Logistics: (research) projects

- Group size 2-4 (exceptions possible).
- **Elevator pitches** next week:
  - In class: **Thursday 02/04**.
- Great opportunity to recruit!

How to pick a project?
- Any research/engineering challenges in your ongoing projects?
- Interested in Perceptual Robotics or Software Engineering?
  - Talk to us about potential research projects.
- What novel technique would make you more productive?
- What novel application would make your everyday life easier?
Logistics: example projects

Research
● Do existing code coverage tools compute significantly different results?

Engineering challenges
● Elevation based navigation/optimization (min and max)
  ○ You’ll learn and implement routing algorithms (e.g., Dijkstra, A*).
  ○ You’ll solve a multivariate optimization problem.
● Activity monitor (car, bike, foot, ...)
  ○ Design of UI and API use (GPS etc.) is challenging.
  ○ Heuristics vs. user interaction (what activity is being monitored?).
Recap

● Modeling and abstraction
● Software architecture vs. software design
● Examples for software architecture
  ○ Pipe and filter
  ○ Model-View-Controller
● UML crash course
  ○ Class diagrams (overview and basic notation)
Recap

Development process

Specification

Architecture

Design

Source code

Level of abstraction

Today
Today

- More on UML class diagrams
  - Aggregation vs. composition
  - Inheritance or composition

- OO design principles
  - Open/closed principle
  - Liskov substitution principle

- A first design problem (if time permits)
Classes, abstract classes, and interfaces

- MyClass
- MyAbstractClass
  - {abstract}
- `<interface>`
  - MyInterface
Classes, abstract classes, and interfaces

MyClass

```
public class MyClass {
    public void op() {
        ...
    }
    public int op2() {
        ...
    }
}
```

MyAbstractClass

```
public abstract class MyAbstractClass {
    public abstract void op();
    public int op2()
        {
            ...
        }
}
```

MyInterface

```
public interface MyInterface {
    public void op();
    public int op2();
}
```
How do default methods (Java 8) fit into this spectrum?
Classes, abstract classes, and interfaces

```
SequentialList {abstract}
```

```
List
```

```
Deque
```

```
Collection
```

```
LinkedList
```

Classes, abstract classes, and interfaces

SequentialList {abstract} extends LinkedList

List <<interface>>

Deque <<interface>>

Collection <<interface>>

LinkedList implements List

SequentialList implements List

LinkedList implements Deque
Aggregation vs. composition

- Customer
  - Bank

- LinkedList
  - Stack
Aggregation vs. composition

```java
public class Bank {
    ...
    public void newCustomer(Customer c) {
        ...
    }
    ...
}
...
Bank b = new Bank();
Customer c = new Customer();
b.newCustomer(c);
...
```

```java
public class Stack<E> {
    private List<E> l = new LinkedList<>();
    public void push(E elem) {
        ...
    }
    ...
}
...
Stack<Integer> s = new Stack<>();
s.push(...);
...
```
Aggregation vs. composition

```
public class Bank {
    ...

    public void newCustomer(Customer c) {
        ...
    }
    ...
}

... Bank b = new Bank();
Customer c = new Customer();
b.newCustomer(c);
...
Aggregation vs. composition

```java
public class Bank {
    ...

    public void newCustomer(Customer c) {
        ...
    }
    ...
}

... Bank b = new Bank();
Customer c = new Customer();
b.newCustomer(c);
...
```

```java
public class Stack<E> {
    private List<E> l = new LinkedList<>();

    public void push(E elem) {
        ...
    }
    ...
}

... Stack<Integer> s = new Stack<>();
s.push(...);
...
Aggregation vs. composition

Customer

Bank

LinkedList

Stack

Consider object ownership and lifetime!
Design choice: inheritance or composition?

- LinkedList
- Stack

```java
public class Stack<E>{
    private List<E> l = new LinkedList<>();
    ...
}
```

- LinkedList
- Stack

```java
public class Stack<E> extends List<E> {
    ...
}
```
Design choice: inheritance or composition?

Which is the right choice?
Think about **is-a** vs. **has-a** relationship first.

However, both designs might be valid -- what are pros and cons?
Design choice: inheritance or composition?

**Pros**
- No delegation methods required.
- Reuse of common state and behavior.

**Cons**
- Exposure of all inherited methods (a client might rely on this particular superclass -> can’t change it later).
- Changes in superclass are likely to break subclasses.

**Pros**
- Highly flexible and configurable.
- No additional subclasses required for different has-a relationships.

**Cons**
- All interface methods need to be implemented -> delegation methods required, even for code reuse.

Composition over inheritance allows more flexibility.
Design principles: open/closed principle

**Software entities** (classes, components, etc.) should be:
- **open** for extensions
- **closed** for modifications

```java
public static void draw(Figure f) {
    if (f instanceof Square) {
        drawSquare((Square) f)
    } else if (f instanceof Circle) {
        drawCircle((Circle) f);
    } else {
        ...
    }
}
```

Good or bad?
Design principles: open/closed principle

**Software entities** (classes, components, etc.) should be:

- **open** for extensions
- **closed** for modifications

```java
public static void draw(Figure f) {
    if (f instanceof Square) {
        drawSquare((Square) f)
    } else if (f instanceof Circle) {
        drawCircle((Circle) f);
    } else {
        ...
    }
}
```

Violates the open/closed principle!
Design principles: Liskov substitution principle

Subtype requirement

Let object $x$ be of type $T_1$ and object $y$ be of type $T_2$. Further, let $T_2$ be a subtype of $T_1$ ($T_2 <: T_1$). Any provable property about objects of type $T_1$ should be true for objects of type $T_2$.

Is the subtype requirement fulfilled?
Design principles: Liskov substitution principle

Subtype requirement
Let object \( x \) be of type \( T_1 \) and object \( y \) be of type \( T_2 \). Further, let \( T_2 \) be a subtype of \( T_1 \) (\( T_2 <: T_1 \)). Any provable property about objects of type \( T_1 \) should be true for objects of type \( T_2 \).

```
Rectangle
+ width : int
  + setWidth(w: int)
+ height : int
  + setHeight(h: int)
+ getArea(): int
Rectangle r = new Rectangle(1,2);
r.setWidth(2);
r.setHeight(10);
int A = r.getArea();
assertEquals(20, A);
```