SOFTWARE ENGINEERING

COMPONENT LEVEL DESIGN

Component level design is the definition and design of components and modules after the architectural design phase. Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each component for the system development.

A complete set of software components is defined during architectural design. But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code. Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each component.

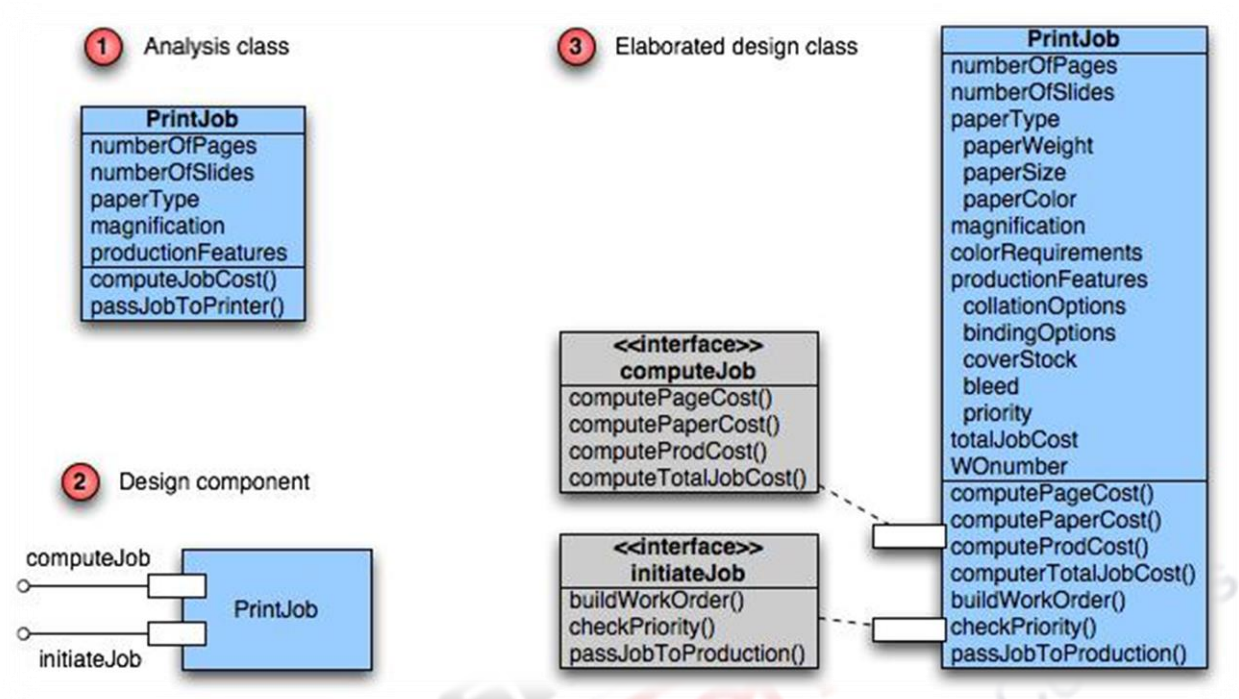
According to OMG UML specification component is expressed as, “A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”

Component Views

- OO View – A component is a set of collaborating classes.
- Conventional View – A component is a functional element of a program that incorporates processing logic, the internal data structures required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

CLASS ELABORATION

Class elaboration focuses on providing a detailed description of attributes, interfaces and methods before the development of the system activities. The following example provides a elaboration design class for “PrintJob”, the elaborated design class provides a detail description of the attributes, interfaces and the operations of the class.



DESIGN PRINCIPLES

The design principles for component level design comprises of the following:

- Design by Contract
- Open-Closed Principle
- Subtype Substitution
- Depend on Abstractions
- Interface Segregation

