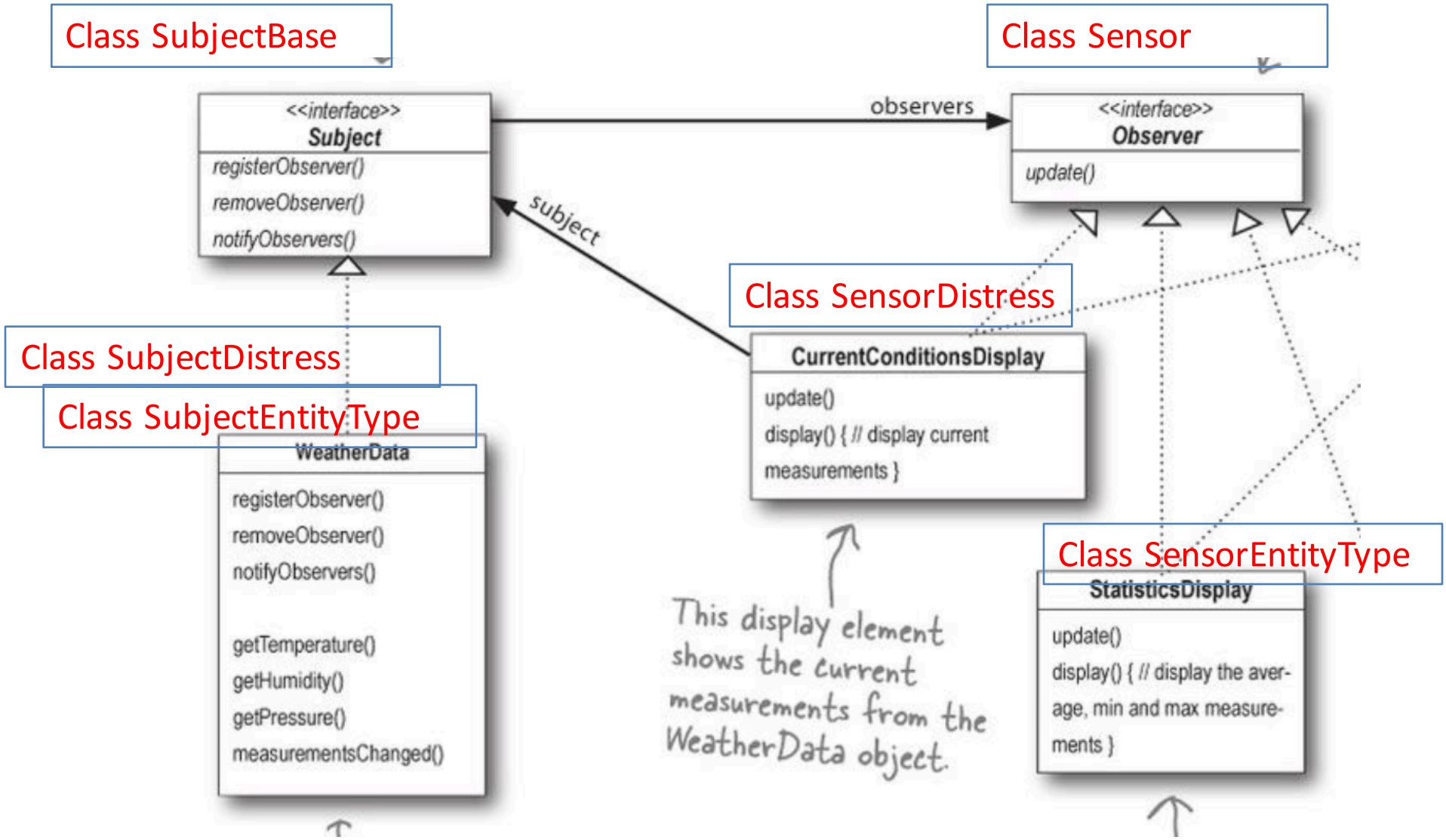
ANNOUNCEMENTS

- Priority 1 and 2 DUE tomorrow.
  - Interface only
  - Functionality in the arena (but not functional)
  - Use your quiz answer to explain and justify your design choices.
- No need to get tests to compile for this.
- Make sure what you want to submit is in devel branch.
- Tag with v.2.interfaces
- Looking for progress!
- Code in devel can be compiled (but not executable).

Sensor Design Based on Observer Patter



EVENTS: convenient data structure in which to pass and store data of various configurations

```
    virtual void Update(const EventBaseClass const * event) {}
    virtual EventBaseClass * get_reading(void) { return event_; }
    bool IsActivated(void) { return activated_; }

private:
  EventBaseClass * event_;  // holds information related to activation
  const ArenaEntity * entity_; // what the sensor is part of
  double range_;
  double fov_angle_; // field of view angle = 360 if omnidirectional
  bool activated_;
```

Sensor Class

Sensor_base.h

Notice Update() – This is the previously named Accept.
Notic get_reading – returns an event.

