# Sequence Diagrams

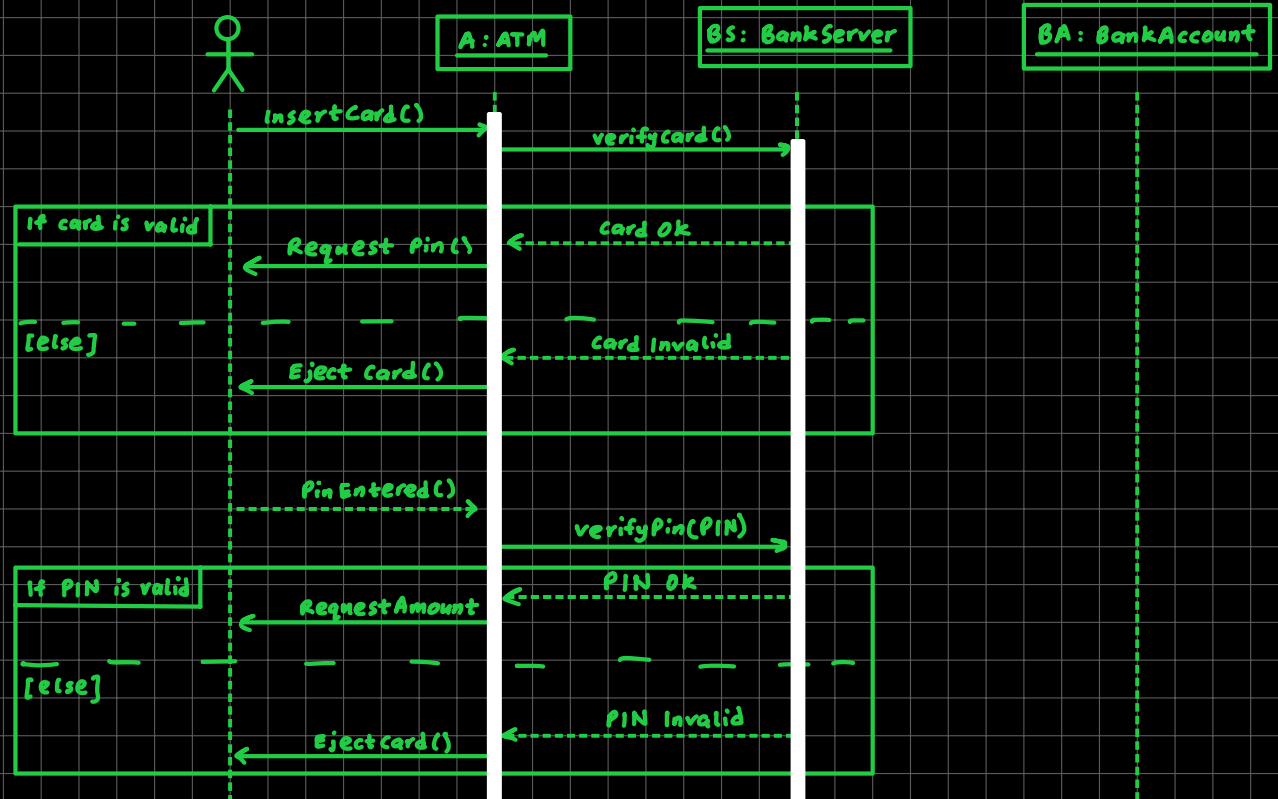
- Interaction Diagrams
- Models dynamic steps of a software system - helps visualize how a system runs
- Shows how actors and objects communicate with each other to perform the steps of a use case or any other functionality - the steps taken altogether is called an interaction
- Shows several different types of communication, including messages, procedure calls, and commands issued by the actor - collectively known as communication

