Assignment Part I - Design Modeling:

1.1 Brainstorming and CRC Cards:
Scenarios #1—Student Registration System

Student	
ID Name Login Logout RequstCourse RegisterCourse DropCourse ViewRepordCard	MessageManager

Professor	
ID Name Login Logout SelectCourseTeach EditCourseInfo ViewCourse RecordGrade	MessageManager

Registar	
ID Name Login Logout MaintainCoureInfo MaintainStudentInfo MaintainProfessorInfo	MessageManager

Security	
ConnectDatabase VerifyInfo	DatabaseManager

DatabaseManager	
CreateDatabase DeleteDatabase CreateTable DeleteTable CreateFields DeleteFields	Security

MessageManager	
PassMessage	Student Processor Registar Security DatabaseManager Course GradeManager

Course		
CourseID Name Semester Room Capacity Professor Update	JudgeCapacity Notify	Student Professor MessageManager

GradeManager	
Course Professor Students ReportGrade	Student Professor MessageManager

Billing	
Semester Students Bill SendBill	Student

Scenarios #2—Mine Pump Control System

Operator	
ID Name ObserveStatusOfSensor StartPump StopPump ReadMethaneSensor	Security WaterSensor

Supervisor	
ID Name ObserveStatusOfSensor StartPump StopPump ResetSystem ReadMethaneSensor	Security WaterSensor

Security	
ConnectDatabase VerifyInfo	DatabaseManager

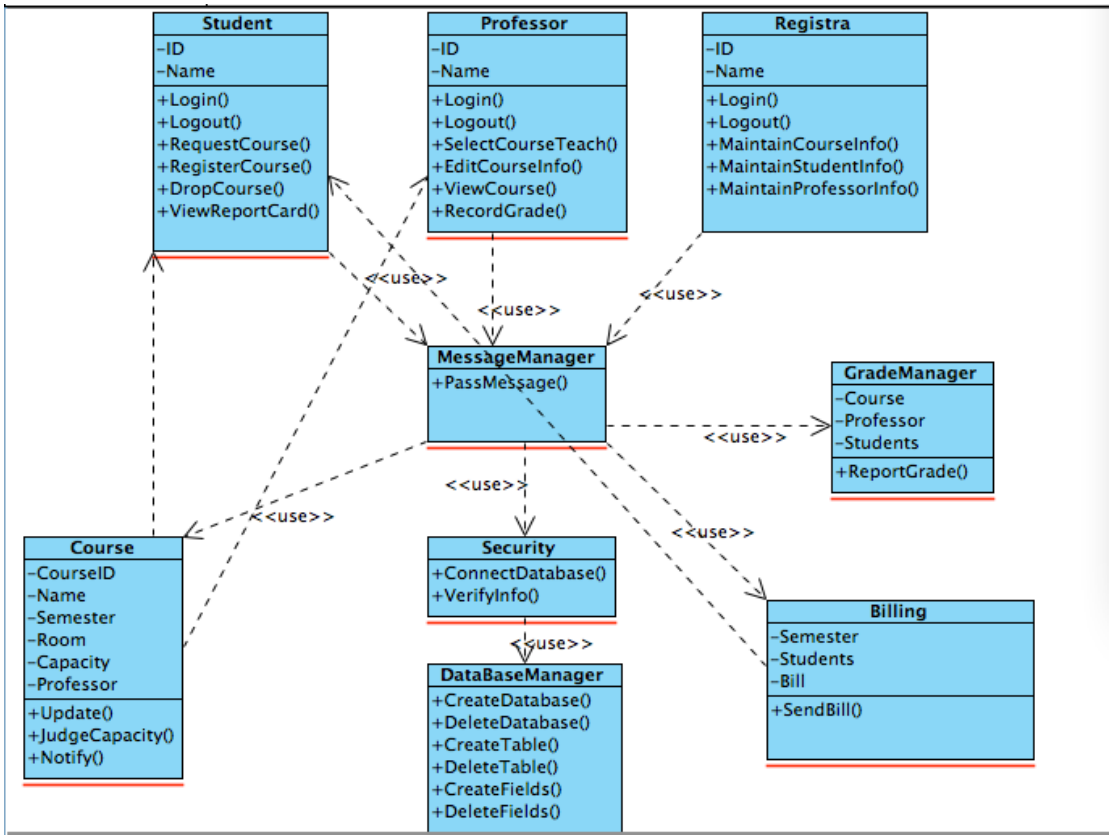
DatabaseManager	
CreateDatabase DeleteDatabase CreateTable DeleteTable CreateFields DeleteFields	Security

WaterSensor	
Level MinimumLevel MaximumLevel CheckLevel SwitchOn SwitchOff	Operator Supervisor

