Sukanya Chinwicha
6313050022B

Course Guideline

- The challenge of OOP is abstractness.
- In workshop, we need consider code, design (structure) to implement. (about word problem)
- Week 08, prepare for exam in next week.
- Code must be the same format/pattern, otherwise you are cheating.
- On workshop, Bring Your Own DEVICES!!
- Be careful if you use online resources, it leads to be plagiarism.

---

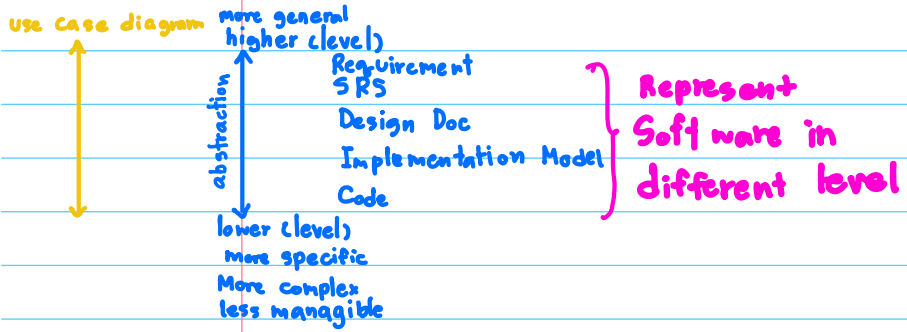
18 Jan 2024 - lecture1

4 pillars of Object Orientation: abstraction, inheritance, encapsulation, polymorphism

Review
กลุ่ม 1.11.36

User of language
-

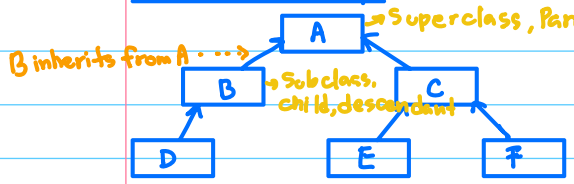
Designer of language
-



OO Languages support abstraction, it can be used to help manage complexity.

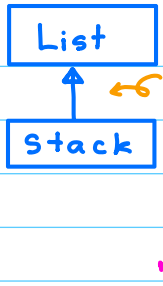
For the same software,
- The data should be consistency
- With higher level, some details may be left out and some are kept
- Some irrelevant details may left out (not too complex)

Inheritance Concept (reuse)



- B can reuse properties which defined by A
- For private, it can be inherited, but cannot access.

Sukanya Chinwicha
6313050022 8



is-a relationship (kind-of relationship)

==

"Type"

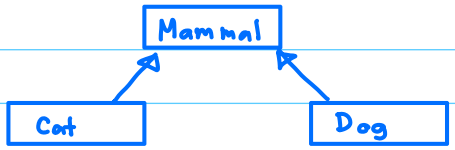
observing this when working with inheritances, will allow you to avoid many potential problems & also allow to to observe the Liskov Substitution Principle (LSP: SOLID Design Principle)

Where we have a variable of supertype, we can assign a value of subtype to it.

Data Type in Java : primitive type
• object reference type

superclass → supertype
subclass → subtype

Polymorphism Concept




```
Mammal m;
```

Polymorphic variable

```
m = new Cat();
m = new Dog();
```

Cat and dog are subtypes

Polymorphism

review delegate ?

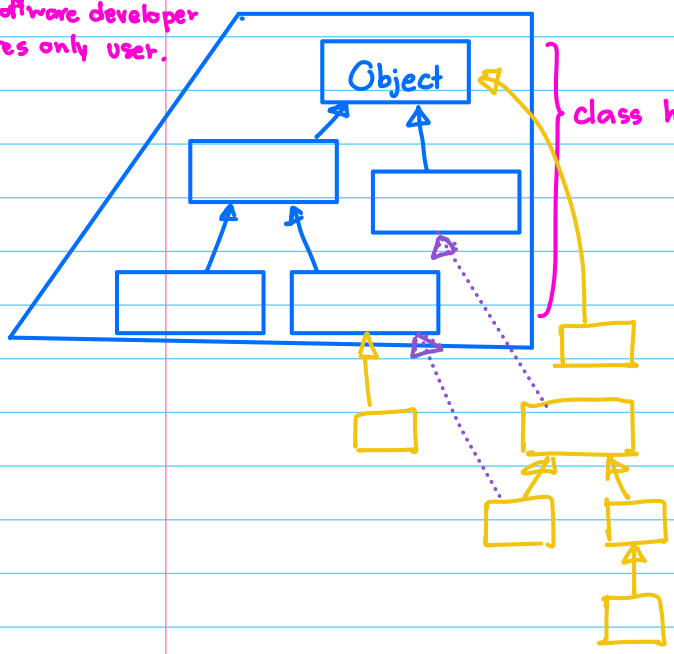
Don't get to play with polymorphism if you don't have inheritance.

Java Class Inheritance → tree structure
- root
- branch
- leaf

in Java has only one root that is "Object".

- user : ?
- designer : ?

Software developer cares only user.

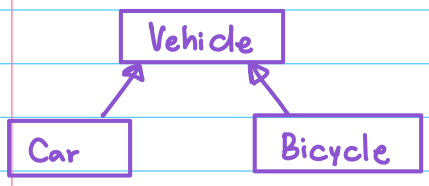


class hierachy

Make sure your library is very flexible enough.

Sukanya Chinwicha
6313050022B

Subtype vs Supertype




```
Car c;
c = new Bicycle();
```

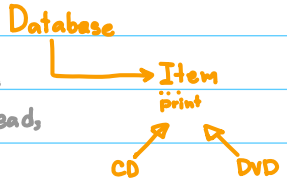
Substitution principle

Compiler check only variable in static type. Cannot overhead look the type.

static type of c is Car. [Compile code]
dynamic type of c is bicycle. [execute code]

With designer language perspective, there are two stages
- Compile code = static
- execute code = dynamic

Potential Problem



- If we have print method in CD class and DVD instead, It will be error. Because from the compiler it reads only static type, when the print occurs in CD and DVD not in the Item. By the way, the compiler will see Item performs it. So it will return "print" does not exist.
- So the "print" should be in "Item" too.

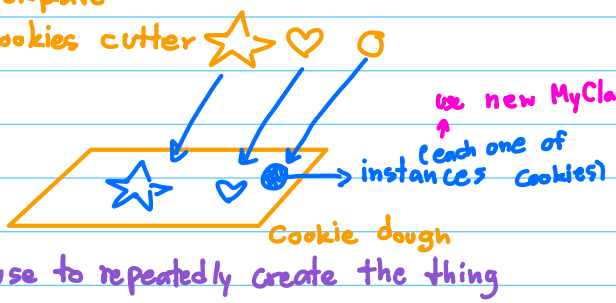
Sukanya Chinwicha
6313050022B

25 Jan 2024

```
public class MyClass {
    instance variables
    methods
}
```

instance variables ← variables that belong to class instances

Class → like template
↘ or cookies cutter



Can use to repeatedly create the thing

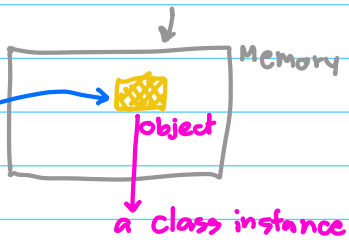
"Instantiation" is the action of creating new class instances from the class(definition).
The resulting instances are typically referred to as "objects".