Design by Contract

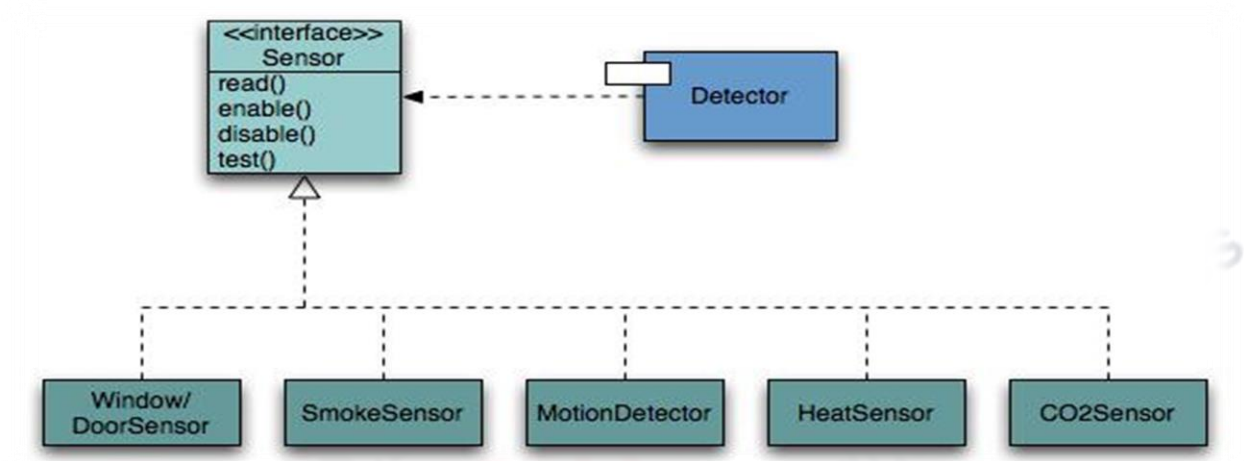
The relationship between a class and its clients can be viewed as a formal agreement, expressing each party's rights and obligations. Consider the following list operation:

public Item remove(int index)

It requires that the specified index is in the range (0 ≤ index < size( )) and ensures that the element at the specified position in this list is removed, subsequent elements are shifted to the left (1 is subtracted from their indices), and the element that was removed is returned.

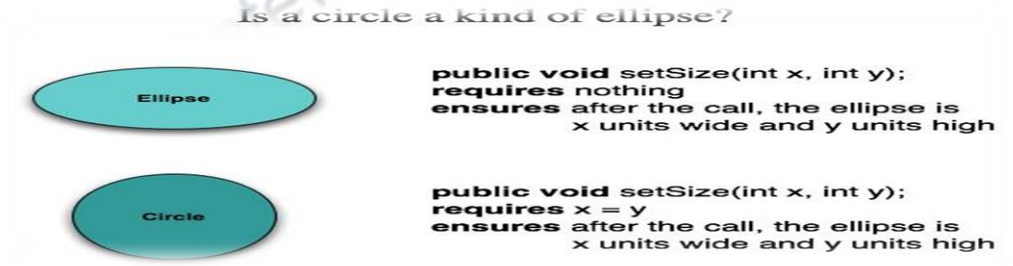
Open-Closed Principle

A module should be open for extension but closed for modification. There should always room for extension without modifying the module. The following module depicts a sensor module which performs the operations – read (), enable(), disable(), test(). This module can be extended for sensing smoke, motion, heat, CO_2 etc.



Substitutability

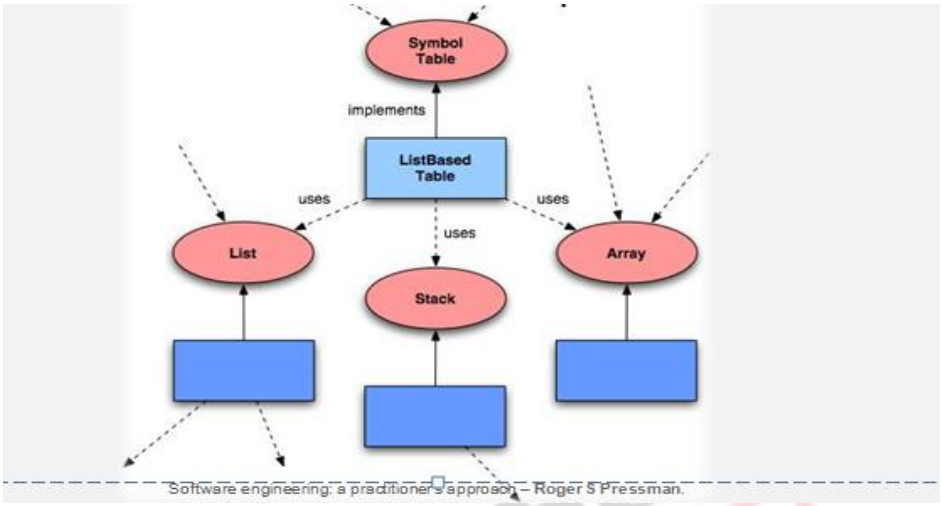
Subclasses should be substitutable for base classes. The subclasses derived from the base class should extend the functionalities of the base class without replacing the functionalities of the derived class.



