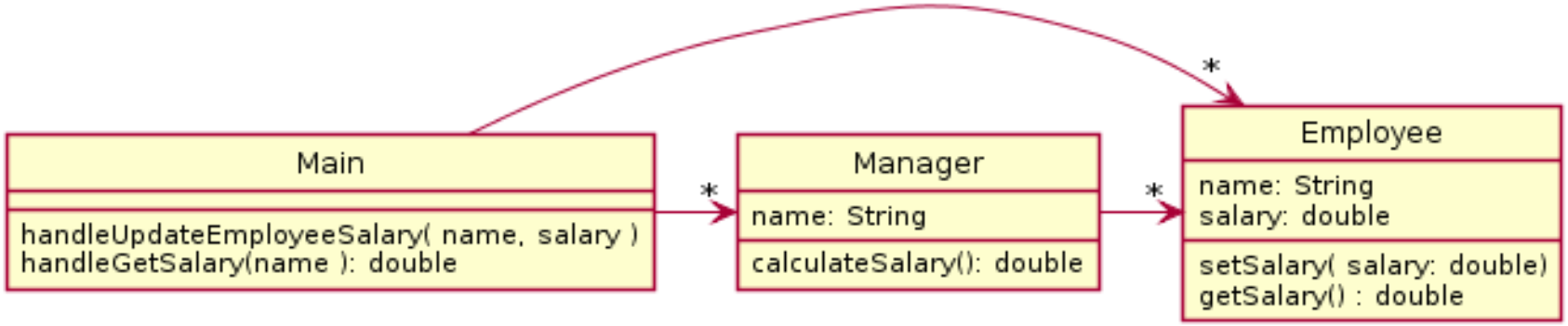


Better Solution



- Anything wrong?
- Room to improve?

Eliminate manager salary field!



Data is technically duplicated if manager contains its own salary field.

What if the two pieces of data were out of sync?

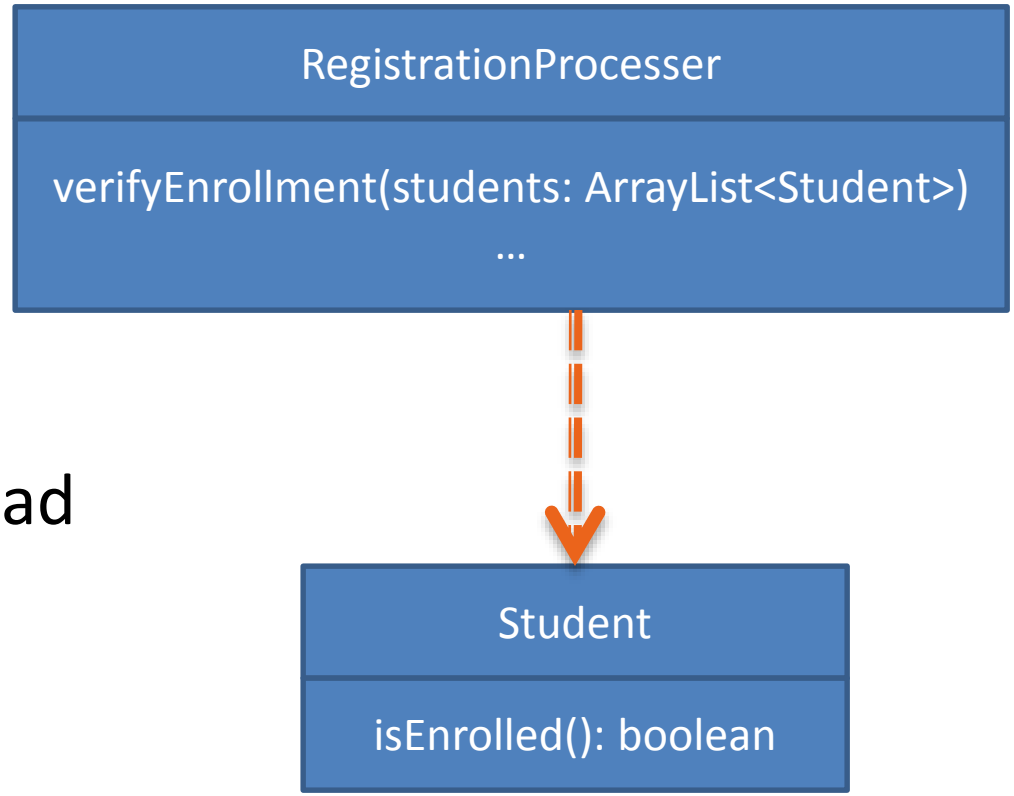
Works well to calculate the salary as needed since it depends upon other data.