Terminology
- Class
- Objects
- Instances
- Instantiation

class → public class ____
{
}
} definition of object

```
MyClass mc = new MyClass();
```

"Instantiation"
↓
MyClass is "instantiated" → With this process, the system will allocate space in memory large enough for MyClass object.



reference (address) → a class instance

If it's not assigned to any, we cannot access the instance. From this case, we assign to "mc".

Sukanya Chinwicha
6313050022B

Class —relationship→ Object

- A single class can be used to instantiated many class instances (objects).

* Don't say
  - The cookie belong to the cookies cutter ✗
  - The cookie cutter contains many Cookies ✗
  - A class contains many objects (instances) ✗
  - Class instances belong to a class ✗

\# = protected
\+ = public
\- = private

In class diagram, the abstract class → see slide (9) to see how abstract class can be used. (It just provide the skeleton, no what happened inside.

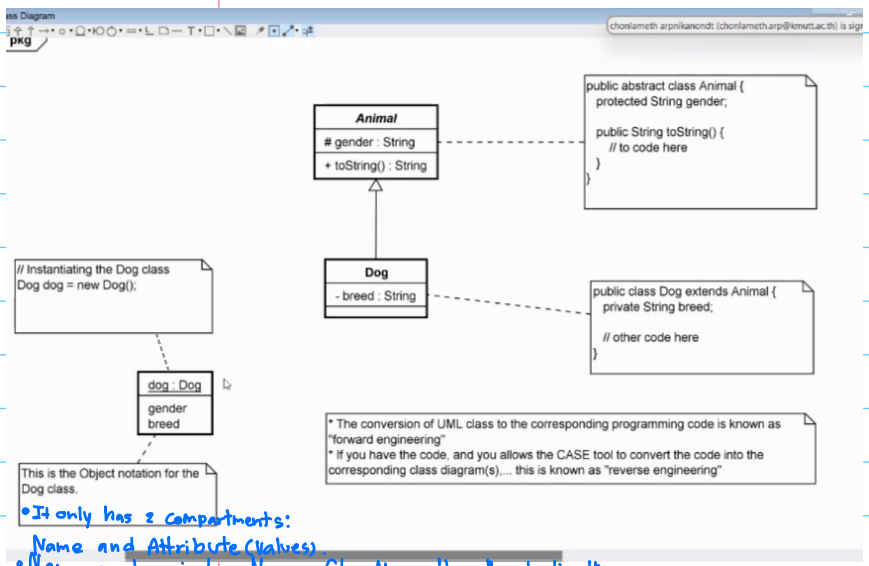
Such as italic font, stereotype <<abstract>>

"Forward engineering" is
the conversion of UML class to the corresponding programming code

"Reverse engineering" is when we do have the code, and you allows the CASE tool to convert the code into the corresponding class diagram(s)... this is known as "reverse engineering"

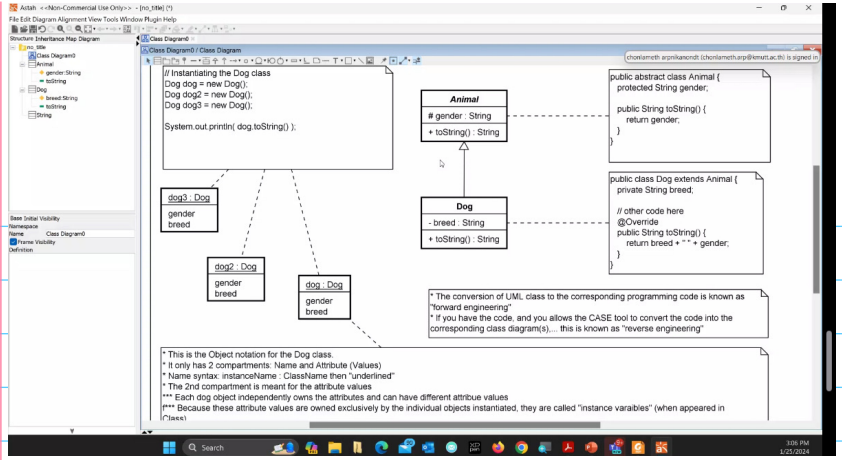
Abstract Concept

Inheritance and Class and Object

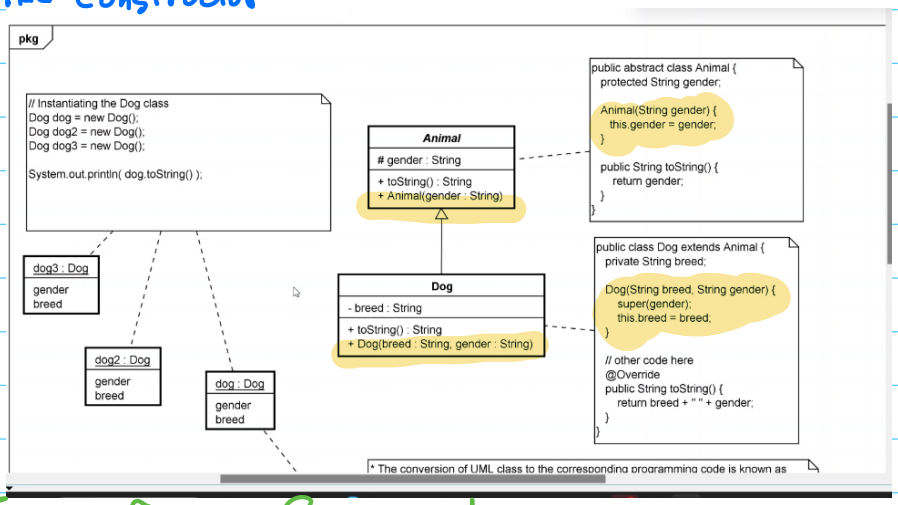


- It only has 2 compartments:
  Name and Attribute (values)
- Name syntax: instanceName : ClassName then " underlined".
- The 2nd compartment is meant for the attribete values
  *** Each dog object independently owns the attributes and can have different attribute values
  *** Because these attribute values are owned exclusively by the individual objects instantiated, they are called "instance variables" (when appeared in Class)
- There is no "method/operation" compartment, because there is no need to. All created objects/instances will always reference the method definition defined in the class
  *** Unlike the attribute values, you can share the method definitions but you cannot share the attribute values.
  *** Each individual object must take care of its own attribute values. Hence, these attributes are called "instance variables."