Dependency Inversion

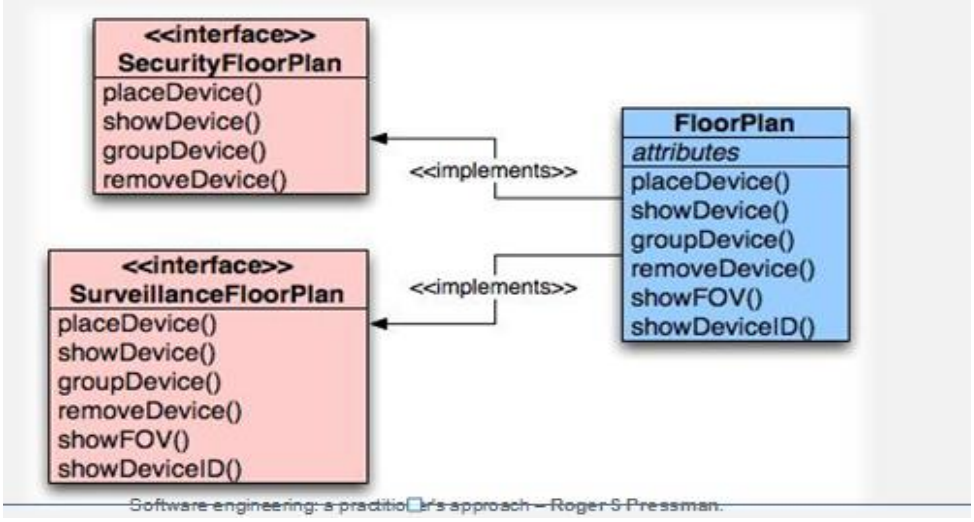
It depends on abstractions and not on concretions. Abstraction should not depend on details, as details should depend on abstraction. It's a bad design principle to allow the high level

or complex modules to majorly depend on low level classes. Here in the example listBasedTable implements SymbolTable and uses the operations such as list, stack and array.



Interface Segregation

Many client-specific interfaces are better than one general purpose interface. For consistency purpose, interfaces should flow from the left-hand side of the component box. The interfaces only that are relevant to the component under development must be depicted.



