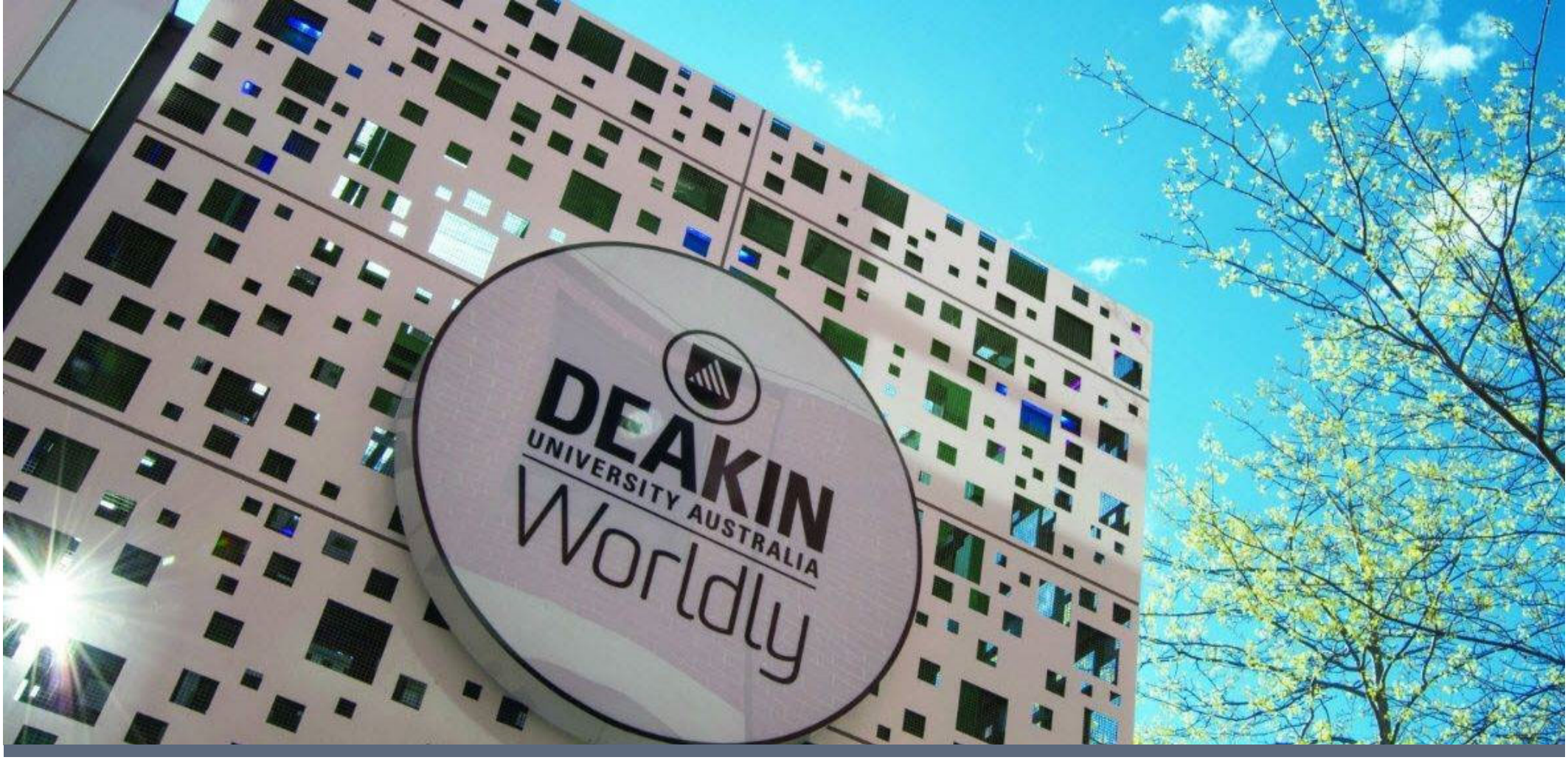
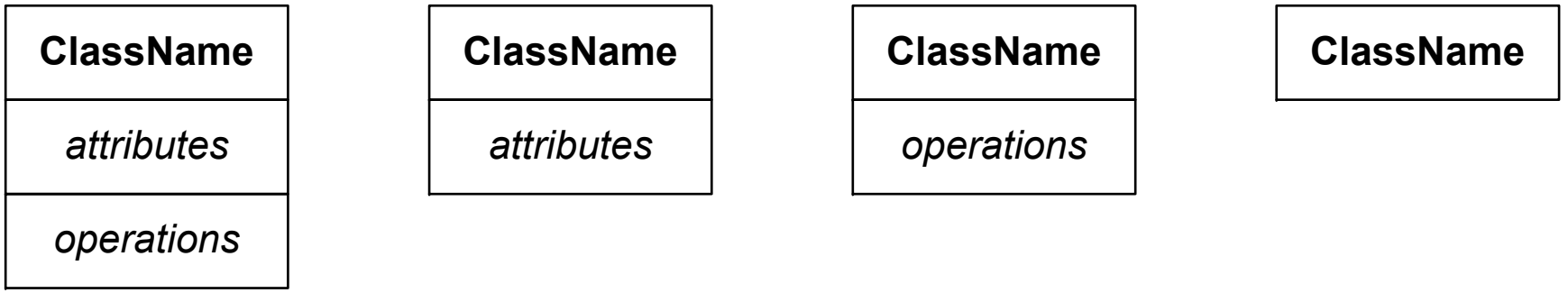


