# Software Development: How to Do It Right in the First Place?

Prof. Jonathan Lee (李允中)

Department of CSIE

National Taiwan University

## Prologue

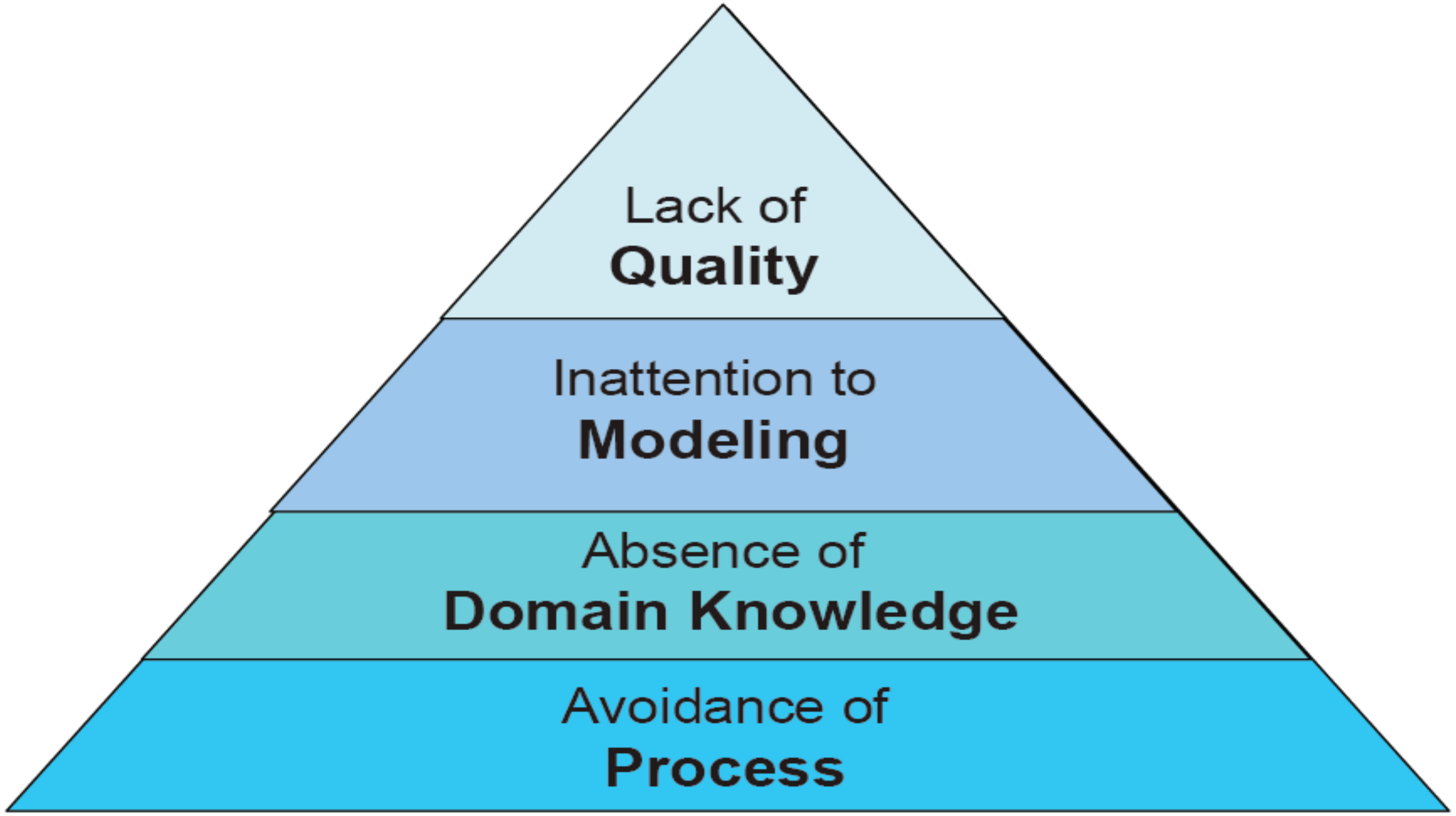
… there is plenty of emotional destruction. A ***team*** of people, probably operating in **crunch mode** on *a death march quest*, went down to defeat. Their hopes and plans for the future were probably clashed. There was probably ***blame and recrimination as the end approached***, and relationships were bent or broken. ***Self-belief*** came ***under attack***, and for some participants it was probably **plowed under…** from "Software Runaways" by Robert Glass

Robert L. Glass, Software Runaways, Prentice Hall PTR, 1998

## Key Characteristics from Prologue

- So, what we can take away from this prologue. Let me try to summarize the key characteristics of this prologue. What we have are the following:
- First, A software project with requirements that can be obtained from clients or products.
- Second, A project team with team members
- Finally, A deadline to beat, namely, a timing constraint

# Dysfunctions of Software Engineering Education



J. Lee and Y. C. Cheng. Change the Face of Software Engineering Education: a Field Report from Taiwan. Information and Software Technology, 53(1): 51-57, Jan 2011.

# Avoidance of Processes

- Avoidance of processes is probably the most serious one among the four dysfunctions.
- Software development processes are **intentionally overlooked** for a very long time because software developers usually don’t like to be **enforced to follow rules**. But, they never have a chance to be reminded that processes happen every day in their own daily lives.
- I can easily refer to several daily activities, such as brushing teeth, taking shower, eating breakfast, and so on so forth. Or, even giving a talk like what I am doing right now. We all follow a certain kind of processes in our daily life. Let me give you another most recent event for example.

## Blackout in Taiwan on 08152017

- Three days ago, there was a nation-wide blackout in Taiwan.
- A natural gas supply disruption to a major power plant in Taoyuan’s Datan Township on Aug 15 caused blackouts throughout the nation and subsequent power rationing.
- All six generators at the power plant shut down abruptly in the afternoon of Aug 15 when a natural gas supply line was cut due to **human errors by skipping activities in the maintenance process of natural gas valves** by the maintenance workers.
- Skipping activity in a process does cause serious **consequences**.

## Uphill Battle

- But, the problem is why not following processes in software development or in important tasks?
- In my viewpoint, one of the key reasons is software engineers like to program right away after they get the job assignments. They always think it would be faster (saving time) and easier (getting results) to program first without a well-prepared plan outlining the activities involved.
- This falls directly to a metaphor of what I called: Up-hill battle. So, what's an uphill battle?

## Steeper Slope Becomes Cliff

- That is, software developers usually jump right to the programming. At the beginning, it does give a quick initial result to the developers, but, it won’t last long before the developers hit a wall trying to fix bugs and to rewrite the code to meet a variety of changes.
- The slope of the hill gets steeper and steeper, and eventually, it comes to be a cliff, making the moving forward almost impossible. As a result, developers have to give it up and start the whole thing again.
- This **vicious cycle** repeats itself again and again in software development due to the avoidance of processes.

## Absence of Domain Knowledge

- Computer Science majors usually don’t have enough domain knowledge or expertise of other disciplines. But, the software we develop is usually a **knowledge-intensive** artifact.
- This is a serious loophole in our computer science curriculum.
- The remedy for the loophole can be alleviated by teaming computer science majors with people from other disciplines, and training them to develop a skillset for communicating with the domain experts to extract and compile viable requirements.

## Inattention to Modeling

- This is similar to the Avoidance to Processes. Programming without thinking ahead, which usually results in a vicious cycle of coding and debugging.
- Most of the time, bug-fixing ends up in vain due to a variety of change, including requirements and designs, or fixing something that is tedious and unimportant.
- It is imperative for the software developers to establish the **awareness** that programs are for **other programmers or future you** to read and understand. It is therefore important that software developers know how to analyze the requirements and come up with a model (for example, design with design patterns) that can be mapped to its requirements.