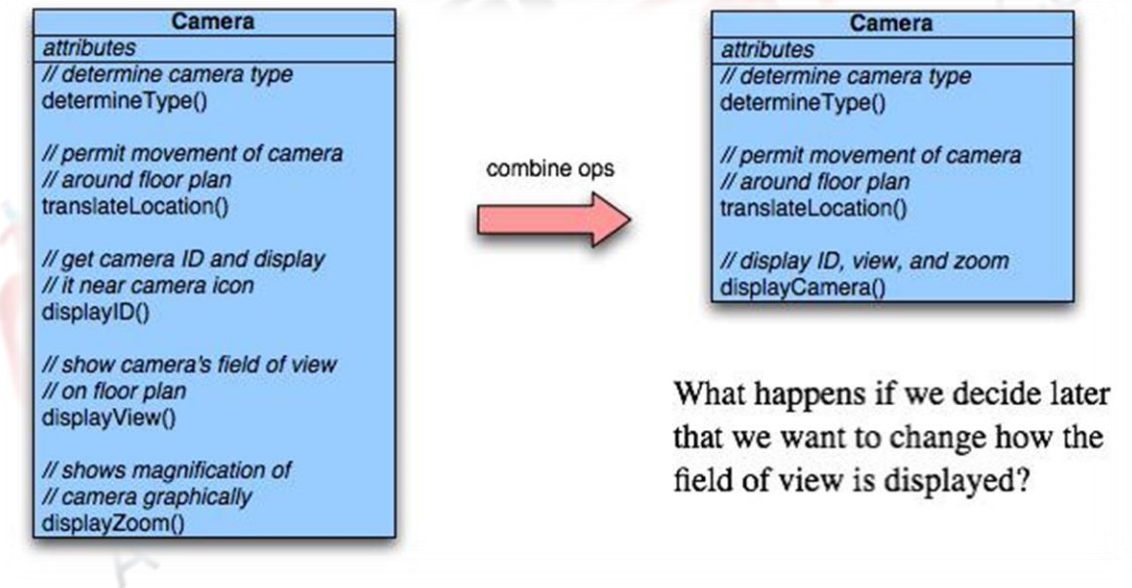
Cohesion

The “single-mindedness” of a module can be given as cohesion. Cohesion implies that a single component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself. Examples of cohesion

- Functional
- Layer
- Communicational

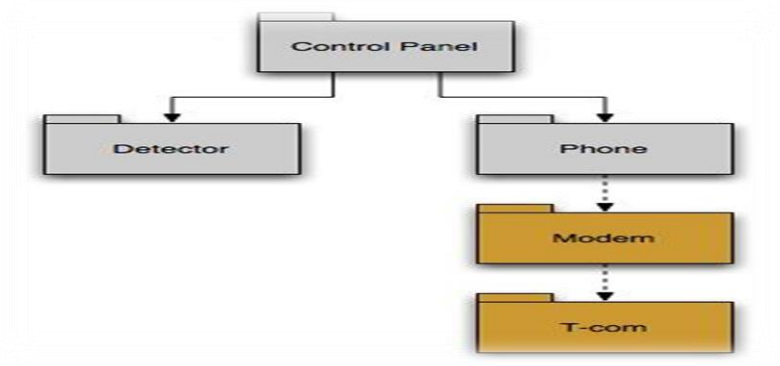
Functional Cohesion

Typically applies to operations where a module performs one and only one computation and then returns a result.



Layer Cohesion

Layer Cohesion applies to packages, components, and classes. It occurs when a higher layer can access a lower layer, but lower layers do not access higher layers. Control Panel provides a good example for layer cohesion, where the higher layer access the lower but the lower layers cant access higher layers.



Communicational Cohesion

All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it. Example: A StudentRecord class that adds, removes, updates, and accesses various fields of a student record for client components.

Coupling

Coupling provides a qualitative measure of the degree to which classes or components are connected to each other.

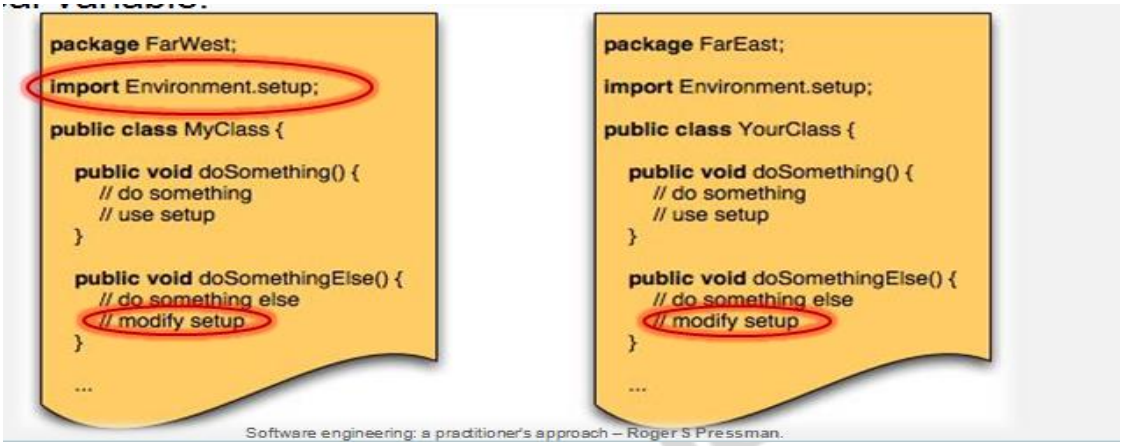
- Avoid
  - Content coupling
- Use caution
  - Common coupling
- Be aware
  - Routine call coupling
  - Type use coupling
  - Inclusion or import coupling

Content Coupling

It occurs when one component "surreptitiously modifies data that is internal to another component". It violates information hiding.

