Putting It All Together

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 14 — 10/09/2008

Lecture Goals

- Review material from Chapter 10 of the OO A&D textbook
    - The OO A&D project life cycle
    - The Objectville Subway Map Example (in python)
    - Dijkstra’s Algorithm
- Discuss the example application of Chapter 10
- Emphasize the OO concepts and techniques encountered in Chapter 10

The OO A&D Project Life Cycle

- All software development projects require a software development life cycle to organize the work performed by their developers
  - Even when there is only one developer and the life cycle being used is "code and fix"
- After nine chapters and ten lectures, we have established a fairly complete look at the various stages of an OO A&D project life cycle
  - The overall life cycle consists of three stages
    - Make sure your software does what the customer wants it to do
    - Apply basic OO principles to add flexibility
    - Strive for a maintainable, reusable design
  - But there are many activities associated with these stages
    - plus iteration and testing!

The Activities

1. Feature List (1)	High-level view of app’s functions
2. Use Case Diagrams (1)	Types of Users and Tasks
3. Break Up the Problem (1)	Divide and Conquer
4. Requirements (1)	Gather requirements for module
5. Domain Analysis (1)	Fill in use-case details
6. Preliminary Design (2)	Class diagram, apply OO principles
7. Implementation (2/3) Repeat steps 4-7 until done	Write code, test it
8. Delivery	You’re Done

Example: Objectville Subway

- To emphasize how much we have learned, the book performs this life cycle on a project that models a subway system and can find the shortest route between any two subway stops
  - We'll review portions of this process
    - And code the solution in Python for comparison
- Vision Statement from Objectville Travel
  - They need a RouteFinder for the Objectville Subway System
  - First step?
    - Feature List

Phase One: Feature List

Objectville RouteFinder Feature List

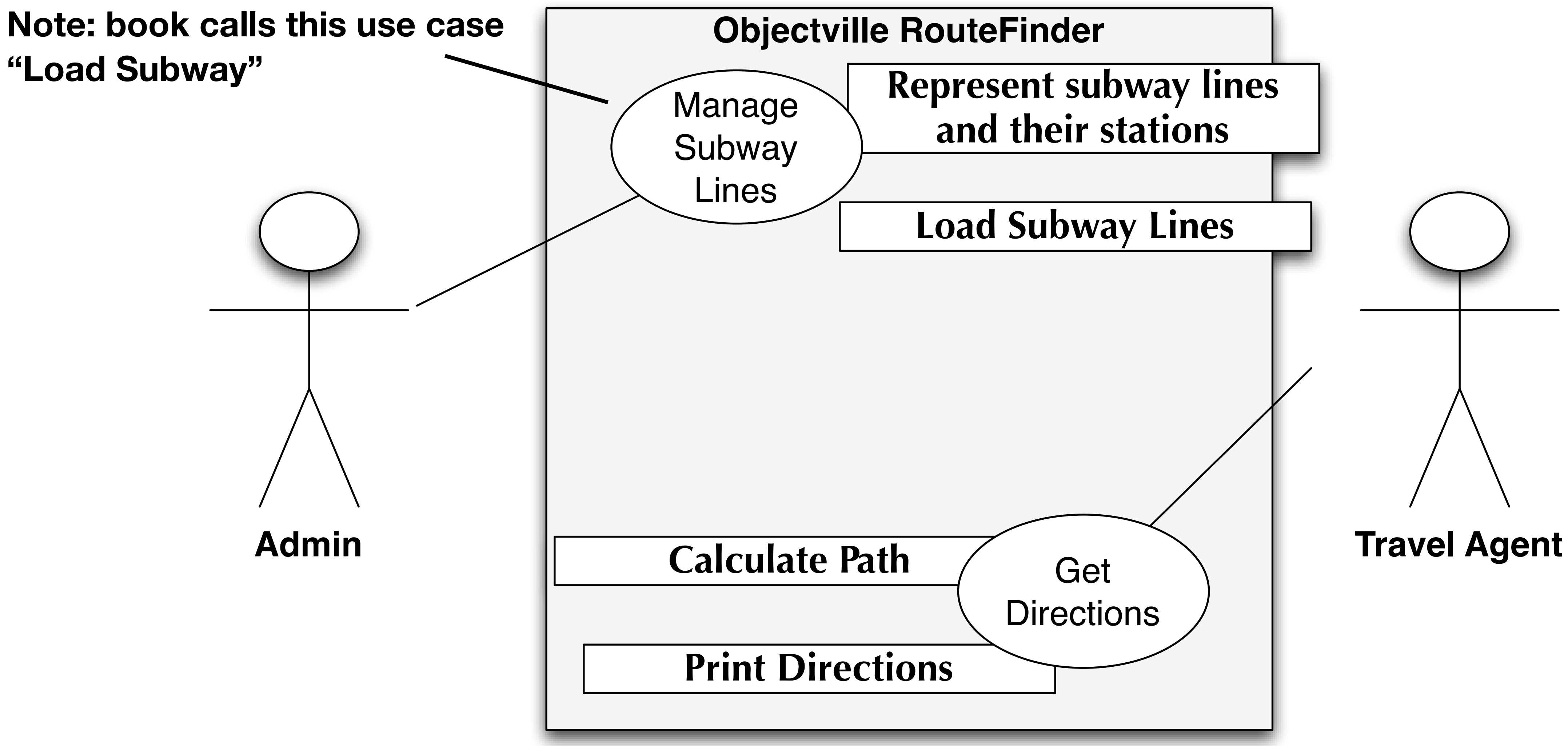
1. Represent subway line and stations along that line.
2. Load subway lines into program, including lines that intersect
3. Calculate path between any two stations in the subway system
4. Print a route between two stations as a set of directions

