

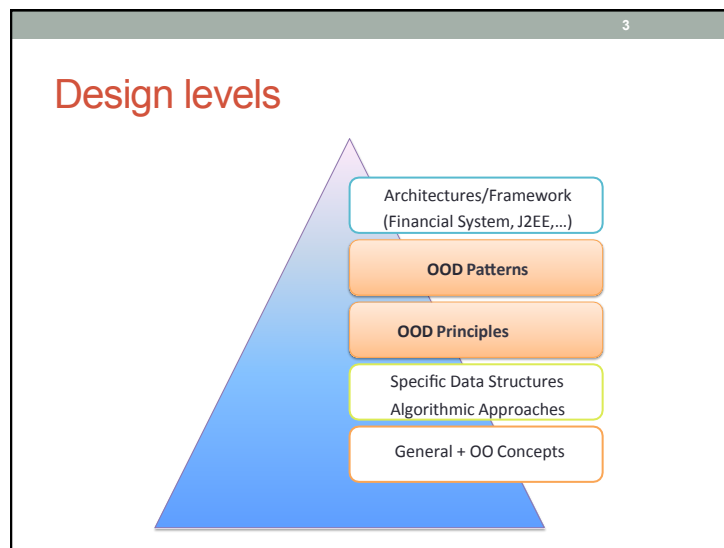
IT4490 - SOFTWARE DESIGN AND CONSTRUCTION

10. DESIGN PRINCIPLES

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn

Content

- ➔ 1. How do you design?
2. Coupling and Cohesion
3. S.O.L.I.D. principles
4. Case study: Reminder program



Key design concepts

General	OO Specific
<ul style="list-style-type: none"> • Cohesion • Coupling • Information hiding <ul style="list-style-type: none"> • Encapsulation • Creation • Binding time 	<ul style="list-style-type: none"> • Behaviors follow data • Class vs. Interface Inheritance <ul style="list-style-type: none"> • Class = implementation • Interface = type • Inheritance / composition / delegation

What's Purpose Of Design?

- What's a design?
 - Express a idea to resolve a problem
 - Use for communications in the team members
- What's a good design?
 - Easy for Developing, Reading & Understanding
 - Easy for Communication
 - Easy for Extending (add new features)
 - Easy for Maintenance

How do you design?

- What principles guide you when you create a design?
- What considerations are important?
- When have you done enough design and can begin implementation?

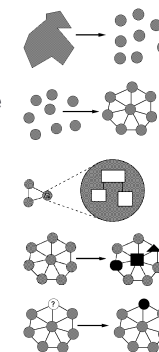
•Take a piece of paper and write down two principles that guide you - considerations that are important or indicators that you have a good design.

Modules

- A **module** is a relatively general term for a class or a type or any kind of design unit in software
- A **modular design** focuses on what modules are defined, what their specifications are, how they relate to each other, but not usually on the implementation of the modules themselves
- Overall, you've been given the modular design so far – and now you have to learn more about how to do the design

Ideals of modular software

- Decomposable – can be broken down into modules to reduce complexity and allow teamwork
- Composable – “Having divided to conquer, we must reunite to rule [M. Jackson].”
- Understandable – one module can be examined, reasoned about, developed, etc. in isolation
- Continuity – a small change in the requirements should affect a small number of modules
- Isolation – an error in one module should be as contained as possible



Content

1. How do you design?
- ➡ 2. Coupling and Cohesion
3. S.O.L.I.D. principles
4. Case study: Reminder program

10

Two general design issues

- **Cohesion** – why are sub-modules (like methods) placed in the same module? Usually to collectively form an ADT
- **Coupling** – what is the dependence between modules? Reducing the dependences (which come in many forms) is desirable

11

Cohesion

- The most common reason to put elements – data and behavior – together is to form an ADT
 - There are, at least historically, other reasons to place elements together – for example, for performance reasons it was sometimes good to place together all code to be run upon initialization of a program
- The common design objective of separation of concerns suggests a module should address a single set of concerns