Sukanya Chinwicha
6313050028



Add Constructor



Interface Concept

-interface

** Abstract by itself

** has NO attributes

*** Except, you can have the constants defined in the attribute compartment

**** To define constants,

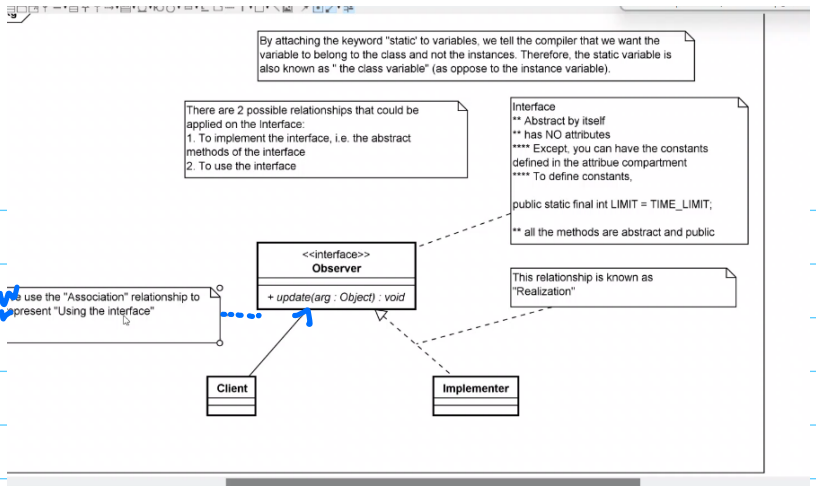
type variable

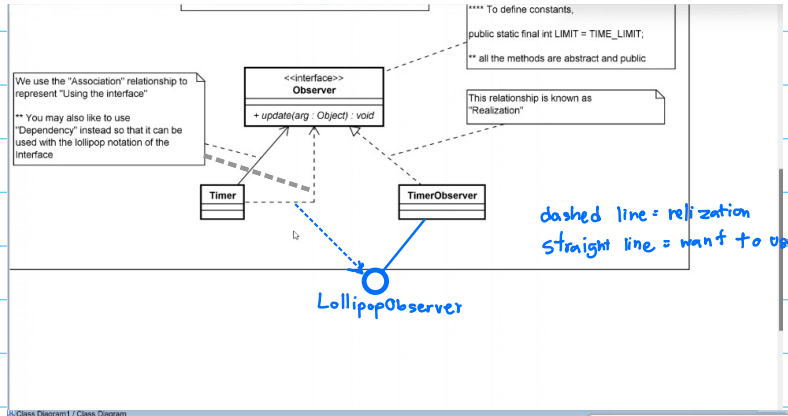
public static final int LIMIT = TIME_LIMIT;

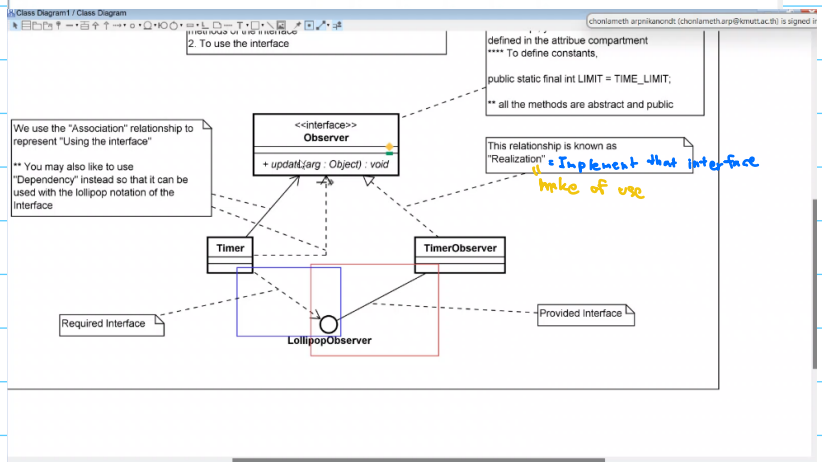
** all the methods are abstract and public

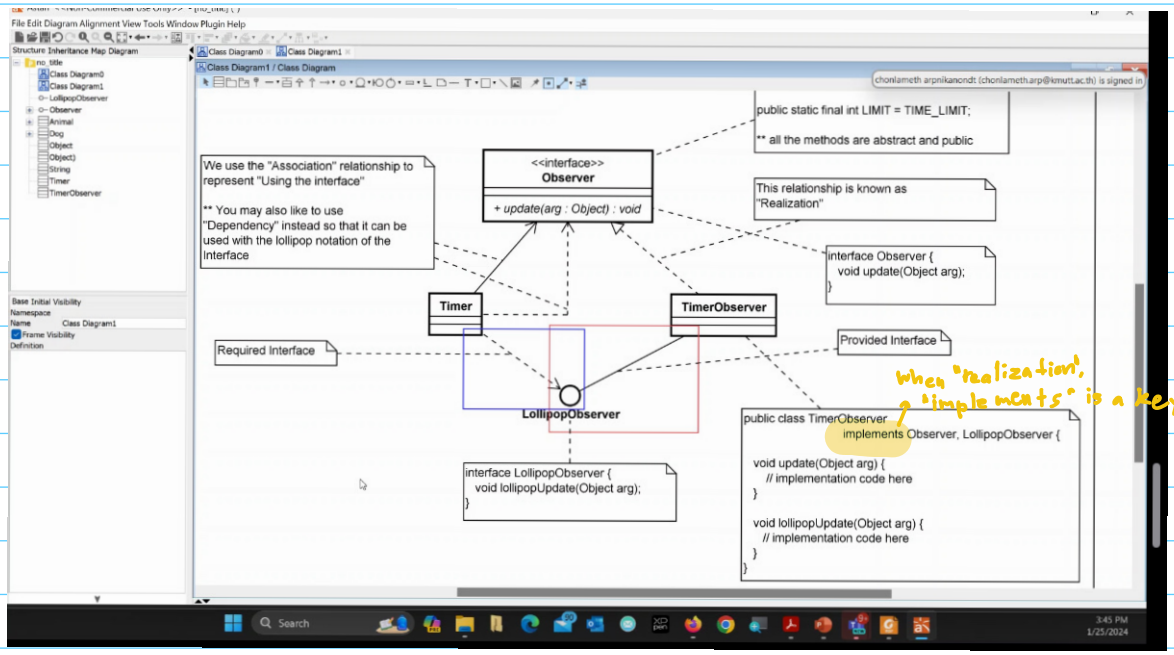
- By attaching the keyword "static" to variables, we tell the compiler that we want the variable to belong to the class and not the instances. Therefore, the static variable is also known as "the class variable" (as oppose to the instance variable).
- There are 2 possible relationships that could be applied on the Interface:
1. To implement the interface, i.e, the abstract methods of the interface.
2. To use the interface