Feature lists are all about understanding what your software is **supposed to do…**

…even (relatively) simple programs like this

Next Up? Use Case Diagram

Phase Two: Use Case Diagram

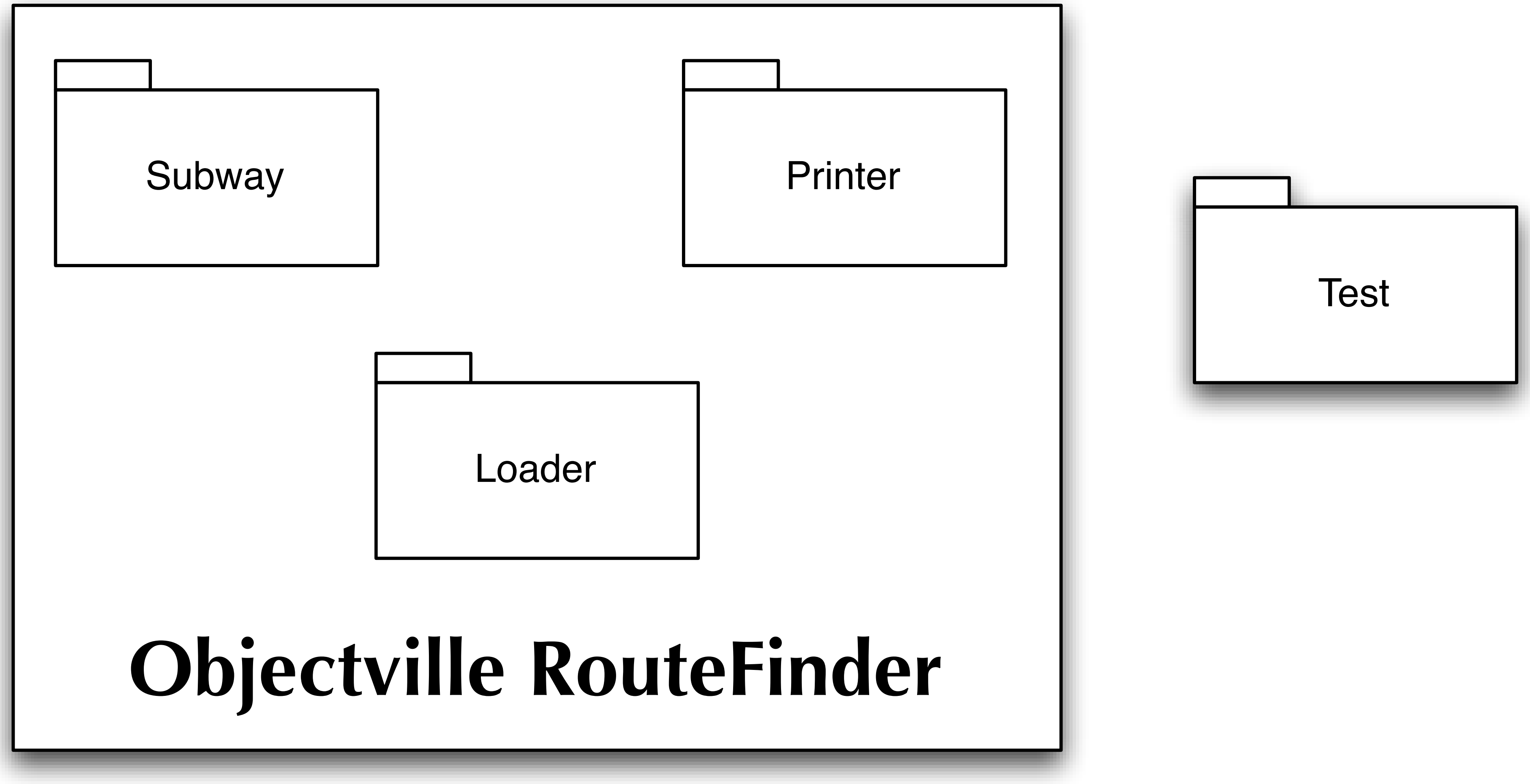


Your use case diagram lets you think about how your software will be **used**, without requiring a lot of detail; features can be assigned to use cases