## Lack of Quality

- There are so many possibilities of lacking quality.
- We should stay focus on the infrastructure and unit testing and integration testing here.
- In order to ensure the quality of software, developers need to be able to write their own test cases for unit testing, and co-work with other developers to come up with scenarios for integration testing.
- All of these require an underlying mechanism to support part of software development processes, that is, DevOps for Continuous Integration and Deployment.

## Take Away from the Pyramid

- Honor the processes, no matter what
- Communicate with domain experts to extract and compile viable requirements.
- Establish awareness that programs are for others and future you to read and understand.
- DevOps Infrastrurcture

## Main Challenges of How to Do it Right

- Create an environment similar to the real world surroundings with timing constraints as described in the prologue
- Software development practice starting from requirements in order to develop software engineering core competence to address the dysfunctions of software engineering education.
  - Honor the processes, no matter what
  - Communicate with domain experts to extract and compile viable requirements.
  - Establish awareness that programs are for others and future you to read and understand.
  - DevOps Infrastrurcture

## Environment and Practices

- What are the main challenges in training software engineers?

- We want to train our software engineers to always start with the requirements and to get used to it, analyze the requirements to establish a design model from the initial design to re-design with design process, map the design to program, and finally test code with test cases.

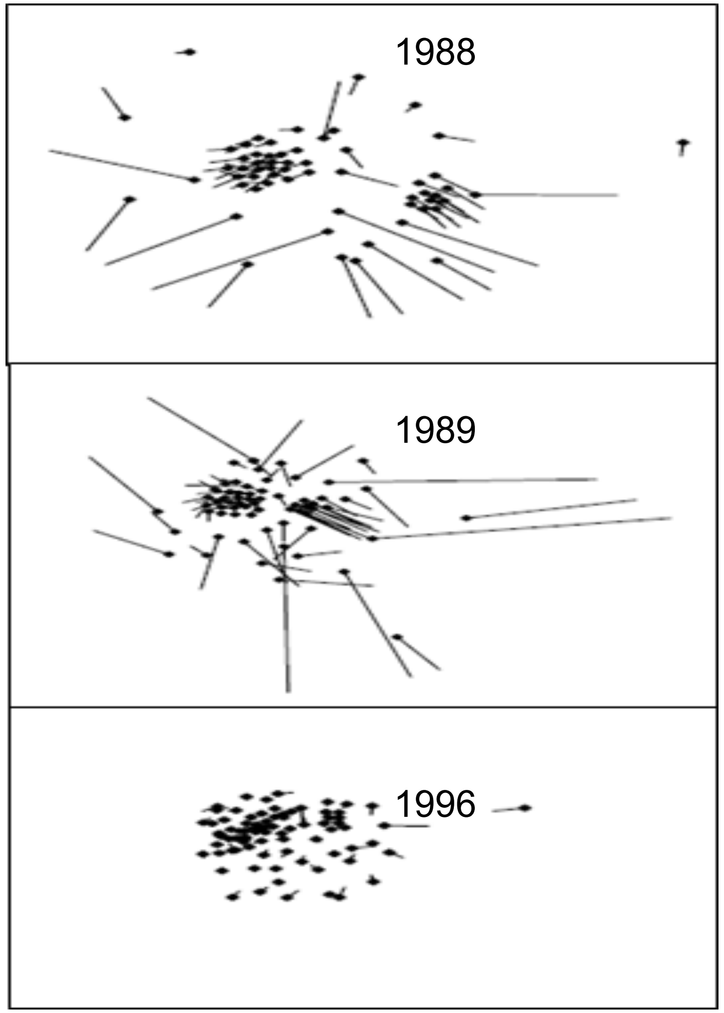
# A Light-Weight Software Development Process

Prof. Jonathan Lee
CSIE Department
National Taiwan University

# Motivation: Software/Code Decay

- A unit of code (in most cases, a module) is decayed if it is *harder to change than it should be*, measured in terms of effort, interval and quality.

- An Example: The Telephone Switches Project
  - Fifteen-year old real-time software system for telephone switches
  - 100,000,000 lines of source code (C/C++) and 100,000,000 lines of header and make files, organized into some 50 major subsystems and 5,000 modules.
  - More than 10,000 software developers have participated.
  - A module within one of the clusters is often changed together with other modules in the cluster but not with other modules
    - Head of each tadpole-like shape corresponds to a module



S.G. Eick ; T.L. Graves ; A.F. Karr ; J.S. Marron ; A. Mockus, **Does code decay? Assessing the evidence from change management data, IEEE Transactions on Software Engineering, 1-12, 27(1), 2001**

## What is a Software Process?

- A *software process* is a set of activities, methods, practices and transformations that people employ to develop and maintain software and the associated products, including requirements documents, project plans, design documents, code, test cases, and etc.
- In Software Engineering, processes are the fundamental for almost everything, for example, software process, DevOps process, project management process, risk management process, change control process, and etc.

## Software Development Process

- Engineering Practices
  - Requirements
  - System Architecture
  - Software Design

- Project Management
  - WBS
  - Meeting Minutes

# Project Management

- Project Plan: Work Breakdown Structure (WBS)
  - Tasks breakdown
  - Estimation: effort, system size, schedule, cost
  - Responsibility: task assignment and commitment
  - Team Work
- Project Monitoring and Control: Meeting minutes
  - Objective
  - Agenda: from WBS or team members' responsibility or risks ..
  - Issues: technical review, verification and validation, schedule deviation
  - Action Items: open, ongoing, suspended, close