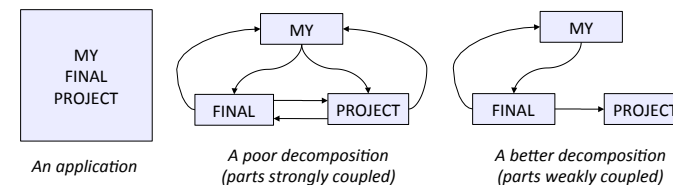
Example considerations

 - Should Item/DiscountItem know about added discount for purchasing 20+ items? Should ShoppingCart know about bulk pricing?
 - Should BinarySearch know the type of the objects it is sorting?
- This kind of questions help make more effective cohesion decisions

Coupling

Roughly, the more coupled k modules are, the more one needs to think of them as a single, larger module

- How are modules dependent on one another?
 - Statically (in the code)? Dynamically (at run-time)? And more
 - Ideally, split design into parts that don't interact much

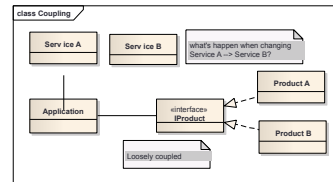


- An artist's rendition – to really assess coupling one needs to know what the arrows are, etc.

Cohesion and Coupling

□ Coupling

Coupling or **Dependency** is the degree to which each program module relies on each one of the other modules.



□ Cohesion

Cohesion refers to the degree to which the elements of a module belong together. **Cohesion** is a measure of how strongly-related or focused the responsibilities of a single module are.

What's Purpose Of Design?

- What's a design?
 - Express a idea to resolve a problem.
 - Use for communications in the team members.
 - What's a good design?
 - "Cohesion and Coupling deal with the quality of an OO design"
 - Easy for Developing, reading & understanding.
 - Easy for Communication.
 - Easy for Extending (add new features)
 - Easy for Maintenance.
- ➔ "Loose coupling and high cohesion" idea!!!

15

Different kinds of dependences

- Aggregation – "is part of" is a field that is a sub-part
 - Ex: A car has an engine
- Composition – "is entirely made of" has the parts live and die with the whole
 - Ex: A book has pages (but perhaps the book cannot exist without the pages, and the pages cannot exist without the book)
- Subtyping – "is-a" is for substitutability
- Invokes – "executes" is for having a computation performed
- In other words, there are lots of different kinds of arrows (dependences) and clarifying them is crucial

16

Law of Demeter

Karl Lieberherr [@](#) and colleagues

- Law of Demeter: An object should know as little as possible about the internal structure of other objects with which it interacts – a question of coupling
- Or... "only talk to your immediate friends"
- Closely related to representation exposure and (im)mutability
- Bad example – too-tight chain of coupling between classes


```

general.getColonel().getMajor(m).getCaptain(cap)
    .getSergeant(ser).getPrivate(name).digFoxHole();
            
```
- Better example


```

general.superviseFoxHole(m, cap, ser, name);
            
```

17

An object should only send messages to ... (More Demeter)

- itself (`this`)
- its instance variables
- its method's parameters
- any object it creates
- any object returned by a call to one of `this`'s methods
- any objects in a collection of the above
- notably absent: objects returned by messages sent to other objects

Guidelines: not strict rules!
But thinking about them
will generally help you
produce better designs

18

Coupling is the path to the dark side

- Coupling leads to complexity
- Complexity leads to confusion
- Confusion leads to suffering
- Once you start down the dark path,
forever will it dominate your destiny,
consume you it will



19

God classes

- **God class**: a class that hoards too much of the data or functionality of a system
 - Poor cohesion – little thought about why all of the elements are placed together
 - Only reduces coupling by collapsing multiple modules into one (and thus reducing the dependences between the modules to dependences within a module)
- A god class is an example of an **anti-pattern** – it is a known bad way of doing things

Content

1. How do you design?
2. Coupling and Cohesion
- ➔ 3. S.O.L.I.D. principles
4. Case study: Reminder program

Principles of OO Class Design

- SRP: The Single Responsibility Principle
- OCP: The Open Closed Principle
- LSP: The Liskov Substitution Principle
- ISP: The Interface Segregation Principle
- DIP: The Dependency Inversion Principle

Principles of OO Class Design

SRP: The Single Responsibility Principle

“There should never be more than one reason for a class to change”

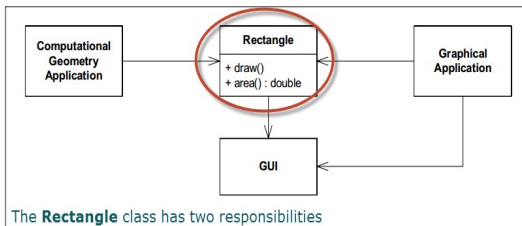
Or

“A class should have one, and only one type of responsibility.”