Features vs. Use Cases

- Use cases reflect usage; Features reflect functionality
  - When we match features to use cases we are indicating that a particular use case depends on a particular feature having been implemented
  - Features and use cases work together, but they are **not** the same thing
- Not all features will match to use cases directly
  - instead indirect relationships may exist
    - The book uses the example of not being able to load subway lines until you have a representation (data structure) for subway lines
    - The use case for loading subway lines will only use the “representation feature” indirectly

Phase Three: The Big Break Up

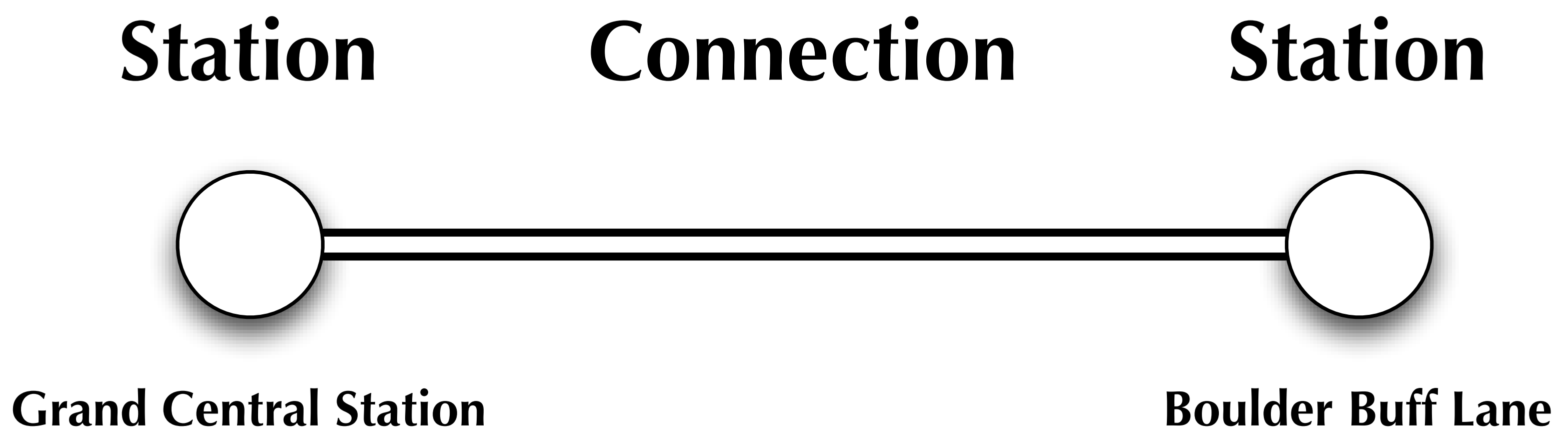


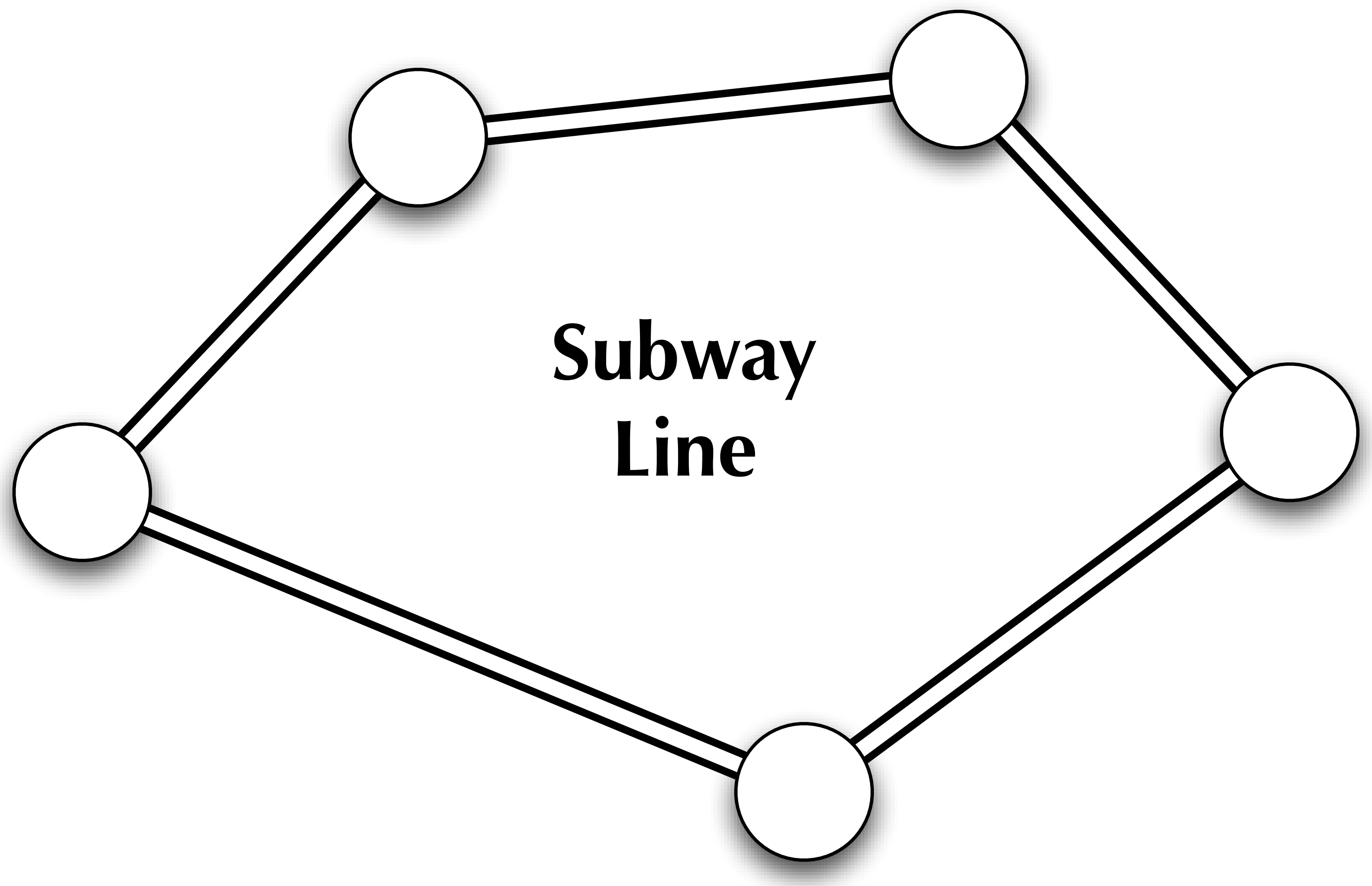
For this small problem, we could have used just a single package, but the book applied the single-responsibility principle and encapsulation to create the modules above

New Step! Understand the Problem

- The next step in the original life cycle is “Requirements”
  - We would focus on generating lists of requirements and filling out the details of our use cases
- But, if you find that you don’t understand the problem well enough to supply the details of a use case, you need to start the requirements process with activities that will help you analyze the problem and gain a deeper understanding
- In the example, we start iteration 1 by focusing on the “Load Subway Lines” use case
  - And then first do some domain analysis to understand the problem domain a bit better and get us the details we need to write the use case

Understanding Our Domain (I)





A subway line is a series of stations, each connected to each other

Understanding Our Domain (II)

```
Ajax Rapids
Algebra Avenue
Boards 'R' Us