## Engineering Practices

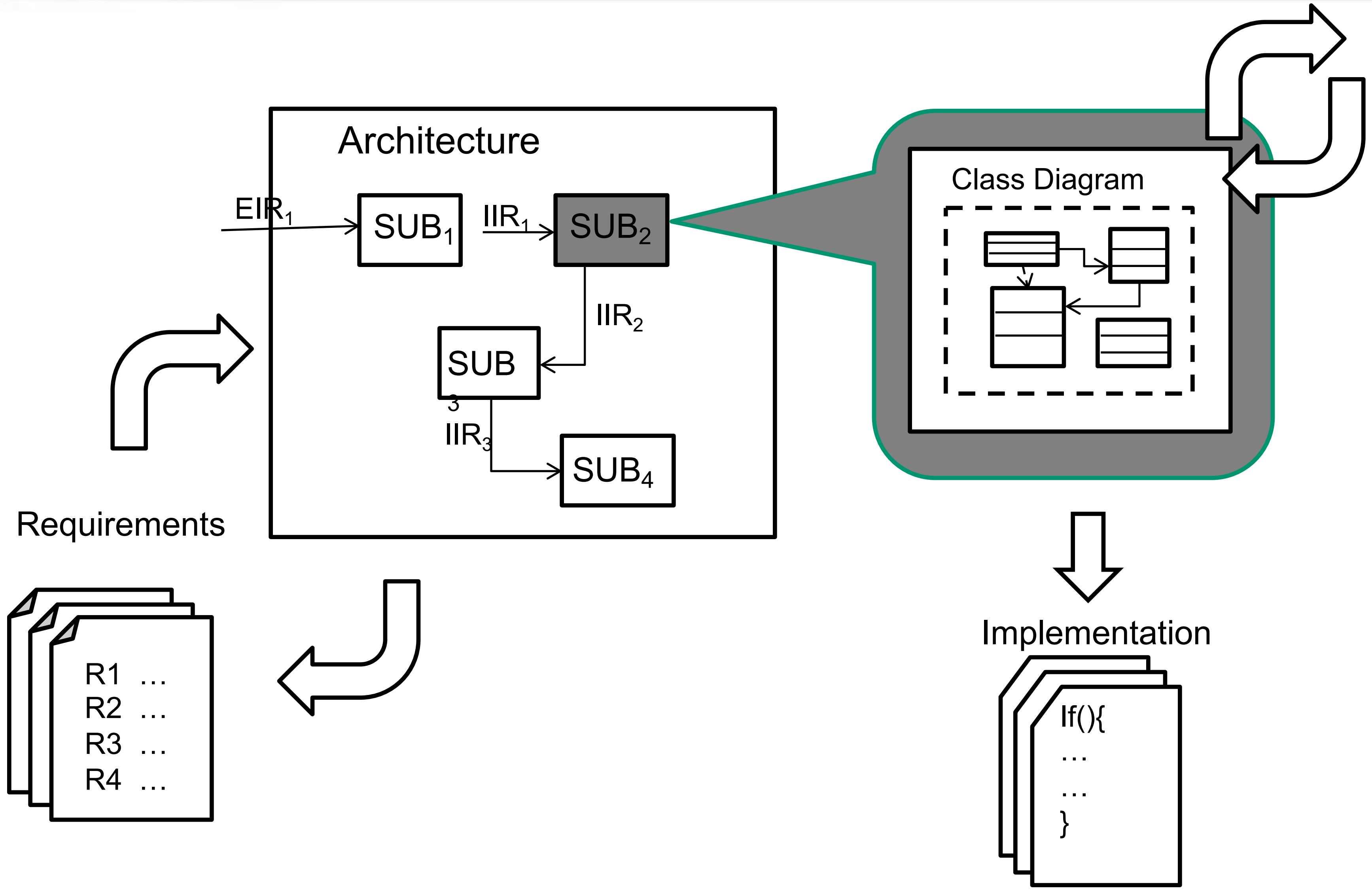
- Requirements:
  - How to elicit and refine requirements: never complete, always vague, and change all the time
  - How to document requirements: functional, nonfunctional, interface requirements
- System Architecture
  - Why do we need a system architecture?
  - How do we iterate between requirements and architecture?
- Software Diagram
  - Initial Design: based on requirements and system architecture
  - Redesign: Design issues and design principles

# Engineering Practices Illustration

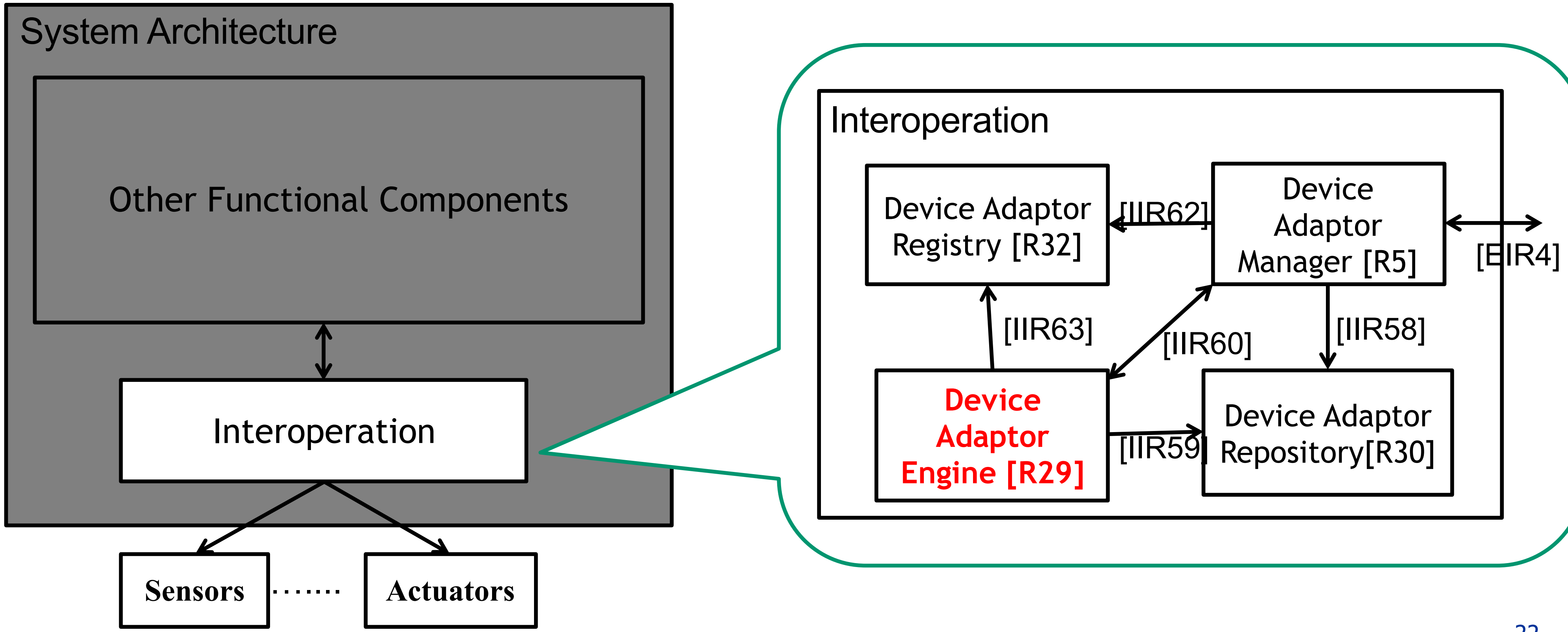
EIR: External Interface Requirement

IIR: Internal Interface Requirement

SUB: Sub-System



# Refine System Architecture


System Architecture
Other Functional Components
Interoperation
Sensors
Actuators
Interoperation
Device Adaptor Registry [R32]
[IIR62]
Device Adaptor Manager [R5]
[EIR4]
[IIR63]
[IIR60]
[IIR58]
Device Adaptor Engine [R29]
[IIR59]
Device Adaptor Repository[R30]

## From Requirements to Design: How to Iterate?

- ❑ Follow requirements to the letter to construct class diagrams in a stepwise way.
- ❑ Find out the most important operations and their scenario.
- ❑ Compile a sequence diagram of the scenario.
- ❑ List the design issues or concerns in conducting the design to serve as a basis for further discussion with requirements providers.

## Where and How to Obtain Requirements?

- ❑ From our clients (customers), users, and general public

- ❑ How to obtain?
  - ➢ Traditional approach: Clients or customers pay us to develop software
  - ➢ More realistic and feasible approach: We, as software developers, pay our clients, customers, users, and general public to obtain requirements

# Patient Monitoring System

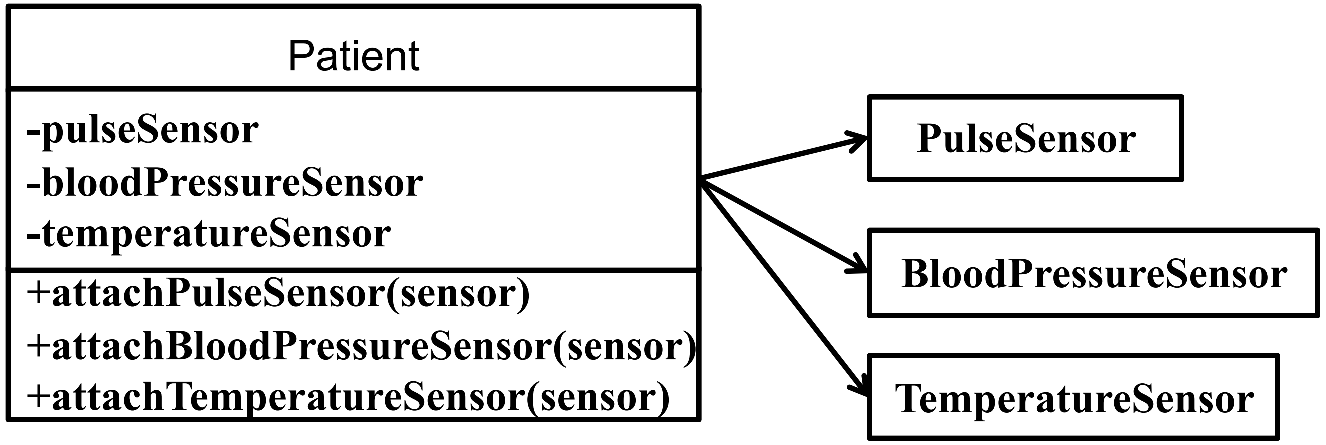
Prof. Jonathan Lee
CSIE Department
National Taiwan University