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);
```

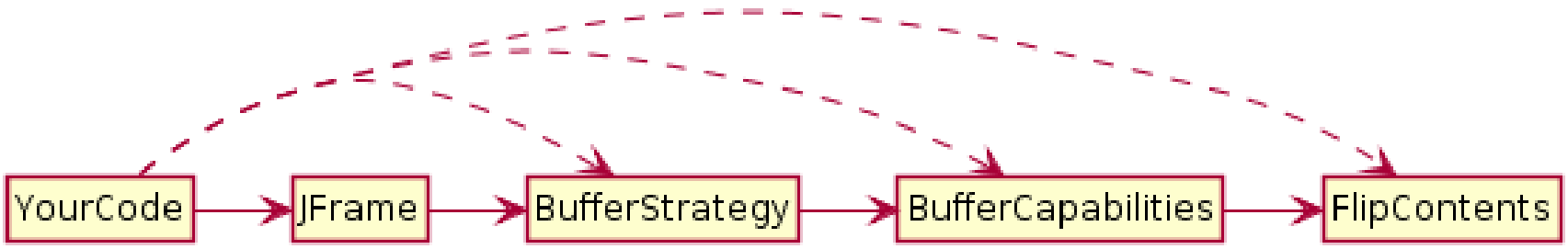
This is generally considered to be a warning sign of excessive dependency and problems.

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

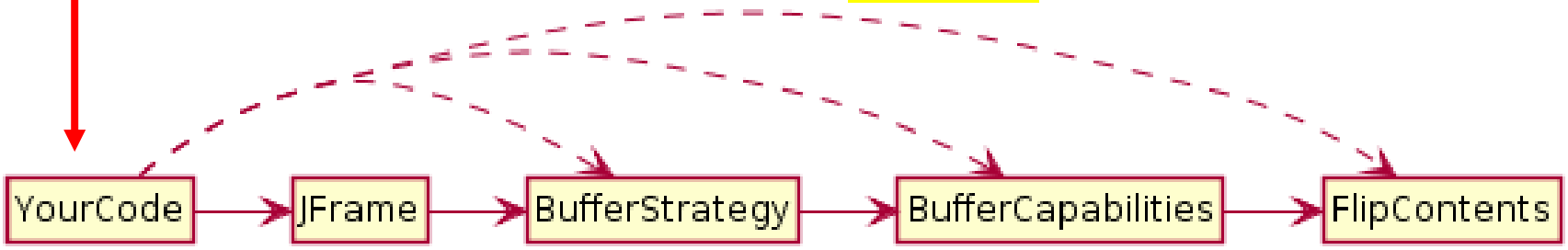