Sukanya Chinwicha
6313050022B

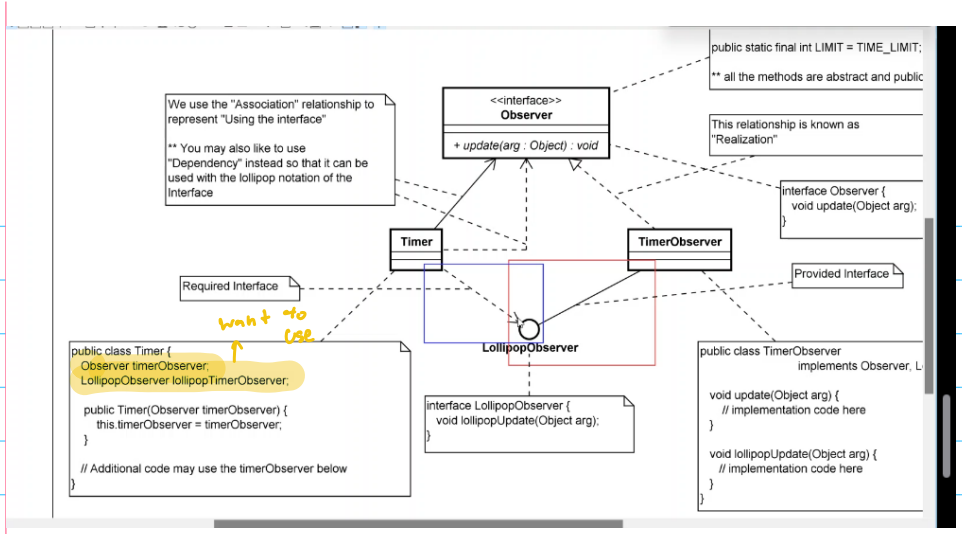






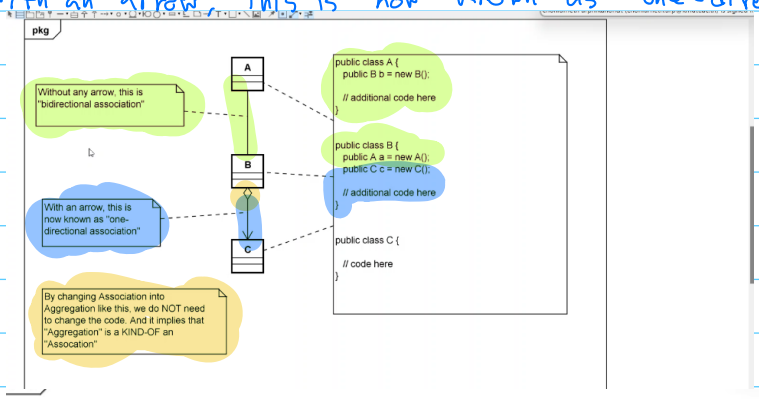


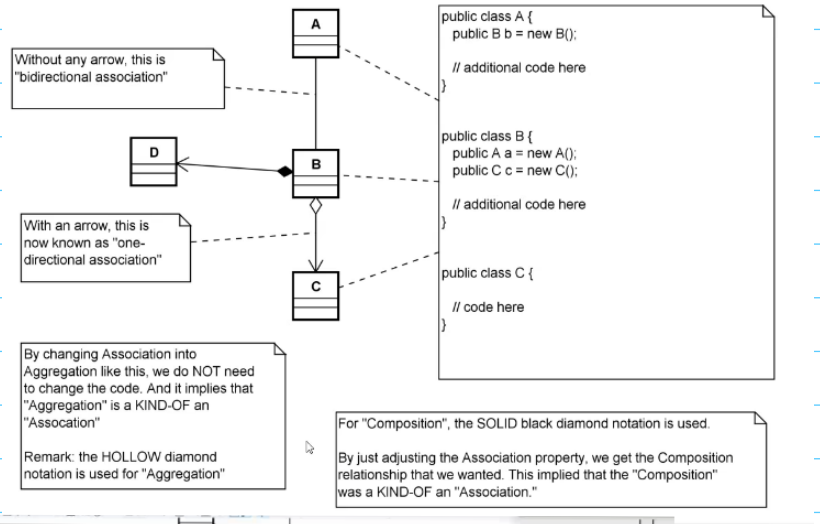
Sukanya Chinwicha
6313050022

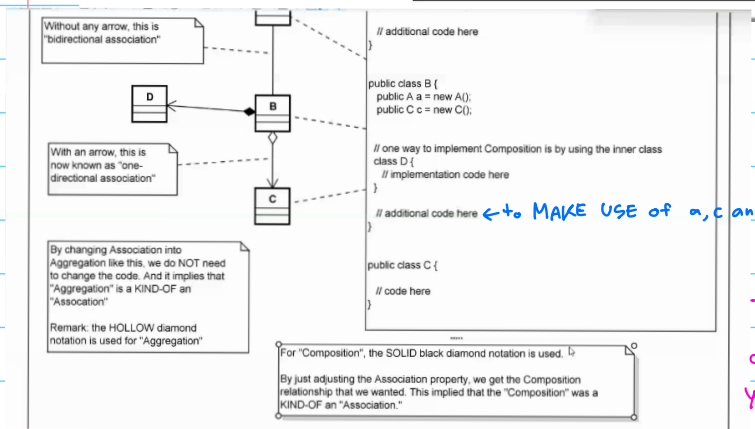


Association, Aggregation and Composition

- Without any arrow, this is "bidirectional association".
- With an arrow, this is now known as "one-directional association".







Typically, if you are NOT absolutely sure that the use of "Aggregation" or "Composition" will be correct. You should stick with using only "Association".

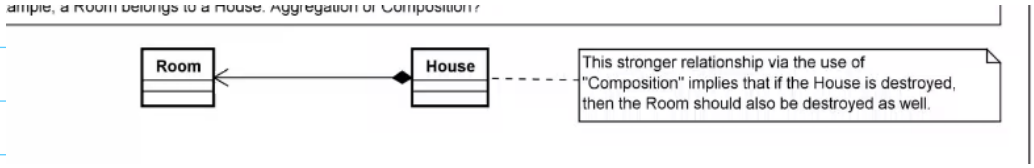
It is the safest way to define the simple association relationship. Sukanya Chinwicha 6313050022-8

And because both the "Aggregation" and "Composition" are a KIND-OF "Association", Association is more generalized and can be used, in general, to mean both "Aggregation" and "Composition".

In terms of defining the relationship, this is known as the HAS-A relationship, or the OWNERSHIP relationship. Then, in this case, "Composition" has a stronger ownership than that of "Aggregation" and than that of "Association".

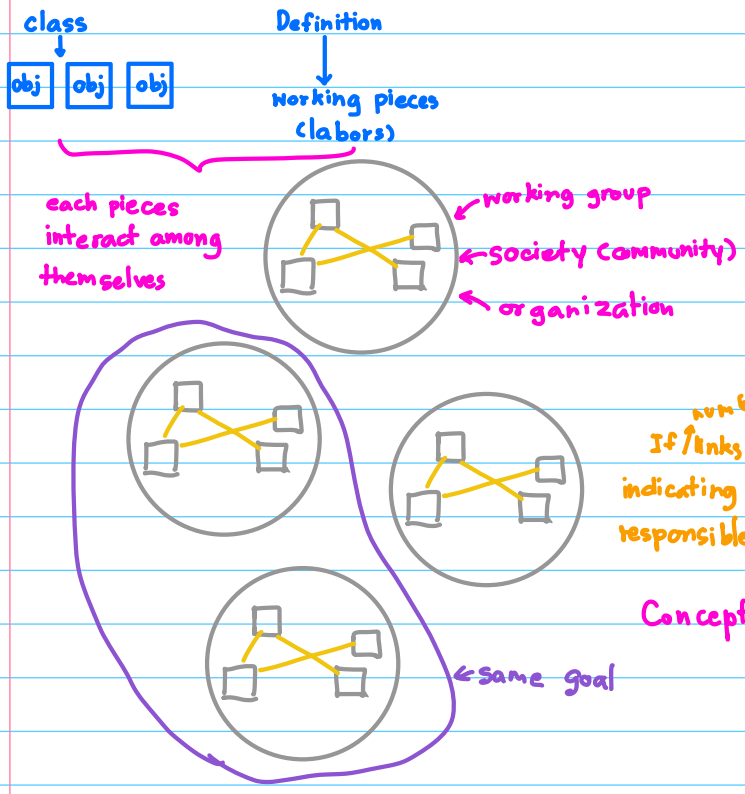
Such ownership, when applied to "Aggregation", it typically means that you can still "SHARE". But, when applied under "Composition", it implies that you do NOT share the objects/instances. "Composition" implies SOLE ownership of the objects.

For example, a Room belongs to a House. Aggregation or Composition?



Sukanya Chinwicha
6313050022

1 Feb 2024



If number of links is too much, indicating each Object is responsible too less.

Concept of Message: normal for OO is communication

How OO works

subsystem must:

- can interact & achieve more complex goal

Client-server on Internet

More links you have, tighter coupling will become.