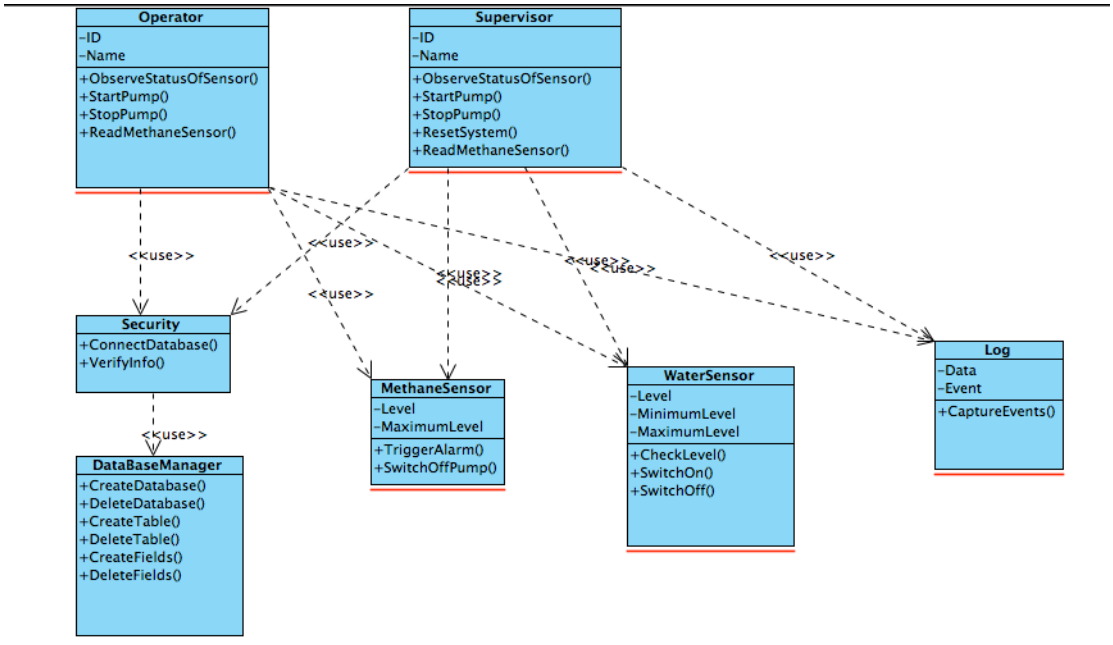
MethaneSensor	
Level MaximumLevel TriggerAlarm SwitchOffPump	Operator Supervisor

Log	
Date Event CaptureEvents	WaterSensor MethaneSensor Operator Supervisor

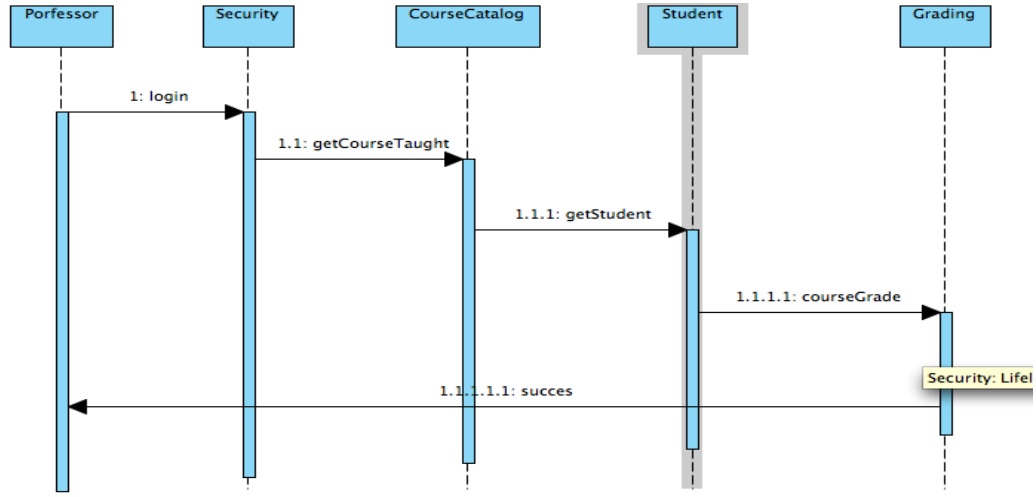
1.2 Class Diagrams:
Scenarios #1—Student Registration System

