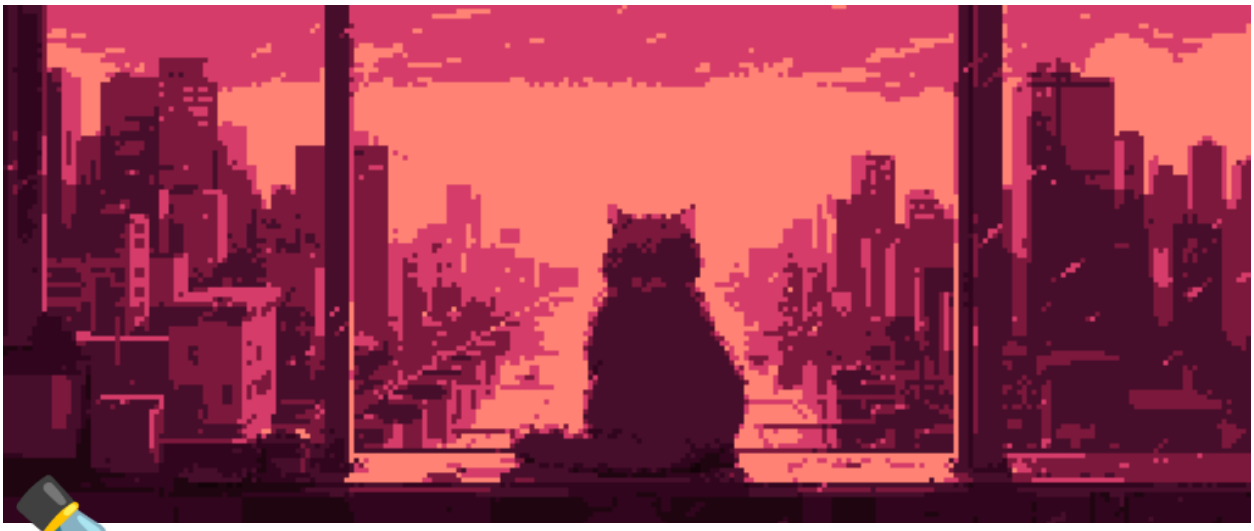


SDA cheatsheet

Created	@December 25, 2023 11:23 AM
Tags	Cheat Sheet SDA
Created by	Muhammad Ali Zahid

Key points

- if a function is static can be called without making an object using scope resolution. static functions can only use static data member.
- use **enum** to easily define mane const values
- Don’t use switch (violates OCP)

- The **Software Development Life Cycle (SDLC)** is a systematic process used by software developers to design, develop, test, and deploy high-quality software. The SDLC consists of a series of phases or stages that guide the development team through the entire software development process. Different models and methodologies exist for SDLC. It’s parts are

    - Requirement analysis
    - Planning
    - Design
    - Implementation
    - testing
    - Development
    - Maintaince
- Characteristics of Good Program

  1. Correctness
  2. Readability
  3. Modularity
  4. Efficiency
  5. Scalability
  6. Maintainability
  7. Portability
  8. Security
  9. Flexibility
  10. Testability
  11. Documentation
  12. User-Friendly Interface
  13.

Error Handling
14.
Version Control

- **Polymorphism(Dynamic binding)**
  - Compile time
    - Function overloading
    - Operator Overloading
  - Run time
    - Virtual Function
- Is-A Rule → Inheritance Relation
- Relation Ships b/w classes

> Association
>
> > Aggregation
> >
> > > Composition

  - Associations
    - relation ships b/w separate classes which is established through their objects (a class obj can access another class stuff)
    - one to one, one to many, many to one, many to many
    - Has a rule applicable . (a class has smth i.e. state has city)

```
#include <iostream>
#include <string>

class Student {
public:
```

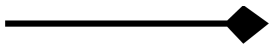
```
    Student(std::string name) : name(name) {}

    std::string getName() const {
        return name;
    }

private:
    std::string name;
};

class Course {
public:
    Course(std::string courseName) : courseName(courseName

    std::string getCourseName() const {
        return courseName;
    }

private:
    std::string courseName;
};

// Association between Student and Course
class Enrollment {
public:
    Enrollment(Student* student, Course* course) : student

    void displayEnrollment() const {
        std::cout << student->getName() << " is enrolled i
    }

private:
    Student* student;
    Course* course;
};
```

```
int main() {
    Student student("John Doe");
    Course course("C++ Programming");
    Enrollment enrollment(&student, &course);
    enrollment.displayEnrollment();

    return 0;
}
```

- Composition
  - Composed objects become part of Composer
  - Child object does not have their lifecycle
  - Child object purpose is served with its relation to the parent object
  - If the parent object is deleted, Child (composed) objects are also deleted
  - i.e tree is composed of wood, if tree is gone so is the wood
  - the arrow head is towards the parent object

```
#include <iostream>
#include <string>

class Engine {
public:
    Engine(std::string type) : type(type) {}

    std::string getType() const {
        return type;
    }

private:
    std::string type;
};

class Car {
```

```
public:
    Car(std::string model, Engine* engine) : model(model),

    void displayCarInfo() const {
        std::cout << "Model: " << model << ", Engine Type
    }

private:
    std::string model;
    Engine* carEngine;  // Composition - Car "has a" Engi
};

int main() {
    Engine carEngine("V8");
    Car myCar("SportsCar", &carEngine);
    myCar.displayCarInfo();

    return 0;
}
```



