Abstract Class & Lambda function

Rudolf Szadkowski

github.com/rudosz/cpp_lecture131

Robot navigating through maze.

1. Start-up the robot.
2. Navigation algorithm provides next target.
3. Robot moves to the target.
4. Repeat until the target is reached.

Straightforward solution:

- for specific robot and algorithm.

→ what it means for scalability?

```
void runSession(Coords end, int maxIterations){
  initializeRobot(current);
  for (size_t i = 0; i < maxIterations; i++){
    Coords nextCoord = planNextCoordinate(
  current, end);
    // Physically drive the robot to the next
  position
    current = driveRobot(nextCoord);
    // Check if the robot has reached the
  destination
    if (current == end) break;
  }
}
```

- Instead of defining the code, we **inject** the **service** code.
- The **client** thus becomes **dependend** on the service.

```
class CompositeNavigationSession {
  public:
    CompositeNavigationSession(
  HexapodRobot& robot, DummyNavigator&
  navigator) : robot(robot), navigator(
  navigator) {};

    void runSession(Coords end, int
  maxIterations);

    private:
      HexapodRobot robot;
      DummyNavigator navigator;

};
```

1. **Client**: target code for injection.
2. **Service**: represents injected code.
3. **Contract**: defines interface between client and service.

CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Polymorphism

```
class A {
  virtual int foo();
};

class B : A {
  int foo() override;
};
```

Lambda Function

```
[](int x, int y) -> int {
  return x + y;
}
```

1. Accepting the subclasses of baseclass.
2. What kind of hiearchy whe should follow? There are many robots.
3. We do not need the implements yet!

```
class Robot {
  virtual Coords drive(Coords){};
};

class HexapodRobot : Robot {
  Coords drive(Coords) override;
};
```

- **Pure virtual function** `virtual int foo(int) = 0;`
- **Abstract class** contains at least one pure virtual function.
- Abstract class **cannot be instantiated.**

```
class IRobot {
  public:
    virtual ~IRobot() {};
    virtual void initialize(Coords start) = 0;
    virtual void shutDown() = 0;
    virtual Coords drive(Coords) = 0;
    virtual Map getMap() const = 0;
};
```

```
auto foo = [capture](params)->returnType{body};
```

- **capture**: puts variables into the lambda.
- **params**: arguments of the lambda.
- **body**: code to be executed.

- **Expression**: above
- **Closure**: constructed by the compiler.
- **Closure type**: type of closure

"Interface" for lambda function

```
using LambdaT = std::function<OutputT(InputT)>;
void clientFunction(LambdaT service);
//...
LambdaT lambda = [](InputT in)->OutputT{return doSomething(in);};
clientFunction(lambda);
```

- Function (client) accepting lambda (service).