No link, no dependencies, no coupling but it's too complex → (God Class) to deal with

↳ good solution for coupling
but not for other metric
such as cohesion

If Object is large, just cohesion↓ coupling↑ (balance both) to find optimum number of Obj. to interact with

Single Responsibility Priciple: SCP

when depency is missing, coupling is better

Sukanya Chinwicha
6313050022B

Code Duplication / Replication is bad → Maintainance harder
to avoid them try to combine them,

Variable, method, Class, package/library, component, reusable solution/framework → encapsulation concept in OOP.

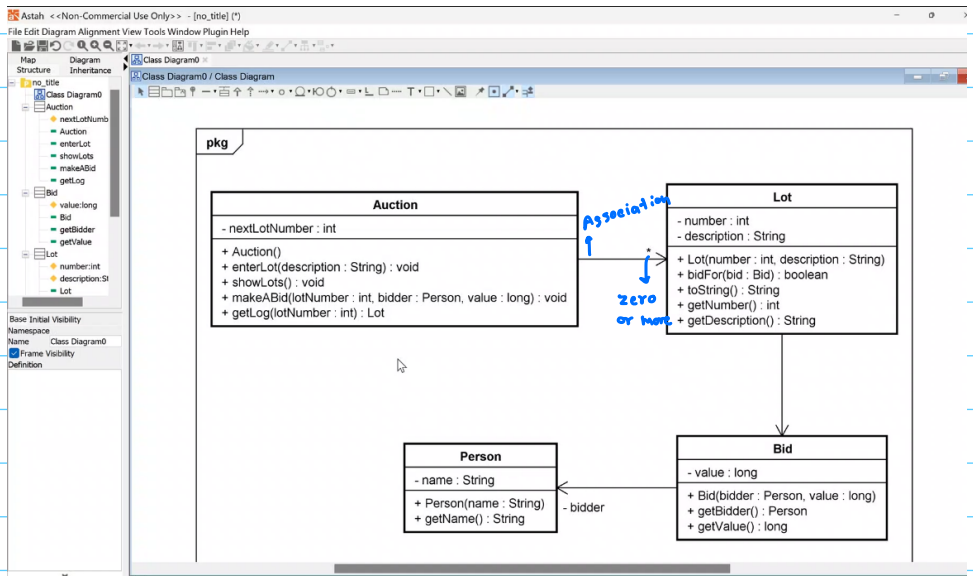
Refactoring = modify existing code, to improving quality.

Refactoring: Modify the existing s.t. the quality is better (no new feature added)
- step1: Features Freezing (no new development in code)
- step2: Refactoring
- Step 3: Regression Testing [If test is done once, after modify, with the same test, it should work correctly.

1 World of Zuul' example in Book

Inheritance can improve code quality as well

From Provided Code, Can Draw UML



Sukanya Chinwicha
63130500228

UML step:

1. Draw Relationship between obj.
2. Add variables and method for each obj.

Question: Generally, we draw UML before coding or coding draw UML, then refactor?

Sukanya Chinwicha
6313050022

8 Feb 2024

SRP - Single Responsibility Principle
- A class should have one, and only one, reason to change.

OCP - open/closed principle
- should be open for extension, but closed for modification.
- to achieve OCP:
  - → Through inheritance
  - → Through composition

LSP - Liskov Substitution Principle
- overridden method (belongs to superclass) of a subclass (overriding method belongs to subclass) → same signature

ISP - Interface Segregation Principle
- to reduce the side efx and frequency of required changes

DIP - Depency Inversion Principle
- both should depend upon abstractions.

Hollywood Principle
- the high-level components tell the low-level components "don't call us, we'll call you".

SOLID
- S: Single Responsibility Principle (SRP)
- O: Open-Closed Principle (OCP)
- L: Liskov Substitution Principle (LSP)
- I: Interface Segregation Principle (ISP)
- D: Dependency Inversion Principle (DIP)

Sukanya Chinwicha
6313050022B

15 February 2024

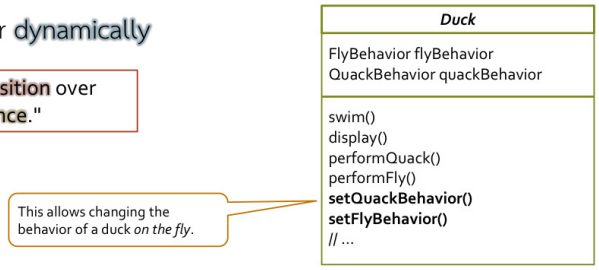
Design Pattern

Strategy

1. decouples interface from implementation
2. shields client from implementation
3. Client configure the Context [Context is not aware which strategy is being used]
4. strategies can be substituted at runtime

Setting behavior dynamically

"Favor composition over inheritance."



Duck Pond Simulation Game: think about SOLID principles

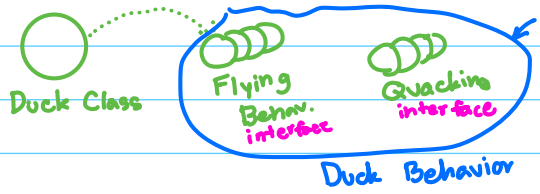
Beauty in OO Design = Simplicity

If it looks messy or complicated, try to refactoring

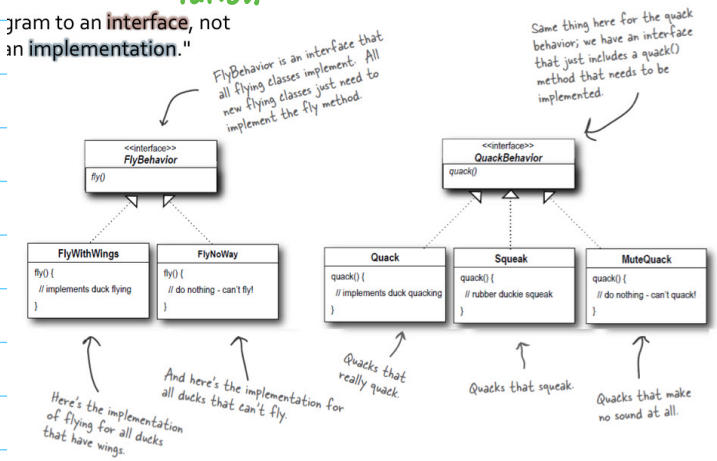
Key Design Principle: identify the aspects of your app. that vary and seperate them from what stays the same.

↓

Get fewer unintended consequences from code changes + more flexibility in system.



→ Program to an interface, not an implementation



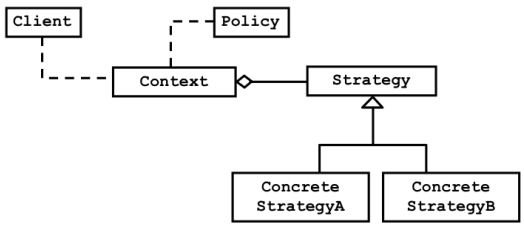
Favor Composition over inheritance

Strategy (aka: Policy) Pattern

1. defines a family of algorithm
2. encapsulates each one
3. make them interchangeable

Strategy lets the algorithm vary independently from clients that use it.