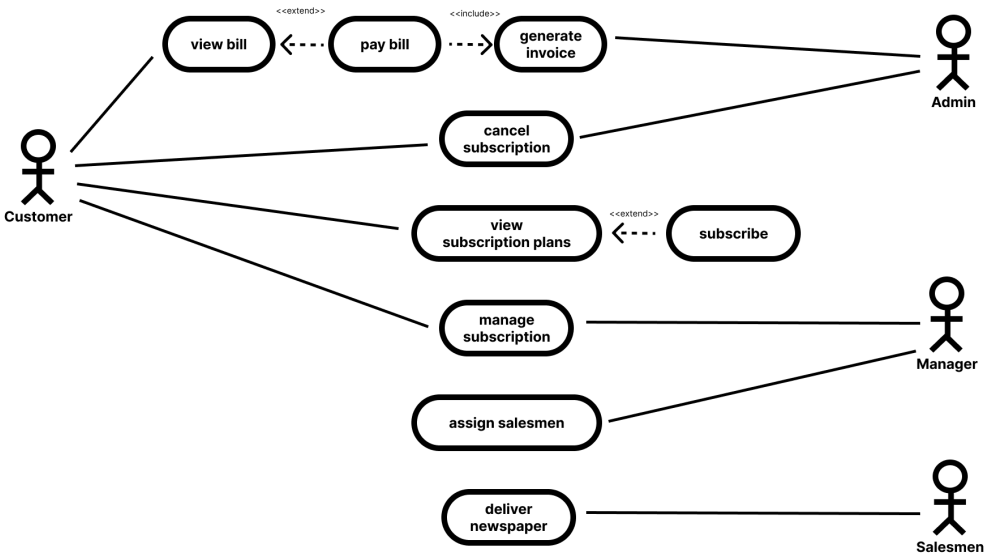
- Aggregation
  - child have their own life cycle but the parent has ownership
  - child have their own independent purpose that they can serve
  - but after the parent dies, the child can not belong to other objects
  - i.e room has bed. if room is demolished the bed is still serving its purpose

```
#include <iostream>
#include <string>
#include <vector>

class Book {
```

```
public:
    Book(std::string title) : title(title) {}

    std::string getTitle() const {
        return title;
    }

private:
    std::string title;
};

class Library {
public:
    Library(std::string name) : name(name) {}

    void addBook(Book* book) {
        books.push_back(book);
    }

    void displayBooks() const {
        std::cout << "Books in library '" << name << "':"
        for (const auto& book : books) {
            std::cout << "- " << book->getTitle() << std:
        }
    }

private:
    std::string name;
    std::vector<Book*> books;  // Aggregation - Library "H
};

int main() {
    Book book1("The C++ Programming Language");
    Book book2("Design Patterns: Elements of Reusable Obje

    Library myLibrary("Tech Library");
```

```
    myLibrary.addBook(&book1);
    myLibrary.addBook(&book2);

    myLibrary.displayBooks();

    return 0;
}
```

- Use Case Analysis
  - Use Case Diagram
    - gives a high level view of the system without going into the implementation detail
    - classifies the actor in to specific classes
    - Extends = not compulsory | Extra Func
    - includes = Compulsory



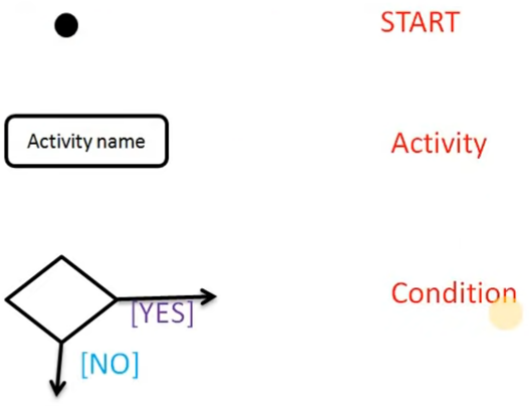
  - Use Case Description

- describe the program flow

Use case : Record a loan
System : Library management system
User : Clerk, Librarian
Precondition : User has logged in
1- User scans barcode of book // usually user initiates a task etc
2- System displays details of the book
3- User scans membership card
4- ~~Use~~ System displays info about member
5- User selects 'Issue Book'
6- System saves the loan & displays a due date
7- User stamps date onto book

