COM S 362
Object-Oriented Analysis & Design

Complexity and the Object Model

Reading

Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications*, Third Edition, 2007.

Read Chapter 1 starting at "The Inherent Complexity of Software" and stop before "Organized and Disorganized Complexity", pp. 7-14.

Read Chapter 2 starting at "Object-Oriented Programming" and stop before "The Meaning of Typing", pp. 41-64.

Complexity

Industrial-Strength Software

- In this class we are interested in ***"industrial-strength software"***
- Software that is inherently complex enough that an individual developer cannot comprehend all the subtleties of its design.
  - Recall example of 100 million line of code in GM car
- Examples:
  - Real-time: reactive systems that are driven by events in the physical world with time and space constraints
  - Concurrency: systems that must maintain data integrity while allowing concurrent updates and queries
  - Dependability: systems that can gracefully recover from multiple hardware and network failures
  - Security: systems that don't accidentally expose information to malicious actors

Complexity

- ***Essential Complexity***: how hard something is to do, regardless of how experienced you are, what tools you use or what new and flashy architecture pattern you used to solve the problem. Some things are just hard and take a long time. –Marin Benčević
- ***Accidental Complexity***: complexity that is the result of things like the tools and languages we use.
- Discussed in classic paper "No Silver Bullet – Essence and Accident in Software Engineering" by Fred Brooks in 1986.
- Even if we remove all accidental complexity with better tools, the essential complexity remains (i.e., don't hope for a magic "silver bullet" tool that will give a 10x improvement in developer productivity).
- We cannot remove essential complexity, the best we can do is mange it with good design.

The Object Model and Principles of Design

Design Principle: Separation of Concerns

- **Separation of Concerns** means design choices related to separate concerns are in separate locations.
- Example: How a sensor takes a reading and when a sensor takes a reading are two different concerns, so split the when into a separate Scheduler class.
- Example: The value of a sensor reading and how the reading is displayed are two different concerns, so create SensorReading and SensorDisplay clases.

Abstraction

- "An ***abstraction*** denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer" –Booch et al.
  - Translation – distinguishes important from unimportant details, depends on the perspective of the user
- Goal: Reduce complexity.

What do you think about this code?

```
public class Sensor {
    …
    public void takeReading() {
        …
    }
    public setConstOffset(double offset) {
    }
    …
}
public class SensorScheduler {
    private Sensor sensor;
    …
    public void timerEvent() {
        sensor.takeReading();
    }
}
```

What do you think about this code?

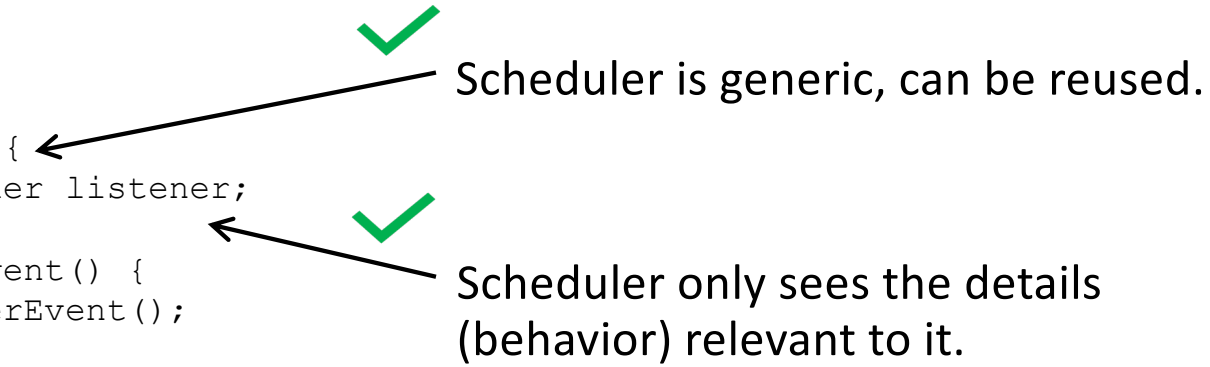
```
public class Sensor {
    …
    public void takeReading() {
        …
    }
    public setConstOffset(double offset) {
    }
    …
}
public class SensorScheduler {
    private Sensor sensor;
    …
    public void timerEvent() {
        sensor.takeReading();
    }
}
```

Is there something unique about a sensor that it needs its own scheduler class? The class seems unnecessarily specific.