Lecture 04. Modelling and Collaboration. Program Design

SIT232 Object-Oriented Development

Class Diagrams

- The Unified Modelling Language (UML) is a de-facto standard for modelling object-oriented systems.
- Classes are represented in UML by a box, which can have up to three sections:
  - Class name (required, in bold);
  - Attributes (optional); and
  - Functions (optional).



Class Diagrams: Examples

Student
– _ID : string – _Name : string – _Course : string
+ Student(id : string, name : string, course : string) + GetID() : string + GetName() : string + GetCourse() : string + ChangeCourse(newCourse : string) + SendFeesInvoice(amount : decimal) + Enrol(unit : string) : bool + Withdraw(unit : string) : bool

Student
– _ID : string + «property» ID : string {readOnly} – _Name : string + «property» Name : string {readOnly} – _Course : string + «property» Course : string {readOnly}
+ Student(id: string, name: string, course: string) + ChangeCourse(newCourse: string) + SendFeesInvoice(amount: decimal) + Enrol(unit: string): bool + Withdraw(unit: string): bool

Student
+ ID : string {readOnly} + Name : string {readOnly} + Course : string {readOnly}
+ Student(id: string, name: string, course: string) + ChangeCourse(newCourse: string) + SendFeesInvoice(amount: decimal) + Enrol(unit: string): bool + Withdraw(unit: string): bool

Class Diagrams: Attributes

visibility name : type multiplicity = default_value {property}

- **visibility** – indicates the visibility (access modifier) of the attribute using special symbols:
  - '+' for public; or
  - '–' for private;
- **name** – the name of the attribute
- **type** – data type for the attribute (simple type or custom type)
- **multiplicity** – optional, indicating how many instances the attribute refers to (usually one unless referring to a collection)
- **default_value** – optional, an equals symbol (=) followed by the attribute's default value
- **property** – optional, surrounded by braces ('{' and '}'), indicates any additional properties about the attribute, e.g., readOnly
- **static attributes** are underlined

Class Diagrams: Operations (Methods)

visibility name(parameters) : return_type

- **visibility** is the same as for attributes
- **name** – the name of the operation
- **parameters** – optional, the parameters to the operation use a similar syntax to attributes
- **return_type** – indicates the data type; blank for no return value (void). Parenthesis are mandatory
- **static operations** are underlined

Object and Class Relationships

- Object relationships and class relationships are different but closely related
- **Object relationships** are implemented via a reference (link) and base on instances of a class
- **Class Relationships** (four types)
  - Association
  - Aggregation
  - Composition
  - Inheritance (does not result in object relationship)

Object Relationship

- Link is usually uni-directional, i.e., one object can invoke the services/methods of another object, but not vice-versa
- Direction of the link is often referred to navigability
- Though the direction does not prevent data from travelling in both directions, e.g., through **output parameters** and **return values**, e.g.

```
bool Account.TryWithdraw(decimal amount, out int transactionID)
```

Class Relationship: Association

- **Association** is a semantically weak relationship (a semantic dependency) between otherwise unrelated objects.
- An association is a “using” relationship between two or more objects in which the objects have their own lifetime and there is no owner.
- The objects that are part of the association relationship can be created and destroyed independently.

Class Relationship: Association

Example: relationship between a **doctor** and a **patient**.

- A doctor can be associated with multiple patients.
- One patient can visit multiple doctors for treatment or consultation.
- Each of these objects has its own life cycle and there is no “owner” or parent.