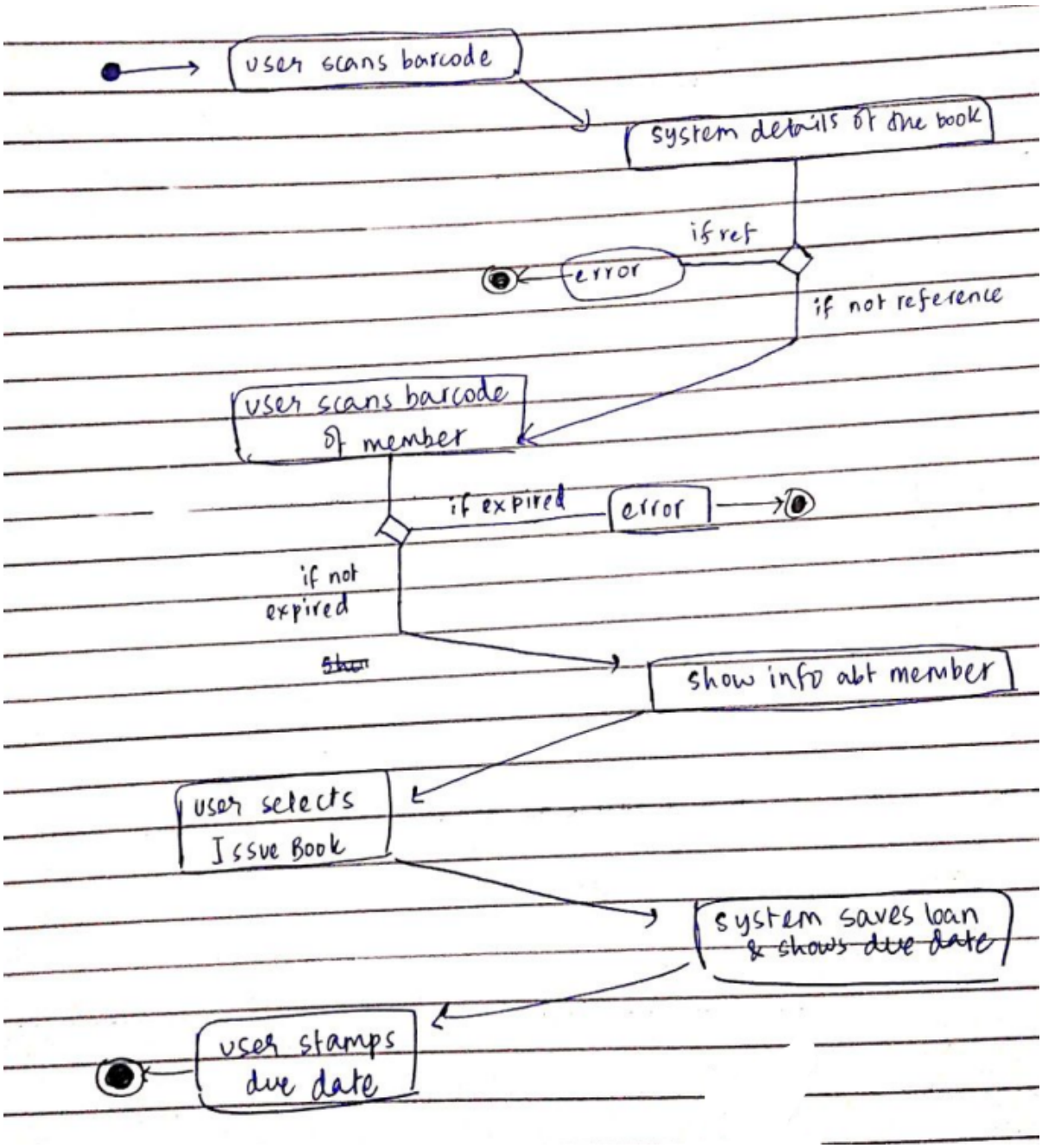
Alternate paths
2B: If the book belongs to the reference section then show an appropriate error msg & halt transaction
4B- If membership expired, system shall stop execution