Principles of OO Class Design

SRP: The Single Responsibility Principle (cont)

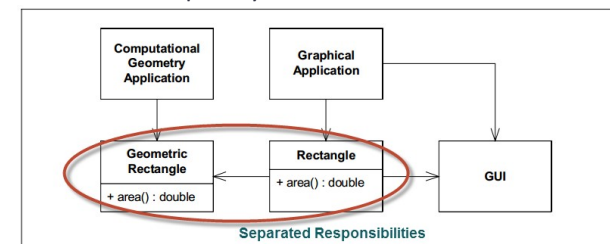
- Two applications are using this Rectangle class:
 - Computational Geometry Application uses this class to calculate the Area
 - Graphical Application uses this class to draw a Rectangle in the UI



Principles of OO Class Design

SRP: The Single Responsibility Principle (cont)

- A better design is to separate the two responsibilities into two completely different classes



- Why is it important to separate these two responsibilities into separate classes?

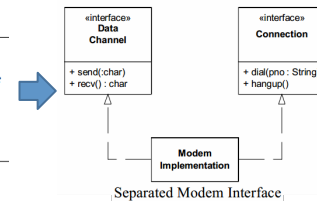
Principles of OO Class Design

SRP: The Single Responsibility Principle (cont)

- What is a Responsibility?
 - A reason for change
 - “Modem” sample
 - dial & hangup functions for managing connection
 - send & rcv functions for data communication
- ➔ Should separate into 2 repositories!

Modem.java -- SRP Violation

```
interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char rcv();
}
```



Principles of OO Class Design

OCP: The Open Closed Principle

“Software entities(classes, modules, functions, etc.) should be open for extension, but closed for modification.”

Bertrand Meyer, 1988

Or

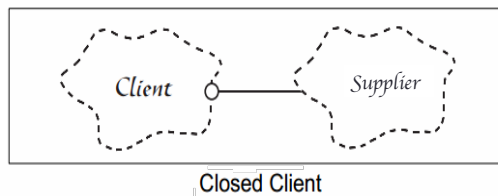
“You should be able to extend a classes behavior, without modifying code”

- “Open for Extension”
 - The behavior of the module/class can be extended
 - The module behave in new and different ways as the requirements changes, or to meet the needs of new applications
- “Closed for Modification”
 - The source code of such a module is inviolate
 - No one is allowed to make source code changes to it

Principles of OO Class Design

OCP: The Open Closed Principle (cont)

- Client & Supplier classes are concrete
 - If the Supplier implementation/class is changed, Client also needs change.
- ➔ How to resolve this problem?

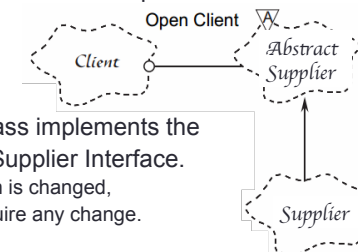


Principles of OO Class Design

OCP: The Open Closed Principle (cont)

- Change to support Open-Closed Principle.

➔ Abstraction is the key.



- The Concrete Supplier class implements the Abstract Supplier class / Supplier Interface.
 - The Supplier implementation is changed,
 - the Client is likely not to require any change.

➔ The Abstract Supplier class here is closed for modification and the Concrete class implementations here are Open for extension.

Principles of OO Class Design

LSP: The Liskov Substitution Principle

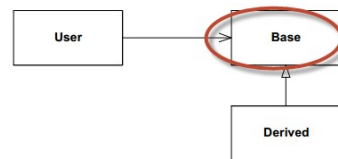
- “Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.”

• Or

“Subclasses should be substitutable for their base classes.”

User, Based, Derived, example.
void User(Base& b);

Derived d;
User(d);

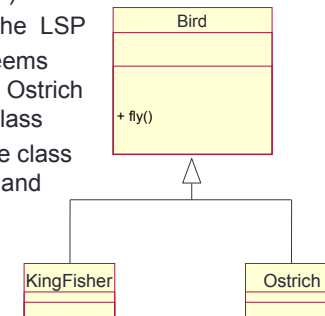


22

Principles of OO Class Design

LSP: The Liskov Substitution Principle (cont)

- Ostrich is a Bird (definitely!!!)
- Can it fly? No! => Violates the LSP
- Even if in real world this seems natural, in the class design, Ostrich should not inherit the Bird class
- There should be a separate class for birds that can't really fly and Ostrich inherits that.