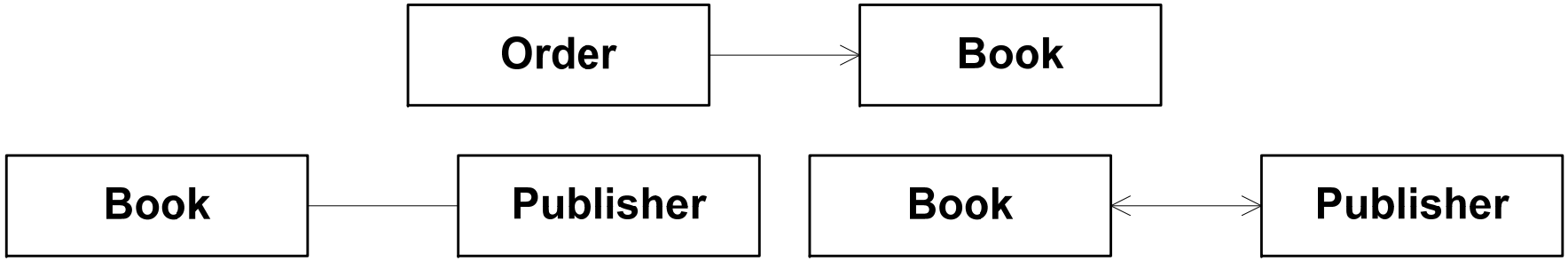
```
public class Doctor {
   private Patient[] patients;
   // ... other members of the Doctor class
}

public class Patient {
   int patientId;
   string name;
   int age;
   // ... other members of the Patient class
}
```

Class Relationship: Association

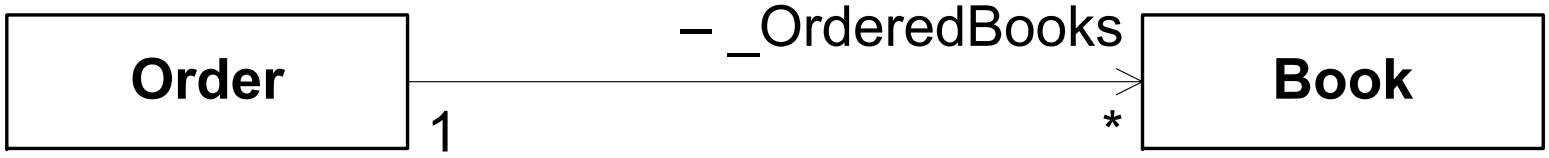
In UML, an association relationship is represented by a single arrow.

- An association relationship can be represented as one-to-one, one-to-many, or many-to-many (also known as cardinality).
- Essentially, an association relationship between two or more objects denotes a path of communication (also called a link) between them so that one object can send a message to another.



Class Relationship: Adding Multiplicity

- Multiplicity depicts the cardinality of a class in relation to another.
- For example, one fleet may include multiple airplanes, while one commercial airplane may contain zero to many passengers.
- The notation '0..*' in a diagram means "zero to many"
- '*' means zero or more when shown on its own
- Instead of '1..1', just show '1'



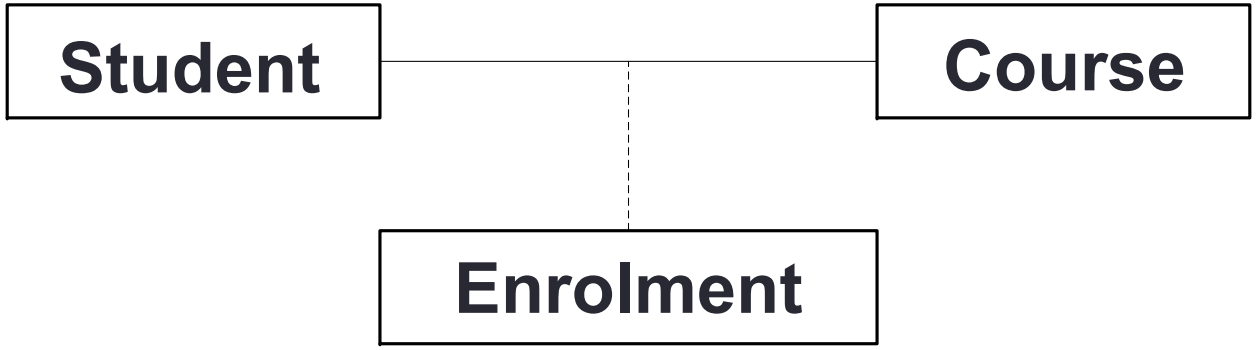
Class Relationship: Association classes

- An **association class**, in UML, is a class that is part of an association relationship between two other classes.
- You can attach an association class to an association relationship to provide additional information about the relationship.
- An association class is identical to other classes and can contain operations, attributes, as well as other associations.