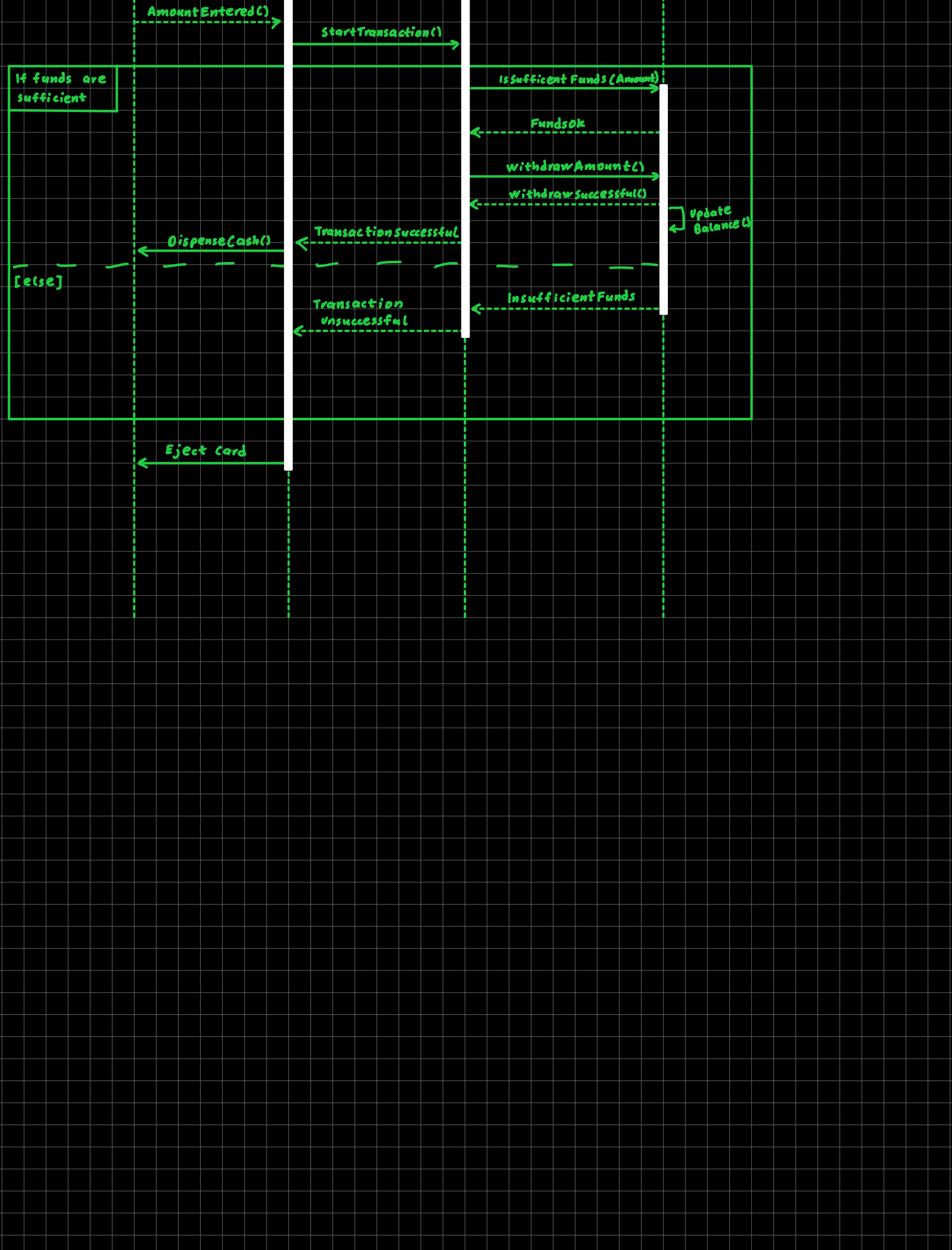
→ Elements:
- Instances of classes or actors ⟶ objects as boxes with class and object identifier underlined
- Messages → shown by arrows  actors as stick-person

- Sequence diagrams
- Shows sequence of messages exchanged by set of objects (optionally by an actor) performing a certain task
- Diagram that shows object interactions (function calls) required to execute a use case - sir def
- Provides a dynamic overview

- Head of Arrow shows class in which function is implemented
- Tail of Arrow shows the class in which the function initiated the call
- Lifeline → vertical dashed line

- Objects arranged left to right
- Actor initiates the request is shown on left
- Vertical dimension shows time
- Top of the diagram is the starting point, and time progresses downwards
- Lifeline attached to each object / actor
- Lifeline becomes activation box during the period that the object is performing computations
→ Object is said to have live activation during this time
- Message represented as arrow b/w activation boxes of sender and receiver.
- Message is given a label, an argument is optional
- Loop and if...else structures
- Recursive structures




AmountEntered()
StartTransaction()
If funds are sufficient
IsSufficentFunds(Amount)
Fundsok
withdrawAmount()
withdrawSuccessful()
Update Balance()
TransactionSuccessful
DispenseCash()
[else]
InsufficientFunds
Transaction unsuccessful
Eject Card