Break Neck Pizza
Choc-O-Holic, Inc.
CSS Center
Design Patterns Plaza
DRY Drive
EJB Estates
GoF Gardens
Head First Labs
Head First Lounge
Head First Theater
HTML Heights
Infinite Circle
JavaBeans Boulevard
JavaRanch
JSP Junction
LSP Lane
Mighty Gumball, Inc.
Objectville Diner
Objectville Pizza Store
OCP Orchard
OOA&D Oval
PMP Place
Servlet Springs
SimUDuck Lake
SRP Square
The Tikibean Lounge
UML Walk
Weather-O-Rama, Inc.
Web Design Way
XHTML Expressway

Booch Line
Ajax Rapids
UML Walk
Objectville Pizza Store
Head First Labs
```

The client supplied this input file.

It starts by listing all stations

Then defines subway lines

Use Case Details

Load Subway Network

1. The admin supplies a file of stations and lines.
2. The system reads in the name of a station.
3. The system **validates that the station doesn't already exist**.
4. The system adds the new station to the subway.
5. The system repeats steps 2-4 until all stations are added.
6. The system reads in the name of a line.
7. The system reads in two stations that are connected.
8. The system **validates that the stations exist.**
9. The system creates a new connection between the two stations, **going in both directions**, on the current line.
10. The system repeats steps 7-9 until the line is complete.
11. The system repeats steps 6-10 until all lines are entered.

Use case documents the algorithm for reading input file.

This is a horrible use case because:

a. too low level

b. too brittle

c. all of the above

“Yeah, I think I can think of something better…” *

Load Subway Network

1. The admin requests system to load subway file.
2. The system validates that file exists.
3. The system loads data.
4. The system validates data.
5. The system displays subway.