Class Relationship: Association classes

Example: relationship between a **student** and a **course,** and **enrolment** as a class specifying this relation.

- Class Student represents a student and has an association with a class called Course, which represents an educational course.
- The Student class can enroll in a course.
- An association class called Enrollment further defines the relationship by providing section, grade, and semester information related to the association relationship.
- In UML, an association class is connected to an association by a dotted line.



Class Relationship: Aggregation

- **Aggregation** is a specialized form of association between two or more objects in which each object has its own life cycle, but there exists an **ownership** as well.
- Aggregation is a typical whole/part or parent/child relationship, but it may or may not denote physical containment.
- An essential property of an aggregation relationship is that the whole or parent (i.e. the owner) can exist without the part or child and vice versa.

Class Relationship: Aggregation

Example: relationship between an **employee** and **departments** in an organization.

- An employee may belong to one or more departments in an organization.
- If an employee's department is deleted, the employee object would not be destroyed, but would live on.
- A department may "own" an employee, but the employee does not own the department, i.e. relation cannot be reciprocal.

Class Relationship: Composition

- **Composition** is a specialized form (a strong type) of aggregation.
- If the parent object is destroyed, then the child objects also cease to exist.
- Like aggregation, composition is also a whole/part or parent/child relationship. However, in composition the life cycle of the part or child is controlled by the whole or parent that owns it.

Class Relationship: Composition

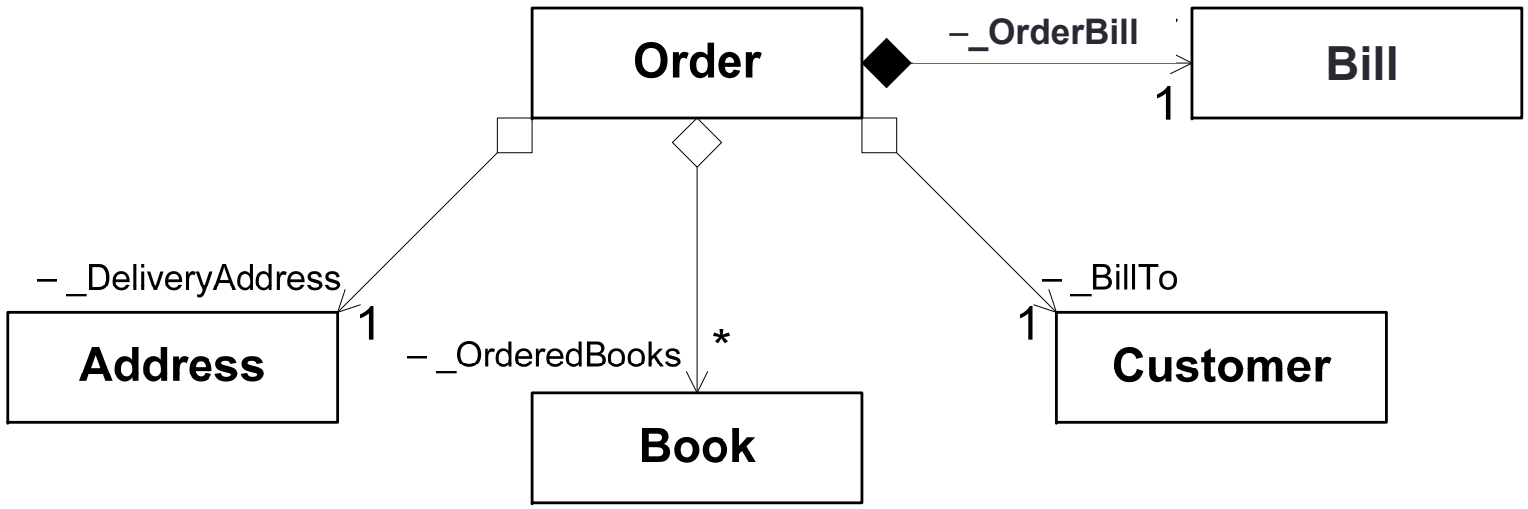
Example: relationship between a **house** and **rooms**.

- A house may be composed of one or more rooms.
- If the house is destroyed, then all of the rooms that are part of the house are also destroyed.

```
public class House
{
   private Room room;
   public House()
   {
      room = new Room();
   }
}
```

Class Relationship: Aggregation and Composition

- Aggregation is usually represented in UML using a line with a hollow diamond.
- Composition is represented in UML using a line connecting the objects with a solid diamond at the end of the object that owns the other object.