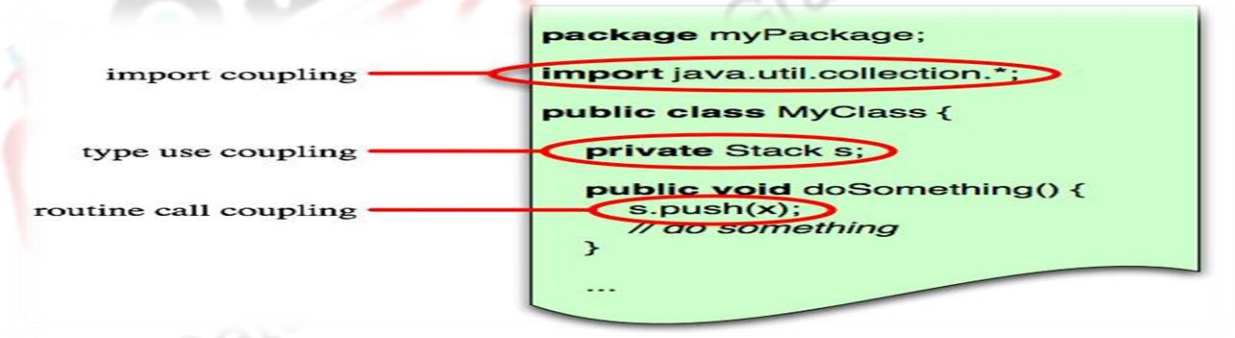
Common Coupling

It occurs when a number of components all make use of a global variable. In the example, the components make use of a global variable “setup”.



Routine Coupling

When certain types of coupling occur routinely in object-oriented programming they are considered as routine coupling.



COMPONENT-LEVEL DESIGN

Component level design is the definition and design of components and modules after the architectural design phase. Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each component for the system development. Component level design includes the following actions:

1. Identify Design Classes in Problem Domain
2. Identify Infrastructure Design Classes

3. Elaborate Design Classes
4. Describe Persistent Data Sources
5. Elaborate Behavioral Representations
6. Elaborate Deployment Diagrams
7. Refactor Design And Consider Alternatives

Steps 1 & 2 – Identify Classes

1. Most classes from the problem domain are analysis classes created as part of the analysis model
2. The infrastructure design classes are introduced as components during architectural design

Step 3 – Class Elaboration

a) Specify message details when classes or components collaborate

b) Identify appropriate interfaces for each component

c) Elaborate attributes and define data structures required to implement them

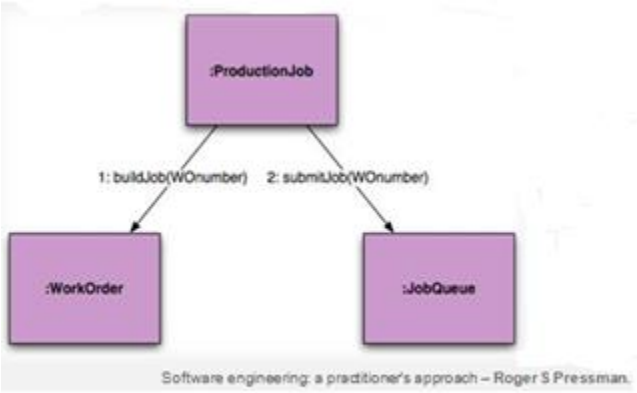
d) Describe processing flow within each operation in detail

3a. Collaboration Details

Messages can be elaborated by expanding their syntax in the following manner:

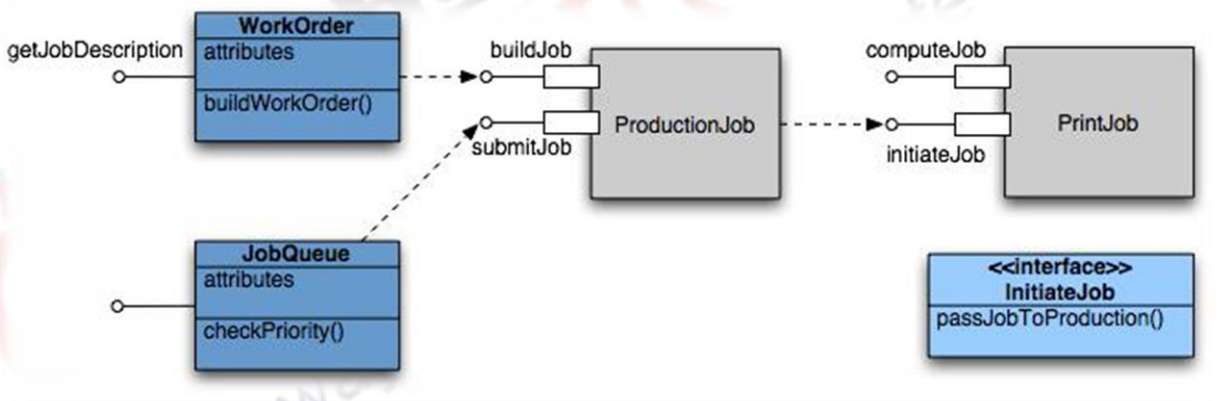
[guard condition] sequence expression (return value) := message name (argument list)

The following example of "ProductionJob" depicts the collaboration details for job production module.



3b. Appropriate Interfaces

PrintJob interface "initiateJob", which does not exhibit sufficient cohesion because it performs three different sub functions – refactoring can be performed.



3c. Elaborate Attributes

Analysis classes will typically only list names of general attributes (ex. paperType). All the attributes during component design are listed. UML syntax:

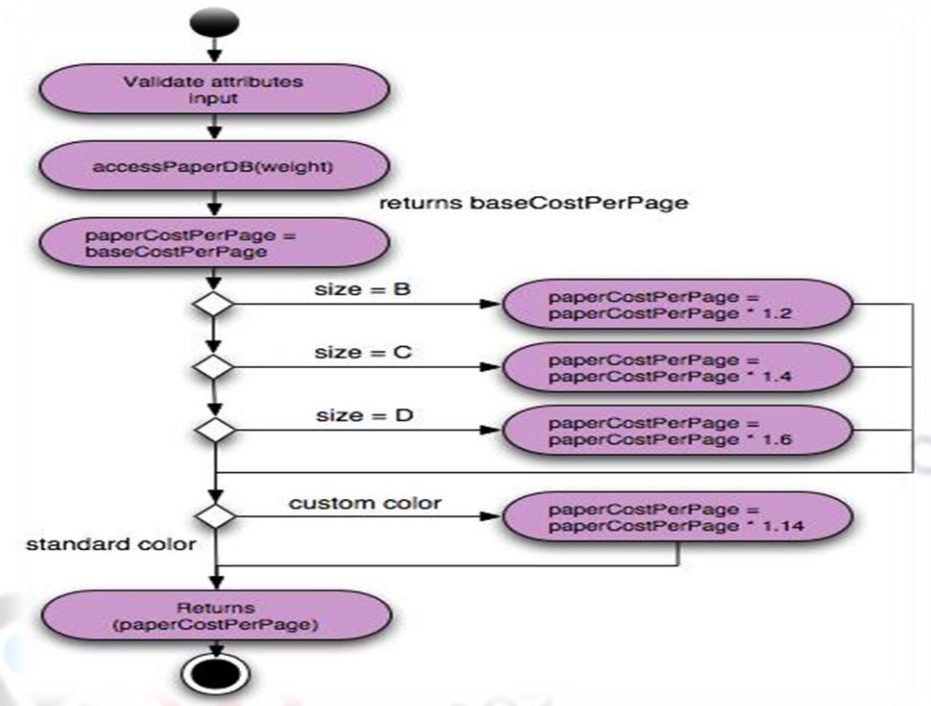
name : type-expression = initial-value { property string }

For example, paperType can be broken into weight, size, and color. The weight attribute would be:

paperType-weight: string = "A" { contains 1 of 4 values – A, B, C, or D }

3d. Describe Processing Flow

The process follow is described in detail using activity diagram. The activity diagram for computePaperCost( ) is shown below which provides a detail description of the modules.