Sensor_distress.cc

```
class SensorDistress : public SensorBase {
 public:
   SensorDistress(ArenaEntity* entity);

  /**
   * @brief Get the current activation reading from the sensor.
   */
  EventDistress * get_reading(void);
  void Update(const EventDistress const * event);
```

Subject Base Class

Subject_base.h

```
class SubjectBase {
 public:
  SubjectBase(void) : max_range(0) {}

  virtual void RegisterObserver(SensorBase * sensor);
  virtual void StateChange(const EventBaseClass * const event);
  virtual void Notify(const EventBaseClass * const event);

 private:
   std::vector<class SensorBase *> observers_;
   int max_range;
};
```

Register: Sensors register with Arena.

StateChange: Arena sends information related to subject.

Notify: When appropriate, all observers (i.e. relevant sensors) notified.

Subject_base.cc

```
void SubjectBase::RegisterObserver(SensorBase * sensor) {
  observers_.push_back(sensor);
  if (sensor->get_range() > max_range) {
    max_range = sensor->get_range();
  }
}

void SubjectBase::Notify(const EventBaseClass * const event) {
  for (auto obs : observers_) {
    obs->Update(event);
  }
}

void SubjectBase::StateChange(const EventBaseClass * const event) {
    Notify(event);
}
```

Subject Distress Class

Subject_distress.h

```
class SubjectDistress : public SubjectBase {
 public:
  SubjectDistress(void) {}
};
```

Subject_entity_type.h

```
class SubjectEntityType : public SubjectBase {
 public:
  SubjectEntityType(void) {}
};
```

```
SubjectDistress * subject_distress_;
SubjectEntityType * subject_entity_type_;
```

ARENA

Subject Objects in Arena.

Changing State in Arena

```
/* Send out distress signals and entity types */
for (auto robot : robots_) {
  if (robot->InDistress()) {
    subject_distress_->StateChange(
      new EventDistress(robot->get_id(), robot->get_pos()));
  }
}
/* Transmit entity type to observers */
for (auto ent : entities_) {
  subject_entity_type_->StateChange(
    new EventEntityType(ent->get_id(),ent->get_pos(),ent->get_type()));
}
```

Create same functionality for Proximity and Touch – why not efficient?

DISTRESS SENSOR TESTS

- Constructor
- Distress Call in range
- Distress Call at range
- Multiple distress calls
- Reset
- Distress Call of self
- Distress Call out of range
- Distress Call in robot
- Set Range: too big, negative, float, normal, 0
- Set FOV: too big, negative, float, normal, 0

NEED LOTS OF SETUP

TEST FEATURE

```
class SensorDistressTest : public ::testing::Test {
 protected:
  virtual void SetUp() {
    // Initialize Distress Sensors for Collections of TESTS
    robot = new csci3081::Robot();
    sensor = new csci3081::SensorDistress(robot);
    robot_id = robot->get_id();

    event_own_distress.set_id(robot_id);

    event_internal.set_id(robot_id+1);
    event_internal.set_pos(csci3081::Position(10,0));

    event_at_range.set_id(robo
    event_at_range.set_pos(csc

    event_in_range.set_id(robo
    event_in_range.set_pos(csc

    event_out_of_range.set_id(
    event_out_of_range.set_pos
  }
```

```
// Default robot is at position (0,0) heading 0
csci3081::Robot * robot;
csci3081::SensorDistress * sensor;
int robot_id;

// Default range and fov is 50 and 360 degrees
csci3081::EventDistress event_own_distress;
csci3081::EventDistress event_internal;
csci3081::EventDistress event_in_range;
csci3081::EventDistress event_at_range;
csci3081::EventDistress event_out_of_range;
```

TESTS using TEST FEATURE

```
TEST_F(SensorDistressTest, Constructor) {
  // get range, fov, and activated
  EXPECT_EQ(sensor->get_range(), DEFAULT_RANGE) << "FAIL: Range:Constructor";
  EXPECT_EQ(sensor->get_fov_angle(), DEFAULT_FOV_ANGLE) << "FAIL: FOV:Constructor";
  EXPECT_EQ(sensor->IsActivated(), false) << "FAIL: Active:Constructor";
 }

TEST_F(SensorDistressTest, DistressCallInRange) {
   // A robot distress in range, not itself
   // should activate the sensor
   sensor->Update(event_in_range);
   EXPECT_EQ(sensor->IsActivated(), true);
 }
```

```
TEST_F(SensorDistressTest, SetRange) {
  // Bad input that should be ignored
  int range = sensor->get_range();
  sensor->set_range(1000);
  EXPECT_EQ(sensor->get_range(), range) << "FAIL: Too big, Range.";
  sensor->set_range(-10);
  EXPECT_EQ(sensor->get_range(), range) << "FAIL: Negative, Range.";

  // Input should be converted to an int
  sensor->set_range(1.2);
  EXPECT_EQ(sensor->get_range(), 1) << "FAIL: Double, Range";
```

TESTS

If you did not do the observer pattern, there wasn't much going on with your sensors.

Practice writing tests for that part of your code that you need to test – in other words, the in-range, out-of-range, etc. tests that are shown here.