Principles of OO Class Design

LSP: The Liskov Substitution Principle (cont)

- “Inheritance” ~ “is a” relationship
 - But, easy to get carried away and end up in wrong design with bad inheritance.
 - The LSP is a way of ensuring that inheritance is used correctly
- Why The LSP is so important? If not LSP,
 - Class hierarchy would be a **mess** and if subclass instance was passed as parameter to methods method, strange behavior might occur.
 - Unit tests for the Base classes would never succeed for the subclass.
 - LSP is just an extension of Open-Close Principle!!!

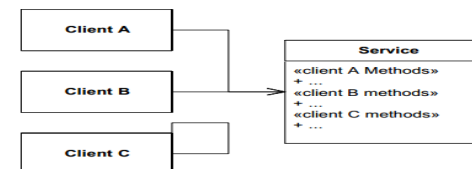
Principles of OO Class Design

ISP: The Interface Segregation Principle

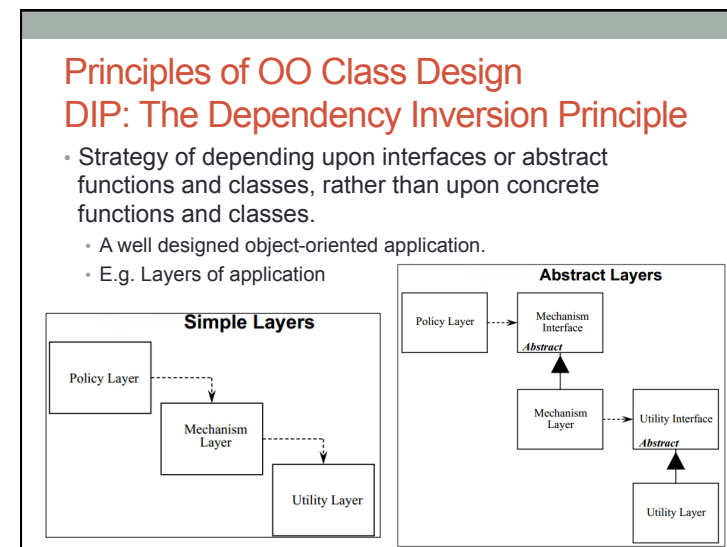
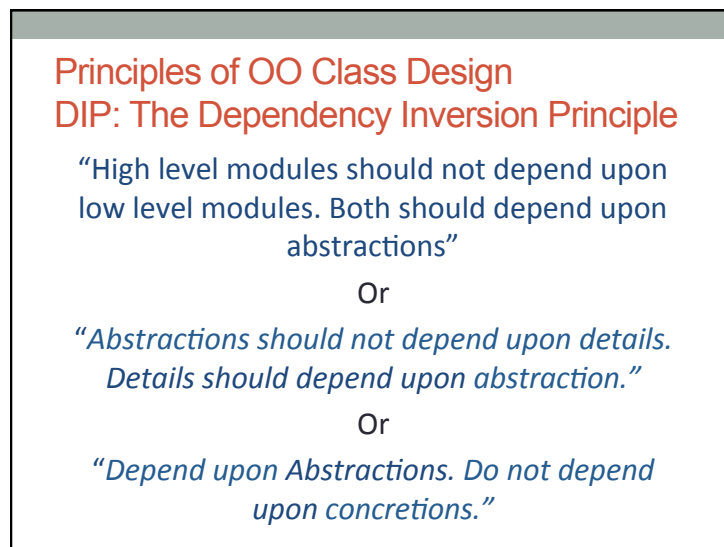
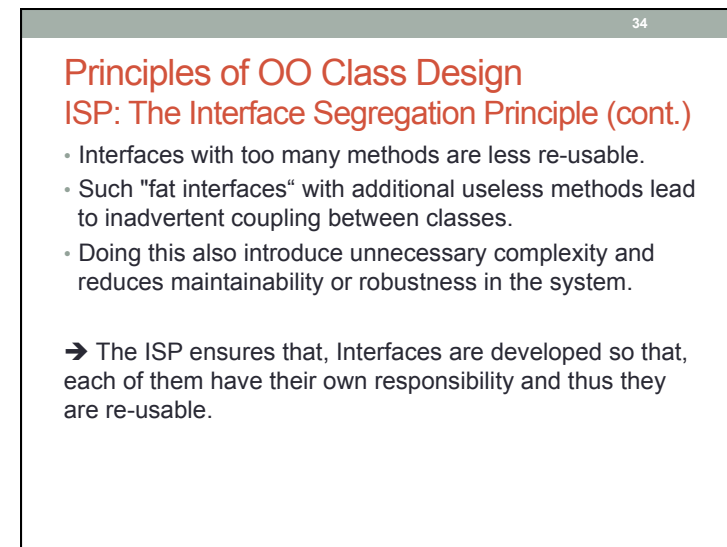
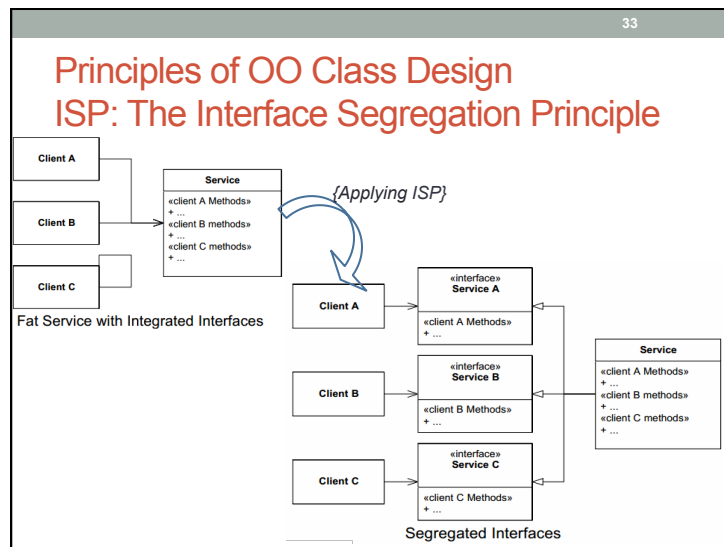
- “Client should not be forced to depend upon interface that they do not use.”