## Requirements

- Patients in an intensive-care ward in a hospital are monitored by electronic analog devices attached to their bodies by sensors of various kinds.
- Through the sensors the devices measure the patients’ vital factors: one device measure pulse rate, another blood pressure, another temperature, and so on.
- A program is needed to read the factors, at a frequency specified for each patient, and store them in a database.
- The factors read are to be compared with safe ranges specified for each patient, and readings that exceed the safe ranges are to be reported by alarm messages displayed on screen of the nurses station. An alarm message is also to be displayed if any analog device fails.

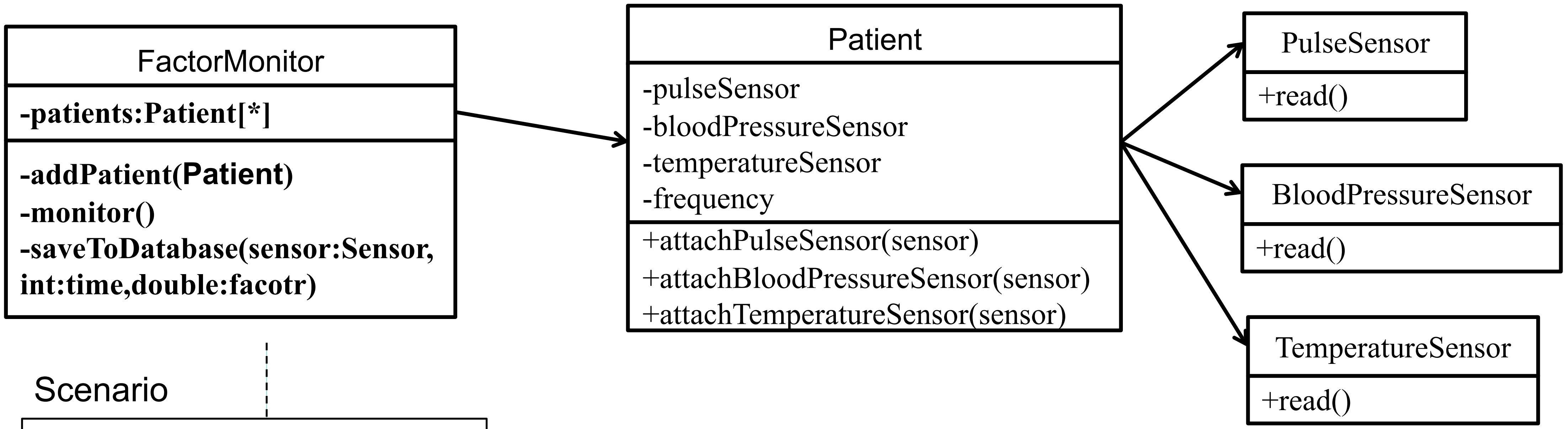
# Requirements Breakdown$_1$

- Patients in an intensive-care ward in a hospital are monitored by electronic analog devices attached to their bodies by sensors of various kinds.
- Through the sensors the devices measure the patients' vital factors: one device measure pulse rate, another blood pressure, another temperature, and so on.



# Requirements Breakdown$_2$

- A program (Factor Monitor) is needed to read the factors, at a frequency specified for each patient, and store them in a database.

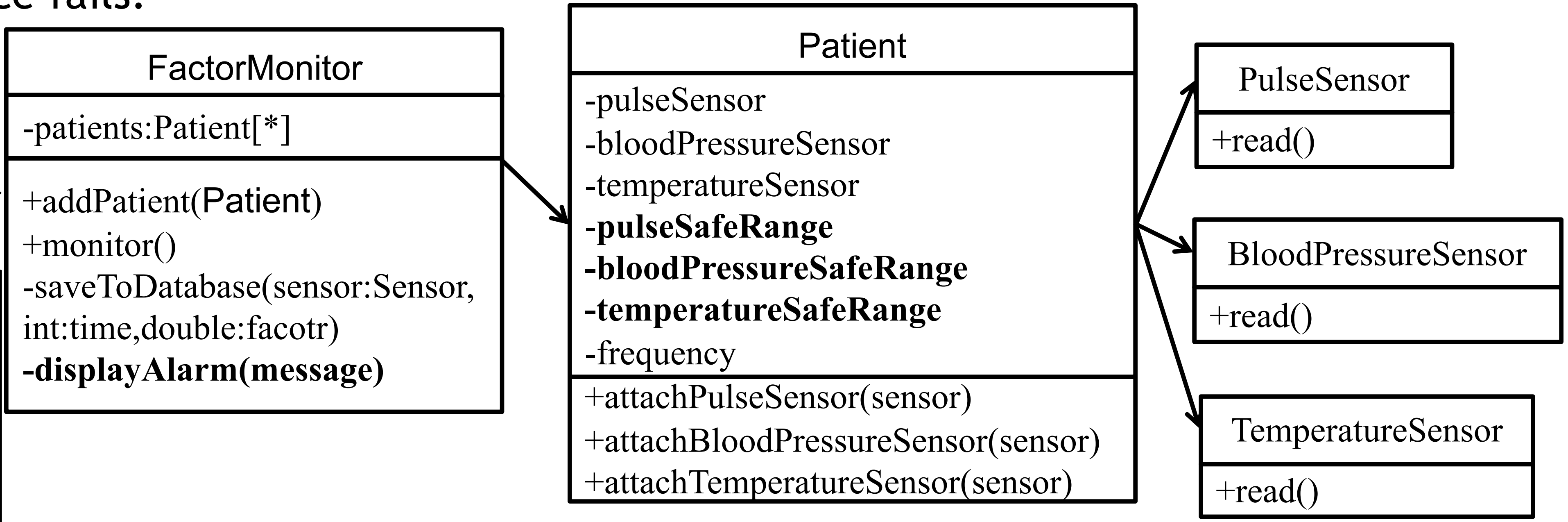


Scenario

```
monitor(){
 for(int time=0;;time++){
   for(Patient patient:patients)
     if(time%patient.getFrequency()==0){
       for(Sensor sensor:patient.getSensors()){
           factor = sensor.read()}
        saveToDatabase(sensor,time,factor)
           …
}}}
```