# State Diagram

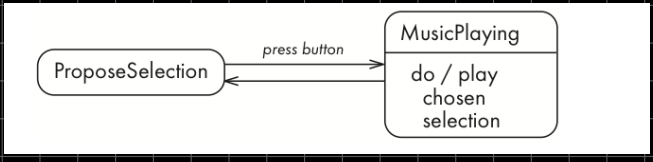
- Describes externally visible behaviour of a system / of an object
- Used in design and analysis
- Rounded Rectangles for states
- Transition represents change of state in response to an event, and occurs instantaneously → Label: What causes transition
- Black Circle represents start state, the machine automatically transitions from start state to a regular state
- Only 1 start state
- Black with a ring around it represents an end state.
- There can be multiple end states

- 2 computations in state diagram: activities, actions

- **Activity:**

→ Sth that occurs over a period of time while the system is in a state

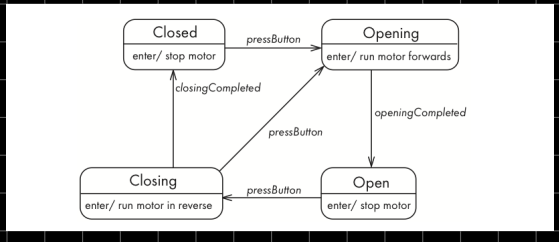
→ The system might take a transition out of the state in response to the completion of an activity, but if some other transition is triggered, then the system has to terminate the activity




- Written in the box after keyword "do" and "/"

- **Action:**
- Sth that takes place instantaneously in any of the following situations:

① When the system takes a specific transition
② Upon entry into a state, regardless of any transition taken
③ Upon exiting a particular state, no matter which transition is taken



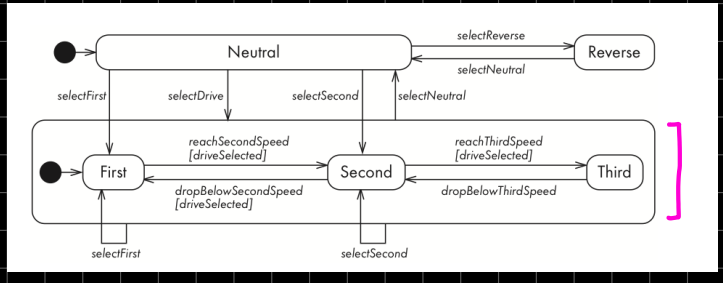

- If action is performed during transition "event / action"

- Upon entering a state "enter/action"
- Upon exiting a state "exit/action"

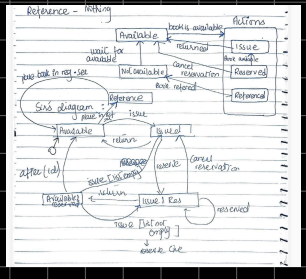
- **Nested Substates and guard conditions**
- State diagram nested inside a state
- States of inner diagrams are called substates

- Making initial state is optional if defined

- enter/ exit/ do while in state
→ Action keywords



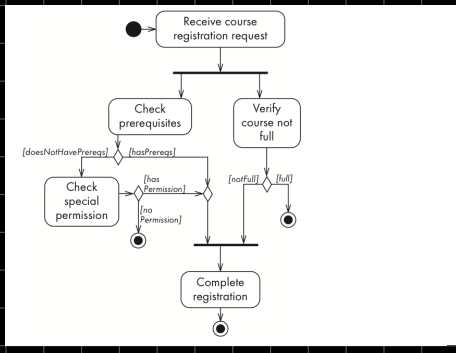

Drive State



- reachSecondSpeed [driveSelected] has a guard condition. If condition in square brackets is True, only then the indicated event (reachSecondSpeed) occurs
- guard condition only evaluated when the associated event occurs

# Activity Diagram

- Transitions caused by internal events
- Used to understand the flow of work an object / component performs
- " " visualize interactions b/w use cases
- May describe an algo / business process



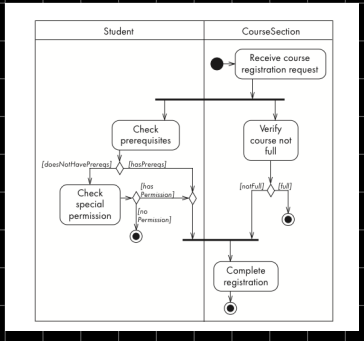

- A *fork* has one incoming transition and multiple outgoing transitions. The result is that execution splits into two concurrent threads.
- A *join* has multiple incoming transitions and one outgoing transition. The outgoing transition will be taken only when all incoming transitions have been triggered. The incoming transitions must be triggered in separate threads.
- A *rendezvous* has multiple incoming and multiple outgoing transitions. Once all the incoming transitions are triggered, the system takes all the outgoing transitions, each in a separate thread.