Message chains are not better if you space them across multiple lines, but it does make it more obvious what the problem is.

```
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

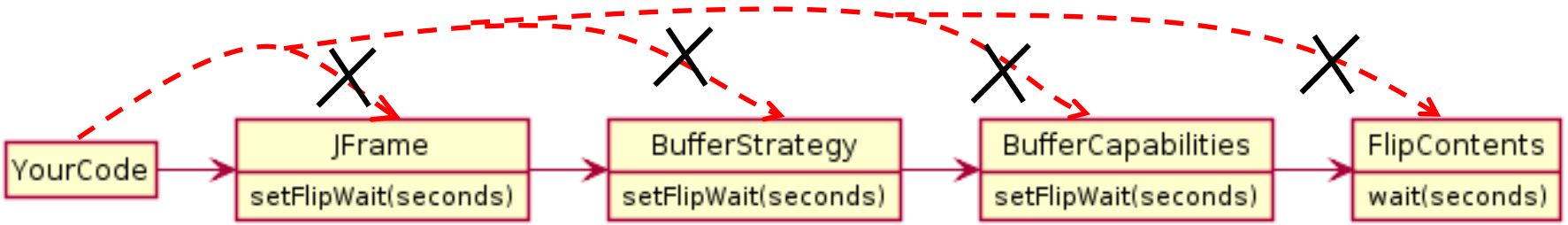
So client is highly coupled with "how" things work



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);
```

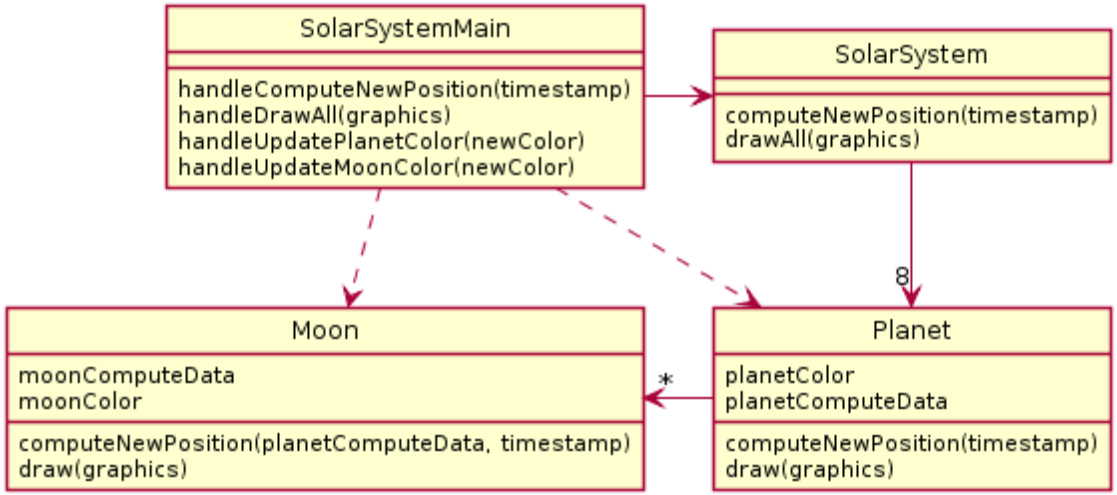