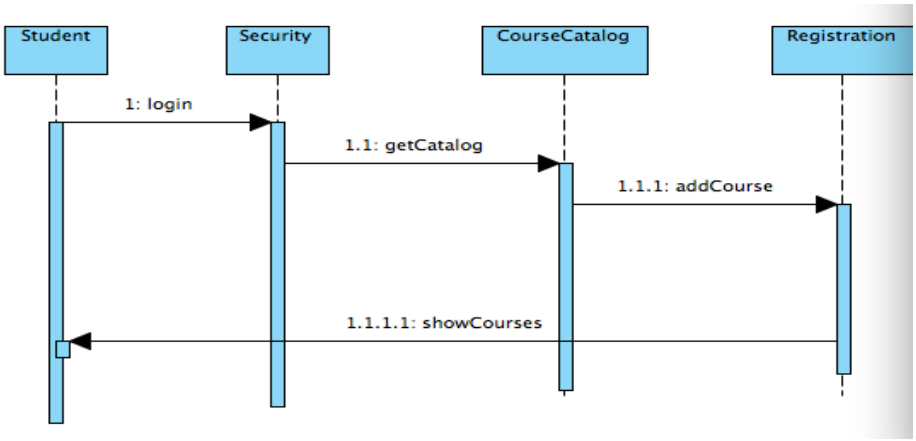


Scenarios #2—Mine Pump Control System

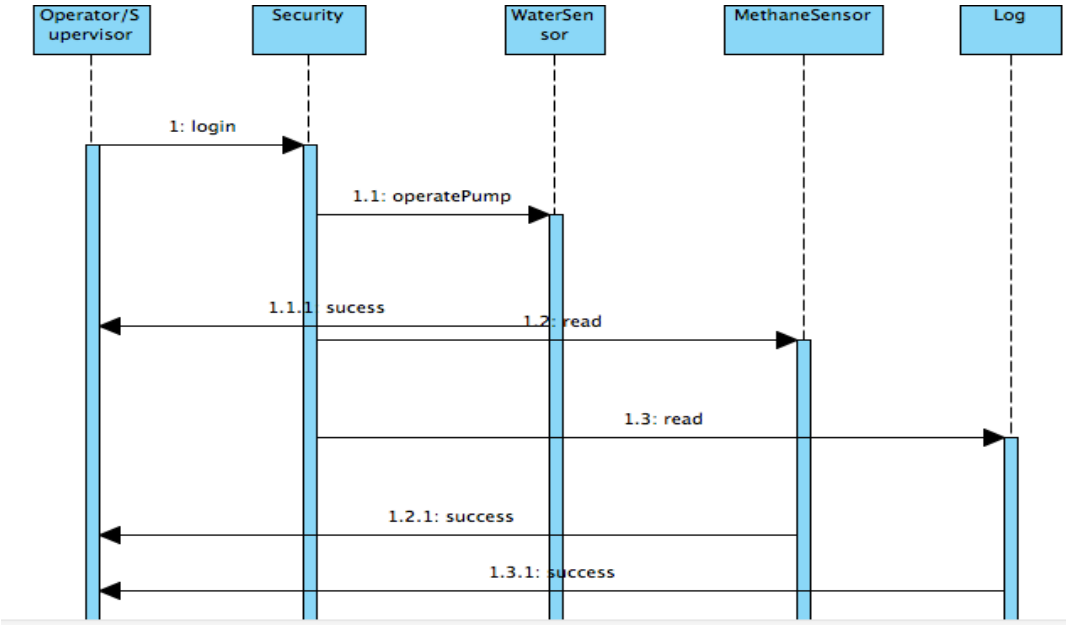


1.3 Sequence Diagrams:
Scenarios #1—Student Registration System





Scenarios #2—Mine Pump Control System



1.4 Inheritance Versus Composition and Architectural Evaluation Discussion

Administrator and service engineer could not inherit from supervisor or operator directly. There should exist multiple inheritances because there is no “is-a” typology in this situation.

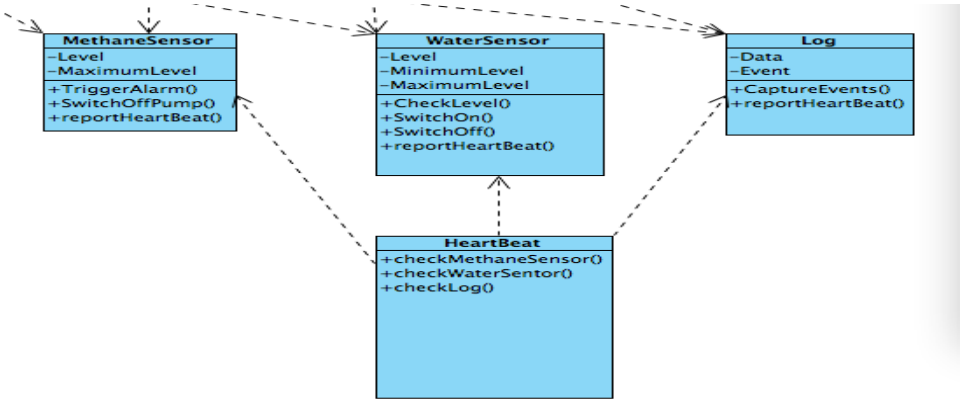
Make sure inheritance models the is-a relationship My main guiding philosophy is that inheritance should be used only when a subclass is-a superclass. An important question to ask yourself when you think you have an is-a relationship is whether that is-a relationship will be constant throughout the lifetime of the application and, with luck, the lifecycle of the code.

Don't use inheritance just to get code reuse If all you really want is to reuse code and there is no is-a relationship in sight, use composition.

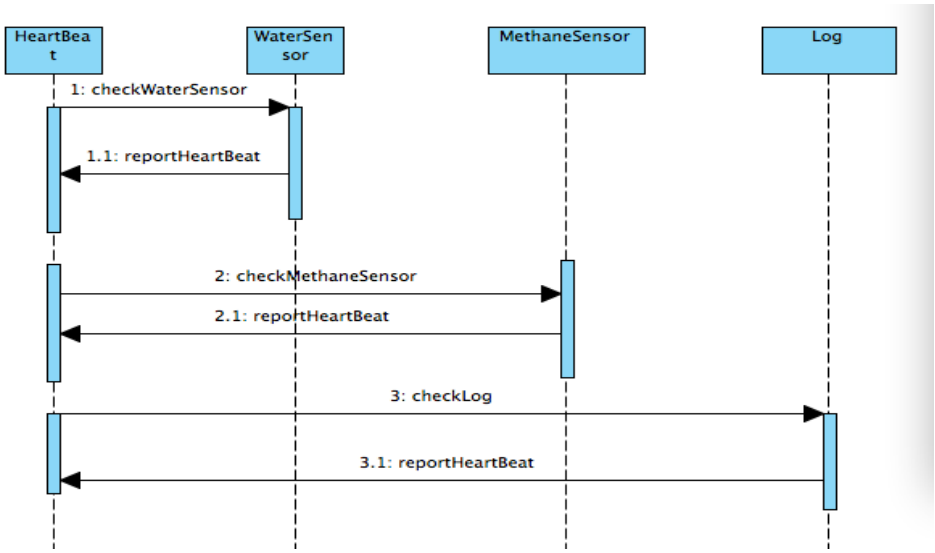
Don't use inheritance just to get at polymorphism If all you really want is polymorphism, but there is no natural is-a relationship, use composition with interfaces.

One drawback to using composition in place of inheritance is that all of the methods being provided by the composed classes must be implemented in the derived class, even if they are only forwarding methods. In contrast, inheritance does not require all of a base class's methods to be re-implemented within the derived class. Rather, the derived class need only implement (override) the methods having different behavior than the base class methods. This can require significantly less programming effort if the base class contains many methods providing default behavior and only a few of them need to be overridden within the derived class.