Class Relationship

To implement a relationship in a program there are two tasks:

- Declare a variable in a dependent object to reference the 'provider', e.g.
  - `private Student _SingleStudent;`
  - `private Student [] _StudentArray = new Student[size];`
  - `private List<Student> _Enrolment = new List<Student>();`
- Store the memory address of 'provider' as a reference within the dependent object, e.g.

```
public void EnrolStudent(Student student)
{
    _Enrolment.Add(student);
}
```

Program Development: Focus on Design

- **Design first**
  - Consider decomposition of a problem into sub-problems
  - Identify interaction between sub-problems
  - Represent problem via diagrams
  - Think about proper data structures
  - Write pseudo-code
- **Then code**
  - Implement data structures and functionality
  - Care about debugging
  - Test the program against various scenarios

Design first, then code: Why?

- What are the advantages of designing first and then writing code?
- What are the disadvantages?

Design first, then code: Why?

- **What are the advantages of designing first and then writing code?**
  - allows us to identify the main components of the code (and their relationships and interactions) before writing code
    $\Rightarrow$ saves us time when coding
  - allows us to identify potential issues and performance bottlenecks before the code is complete
    $\Rightarrow$ saves us time (and money!)
- **What are the disadvantages?**
  - we can only see issues with the design after we have produced code and tested it
  - there is a delay until we get a ‘product’

What is a good design?

What makes a good design?

A good design is …

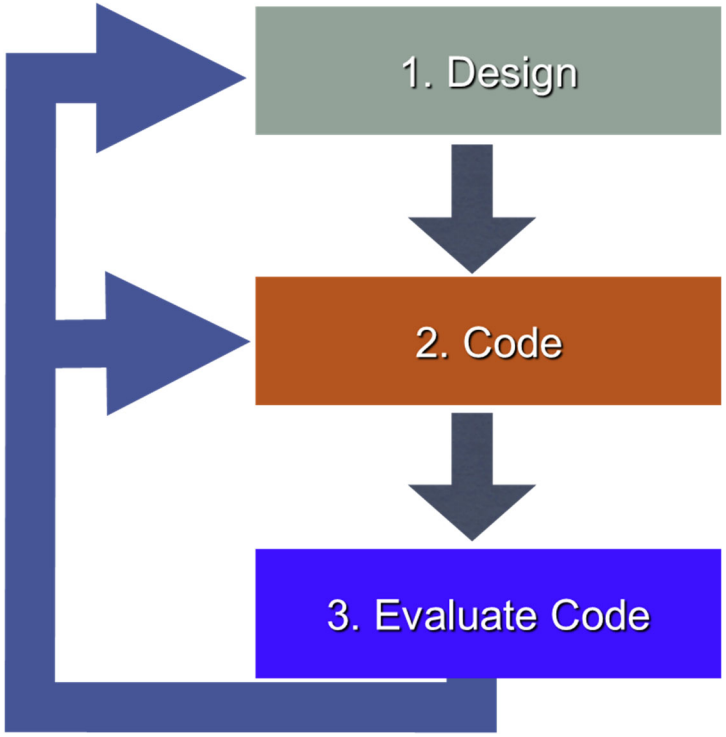
- **Extensible** - new components/modules can be easily added
- **Adaptable** - can adapt to changes easily
- **Reusable** - the design (or parts of it) can be used again with little or no modifications

Design is an Iterative Process

- We cannot say if a design is good until we have implemented the code
- If the code does not exactly meet the specification, then
  - **if the code does not conform to the design**
    $\Rightarrow$ modify code; repeat
  - **if the code conforms to the design**
    $\Rightarrow$ modify design; repeat

Iterative Lifecycle

- Analyze (evaluate)
- Admit - admit that you have an issue with (parts of) your code
- Adapt - change the code/design accordingly



Evaluation

There are many measures that could be used to evaluate code

- Does it meet the requirements?
- Does it behave as expected for a variety of inputs: both expected and unexpected?
- Is it fast enough? When is it slow? How frequent is the slow case?
- What are the performance bottlenecks?
- Other quality measures:
  - Is the code easy to maintain?
  - Is the code safe/secure?

Software has a Life Cycle

- We can consider any piece of software to go through distinct stages, just as many living entities do.
- This changes our expectations as to what the software should be doing at any given time.
- This also changes what we should be doing at the software development stages.

Software has a Life Cycle