An activity diagram also has two types of nodes for branching within a single thread. These are represented as small diamonds:

- A *decision node* has one incoming transition and multiple outgoing transitions, each with a Boolean guard in square brackets. Exactly one of the outgoing transitions will be taken.
- A *merge node* has two incoming transitions and one outgoing transition. It is used to bring together paths that had been split by decision nodes.

- Optional activity written in description

- Also has guard conditions




- Swimlanes used to partiton classes based on their activities
- Read code at end of book

# Solid Principles

1. Single Responsibility principle
2. Open close principle
3. Liskov's principle
4. Interface segregation principle
5. Dependency inversion principle

## 1- Single Responsibility Principle:

- A module (class / Function) should focus on a single entity or task violations

**Violations:**

→ searchAndSort(...)
→ class studentAndTA{...}
→ class Emp {string street[N], string city[N]}

why?

- can introduce more bugs, merge conflicts

```
public class Student
{
    public int StudentId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }

    public void Save()
    {
        // Code to save student to database
    }

    public void Email(string subject, string body)
    {
        // Code to Email student
    }

    public void Enrol(Course course)
    {
        // Code to enrol the student in a course
    }
}
```

- seperate class for everything

- single Responsibility ≠ single job
→ The module has everything that is very closely related e.g: class student needs to hold student details but has functions save() , Email() , Enrol() which is not the class's responsibility

## 2- open close Principle:

- software Entity (class / Function etc) should be open for extension but closed for modification

- why?
- unit tests would break
- Additional issues if edited methods called

- **Decorator Pattern**

```
public class Service : IService
{
    public int DoSomething(int input)
    {
        return input + 2;
    }
}
```

```
public class NewService : IService
{
    public readonly IService _service;

    public NewService(IService service)
    {
        _service = service;
    }

    public int DoSomething(int input)
    {
        // Do something new here
        ValidateInput(input);

        var value = _service.DoSomething(input);

        // Maybe add something here as well
        return value;
    }
}
```

- Make new class extending the one in which modification is to be made.

- Works either you want to add functionality before or after the already written code

- Works only if the module is public and can be inherited

- **Extension Pattern:**

```
namespace Extensions
{
    public static class ExtensionMethods
    {
        public static void DoSomething(this Student student)
        {
            Console.WriteLine($"Hello {student.FirstName}");
        }
    }
}
```

```
using System;
using Extensions;

public class Program
{
    public static void Main()
    {
        var student = new Student();
        student.StudentId = 1;
        student.FirstName = "Alex";
        student.LastName = "Hyett";
        student.Email = "hi@alexhyett.com";

        student.DoSomething();
    }
}
```

- public static class
- method to be inside public static class and must be static
- The first parameter defines which type is being extended, it is preceeded by 'this' keyword.
- only 1 parameter can be extended in a method
- use namespace in main program

```
void foo(void* shape, int type) {
    switch(type) {
        case 'C':
            Circle* cir = (Circle*) shape;
            cir->draw();
            break;
        case 'R':
            Rect* rec = (Circle*) shape;
            rec->draw();
            break;
        ...
    }
}
```

class example using polymorphism

```
void foo(Shape* shape) {

    shape->draw();

}
```

- can't switch or change at runtime

## 3- Liskov's Principle → square is a rectangle analogy

- To be able to replace a super-type reference with a sub-type reference without affecting program's correctness

- Example
- Child inherits Parent but logically child can't do some things that parents do, so it breaks this principle.
- Instead we make parent and child inherit a human class.