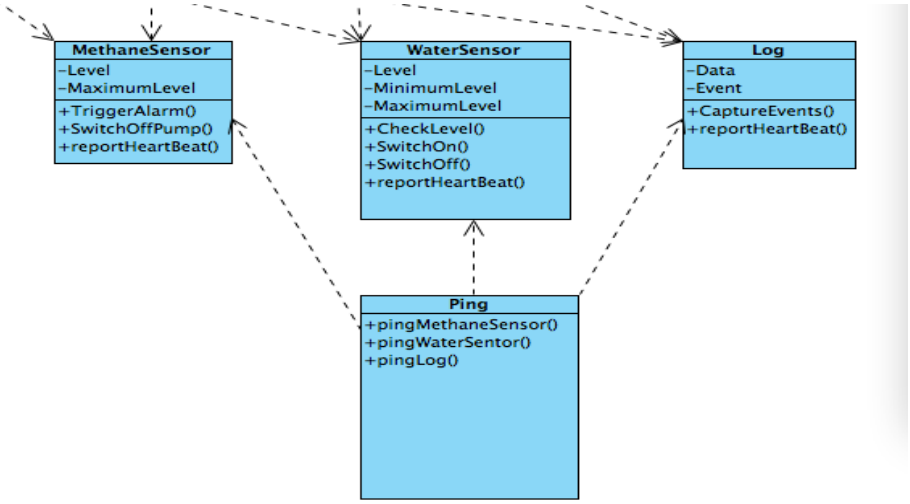
Assignment Part II – Architecture Analysis:
2. Architectural Capabilities
2.1 Fault Detection
HeartBeat Monitor Class Diagram



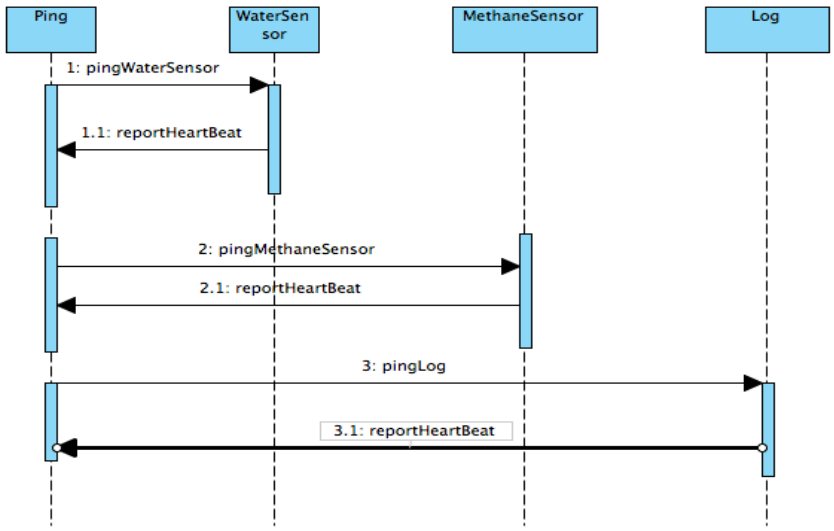
HeartBeat Monitor Sequence Diagram



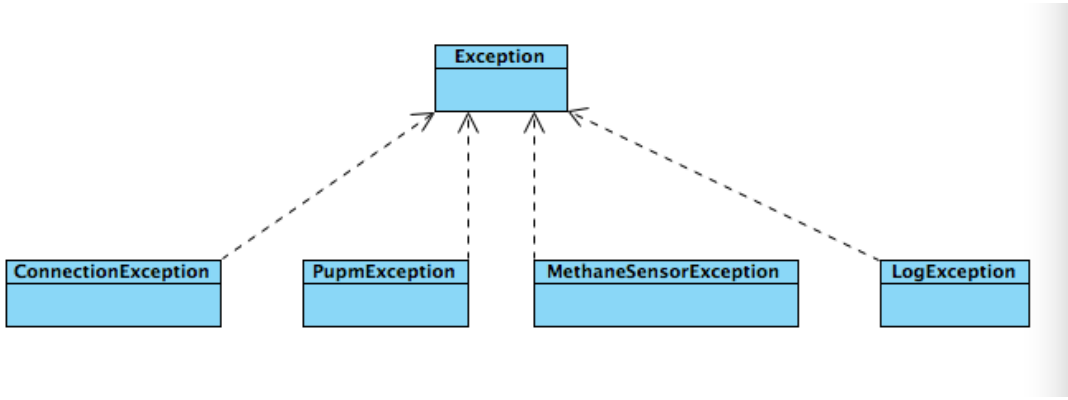
Ping Monitor Class Diagram



Ping Monitor Sequence Diagram

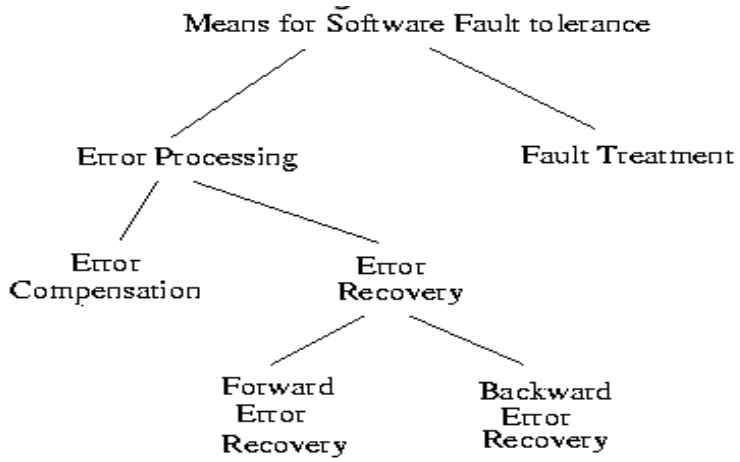


Exception Class Hierarchy



These designs should be generic. We could use inheritance and polymorphism to realize our requirements.