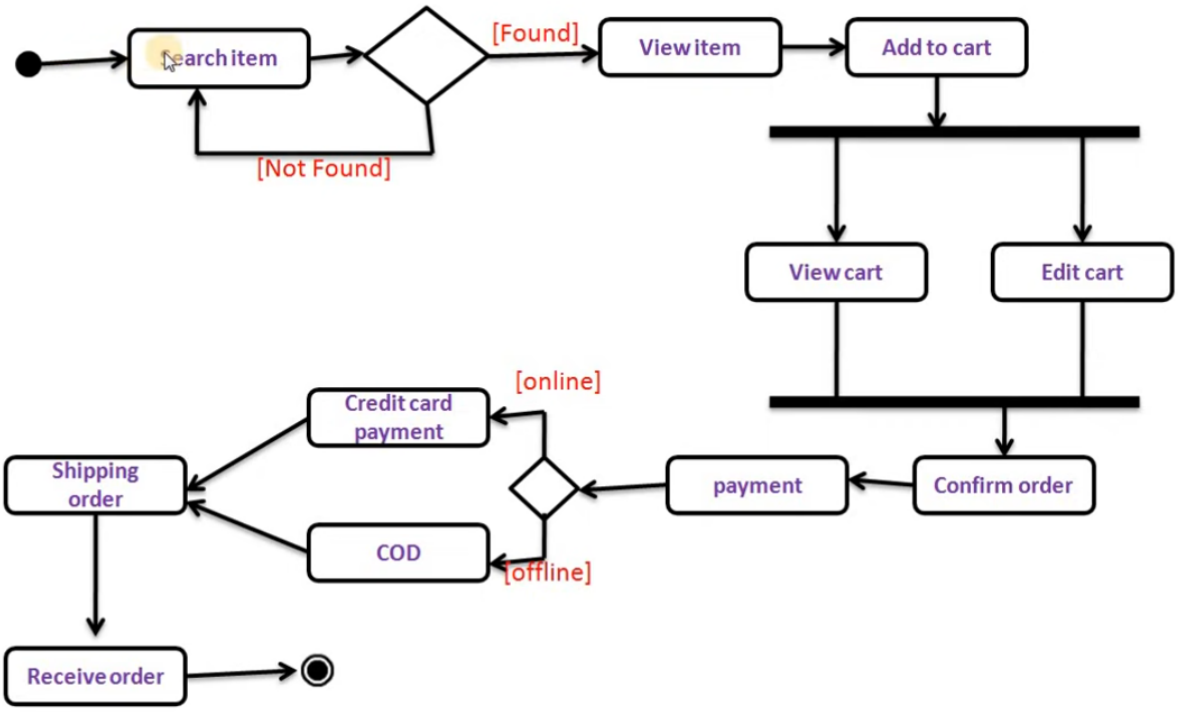
- Activiti Diagram

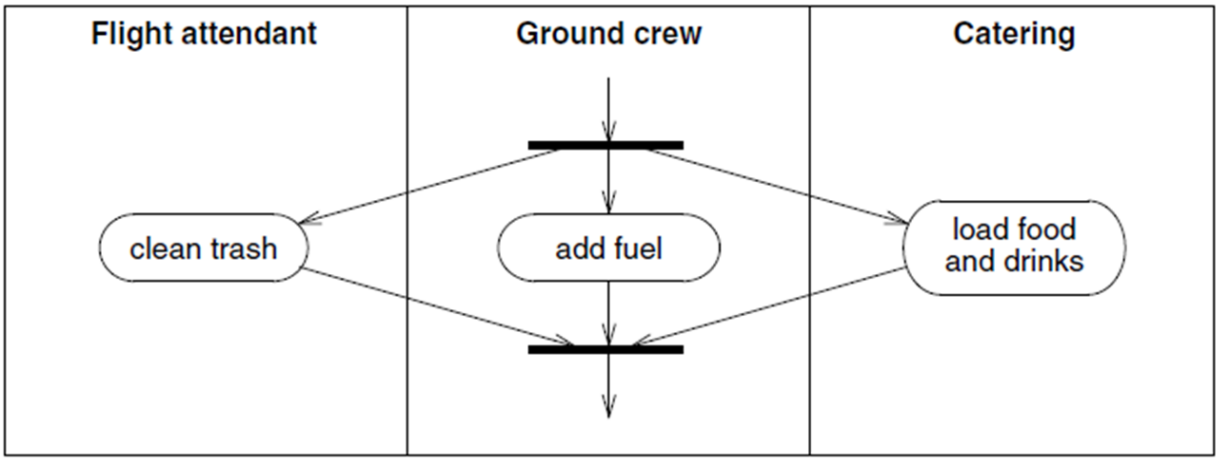


Parallel activity

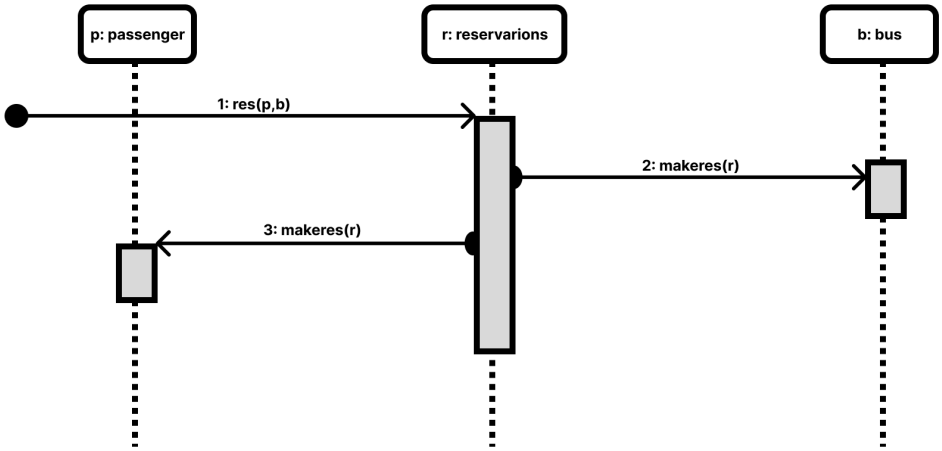
END

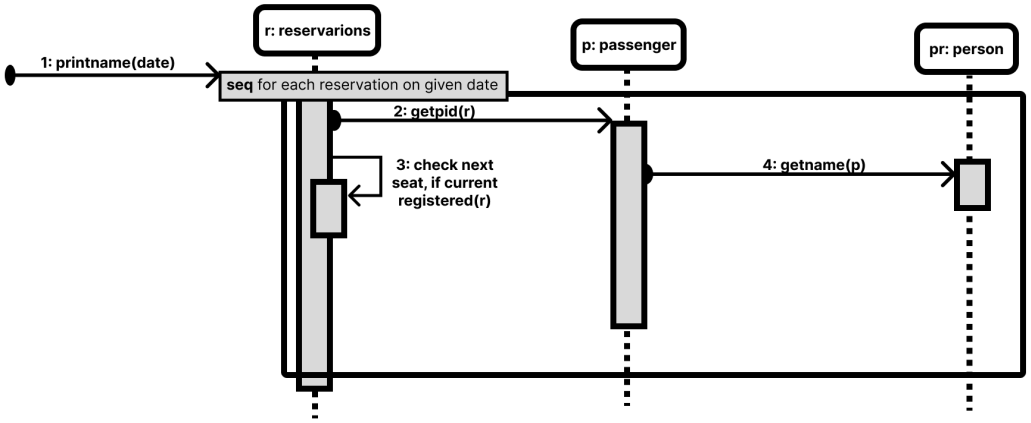






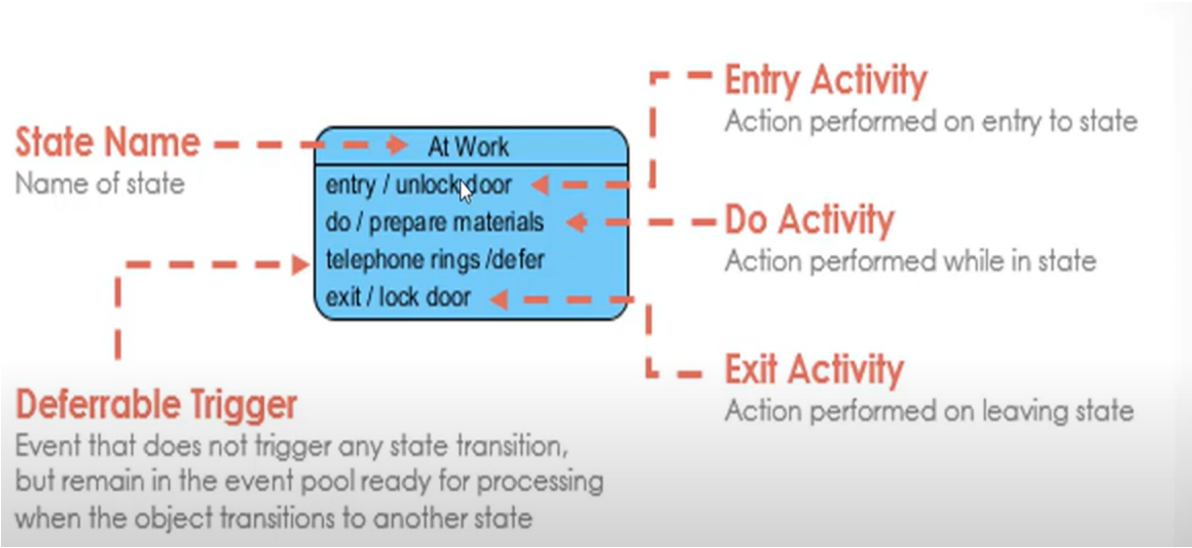
- **Sequence Diagram**
  - Shows the object Interactions required to execute a user case
  - Shows a dynamic view of the systems (function calls)
  - the dotted line shows the timeline, time flows from to up to down

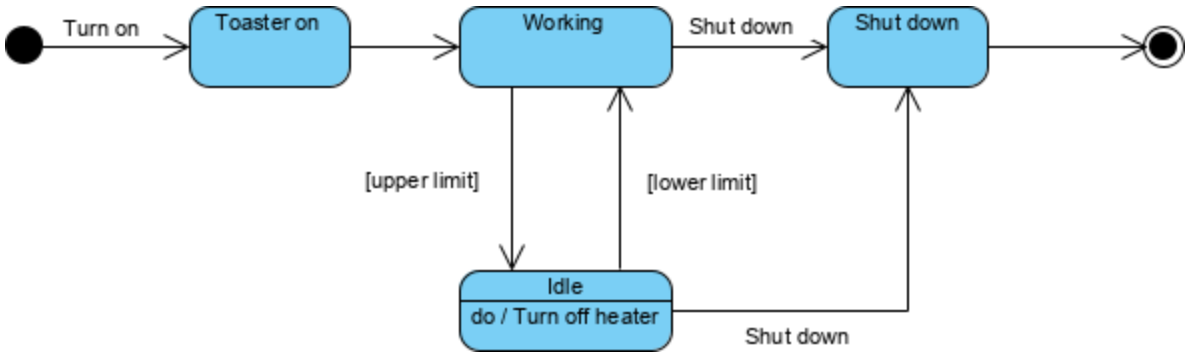




- State Diagram
  - specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events
  - transitions must have label
  - The UML notation for a change event is the keyword *when* followed by a parenthesized Boolean expression

- UML notation for an absolute time is the keyword *when* followed by a parenthesized expression involving time.
- The notation for a **time interval** is the keyword *after* followed by a parenthesized expression that evaluates to a time duration.
- States often correspond to **verbs** with a suffix of “ing” (Waiting, Washing) or the duration of some **condition** (Armed, Purchased).
- A ***guard condition*** is a Boolean expression that must be true in order for a transition to occur.
- **For example**, a traffic light at an intersection may change only if a road has cars waiting.
- A ***do-activity*** is an activity that continues for an **extended time**.