# Requirements Breakdown$_3$

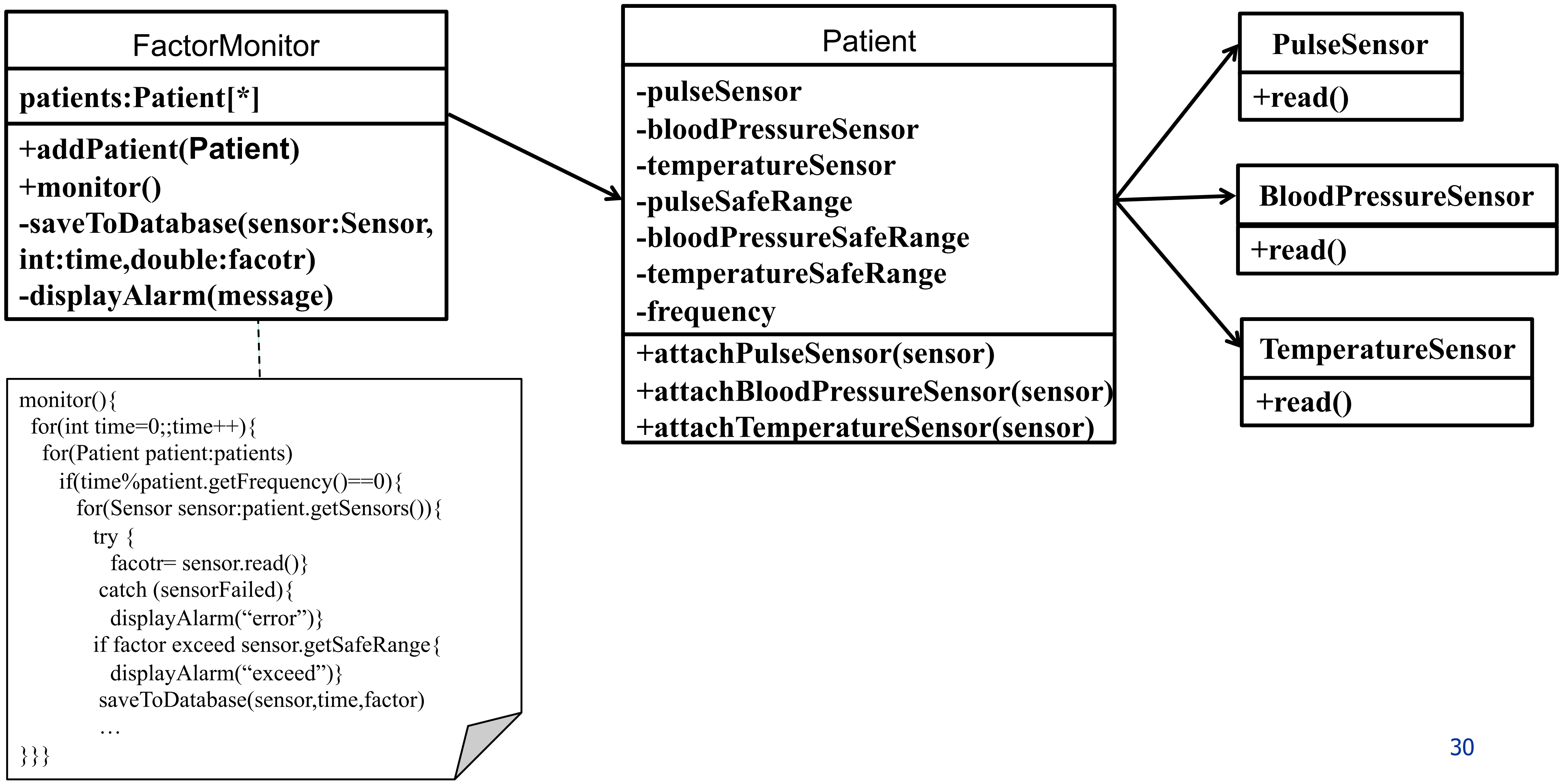
- The factors read are to be compared with safe ranges specified for each patient, and readings that exceed the safe ranges are to be reported by alarm messages displayed on screen of the nurses station. An alarm message is also to be displayed if any analog device fails.


FactorMonitor
-patients:Patient[*]
+addPatient(Patient)
+monitor()
-saveToDatabase(sensor:Sensor, int:time,double:facotr)
**-displayAlarm(message)**
Patient
-pulseSensor
-bloodPressureSensor
-temperatureSensor
**-pulseSafeRange**
**-bloodPressureSafeRange**
**-temperatureSafeRange**
-frequency
+attachPulseSensor(sensor)
+attachBloodPressureSensor(sensor)
+attachTemperatureSensor(sensor)
PulseSensor
+read()
BloodPressureSensor
+read()
TemperatureSensor
+read()

Scenario

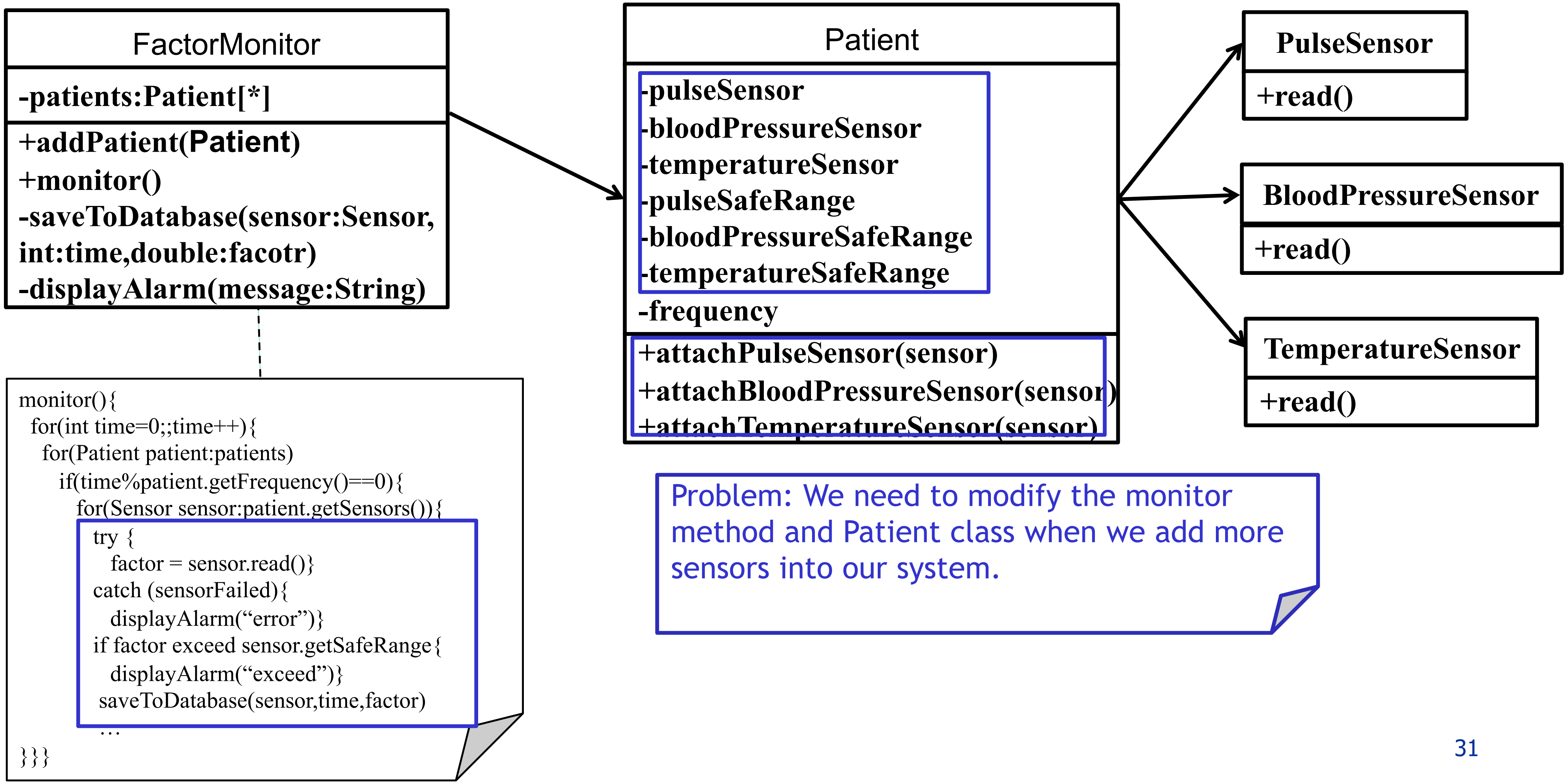
```
monitor(){
 for(int time=0;;time++){
   for(Patient patient:patients)
     if(time%patient.getFrequency()==0){
       for(Sensor sensor:patient.getSensors()){
         try {
           factor = sensor.read()}
         catch (sensorFailed){
           displayAlarm("error")}
         if factor exceed sensor.getSafeRange{
           displayAlarm("exceed")}
          saveToDatabase(sensor,time,factor)
          …
}}}
```