2.2 Fault Recovery



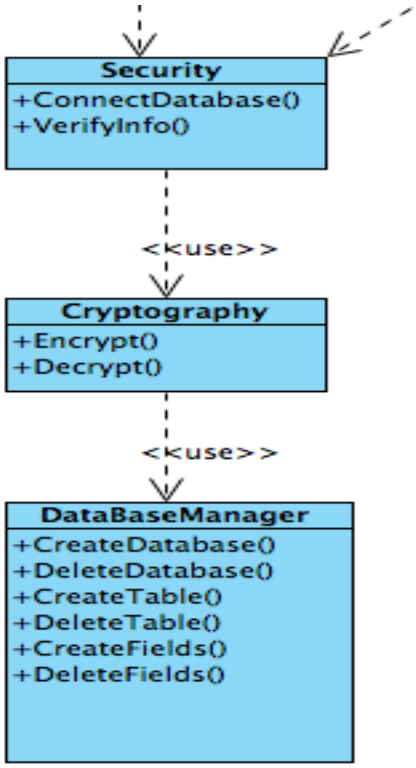
There are two strategies for software fault tolerance - *error processing* and *fault treatment*. Error processing aims to remove errors from the software state and can be implemented by substituting an error-free state in place of the erroneous state, called *error recovery*, or by compensating for the error by providing redundancy, called *error compensation*. Error recovery can be achieved by either forward or backward error recovery. The second strategy, fault treatment, aims to prevent activation of faults and so action is taken before the error creeps in. The two steps in this strategy are fault diagnosis and fault passivation.

2.3 Performance and Concurrency
Sensor Input: Medium
Alarm Triggering: High
Pump Actions: High
User Input: Low

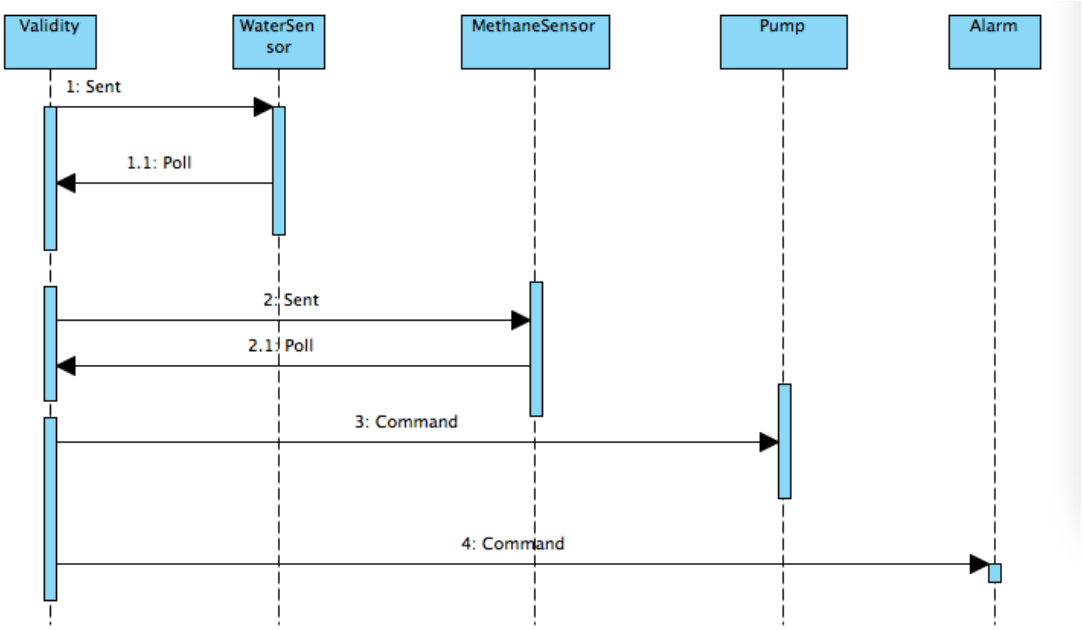
We could build a queue to process many high priority processes.

We can apply a scheduling algorithm to this system, arranging the incoming event. Base on the function and emergence level we can assign different event a relative priority.