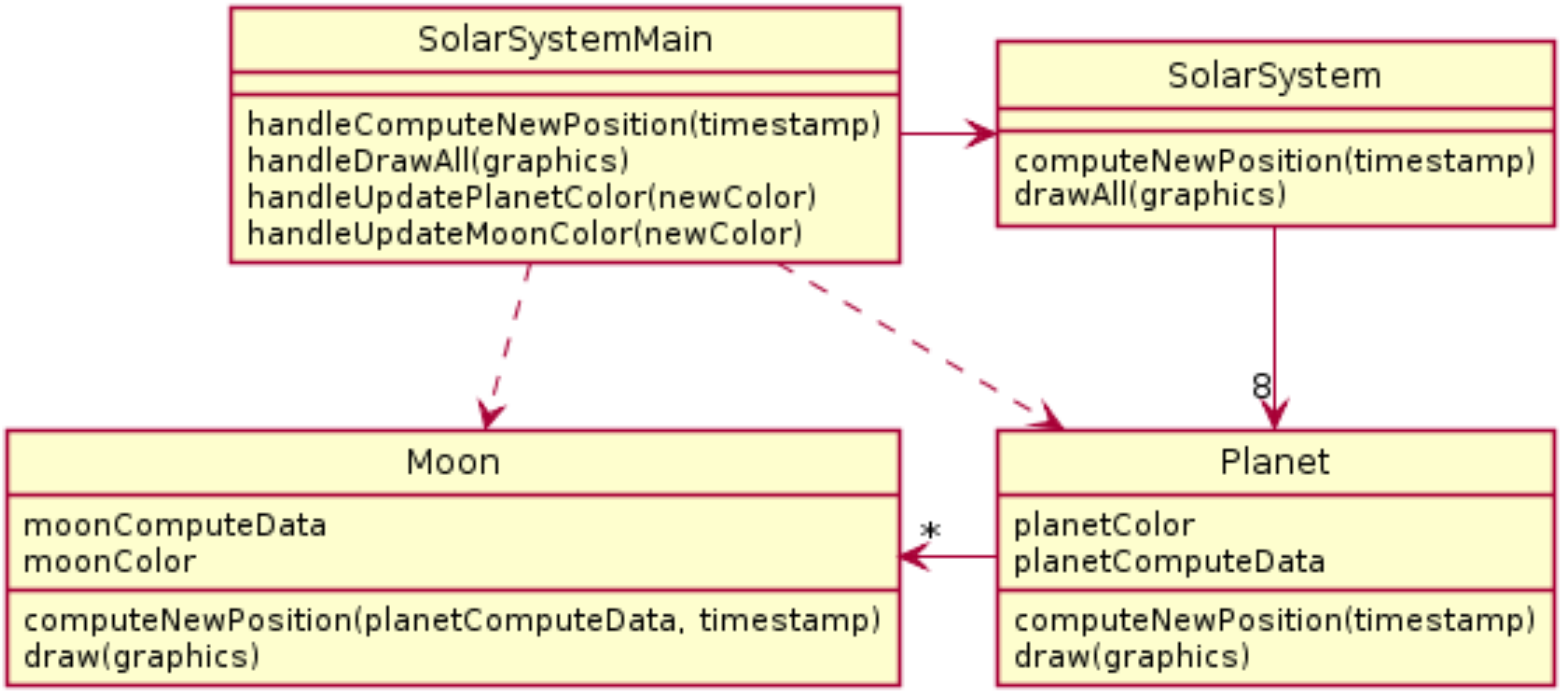
This approach also actually decouples:

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

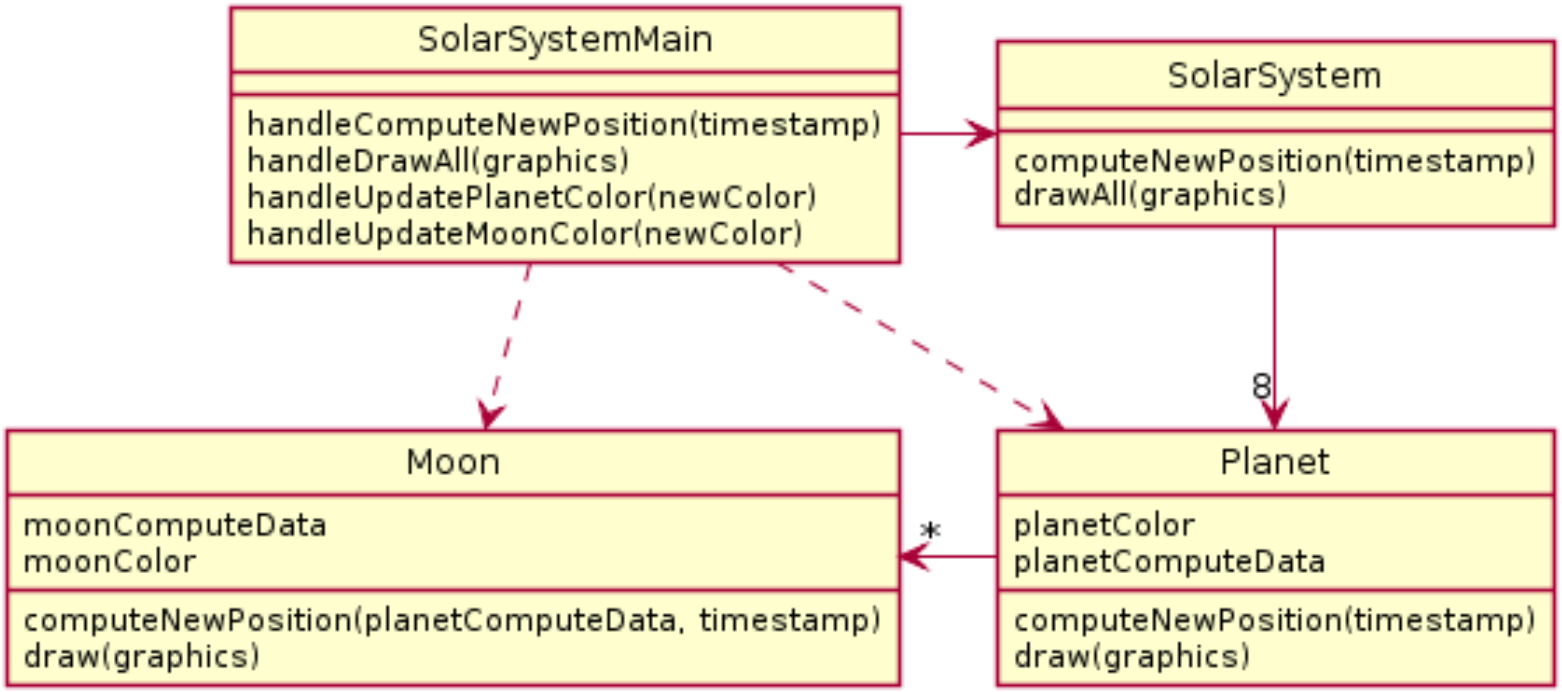
Solar System Problem
In-Class Quiz Questions #4 & #5

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.





- What is wrong here?