## 4- Interface Segregation Problem

- Many small specific interfaces are better than a single large interface

- In bulky interfaces, methods are implemented that might not be used

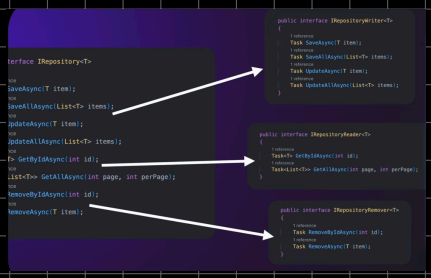
Initial Problem

```
public interface IRepository
{
    Task SaveAsync(Student student);
    Task SaveAsync(int studentId, string firstName, string lastName);
    Task SaveAsync(int studentId, string firstName, string lastName, string e
    Task SaveAllAsync(List<Student> students);
    Task UpdateAsync(Student student);
    Task UpdateAsync(int studentId, string firstName, string lastName);
    Task UpdateAsync(int studentId, string firstName, string lastName, string
    Task UpdateAllAsync(List<Student> students);
    Task<Student> GetStudentByIdAsync(int studentId);
    Task<List<Student>> GetAllStudentsAsync(int page, int perPage);
    Task RemoveStudentAsync(int studentId);
}
```

→

```
public interface IRepository<T>
{
    1 reference
    Task SaveAsync(T item);
    1 reference
    Task SaveAllAsync(List<T> items);
    1 reference
    Task UpdateAsync(T item);
    1 reference
    Task UpdateAllAsync(List<T> items);
    1 reference
    Task<T> GetByIdAsync(int id);
    1 reference
    Task<List<T>> GetAllAsync(int page, int perPage);
    1 reference
    Task RemoveByIdAsync(int id);
    1 reference
    Task RemoveAsync(T item);
}
```

→



↓

```
public interface IRepository<T> :
    IRepositoryReader<T>, IRepositoryWriter<T>, IRepositoryRemover<T>
{
}
```

Liskov

```
class Circle {
    … scale(...);
    … rotate(...);
    … shear(...);

    … circum(...);
    … area(...);
    … dia(...);
}
```

```
class Parent1 {
    … scale(...) = 0;
    … rotate(...) = 0;
    … shear(...) = 0;
}
```

```
class Client1 {
    Circle* cir;
    void foo() {
        cir->scale(25);
    }
}

class Client2 {
    Circle* cir;
    void bar() {
        cout << cir->area();
    }
}
```

```
class Parent2 {
    … circum(...) = 0;
    … area(...) = 0;
    … dia(...) = 0;
}
```

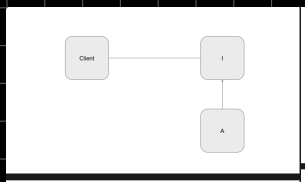
```
class Client1 {
    Parent1* cir = new Circle(...);
    void foo() {
        cir->scale(25);
        cout << cir->area();   // error
    }
}

class Client2 {
    Parent2* cir = new Circle(...);
    void bar() {
        cout << cir->area();
    }
}
```

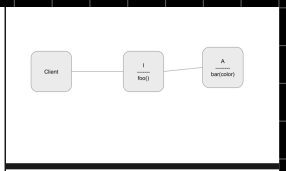
```
class Circle : public Parent1, public Parent2 {
    … scale(...);
    … rotate(...);
    … shear(...);

    … circum(...);
    … area(...);
    … dia(...);
}
```

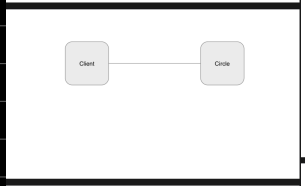
## 5- Dependency Inversion Principle:

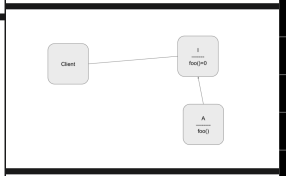
- High level modules shouldn't depend on low-level modules. They should depend on abstraction, e.g: nested classes. List depend on node | or direct association

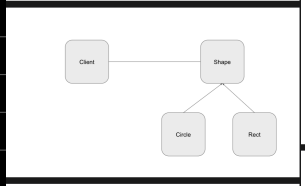
- If a module is expected to change then do not directly interact with it
- Inclusion of an interface or parent class for
- Dependency Injection

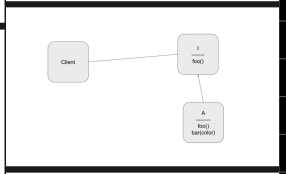


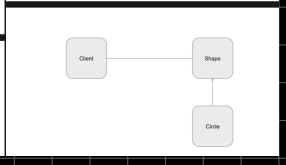








# Two Design Principles

1. Reduce coupling where ever possible
2. Increase cohesion where ever possible

1. Coupling occurs when there are interdependencies b/w modules