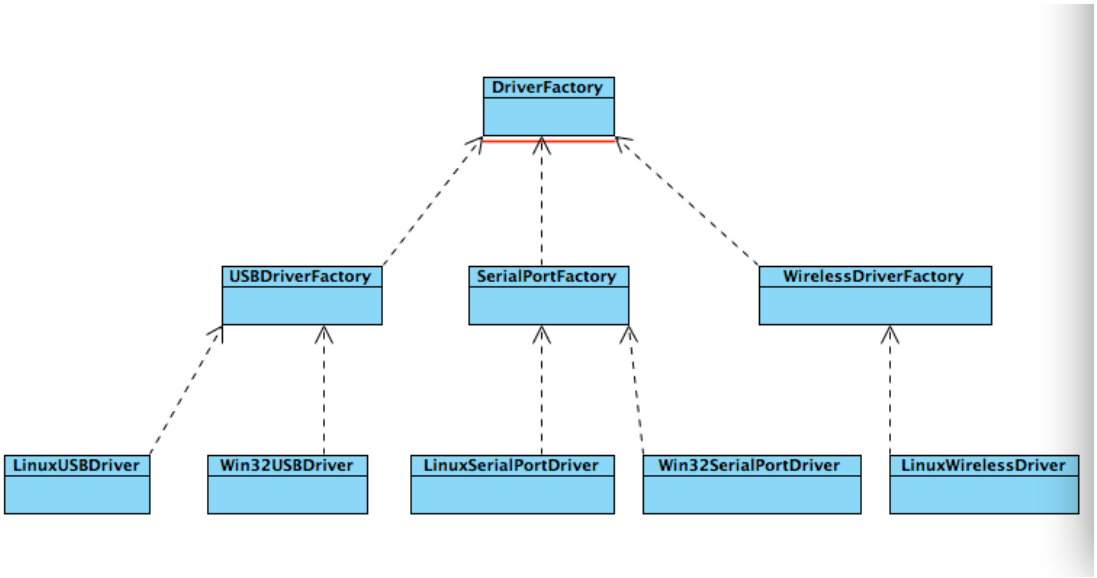
2.4 Security



2.5 Validity/Testability

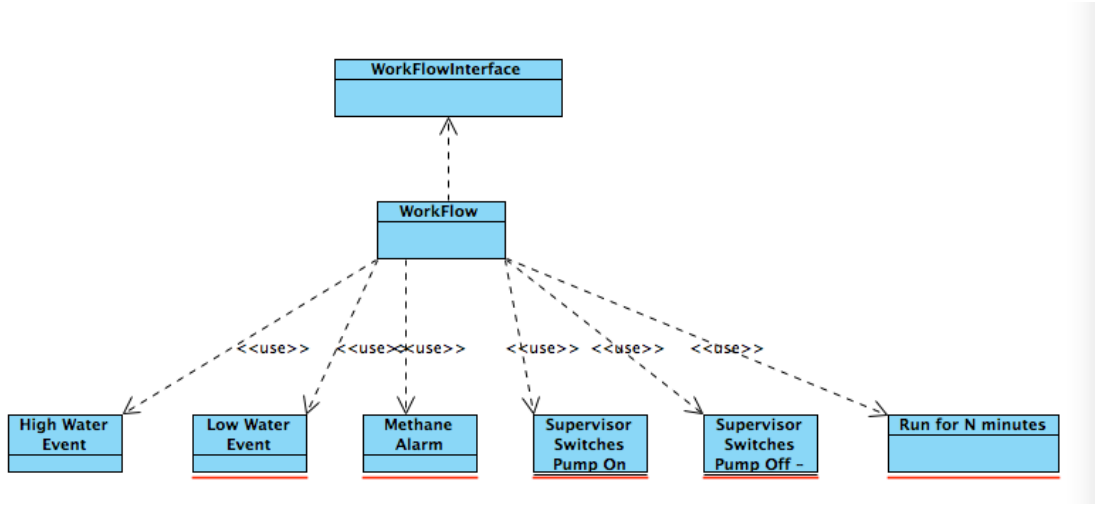


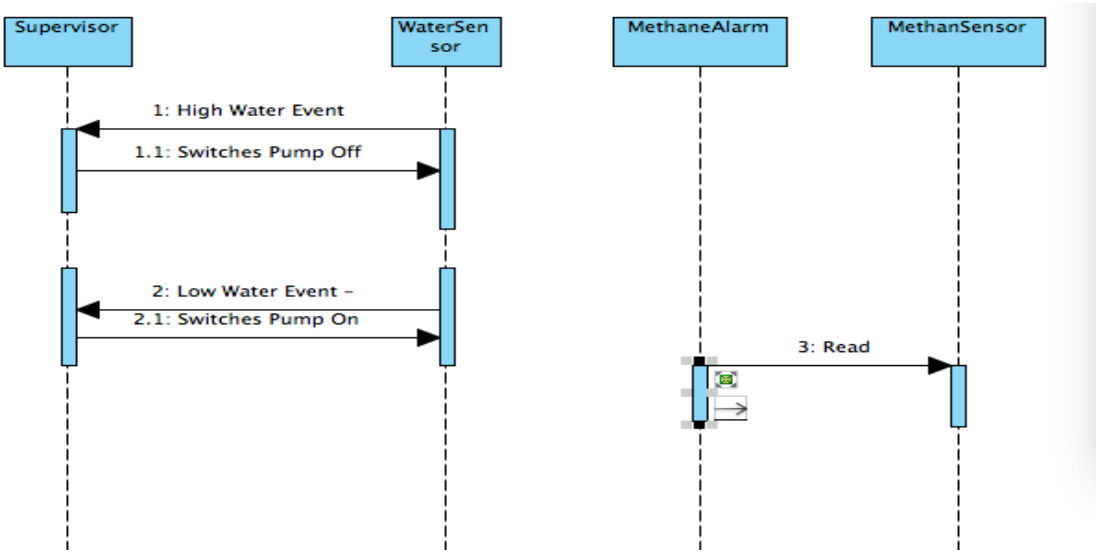
3. Abstract Factory Design Pattern (mine pump scenario only):



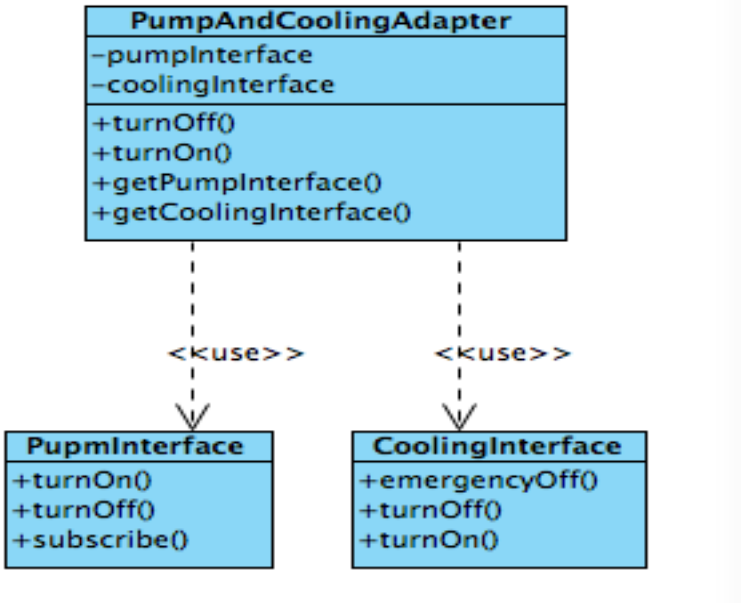
Singleton pattern is to assure only one and same instance of object every time.