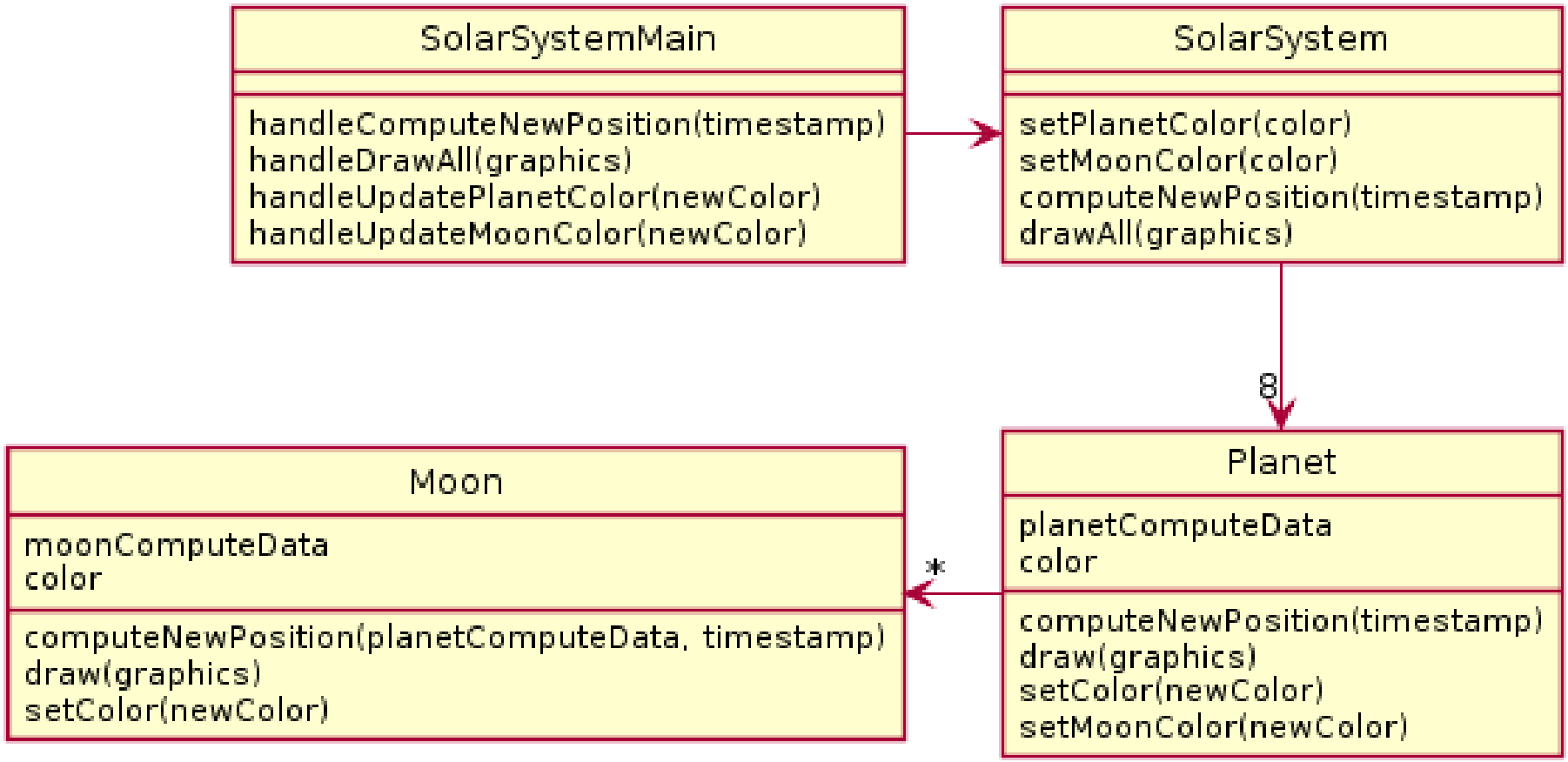


- What is wrong here?