- Strategy Pattern: Structure



Strategy Pattern

- Name
  - Strategy (aka Policy)
- Applicability
  - many related classes differ only in their behavior
  - many different variants of an algorithm
  - need to encapsulate algorithmic information

Update strategy

Semantic \ Qualities	Design abstraction (i.e. decoupling)	Efficiency	Simplicity
Push	bad	good	good
Pull	good	bad	good
Change Manager	good	good	bad

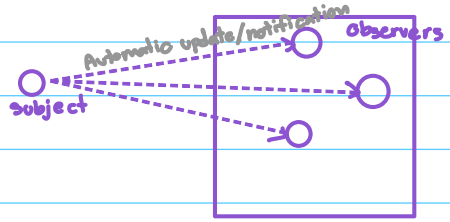
Sukanya Chinwicha
6313050022

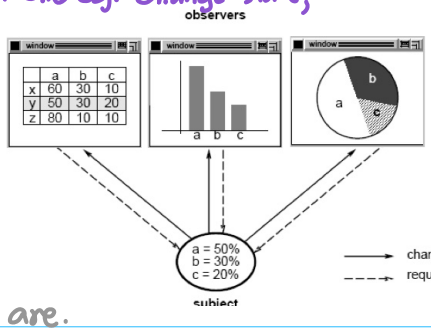
Design Patterns Observer: Keeping Your Obj. in the know

← Observer Pattern [Subject-Observer (Publisher-Subscriber)]

use when
- abstraction has two aspects, one dependent on the other
- change one, and require change others

- defines a one-to-many dependency between objects so that when one obj. change state, all of its dependents are notified and updated automatically





- obj. should notify other obj. without making assumptions about who these obj. are.
- Strive for loosely coupled designs between objects that interact
  When two objects are loosely coupled, they can interact, but have very little knowledge of each other.
- Consequences
  + Modularity: subject & observers may vary independently
  + Extensibility: can define and add any number of observers
  + Customizability: different observers provide different views of subject.
  - Because observers don't know about each other, a simple update to an observer might cause a long chain of other updates.
  - Update overhead: might need hints... Complicated observers have to do a lot of work to figure out what changed when they recieve a notification

---

Factory Pattern Design

- defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Depency Inversion Principle: DIP

- Depend upon abstractions. Do not depend upon concrete classes
- Guidelines:
  1. No variable should hold a reference to a concrete class.
  2. No class should derive from a concrete class.
  3. No method should override an implemented method of any of its base classes.

Factory Method Pattern

- Name: Factory Method (aka Virtual Constructor)
- Intent: define an interface for creating an object, but let subclass decide which class to instantiate
- Applicability: use the Factory Method pattern when
  1. a class cannot anticipate the class of objects it must create
  2. a class wants its subclasses to specify the objects it creates
  3. classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge

Sukanya Chinwicha
6313050022B

of which helper subclass is the delegate.

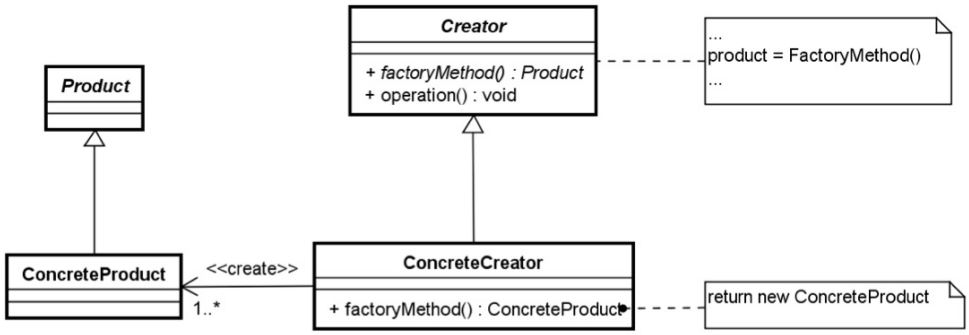
•Participants

1. Product: defines the interface of objects the factory method creates.
2. ConcreteProduct: Implements the product interface.
3. Creator:
   - → Declares the factory method, which returns an object of the product
   - → May also define a default implementation that returns a default ConcreteProduct object.
4. ConcreteCreator: overrides the factory method to return an instance of a ConcreteProduct.

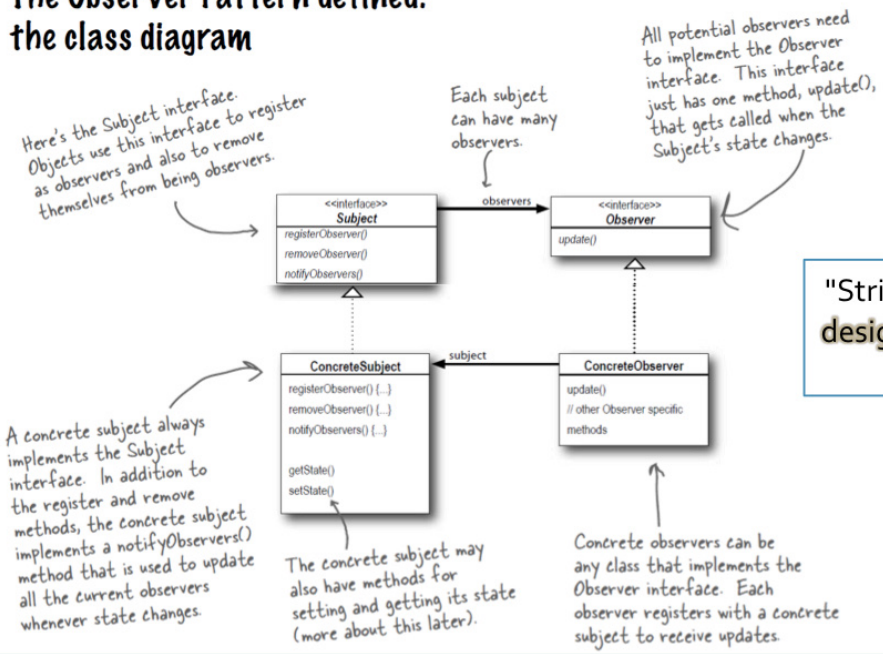
-Consequences

+ Eliminate the need to bind application-specific classes into the code (client)
  + Client deals with the product interface: therefore, it can work with any user-defined ConcreteProduct classes
- Clients might have to subclass the Creator class just to create a particular ConcreteProduct object
  - Fine if the client to subclass the Creator class anyway

+ Provide hooks for subclasses
  + The factory method is called only by Creators.
  + Creating objects with a factory method is always more flexible than creating an object directly.
  + Factory Method gives subclasses a hook for providing an extended version of an object.