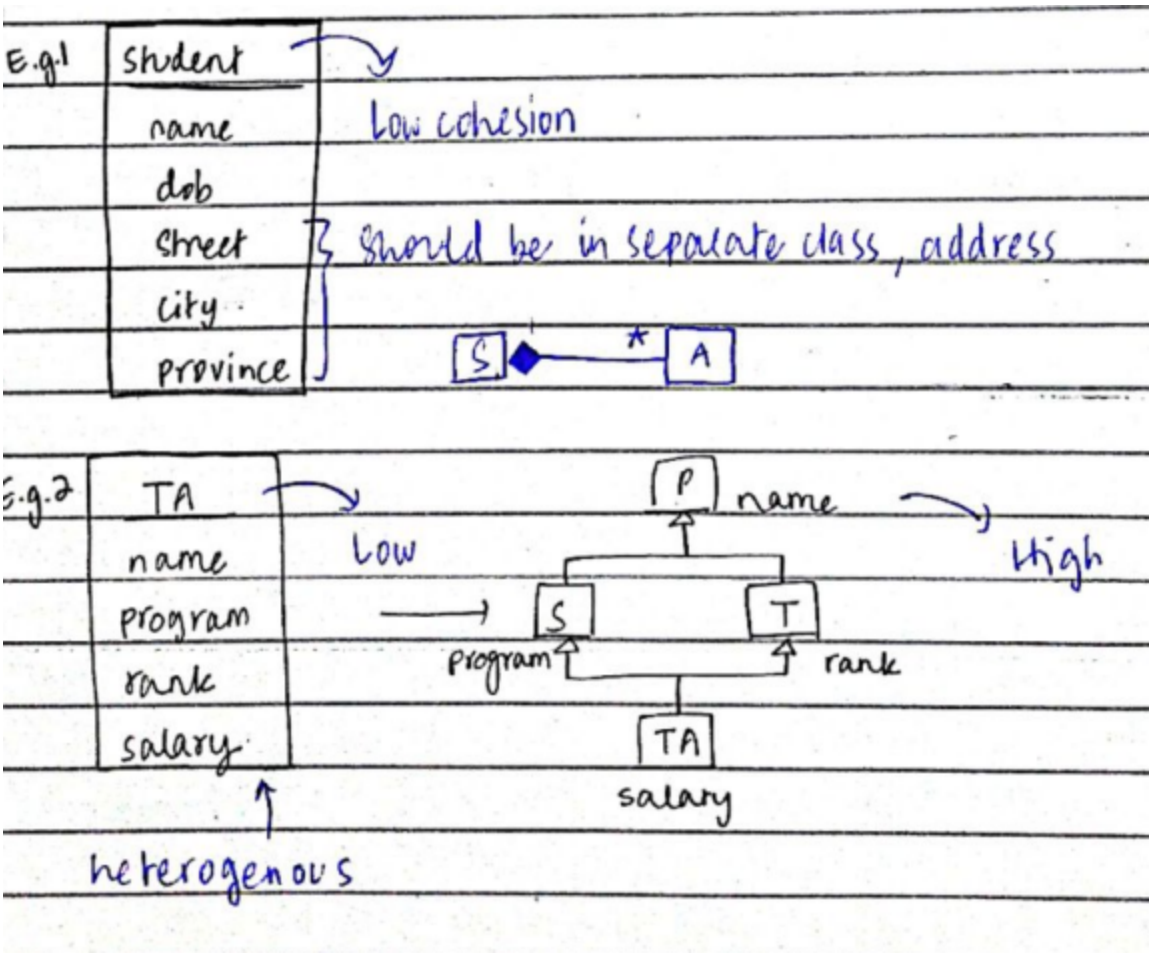


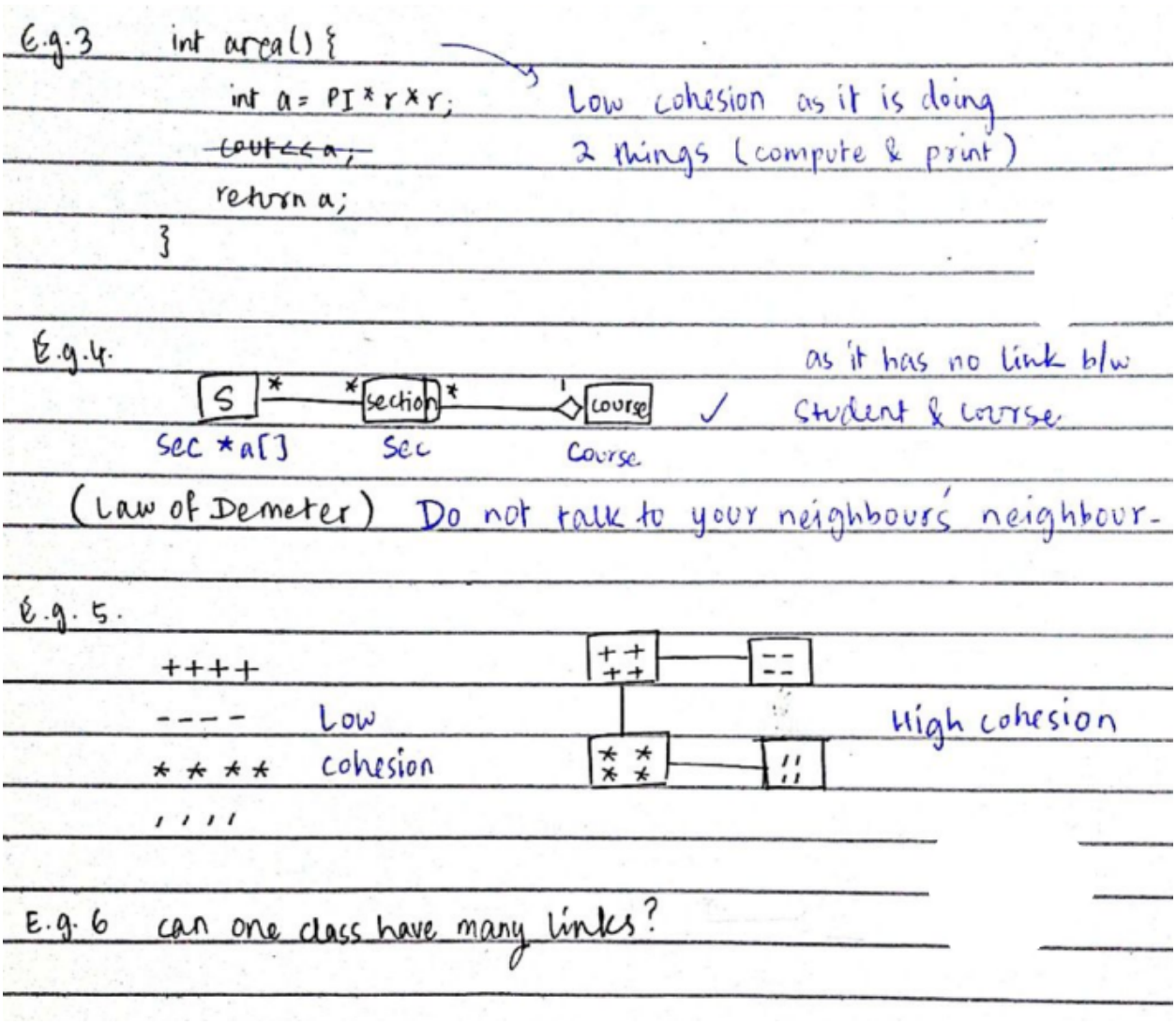


- Interfaces

    - idea: different UI for different classes in accordance to the actions the can perform
    - used to prevent complications
- SOLID design principles
    - **Single-responsibility principle:** A class should have only one responsibility and one reason to change.
    - **Open-closed principle:** Software entities should be open for extension, but closed for modification. We can add features without having to change existing code.
    - **Liskov's substitution principle:** Subtypes(child) should be substitutable for their base(parent) types without breaking the program logic.
    - **Interface segregation principle**: Clients should depend on interfaces that they use only. A specific interface for each user type.
    - **Dependency inversion principle:** High-level modules should depend on abstractions, not on low-level details. If a class is using another class that changes frequently, than invert the dependency using an interface
        - A → B then A → I ← B where I is the interface
- Cohesion
    - modules should have high cohesion
    - refers to the degree to which the elements within a module work together to fulfill a single, well-defined purpose.
    - A module with high cohesion means that its elements are closely related and focused on a single task, while a module with low cohesion means that its elements are loosely related and serve multiple purposes.
- Coupling
    - modules should have low coupling
    - refers to the degree of interdependence between the modules.