4b. methodChain to update a moon

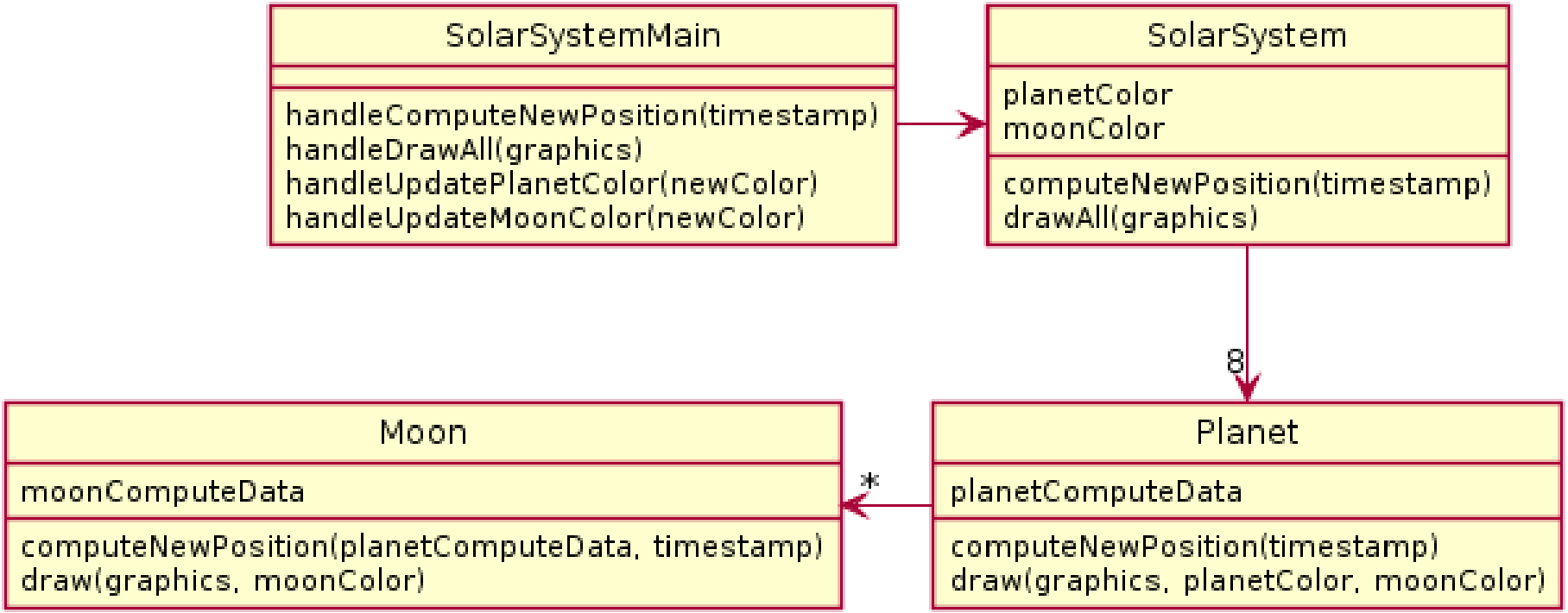
```
ss.getPlanets().get(0).getMoons().get(0).setColor(color);
```

Partial Solution



Better Solution

Eliminate Data Duplication



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

Goals

- Learn 3 essential object oriented design terms:
  - Encapsulation (done- last class)
  - Coupling
  - Cohesion
- Static fields (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 is more cohesive?

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 more cohesive?

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.

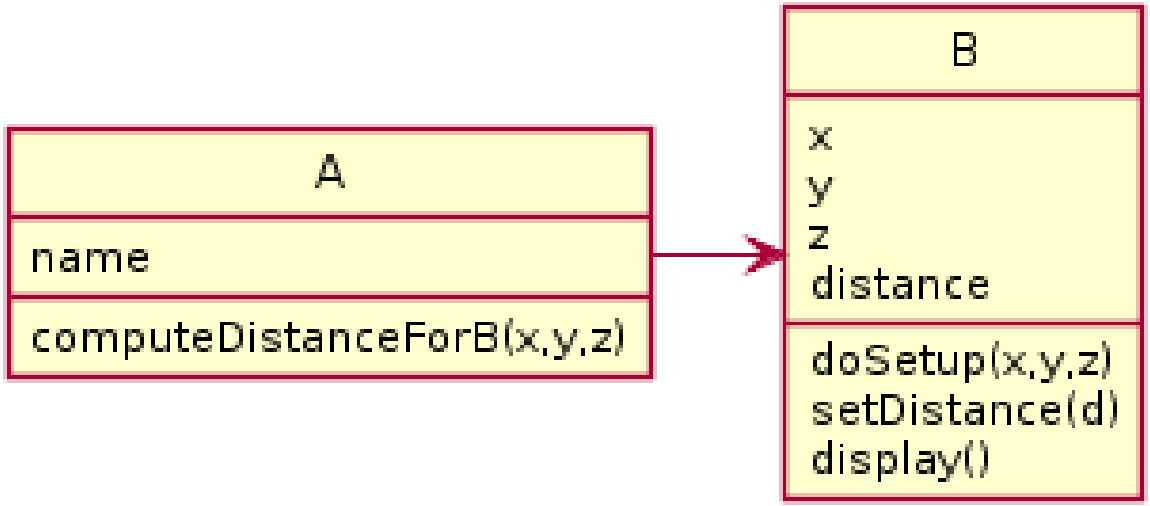
  - Your textbook

On to coupling...