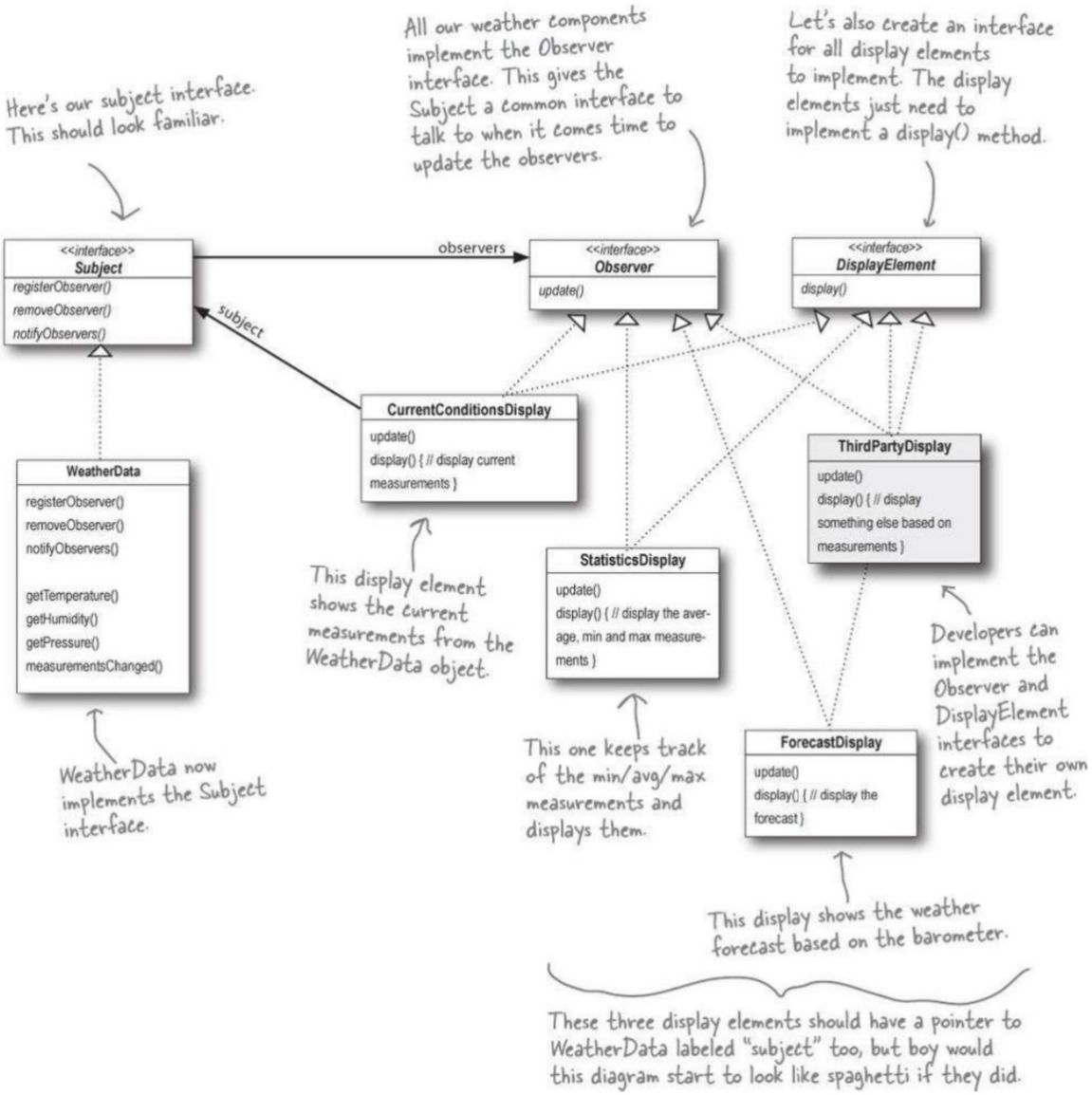
The Observer Pattern defined: the class diagram

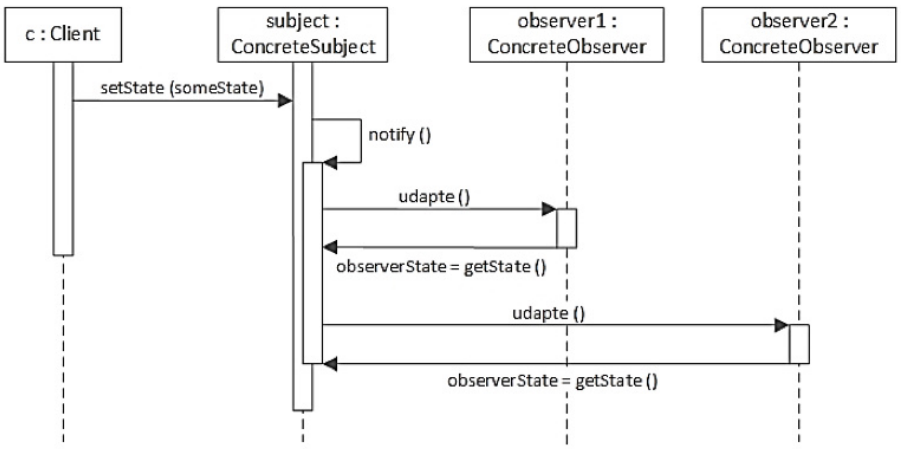


"Strive for **loosely coupled designs** between objects that interact."

When two objects are ***loosely coupled***, they can interact, but have very **little knowledge** of each other.

Sukanya Chinwicha
6313050022B





Sukanya Chinwicha
6313050022

Review : Adv. JAVA Block 1.

Week 1 Designing classes

Main concept: Responsibility-driven design, Coupling, Cohesion, Refactoring

Software is continuously changed, that cannot be maintained will die.

Code Quality concept: 4 Pillars of OOP can help

1. Coupling = links between seperate units of a program [tightly coupled = two classes depend closely on many details of each other]
   - aim for loose coupling
     - to underestand one class without reading others.
     - to change one class without affecting others.
     - ∴ improves maintainability
2. Cohesion = number and diversity of tasks that a single unit is responsible for.
   - aim for high cohesion [each unit is responsible for one single logical task][applies to classes/method]
     - easier to understand what a class or method does
     - easier to use descriptive names
     - easier to reuse classes or methods
   - for classes: classes represent one single, well defined entity
   - for methods: method should be responsible for one and only one well defined task

Warning!! Code Duplication >> indicator of bad design → maintainance harder

Good quality code = avoid duplication, high cohesion, low coupling

Responsibity-driven design: RDD

- leads to low coupling
- each class should be responsible for manipulating its own data
- class that owns the data should be responsible for processing it

↓coupling + RDD to→ localize change [when change, few classes as possible should be affected]

thinking ahead to make those changes easy

Refactoring

- classes are maintained → often code is added → classes&methods become longer → TIME TO REFACTOR [maintain cohesion and low coupling]
- NOTE:
  1. When refactoring code, seperate the refactoring from making other changes
  2. do refactoring only, without changing functionality.
  3. test before and after refactoring to ensure nothing was broken.

Sukanya Chinwicha
6313050022B

Design Guidelines

- A method is too long if it does more than one logical task
- A class is too complex if it represents more than one logical entity.

Enumerated Types

- language feature
- use enum instead of class to introduce a type name
- Simplest use is to define a set of significant names
- Basic enumerated type

```
public enum CommandWord
{
    // A value for each command word,
    // plus one for unrecognised commands.
    GO, QUIT, HELP, UNKNOWN;
}
```

- each name represents an object of the enumerated type, e.g., CommandWord.HELP
- enumerated objects are not created directly by the programmer

Highlight Code must be understandable and maintainable

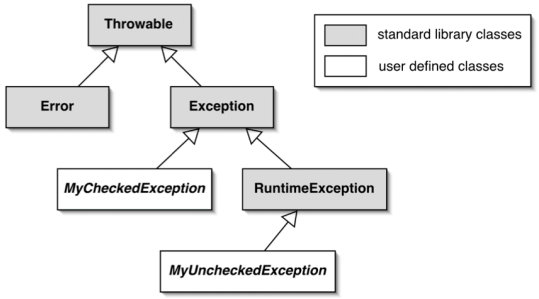
Exception Handling

Exception-throwing principles

- No special return value needed
- Error cannot be ignored in the client
  - normal flow-of-control is interrupted
- Specific recovery actions are encouraged.
- An exception object is constructed: new ExceptionType ("...")
- The exception object is thrown: thrown
- Javadoc documentation: @throws ExceptionType reason

```
/**
 * Look up a name or phone number and return the
 * corresponding contact details.
 * @param key The name or number to be looked up.
 * @return The details corresponding to the key,
 *         or null if there are none matching.
 * @throws IllegalArgumentException if
 *         the key is invalid.
 */
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new IllegalArgumentException(
                        "null key in getDetails");
    }
    return book.get(key);
}
```