# Initial Design



```
monitor(){
 for(int time=0;;time++){
   for(Patient patient:patients)
      if(time%patient.getFrequency()==0){
         for(Sensor sensor:patient.getSensors()){
           try {
             facotr= sensor.read()}
            catch (sensorFailed){
             displayAlarm("error")}
           if factor exceed sensor.getSafeRange{
             displayAlarm("exceed")}
            saveToDatabase(sensor,time,factor)
            …
}}}
```

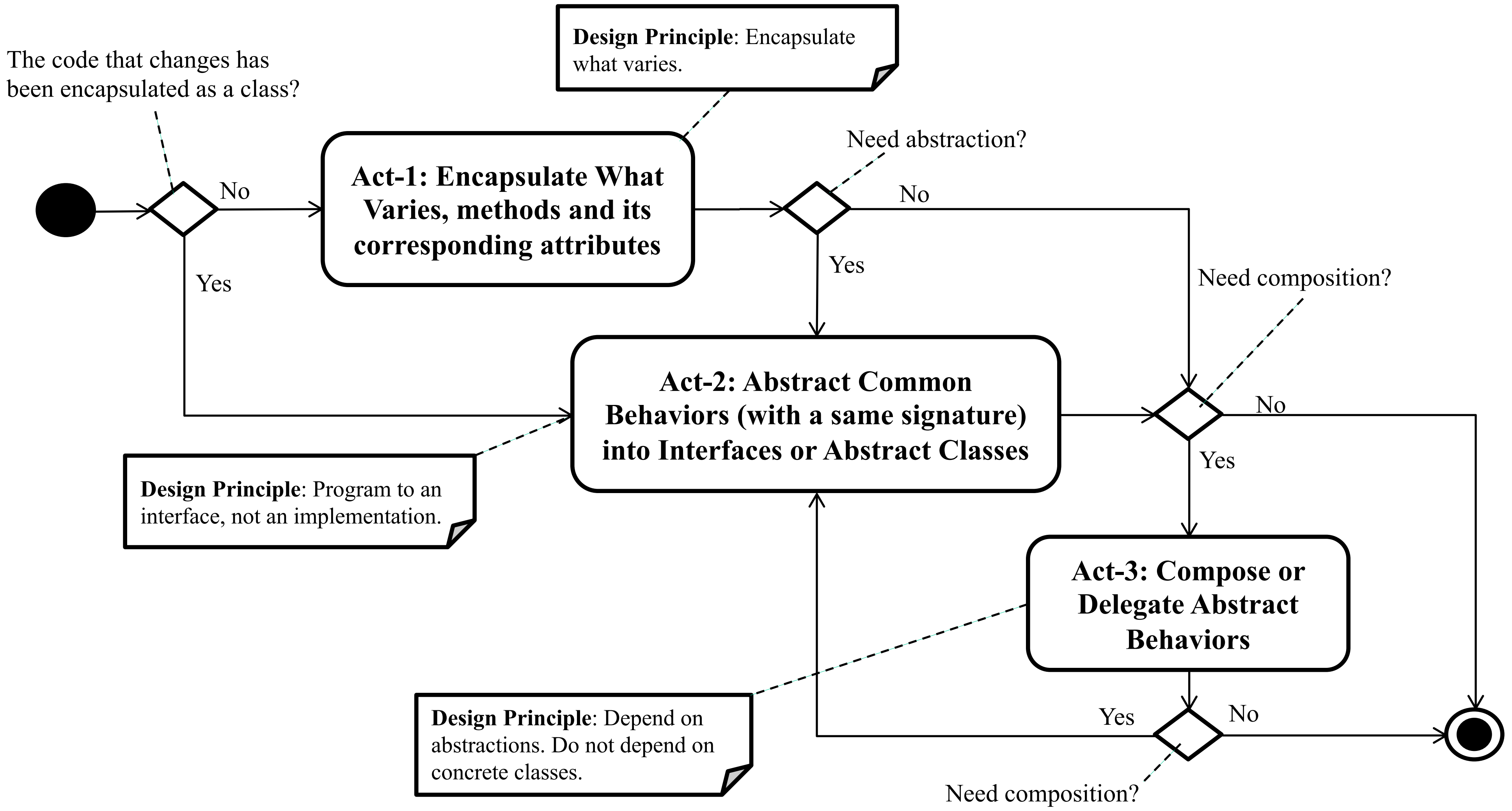
# Problem with Initial Design



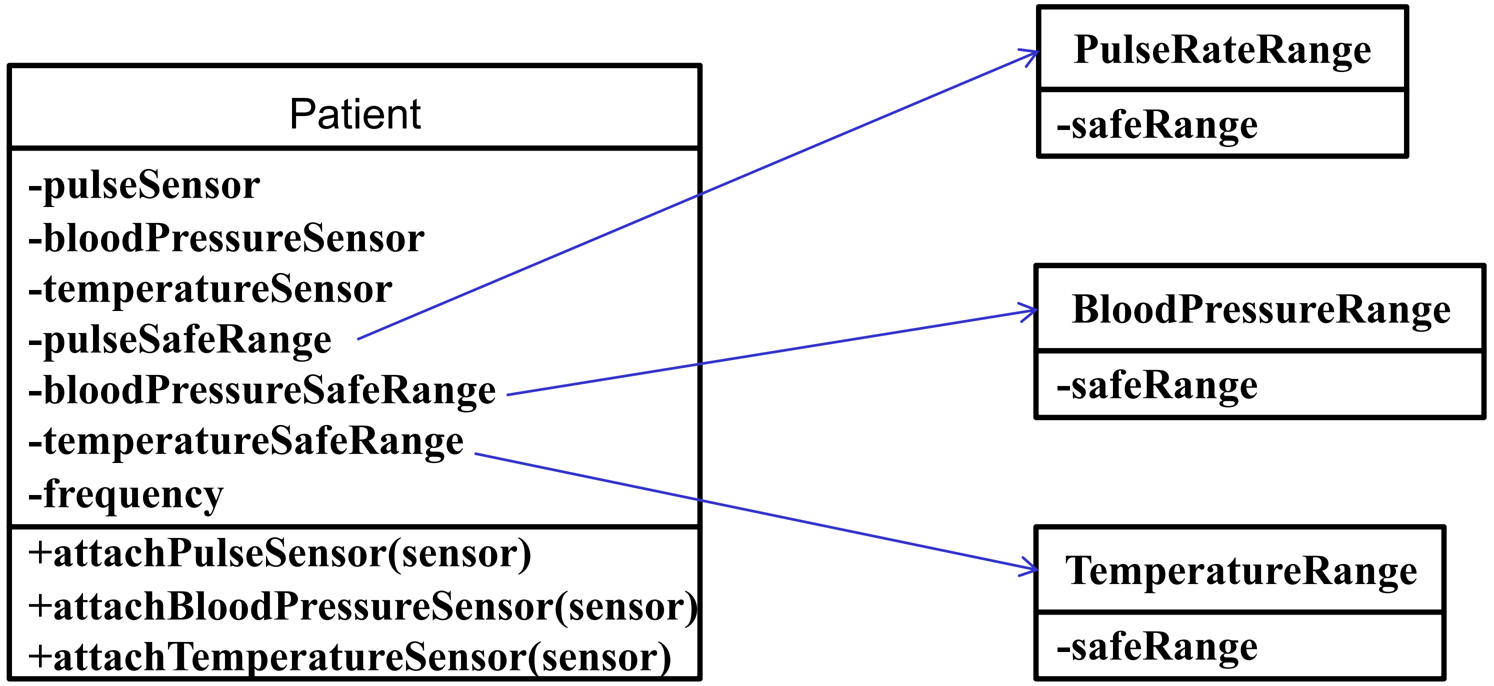
```
monitor(){
  for(int time=0;;time++){
    for(Patient patient:patients)
       if(time%patient.getFrequency()==0){
          for(Sensor sensor:patient.getSensors()){
             try {
                factor = sensor.read()}
             catch (sensorFailed){
                displayAlarm("error")}
             if factor exceed sensor.getSafeRange{
                displayAlarm("exceed")}
              saveToDatabase(sensor,time,factor)
              …
}}}
```