    - A module with high coupling means that it is closely connected to other modules and changes in one module may affect other modules. A module with low

coupling means that it is independent of other modules and changes in one module have little impact on other modules.





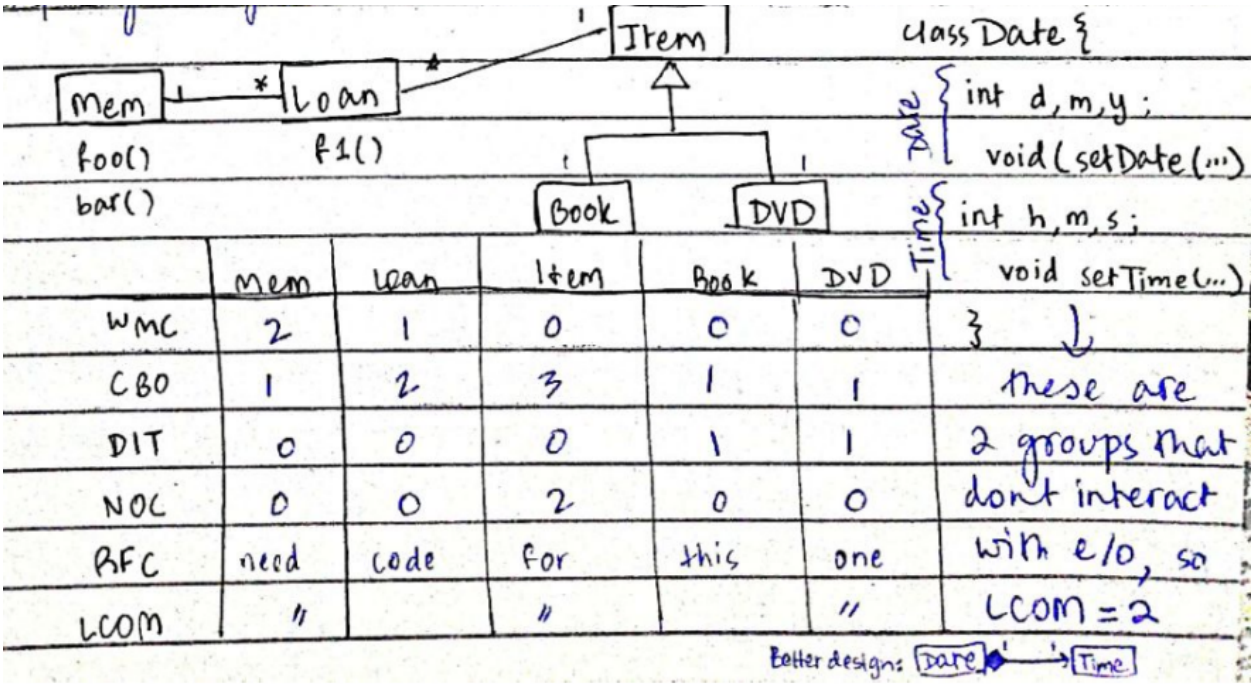
- Singleton
  - a class whose objects can be made once
  - memory optimization
  - all objects created will point to one object

```
class Singleton {
  private:
          static Singleton *ins;
          Singleton(){}
  public:
          static Singleton* getIns(){
```

```
                if(ins == NULL )
                          ins=New Singleton();
                return ins;
            }
};
```

- **2 tier Architecture**
  - **Client Layer**(interface & business layer)
  - **DB layer**
  - communicates through APIs directli
  - disadvantage
    - scalability
    - security
  - Advantage
    - maintenance
    - data buffer
- **3 tier Architecture**
  - **Client Layer**(interface)
  - **Application Layer**(business layer)
  - **DB Layer**
  - client communicates with application and then application communicates with DB
  - Advantages
    - scalability
    - security
  - Disadvantage
    - maintenance

- **Object Oriented Metrics**
  - Weighted methods per class = number of functions in class
    - desirable = low
  - Coupling between objects = number of pointers in class
    - desirable = low
  - Depth of inheritance tree
    - desirable = low-intermediate
  - Number of Childs
    - desirable = low-intermediate
  - Response for a class = number of other functions a function calls
    - desirable = low-intermediate
    - R = M + A
  - Lack of cohesion of Methods
    - desirable = low-intermediate