• Or

- “Many client specific interfaces are better than one general purpose interface.”



Fat Service with Integrated Interfaces



Content

1. How do you design?
2. Coupling and Cohesion
3. S.O.L.I.D. principles
- ➡ 4. Case study: Reminder program

38

Design exercise

- Write a typing break reminder program
 - Offer the hard-working user occasional reminders of the health issues, and encourage the user to take a break from typing
- Naive design
 - Make a method to display messages and offer exercises
 - Make a loop to call that method from time to time
(*Let's ignore multi-threaded solutions for this discussion*)

39

TimeToStretch suggests exercises

```
public class TimeToStretch {
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }
    public void suggestExercise() {
        ...
    }
}
```

40

Timer calls run() periodically

```
public class Timer {
    private TimeToStretch tts = new TimeToStretch();
    public void start() {
        while (true) {
            ...
            if (enoughTimeHasPassed) {
                tts.run();
            }
            ...
        }
    }
}
```

41

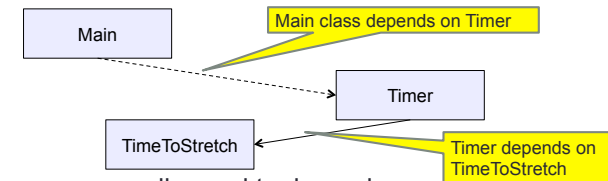
Main class puts it together

```
class Main {
    public static void main(String[] args) {
        Timer t = new Timer();
        t.start();
    }
}
```

42

Module dependency diagram

- An arrow in a module dependency diagram indicates “depends on” or “knows about” – simplistically, “any name mentioned in the source code”



- Does **Timer** really need to depend on **TimeToStretch**?
- Is **Timer** re-usable in a new context?