Problem: We need to modify the monitor method and Patient class when we add more sensors into our system.

## Software Design Process

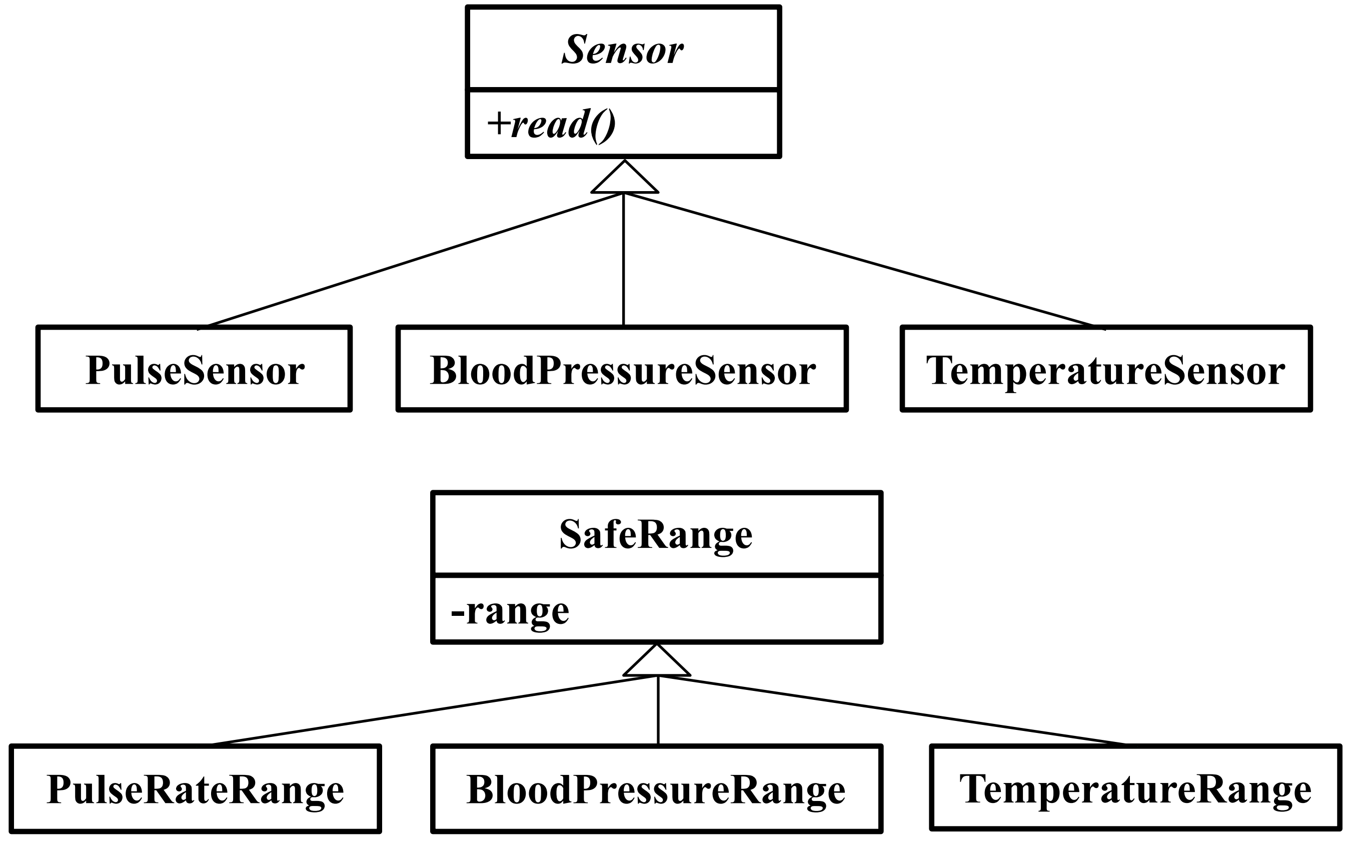

The code that changes has been encapsulated as a class?
Design Principle: Encapsulate what varies.
No
Yes
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
No
Yes
Design Principle: Program to an interface, not an implementation.
Act-3: Compose or Delegate Abstract Behaviors
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No
Need composition?

# Act-1: Encapsulate What Varies



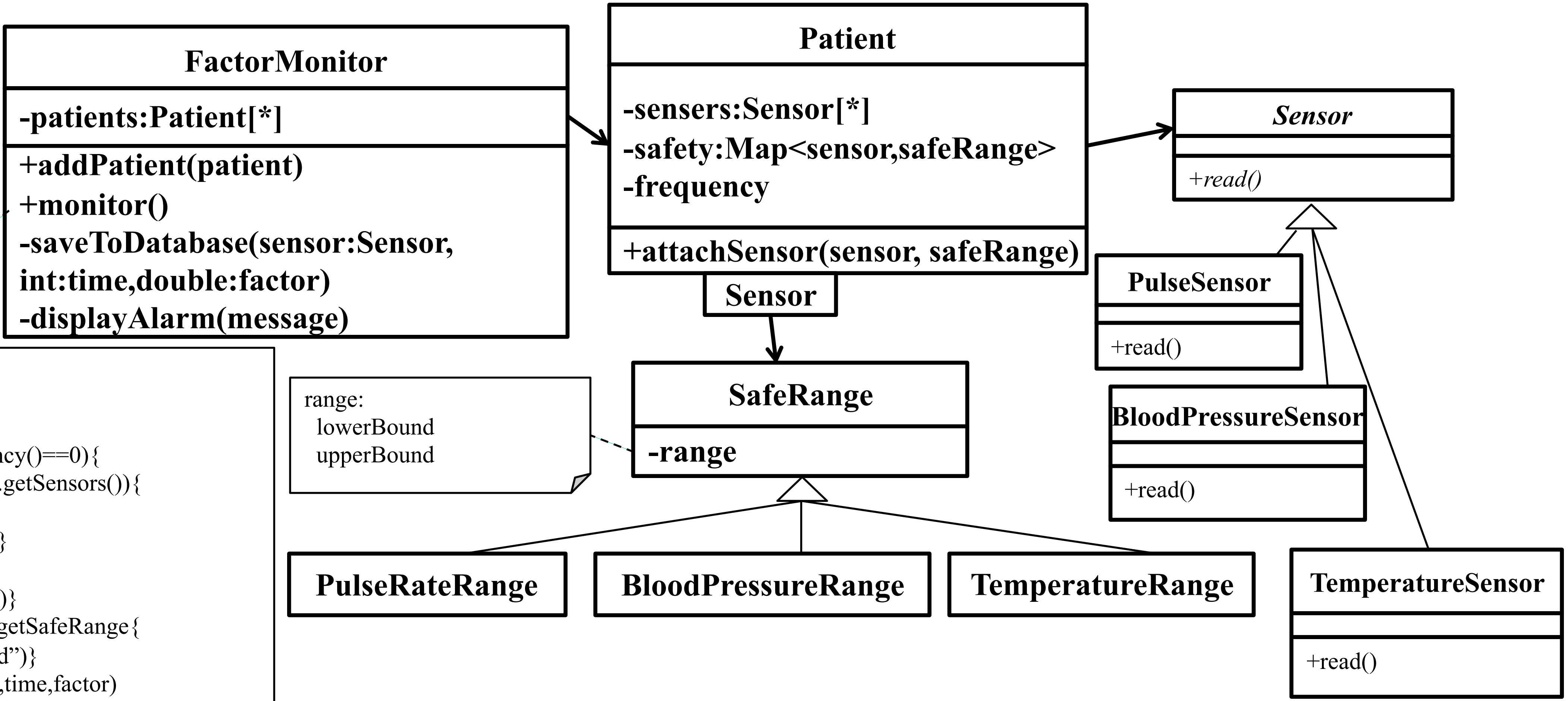
Act-1: Since the sensors are already concrete classes, we just move them into the class