Sensors are full of state and behavior that are completely irrelevant to a scheduler, why expose the scheduler to such details?

Applying Abstraction

```
interface EventListener {
    public void triggerEvent();
}
public class Sensor implements EventListener {
    …
    public void triggerEvent() {
        // take reading
        …
    }
    …
}
public class Scheduler {
    private EventListener listener;
    …
    public void timerEvent() {
        listener.triggerEvent();
    }
}
```



Encapsulation and Information Hiding

- "***Encapsulation*** is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface for an abstraction and its implementation." –Booch et al.
  - Translation: Hide the "secrets" and provide an interface for the abstraction.
- Goal: keep changes local
- Achieved by ***information hiding***
  - Information is more than data, it includes state, structure and behavior

What do you think about this code?

```
public class Sensor {
    private SensorValue value;
    private List<SensorObserver> listeners;
    …
    public SensorValue getValue() {
        return value;
    }
    public void setValue(double value) {
        this.value = value;
        update();
    }
    public List<SensorObserver> getListeners() {
        return listeners;
    }
    public void setListeners(List<SensorObserver> listeners) {
        this.listeners = listeners;
    }
}
```

What do you think about this code?

```
public class Sensor {
    private SensorValue value;
    private List<SensorObserver> listeners;
    …
    public SensorValue getValue() {
        return value;
    }
    public void setValue(double value) {
        this.value = value;
        update();
    }
    public List<SensorObserver> getListeners() {
        return listeners;
    }
    public void setListeners(List<SensorObserver> listeners) {
        this.listeners = listeners;
    }
}
```

Seems reasonable, obtaining a value is an essential behavior of a sensor.

Hmm, do we want users to decide when and how the sensor value is set?

Really?!?!

Bad Habit: Getters and Setters Everywhere

```
public class Sensor {
    private SensorValue value;
    private List<SensorObserver> listeners;
    …
    public SensorValue getValue() {
        return value;
    }
    public void setValue(double value) {
        this.value = value;
        update();
    }
    public List<SensorObserver> getListeners() {
        return listeners;
    }
    public void setListeners(List<SensorObserver> listeners) {
        this.listeners = listeners;
    }
}
```

Beginners Version of Encapsulation:
Make all variables private and give them all public getters and setters.

Good Encapsulation: Hiding Secrets – Keeping Changes Local

```
public class Sensor {
    private SensorValue value;
    private List<SensorObserver> listeners;
    …
    public SensorValue getValue() {
        return value;
    }
    public void applySource(Source source) {
        …
    }
    public void addObserver(SensorObserver listener) {
        …
    }
    public void removeObserver(SensorObserver listener) {
        …
    }
}
```

Instead of forcing a value on the sensor, we let the sensor decide when and how to take a reading from the source.

Sensor manages observers, no longer exposes object's secrets.

Modularity

- "***Modularity*** is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules." –Booch et al.
  - Translation – Make modules that don't depend much on each other (loosely coupled) and where the internals naturally belong together (cohesive).
- Goal: Make changeable and reusable modules.

Loosely Coupled

- Question: How should one module depend on another?
- Keep dependencies between modules small, change in one module should not affect change in others.
- Modules have interfaces
  - The interface should expose only the abstractions.
  - Why? Because abstractions are essential characteristics (they don't change).
  - We don't want our interface to change, because when an interface changes every code that uses the module might break.
  - On the other hand, extending an interface is not as costly.
  - Design principle: open for extension, closed for modification.

High Cohesion

- Question: What goes in a module?
  - Modules should contain abstractions that logically belong together.
  - Include parts of the code that are dependent on each other (i.e., change together).
  - Parts that change independently of each other should go in separate modules.
- “Each module’s structure should be simple enough that it can be understood fully” –Britton and Parnas

Hierarchy

- "***Hierarchy*** is a ranking or ordering of abstractions." –Booch et al.
  - Translation – With modularity we determined which abstractions go together, with hierarchy we give abstractions an ordering from most generic to most specific.
- Goal: Provide structure to the dependencies.
- Two main types of dependency: "has-a" (e.g., aggregation) and "is-a" (e.g., inheritance)