43

Decoupling

- **Timer** needs to call the **run** method
 - **Timer** doesn't need to know what the **run** method does
- Weaken the dependency of **Timer** on **TimeToStretch**
- Introduce a weaker specification, in the form of an interface or abstract class

```
public abstract class TimerTask {
    public abstract void run();
}
```

- **Timer** only needs to know that something (e.g., **TimeToStretch**) meets the **TimerTask** specification

44

TimeToStretch (version 2)

```
public class TimeToStretch extends TimerTask {
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }

    public void suggestExercise() {
        ...
    }
}
```

45

Timer v2

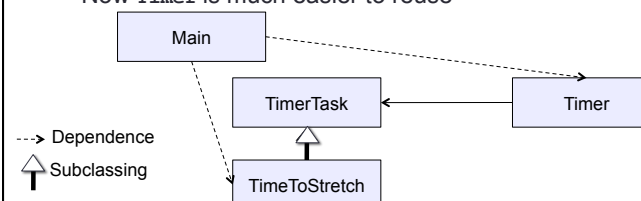
```
public class Timer {
    private TimerTask task;
    public Timer(TimerTask task) { this.task = task; }
    public void setTask(TimerTask task){this.task = task;}
    public void start() {
        while (true) {
            ...
            if (enoughTime)
                task.run();
        }
    }
}
```

- Main creates the **TimeToStretch** object and passes it to **Timer**
Timer t = new Timer(new TimeToStretch());
t.start();
t.setTask(new TimeToSave());
t.start();

46

Module dependency diagram

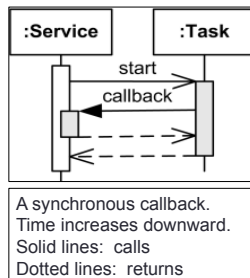
- **Main** still depends on **Timer** (is this necessary?)
- **Main** depends on the constructor for **TimeToStretch**
- **Timer** depends on **TimerTask**, not **TimeToStretch**
 - Unaffected by implementation details of **TimeToStretch**
 - Now **Timer** is much easier to reuse



47

callbacks

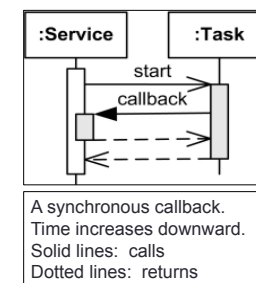
- **TimeToStretch** creates a **Timer**, and passes in a reference to itself so the **Timer** can call it back
- This is a **callback** – a method call from a module to a client that notifies about some condition
- Use a callback to invert a dependency
 - Inverted dependency: **TimeToStretch** depends on **Timer** (not vice versa)
 - Side benefit: **Main** does not depend on **Timer**



48

Callbacks

- Synchronous callbacks
 - Ex: **HashMap** calls its client's **hashCode**, **equals**
 - Useful when the callback result is needed immediately by the module
- Asynchronous callbacks
 - Examples: GUI listeners
 - Register to indicate interest and where to call back
 - Useful when the callback should be performed later, when some interesting event occurs



49

TimeToStretch v3

```
public class TimeToStretch extends TimerTask {
    private Timer timer;
    public TimeToStretch() {
        timer = new Timer(this);
    }
    public void start() {
        timer.start();
    }
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }
    ...
}
```

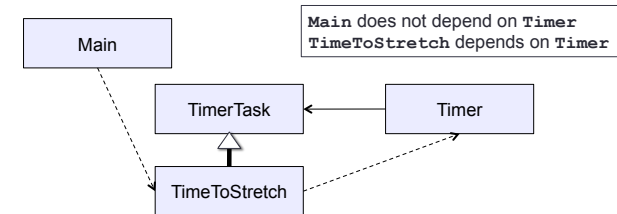
Register interest with the timer

Callback entry point

50

Main v3

- `TimeToStretch tts = new TimeToStretch();`
`tts.start();`
- Use a callback to invert a dependency
- This diagram shows the inversion of the dependency between `Timer` and `TimeToStretch` (compared to v1)



51

How do we design classes?

- One common approach to class identification is to consider the specifications
- In particular, it is often the case that
 - *nouns* are potential classes, objects, fields
 - *verbs* are potential methods or responsibilities of a class

52

Design exercise

- Suppose we are writing a birthday-reminder application that tracks a set of people and their birthdays, providing reminders of whose birthdays are on a given day
- What classes are we likely to want to have? Why?

Class shout-out about classes

53

More detail for those classes

- What fields do they have?
- What constructors do they have?
- What methods do they provide?
- What invariants should we guarantee?

In small groups, ~5 minutes