	Mem	Loan	Item	Book	DVD
WMC	2	1	0	0	0
CBO	1	2	3	1	1
DIT	0	0	0	1	1
NOC	0	0	2	0	0
RFC	need	code	for	this	one
LCOM	"		"		"

these are 2 groups that dont interact with e/o, so LCOM = 2

- **Design Patterns**
  - is a general repeatable solution to a commonly occurring problem in software design. like a template
  - Benefits
    - reusable
    - maintainable
    - scalable
    - encapsulation
    - flexible
    - readability
    - Abstraction
    - collaborative
    - Optimized
  - Drawbacks
    - complex
    - rigid
    - overhead
    - lack of creativity
- types of **design patterns**
  - **Creational patterns** deal with object creation mechanisms, helping you choose the most appropriate way to instantiate objects based on your specific needs
    - **Singleton:** Ensures only one instance of a class exists, making it suitable for managing global resources or configurations.(already done)
    - **Factory Method:** Creates objects without specifying the exact class to be instantiated, promoting flexibility and decoupling.

```
// before
// Client code directly creates concrete product instances
void clientCode() {
    Product* product;
    if (condition) {
        product = new ConcreteProduct1(); // Client knows
    } else {
        product = new ConcreteProduct2(); // Client knows
    }
    product->use();
}
```

```
// after
// Client code delegates object creation to the factory me
void clientCode() {
    Creator* creator;
    if (condition) {
        creator = new ConcreteCreator1(); // Client choose
    } else {
        creator = new ConcreteCreator2(); // Client choose
    }
    Product* product = creator->factoryMethod(); // Client
    product->use();
}
```

- Structural Patterns
  - **Composite Design Pattern:** Treats a group of objects as a single object, simplifying the handling of complex structures.

```
// before
class File {
public:
    void ls() {
        std::cout << "File: " << name << std::endl;
```

```
    }
private:
    std::string name;
};

class Directory {
public:
    void ls() {
        std::cout << "Directory: " << name << std::endl
        for (const auto& file : files) {
            file->ls();
        }
    }
private:
    std::string name;
    std::vector<File*> files;
};
```

```
//after
class FileSystemItem {
public:
    virtual ~FileSystemItem() {}
    virtual void ls() = 0;
    virtual void add(FileSystemItem* item) = 0;
};

class File : public FileSystemItem {
public:
    void ls() override {
        std::cout << "File: " << name << std::endl;
    }
    void add(FileSystemItem* item) override {
        // Do nothing (files can't contain other items)
    }
private:
```

```
    std::string name;
};

class Directory : public FileSystemItem {
public:
    void ls() override {
        std::cout << "Directory: " << name << std::endl
        for (const auto& item : items) {
            item->ls(); // Uniform treatment of items
        }
    }
    void add(FileSystemItem* item) override {
        items.push_back(item);
    }
private:
    std::string name;
    std::vector<FileSystemItem*> items;
};
```

- **Adapter Pattern**: Makes incompatible interfaces work together by wrapping one interface in another.

```
// EnglishBook interface
class EnglishBook {
public:
    virtual ~EnglishBook() {}
    virtual string readPage() = 0; // Returns a string
};

// SpanishSpeaker interface
class SpanishSpeaker {
public:
    virtual ~SpanishSpeaker() {}
    virtual void listenTo(string text) = 0; // Expects
};
```

```
// TranslatorAdapter class
class TranslatorAdapter : public EnglishBook {
public:
    TranslatorAdapter(SpanishSpeaker* speaker) : speake

    string readPage() override {
        string englishText = originalBook->readPage();
        string spanishText = translateToSpanish(englisl
        speaker->listenTo(spanishText); // Adapt the te
        return spanishText; // Return the translated te
    }

private:
    EnglishBook* originalBook;
    SpanishSpeaker* speaker;
};
```

- **Behavioral Design Pattern** focus on how objects communicate and collaborate with each other, defining communication models and algorithms for specific tasks.
  - **Observer Pattern**: Defines a one-to-many dependency where an object notifies all its dependents of any state changes.

```
//before
class Subject {
public:
    void changeState() {
        // State change logic
        state = newState;
        // Manually notify observers (tightly coupled)
        for (Observer* observer : observers) {
            observer->update();
        }
    }
```

```
private:
    std::vector<Observer*> observers;
    int state;
    int newState;
};

class Observer {
public:
    virtual void update() = 0;
};

class ConcreteObserver : public Observer {
public:
    void update() override {
        // React to the state change
        std::cout << "Observer notified of state change
    }
};
```

```
//after
class Subject {
public:
    void attach(Observer* observer) {
        observers.push_back(observer);
    }

    void detach(Observer* observer) {
        observers.erase(std::remove(observers.begin(),
    }

    void notify() {
        for (Observer* observer : observers) {
            observer->update();
        }
    }
```

```
    void changeState() {
        // State change logic
        state = newState;
        notify(); // Notify observers automatically
    }
private:
    std::vector<Observer*> observers;
    int state;
    int newState;
};

// Observer interface remains the same

int main() {
    Subject subject;
    ConcreteObserver observer;
    subject.attach(&observer);

    subject.changeState(); // Observer gets notified
}
```

- Template Method Pattern

```
#include <iostream>

class Beverage {
public:
    void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
```

```
protected:
    virtual void boilWater() {
        std::cout << "Boiling water...\n";
    }

    virtual void brew() = 0; // Abstract, must be imple

    virtual void addCondiments() = 0; // Abstract, must

private:
    void pourInCup() {
        std::cout << "Pouring into cup...\n";
    }
};

class Coffee : public Beverage {
protected:
    void brew() override {
        std::cout << "Brewing coffee...\n";
    }

    void addCondiments() override {
        std::cout << "Adding sugar and milk...\n";
    }
};

class Tea : public Beverage {
protected:
    void brew() override {
        std::cout << "Steeping tea...\n";
    }

    void addCondiments() override {
        std::cout << "Adding lemon...\n";
    }
};
```

```
int main() {
    Beverage* coffee = new Coffee();
    Beverage* tea = new Tea();

    coffee->prepareRecipe(); // Output: Boiling water.
    tea->prepareRecipe();   // Output: Boiling water..
}
```