Step 4 – Persistent Data

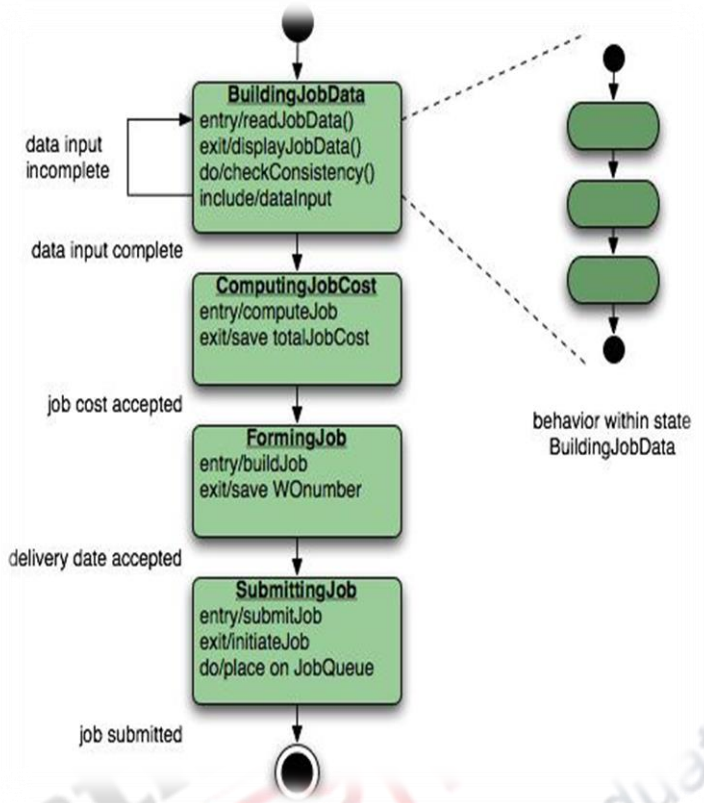
The persistent data sources are described (databases and files) and the classes required to manage them are identified.

Step 5 – Elaborate Behavior

It is sometimes necessary to model the behavior of a design class. Transitions from state to state have the form:

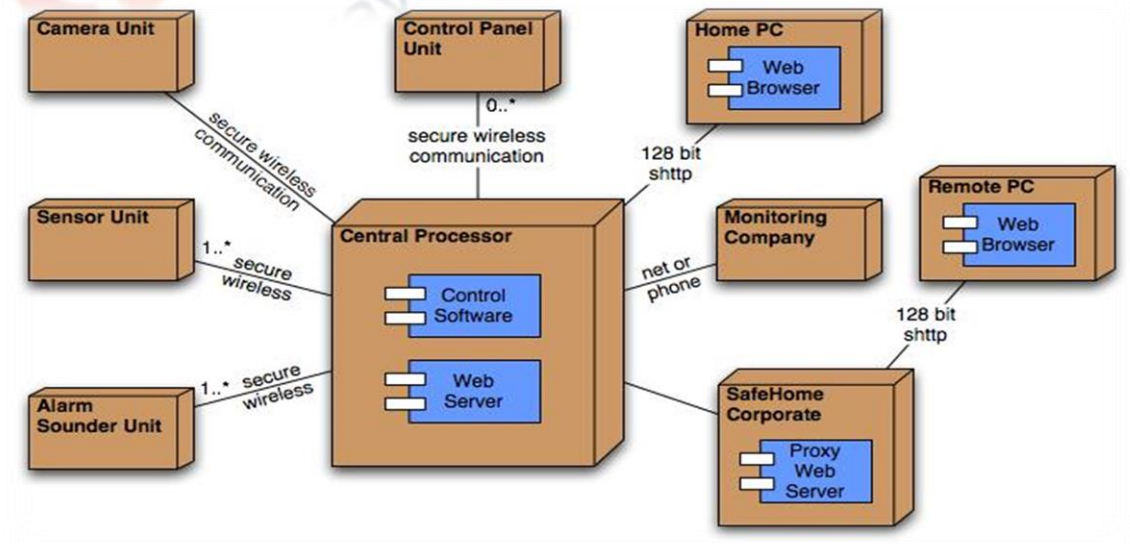
Event-name (parameter-list) [guard-condition] / action expression

The behavior of the system is elaborated using a state diagram to depict the transition of states during work flow. The following state diagram provides the transition of states for BuildingJobData.



Step 6 – Elaborate Deployment Diagrams

Deployment diagrams are elaborated to represent the location of key packages or components



Step 7 – Redesign/Reconsider

The first component-level model you create will not be as complete, consistent, or accurate as the n^{th} iteration you apply to the model. The best designers will consider many alternative design solutions before settling on the final design model.