Use case is still low level, but now less brittle. If the file format changes, we don’t have to update this use case.

Indeed, I think this functionality is better represented as a feature than a use case.

You don’t need use cases for everything.

* Reference to Steve Martin Movie: “Roxanne”

But, back to the original for textual analysis

Load Subway Network

1. The admin supplies a file of stations and lines.
2. The system reads in the name of a station.
3. The system **validates that the station doesn't already exist**.
4. The system adds the new station to the subway.
5. The system repeats steps 2-4 until all stations are added.
6. The system reads in the name of a line.
7. The system reads in two stations that are connected.
8. The system **validates that the stations exist**.
9. The system creates a new connection between the two stations, **going in both directions**, on the current line.
10. The system repeats steps 7-9 until the line is complete.
11. The system repeats steps 6-10 until all lines are entered.

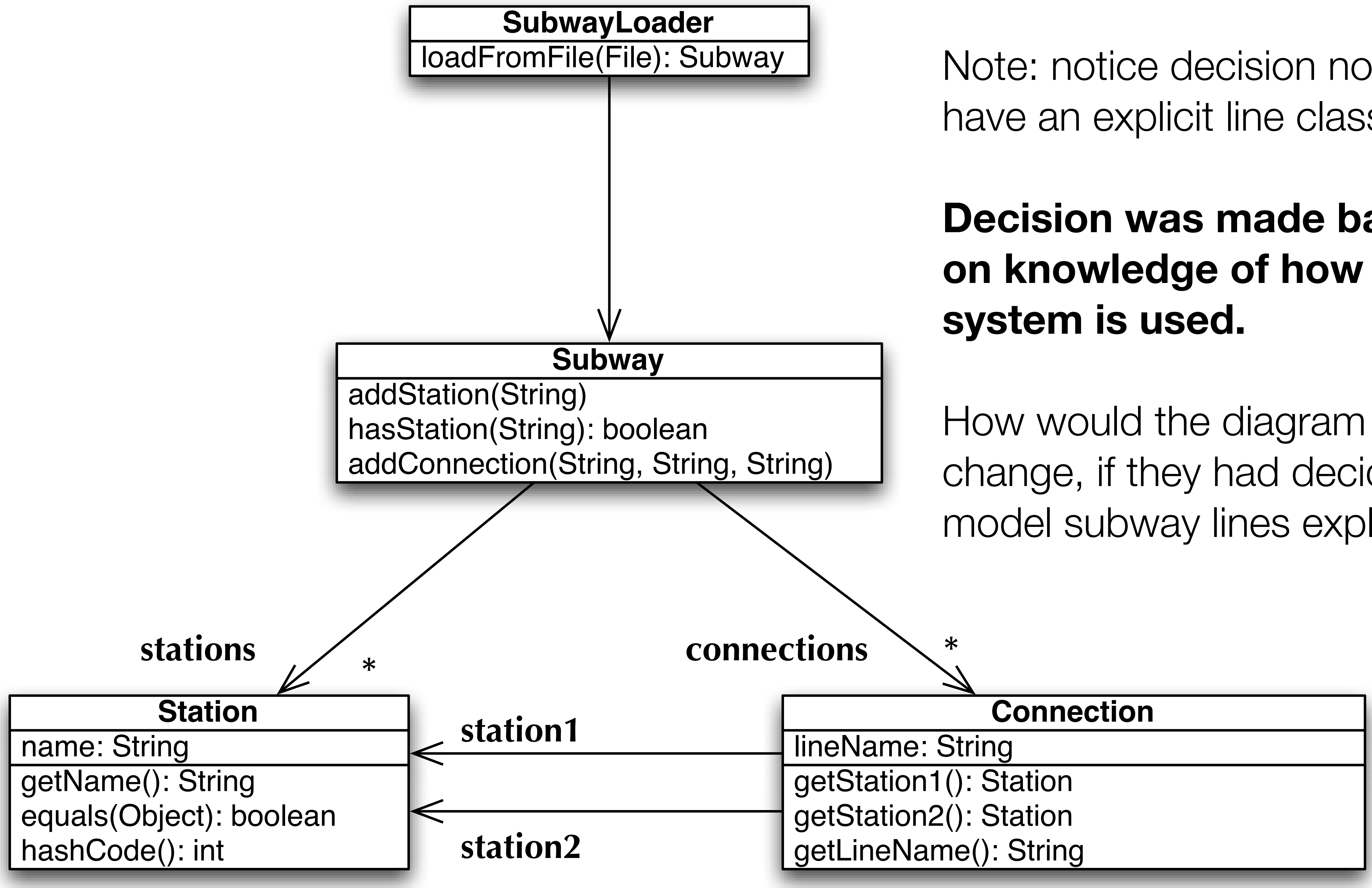
Nouns (Classes):

- ~~admin~~
- ~~system~~
- ~~file~~
- station
- subway
- connection
- line

Verbs (methods)

- supplies a file
- ~~reads in~~
- validates station
- adds a station
- ~~repeats~~
- adds a connection

Class Diagram: Domain Model



Note: notice decision not to have an explicit line class.