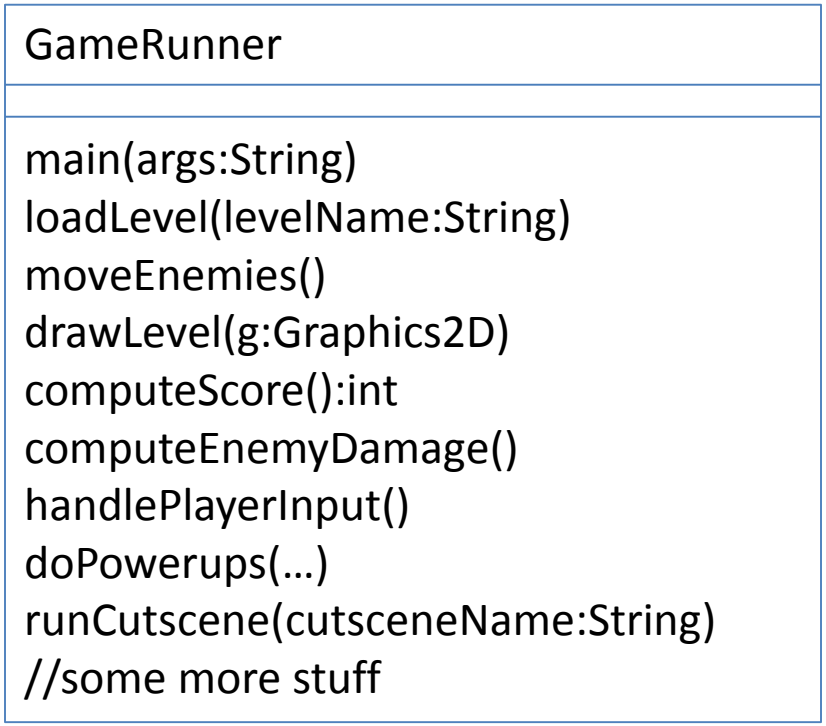
Coupling

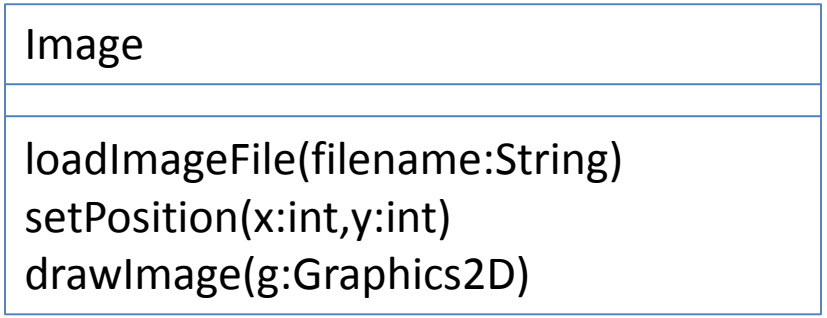
- Coupling is when one object depends strongly on another

```
//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();
```



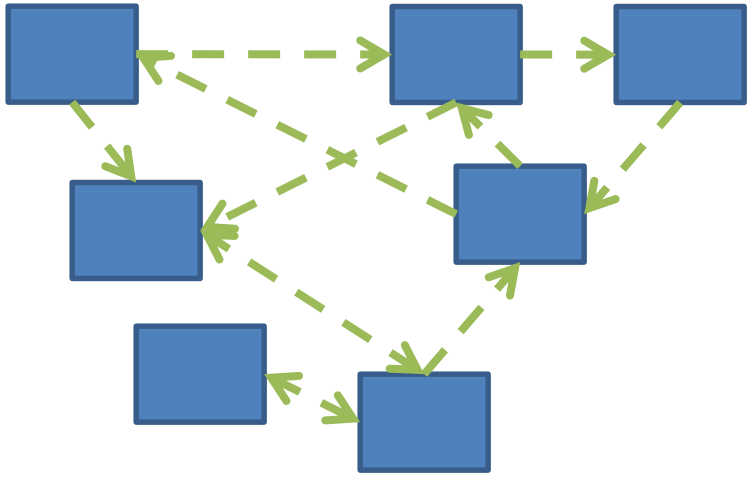
Note that in this design, GameRunner probably had many objects of the image class, but Image does not know the GameRunner class even exists. That's a sign of low coupling between Image and GameRunner.

