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)
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

<table>
<thead>
<tr>
<th>MyClass</th>
<th>MyAbstractClass</th>
<th>MyInterface</th>
</tr>
</thead>
</table>
| public class MyClass {
  public void op() {
    ...
  }
  ...
  public int op2() {
    ...
  }
} | public abstract class MyAbstractClass {
  public abstract void op();
  public int op2();
} | public interface MyInterface {
  public void op();
  public int op2();
} |

How do default methods (Java 8) fit into this spectrum?

Classes, abstract classes, and interfaces

<table>
<thead>
<tr>
<th>LinkedList</th>
<th>List</th>
<th>Deque</th>
</tr>
</thead>
<tbody>
<tr>
<td>SequentialList {abstract}</td>
<td>Collection</td>
<td></td>
</tr>
<tr>
<td>LinkedList</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How do default methods (Java 8) fit into this spectrum?
public class Bank {
  ...
  public void newCustomer(Customer c) {
    ...
  }
  ...
}

public class Stack<E> {
  private List<E> l = new LinkedList<>();
  public void push(E elem) {
    ...
  }
  ...
}

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

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

public class Stack<E> extends List<E> {
  ...
}

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?

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.

Composition over inheritance allows more flexibility.

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.
**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?**

- **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 \( T_2 \).

```java
Rectangle r = new Rectangle(1,2);
r.setWidth(2);
r.setHeight(10);
int A = r.getArea();
assertEquals(20, A);
```

**Is the subtype requirement fulfilled?**