Decision was made based on knowledge of how this system is used.

How would the diagram change, if they had decided to model subway lines explicitly?

Have Use Case, Have Class Diagram, Will Code

```python
class Station(object):
    """The Station class represents a single named station on a subway line."""

    def __init__(self, name):
        """Every Station object has a name attribute."""
        self.name = name

    def __eq__(self, obj):
        """Equality of Station objects depends on the lowercase version of
           their names."""
        if not isinstance(obj, Station):
            return False
        return self.name.lower() == obj.get_name().lower()

    def __hash__(self):
        """A Station object's hash code depends on the hash code of
           the lowercase version of its name."""
        return self.name.lower().__hash__()

    def get_name(self):
        """Retrieves the Station object's name."""
        return self.name

```

Python equivalent of Station class shown in text book

Code for Connection

```python
from Station import Station

class Connection(object):
    """The Connection class represents a connection between two subway
       stations along a particular subway line. Note: this class
       is an information holder. It does nothing but store data."""

    def __init__(self, station1, station2, line):
        """Every Connection object has two stations and the name of its line."""
        self.station1 = station1
        self.station2 = station2
        self.line     = line

    def get_station1(self):
        """Retrieves a Connection object's first station."""
        return self.station1

    def get_station2(self):
        """Retrieves a Connection object's second station."""
        return self.station2

    def get_line(self):
        """Retrieves the name of a Connection object's subway line."""
        return self.line

```

equals() method for Connection objects

```
    def __eq__(self, obj):
        """Equality of Connection objects depends on the equality
           of their attributes: station1, station2, line."""
        if not isinstance(obj, Connection):
             return False
        result1 = self.station1 == obj.get_station1()
        result2 = self.station2 == obj.get_station2()
        result3 = self.line == obj.get_line()
        return result1 and result2 and result3
```

Why define an equals() method for Station and Connection?

Because, we don't want the standard equality metric (two variables pointing at the same object). We want two separate objects with the same attributes to be considered equal. Why?

It enables code like this

```python
    def add_station(self, name):
        """If we have never seen the specified station name before, then
           we add it to our collection of station objects."""
        if not self.has_station(name):
            self.stations.append(Station(name))

    def has_station(self, name):
        """Returns True if we have a station with the specified name."""
        station = Station(name)
        return station in self.stations
```

add_station and has_station both take in strings, create Station objects, and then do their jobs. In has_station(), we create a Station object and then check to see if another object that equals it exists in the self.stations collection

What are the trade-offs with this style of coding?

Ease of Expression: whenever I need a new Station object, just create one!

Loss of Efficiency: more station objects == more memory consumed

Code for Subway

```python
from Connection import Connection
from Station    import Station

class Subway(object):
    """The Subway class represents a subway line with stations and
       connections between those stations."""

    def __init__(self):
        """Every Subway object has a collection of stations and a list
           of connections."""
        self.stations = []
        self.connections = []

    def add_station(self, name):
        """If we have never seen the specified station name before, then
           we add it to our collection of station objects."""
        if not self.has_station(name):
            self.stations.append(Station(name))

    def has_station(self, name):
        """Returns True if we have a station with the specified name."""
        station = Station(name)
        return station in self.stations

    def has_connection(self, name1, name2, line):
        """Returns True if we have a connection with the specified atts."""
        connection = Connection(Station(name1), Station(name2), line)
        return connection in self.connections

    def add_connection(self, name1, name2, line):
        """Adds a connection going in both directions to the subway
           as long as specified names reference existing stations."""
        if self.has_station(name1) and self.has_station(name2):
            connection1 = Connection(Station(name1), Station(name2), line)
            connection2 = Connection(Station(name2), Station(name1), line)
            self.connections.append(connection1)
            self.connections.append(connection2)
```