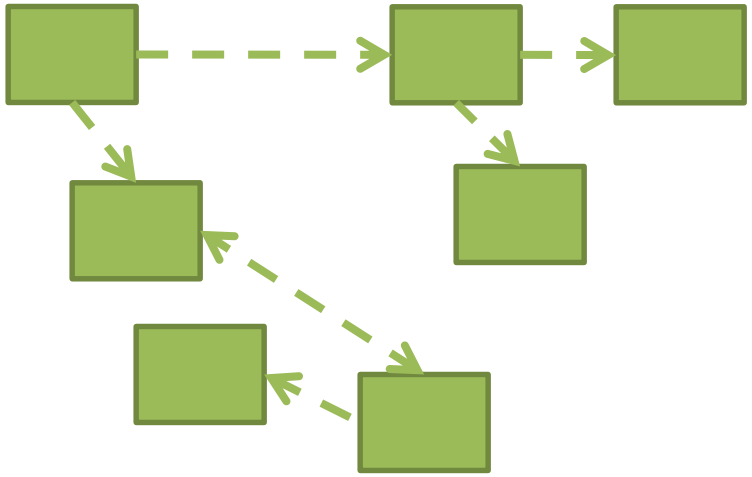




Coupling – UML Diagrams

- Lot's of dependencies ➔ high coupling
- Few dependencies ➔ low coupling

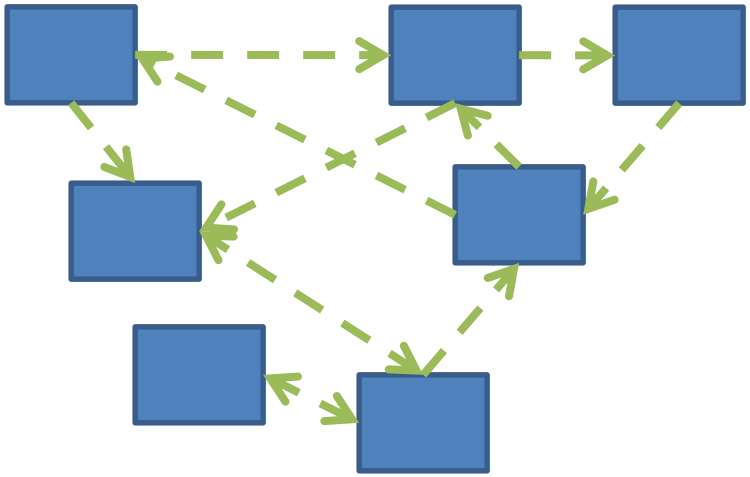


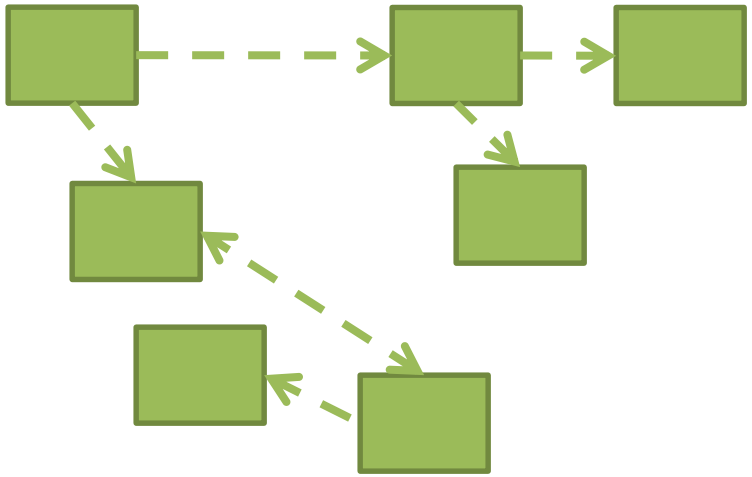


How hard will it be to change code with:
High coupling? Low coupling?

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 "know" of most of the other classes in the system

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

Next Up Static variables

- If time allows!

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

Reminders

- ImplementingDesign2 – see due date on schedule page
- DesignProblem3 – see due date on schedule page
- **Due tonight submit via your git repo:**
  - Implementation of ImplementingDesign1 in code **(20 points)**
  - Demonstrating functionality in main!
  - Final UML: submit to repo along with other files **(5 points)**
  - Reflection on the process: **reflection_questions.txt (5 points)**
- Exam1 Wrapper: **Due first class after break**
  - optional assignment to reflect on exam prep and to earn back % of points on Exam1
  - Complete Moodle Survey

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)