4. Composite Design Pattern (mine pump scenario only):



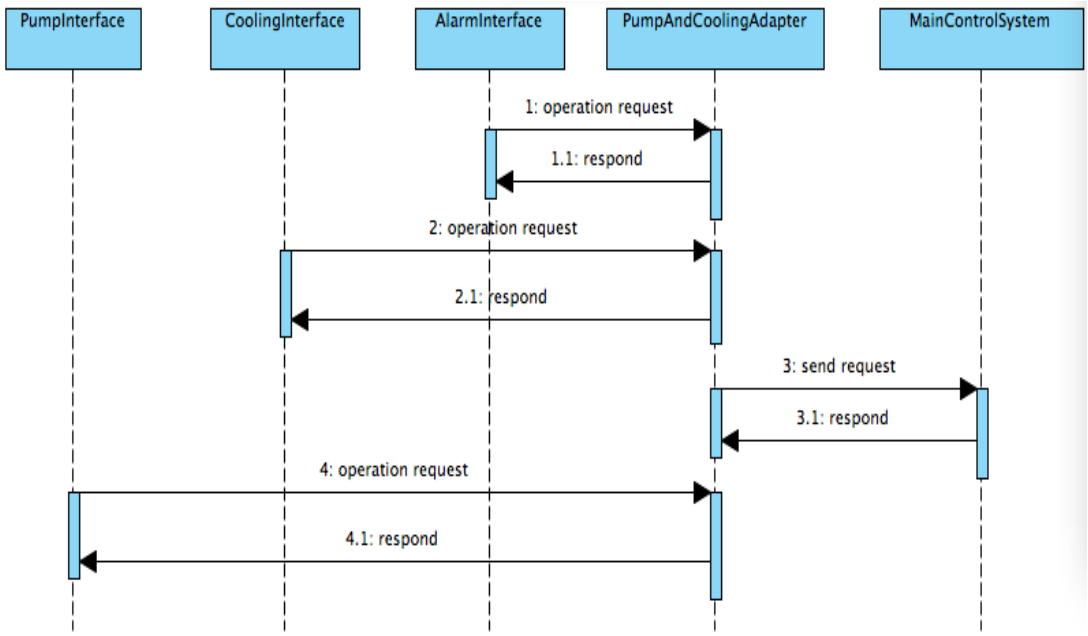


5. Adapter Design Pattern (mine pump scenario only):

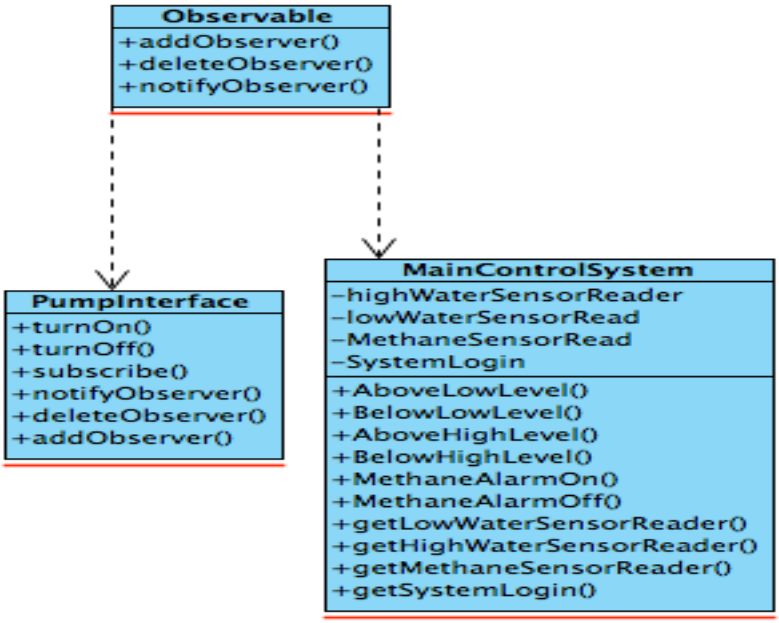


Clearly show above, I introduce a pump interface and cooling interface. PumpAndCoolingAdapter mainly responsible for implementing the function in the pump interface and cooling interface. Here we also design alarm interface to control the alarm for setting.

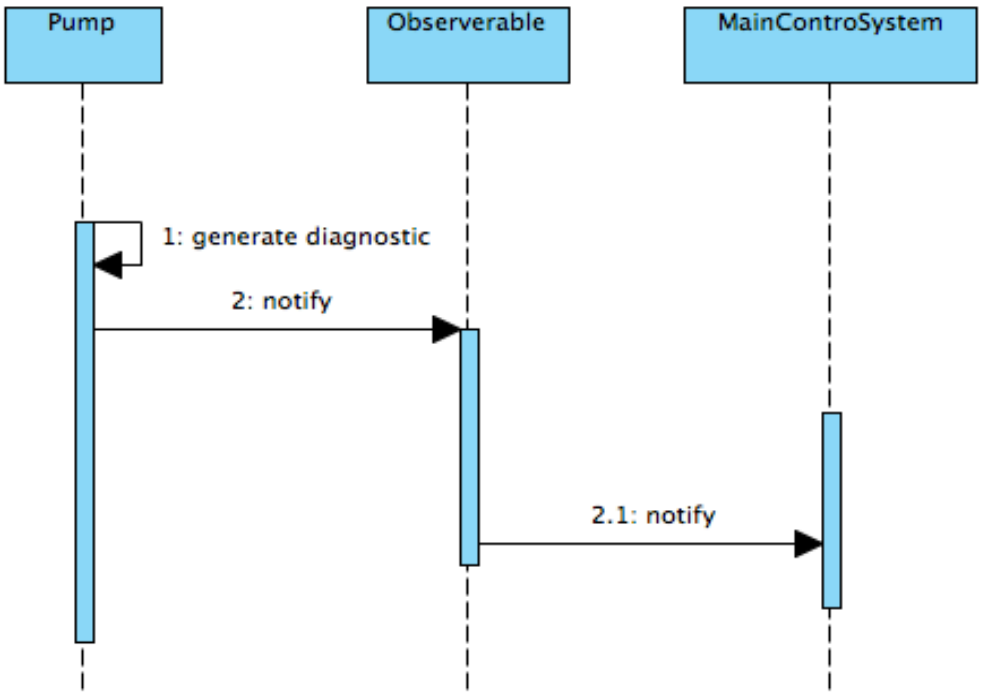
The following diagrams have show that how the operation and behavior exhibited when the methane alarm is activated.