Loose Coupling in RouteFinder

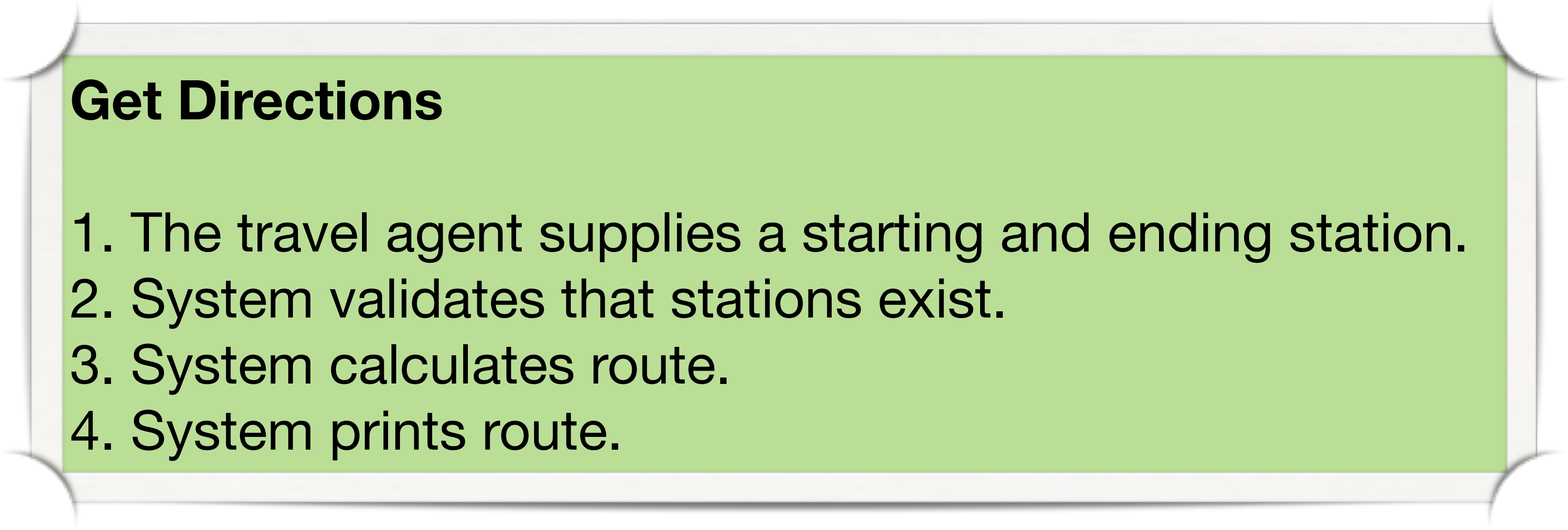
- The book raises an interesting issue about the interface of the Subway class
    - In particular, addStation() and addConnection() accept strings rather Station objects
- Why?
    - To promote lose coupling. If the clients of Subway only ever deal with strings, then they do not depend on classes like Station and Connection
    - Those classes are used internally but not needed externally
- You should only expose clients of your code to the classes that they NEED to interact with
- Classes that the clients don’t interact with can be changed with minimal client code being affected.

Next Steps

- Create Code for Subway Loader
- Create Test Case for Subway Loader
  - Requires adding has_connection() to Subway class
- Demonstration

- We’ve now finished the first use case “Load Subway”
  - Its time to perform a second iteration to tackle “Get Directions”
  - We loop back to the Requirements stage and try to write this use case