# Act-2: Abstract Common Behaviors



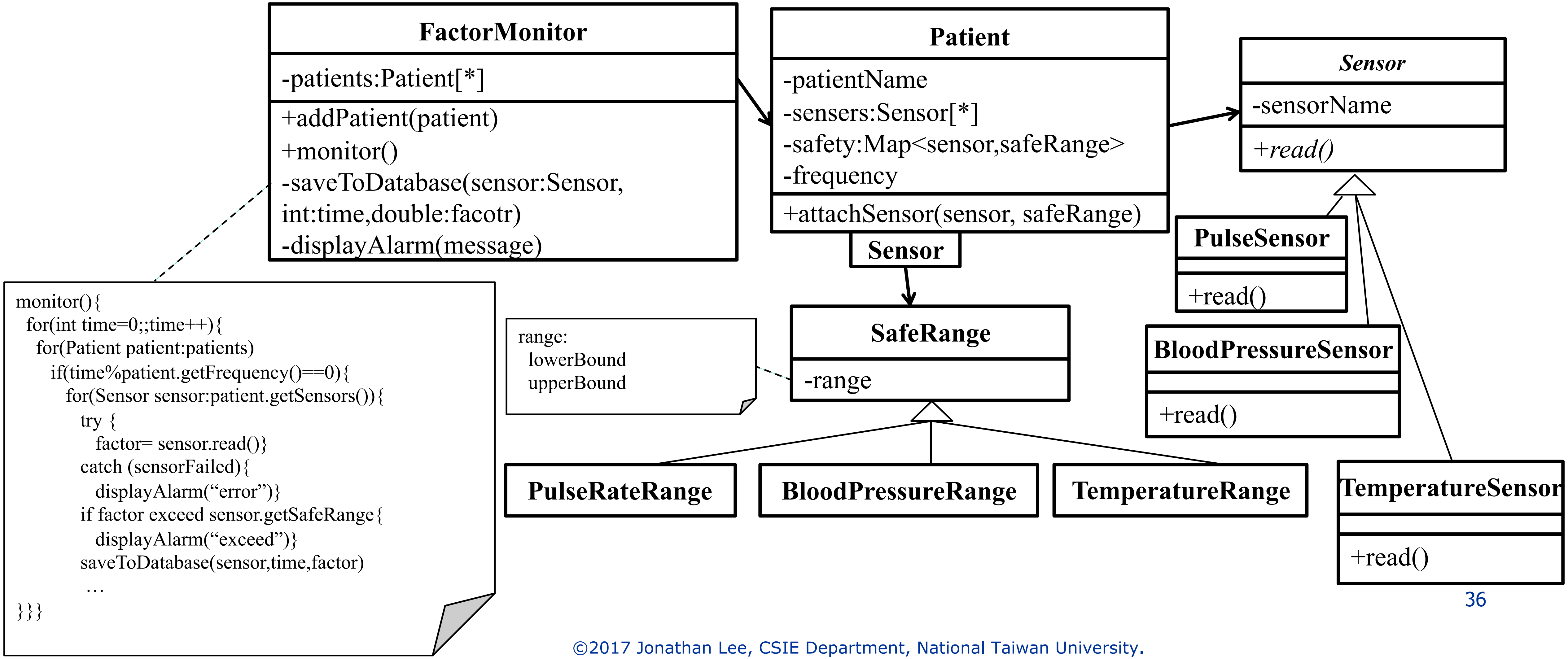
Act-2.2: Abstract common behaviors into abstract classes through polymorphism

# Act-3: Compose Abstract Behaviors



```
monitor(){
  for(int time=0;;time++){
    for(Patient patient:patients)
      if(time%patient.getFrequency()==0){
        for(Sensor sensor:patient.getSensors()){
          try {
            factor = sensor.read()}
          catch (sensorFailed){
            displayAlarm("error")}
          if factor exceed sensor.getSafeRange{
            displayAlarm("exceed")}
          saveToDatabase(sensor,time,factor)
           …
}}}
```

# Final Design



```
monitor(){
  for(int time=0;;time++){
    for(Patient patient:patients)
      if(time%patient.getFrequency()==0){
        for(Sensor sensor:patient.getSensors()){
          try {
            factor= sensor.read()}
          catch (sensorFailed){
            displayAlarm("error")}
          if factor exceed sensor.getSafeRange{
            displayAlarm("exceed")}
          saveToDatabase(sensor,time,factor)
           …
}}}
```

## Test Cases

- ❑ TestCase1: Read data from 3 types of sensor (6 points)
- ❑ TestCase2: Safe range of 3 types of sensor (6 points)
- ❑ TestCase3: Multiple patient (6 points)
- ❑ TestCase4: Same type of sensor attached to a patient (4 points)
- ❑ TestCase5: Sensor fail (6 points)
- ❑ TestCase6: Complex Test Case (2 points)

# Software Engineering Basic Core Competence

- Fundamental
  - Think computationally
- Teamwork
  - Co-work in software development and maintenance
- Problem Space
  - Build abstractions and perform problem decompositions
- Solution Space (I)
  - Analyze and model complex systems

J. Lee, A. Liu, Y-C. Cheng, S.-P. Ma and S.-J. Lee. Execution Plan for Software Engineering Education in Taiwan. In Proceedings of the 19th Asia-Pacific Software Engineering Conference, Hong Kong, Dec. 2012.

## Software Engineering Advanced Core Competence

- Solution Space (II)
  - Develop and verify complex systems
- Software Evolution:
  - Manage and evolve large-scale design and development efforts
- User Experience:
  - Create instinct-awareness user-friendly interfaces

# Practice for Developing Competence

- With the software development practice as demonstrated in my previous slides, we can train software engineers to think computationally every time they encounter requirements in natural language by analyzing the requirements and establishing their corresponding design.
- This practice requires members of a team to work together via Work Breakdown Structure and meeting minutes.
- While reading the requirements, an incremental way of analyzing the requirements is introduced to model classes and relations and with design process.