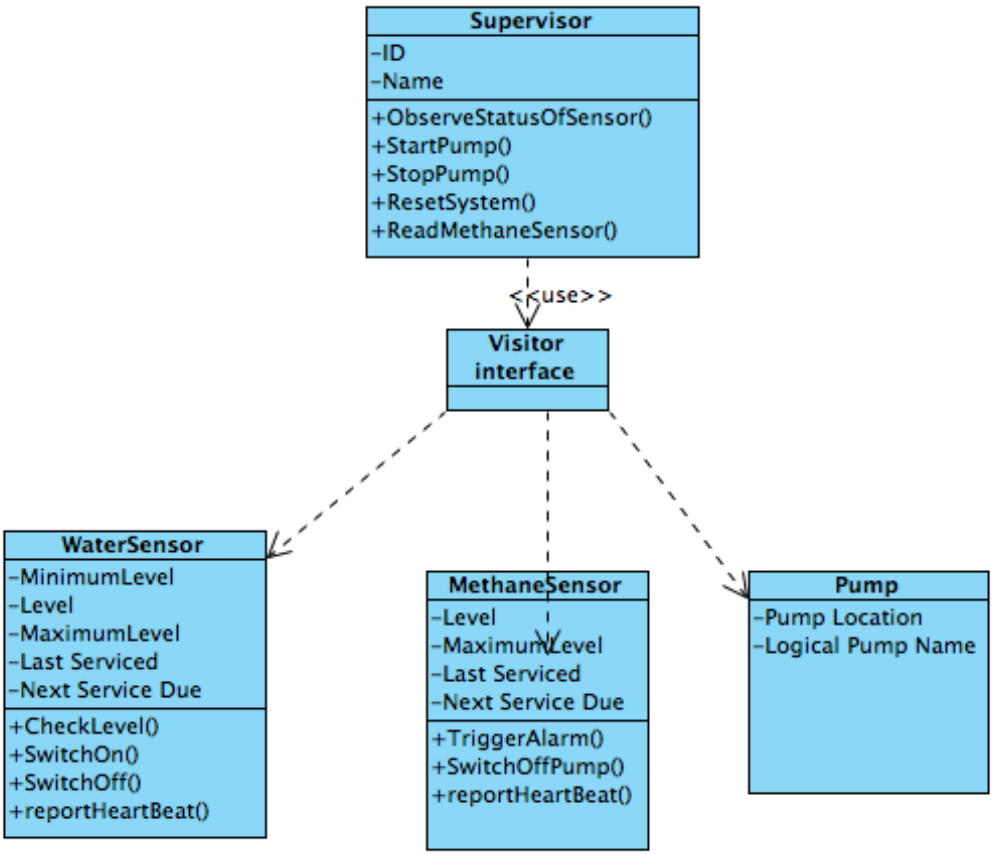
6. Observer Design Pattern (mine pump scenario only):



Here I design a component "Observable" that can receive the signal from the pump.



7. Visitor Design Pattern (mine pump scenario only):



8. Architectural Patterns:
The users and all components share the common resource in the system. We design an entitlement component to authorized the right for accessing this common resource.

9. J2EE Implementation Patterns:

A.Business Delegate:
Use a Business Delegate to reduce coupling between presentation-tier clients and business services. The Business Delegate hides the underlying implementation details of the business service, such as lookup and access details of the EJB architecture.

B. Service Locator:
Create a service locator that contains references to the services and that encapsulates the logic that locates them. In your classes, use the service locator to obtain service instances.
The Service Locator pattern does not describe how to instantiate the services. It describes a way to register services and locate them. Typically, the Service Locator pattern is combined with the Factory pattern and/or the Dependency Injection pattern. This combination allows a service locator to create instances of

services.

C. Session Façade:

Use a session bean as a facade to encapsulate the complexity of interactions between the business objects participating in a workflow. The Session Facade manages the business objects, and provides a uniform coarse-grained service access layer to clients.

The Session Facade abstracts the underlying business object interactions and provides a service layer that exposes only the required interfaces. Thus, it hides from the client's view the complex interactions between the participants. The Session Facade manages the interactions between the business data and business service objects that participate in the workflow, and it encapsulates the business logic associated with the requirements. Thus, the session bean (representing the Session Facade) manages the relationships between business objects. The session bean also manages the life cycle of these participants by creating, locating (looking up), modifying, and deleting them as required by the workflow.

It is important to examine the relationship between business objects. Some relationships between business objects are transient, which means that the relationship is applicable to only that interaction or scenario. Other relationships may be more permanent. Transient relationships are best modeled as workflow in a facade, where the facade manages the relationships between the business objects. Permanent relationships between two business objects should be studied to determine which business object (if not both objects) maintains the relationship.

D. Intercepting Filter:

Use an Intercepting Filter as a pluggable filter to pre and postprocess requests and responses. A filter manager combines loosely coupled filters in a chain, delegating control to the appropriate filter. In this way, you can add, remove, and combine these filters in various ways without changing existing code.

We are able, in effect, to decorate our main processing with a variety of common services, such as security, logging, debugging, and so forth. These filters are components that are independent of the main application code, and they may be added or removed declaratively. For example, a deployment configuration file may be modified to set up a chain of filters. The same configuration file might include a mapping of specific URLs to this filter chain. When a client requests a resource that matches this configured URL mapping, the filters in the chain are each processed in order before the requested target resource is invoked.

E. Fast Lane Reader:

• This pattern is good for faster and more efficient data retrieval only.

• There is a risk of the data being stale and hence this pattern should not be used

when the data being accessed changes rapidly or when the tolerance for slightly stale data is very small.

- Because this pattern bypasses the EJB model and provides direct access for persistent data, the design of the application may become complex for large-scale applications.
- As the pattern name suggests, this pattern is ideal for data read. For updating the data, this pattern should not be used as transactional access is bypassed.