Get Directions Use Case



Get Directions

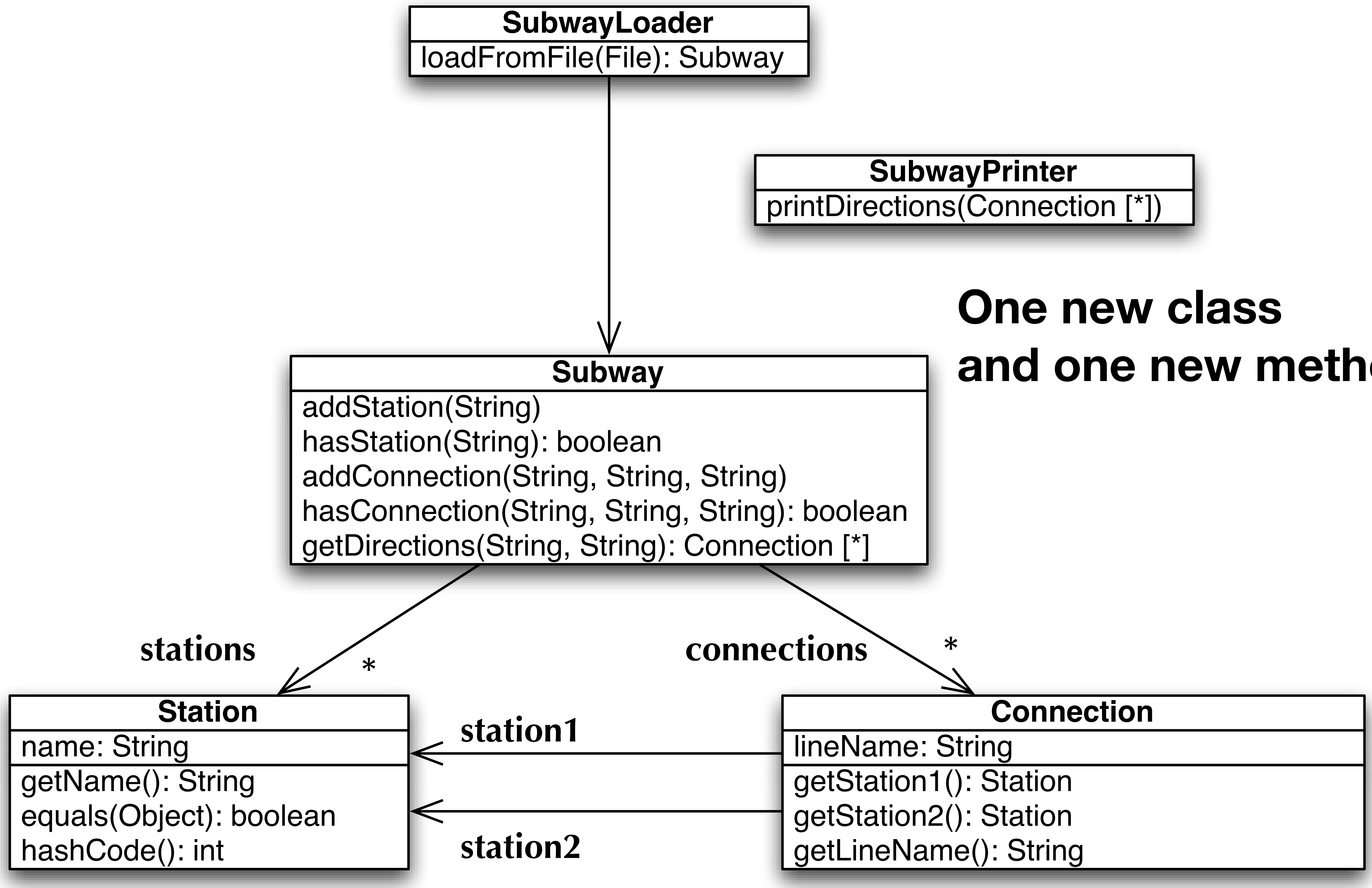
1. The travel agent supplies a starting and ending station.
2. System validates that stations exist.
3. System calculates route.
4. System prints route.

Looks simple enough!

Note: we've switched our focus from code back to customer. We'll use this switch in focus to identify new requirements, do some design, and then go back to focusing on code.

You will shift focus like this multiple times as you progress

Update Class Diagram



One new class and one new method

Changes to Subway

- We need to use Dijkstra’s algorithm to discover the shortest path between any two stations (nodes) on our subway (graph)
  - To do that, we have to update Subway such that
    - It contains a hash table called network that keeps track of what stations are directly reachable from a particular station
      - For example, starting at XHTML Expressway, there are four stations that are directly reachable: Weather-O-Rama, Inc., LSP Lane, Infinite Circle, Choc-O-Holic, Inc.
  - We then implement Dijkstra’s algorithm in the getDirections() method

Layman’s Description of Dijkstra’s Algorithm

- Adapted from Wikipedia: <http://en.wikipedia.org/wiki/Dijkstra's_algorithm>
    - Using a street map, mark streets (trace a street with a marker) in a certain order, until you have a route marked from a starting point to a destination.
    - The order is simple: from all the street intersections of the already marked routes, find the closest unmarked intersection - closest to the starting point (the "greedy" part).
    - Your new route is the entire marked route to the intersection plus the street to the new, unmarked intersection
    - Mark that street to that intersection, draw an arrow with the direction, then repeat
    - Never mark any intersection twice
    - When you get to the destination, follow the arrows backwards. There will be only one path back against the arrows, the shortest one.

Last Step: Implement SubwayPrinter

- Accept route from getDirections()
- print “Start out at <starting point> and get on <first line>”
- Loop until destination is reached
    - If next connection on same line, then print “continue past <station>”
    - Otherwise, print “when you reach <station> switch to <next line>”
- print “Get off at <destination> and enjoy yourself”

- Need to write test case and test results

Demonstration

- Review Dijkstra code in Subway.py
- Review printing code in SubwayPrinter.py
- Try out test code in SubwayPrinter.py

We're Done!

- We'll almost
  - We could always add additional ways to print out the route
  - We could look for additional ways to clean up the current design
  - We could look for additional functionality and continue to iterate
- The important thing is that we've seen how to bring everything we've discussed this semester together to solve a software design problem from start to finish
  - Take a look at the two implementations of this system
    - Find similarities and differences
    - Be sure to understand the code

Coming Up Next

- Midterm and Midterm Discussion (Week 8)
- Framework Project Demonstrations (Week 9)
  - Start of Semester Project
- Design Patterns (Week 10)