Exception class hierarchy



Exception categories:

1. Checked exceptions use for anticipated failures where recovery may be possible
2. Unchecked exceptions use for unanticipated failure where recovery is unlikely

Effect of an exception

- Throwing method finishes prematurely
- No return value is returned
- Control does not return to the client's point of call
- Client may catch an exception

Sukanya Chinwicha
6313050022

Unchecked Exceptions

- Cause program termination if not caught
- IllegalArgumentException is a typical example
- Argument checking

```
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new IllegalArgumentException(
                        "null key in getDetails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
                    "Empty key passed to getDetails");
    }
    return book.get(key);
}
```

- Preventing object creation

```
public ContactDetails(String name, String phone, String address)
{
    if(name == null) {
        name = "";
    }
    if(phone == null) {
        phone = "";
    }
    if(address == null) {
        address = "";
    }

    this.name = name.trim();
    this.phone = phone.trim();
    this.address = address.trim();

    if(this.name.length() == 0 && this.phone.length() == 0) {
        throw new IllegalStateException(
                "Either the name or phone must not be blank.");
    }
}
```

Defining new exceptions

- Extend RuntimeException for an unchecked or Exception for a checked exception
- Define new types to give better diagnostic information – include reporting and/or recovery information

```
public class NoMatchingDetailsException extends Exception
{
    private String key;

    public NoMatchingDetailsException(String key)
    {
        this.key = key;
    }

    public String getKey()
    {
        return key;
    }

    public String toString()
    {
        return "No details matching '" + key +
               "' were found.";
    }
}
```

Assertions

- used for internal consistency checks
- used during development and normally removed in production version E.g. compile time

Exception handling

- Checked exceptions are meant to be caught and responded to
- Used properly, failures may be recoverable

Methods throwing a checked exception must include a Throws clause

```
public void saveToFile(String destinationFile)
    throws IOException
```

Clients catching an exception must protect the call with a Try statement

```
try {
    Protect one or more statements here.
}
catch(Exception e) {
    Report and recover from the exception here.
}
```

1. Exception thrown from here

```
try {
    addressbook.saveToFile(filename);
    successful = true;
}
catch(IOException e) {
    System.out.println("Unable to save to " + filename);
    successful = false;
}
```

2. Control transfers to here

Catching multiple exceptions

```
try {
    ...
    ref.process();
    ...
}
catch(EOFException e) {
    // Take action on an end-of-file exception.
    ...
}
catch(FileNotFoundException e) {
    // Take action on a file-not-found exception.
    ...
}
```

Multi-catch

```
try {
    ...
    ref.process();
    ...
}
catch(EOFException | FileNotFoundException e) {
    // Take action appropriate to both types
    // of exception.
    ...
}
```

Finally clause

```
try {
    Protect one or more statements here.
}
catch(Exception e) {
    Report and recover from the exception here.
}
finally {
    Perform any actions here common to whether or
    not an exception is thrown.
}
```

- executed even if a return statement is executed in try or catch clauses.
- Uncaught or propagated exception still exits via the finally clause

Sukanya Chinwicha
6313050022B

Java Assertion Statement

- two forms:
  - assert boolean-expression
  - assert boolean-expression : expression
- AssertionError is thrown if the expression evaluates to false

```
public void removeDetails(String key)
{
    if(key == null){
        throw new IllegalArgumentException("...");
    }
    if(keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
    assert !keyInUse(key);
    assert consistentSize() :
           "Inconsistent book size in removeDetails";
}
```

- Guidelines:
  1. use for internal checks
  2. remove from production code
  3. don't include normal functionality

  e.g. assert book.remove(name) != null;
  // incorrect use

Error recovery

- clients should take note of error notifications
  - check return values
  - don't ignore exceptions
- include code to attempt recovery
  - often require a loop
- attempting recovery

```
// Try to save the address book.
boolean successful = false;
int attempts = 0;
do {
    try {
        contacts.saveToFile(filename);
        successful = true;
    }
    catch(IOException e) {
        System.out.println("Unable to save to " + filename);
        attempts++;
        if(attempts < MAX_ATTEMPTS) {
            filename = an alternative file name;
        }
    }
} while(!successful && attempts < MAX_ATTEMPTS);

if(!successful) {
    Report the problem and give up;
}
```

Error avoidance

- clients can often use server query methods to avoid errors
  - more robust clients mean servers can be more trusting
  - unchecked exceptions can be used
  - simplifies client logic
- may increase client-server coupling

File output - stages: open, write, close

```
try {
    FileWriter writer = new FileWriter("name of file");
    while(there is more text to write) {
        ...
        writer.write(next piece of text);
        ...
    }
    writer.close();
}
catch(IOException e) {
    something went wrong with accessing the file
}
```

Try-with-resource - ensures 'resources' are closed after use.

```
try(FileWriter writer = new FileWriter("name of file")) {
    while(there is more text to write) {
        ...
        writer.write(next piece of text);
        ...
    }
}
catch(IOException e) {
    something went wrong with accessing the file
}
```

No close() call required in either clause.
See the *weblog-analyzer-v7* project.

- removes need for explicit closure on both successful and failed control flows.
- aka 'automatic resource management' ARM

Text input from file

```
try {
    BufferedReader reader =
        new BufferedReader(new FileReader("filename"));
    String line = reader.readLine();
    while(line != null) {
        do something with line
        line = reader.readLine();
    }
    reader.close();
}
catch(FileNotFoundException e) {
    the specified file could not be found
}
catch(IOException e) {
    something went wrong with reading or closing
}
```

See *tech-support-io*

Highlight

- Key classes for text input/output are FileReader, BufferReader, FileWriter, Scanner
- open, read, close file

```
// Use the correct method to put details
// in the contacts list.
if(contacts.keyInUse(details.getName() ||
        contacts.keyInUse(details.getPhone()) {
    contacts.changeDetails(details);
}
else {
    contacts.addDetails(details);
}
```

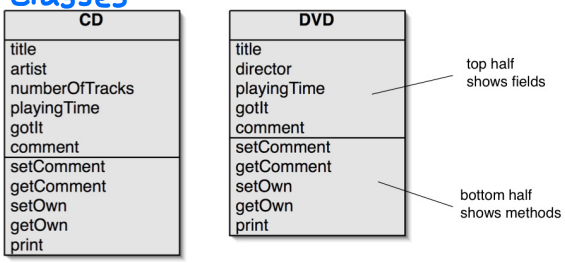
The `addDetails` method could now throw an *unchecked* exception.

Sukanya Chinwicha
6313050022B

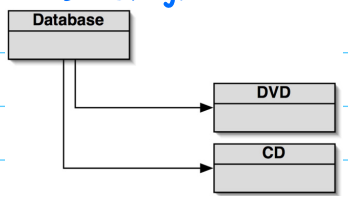
Improving Structure With Inheritance

DoME = Database of Multimedia Entertainment

Classes



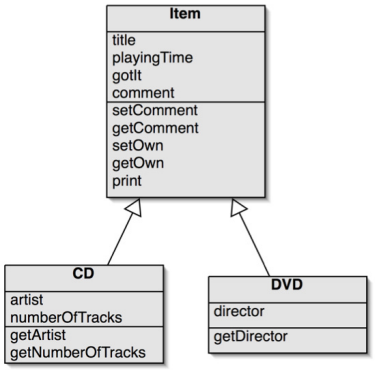
Class diagram



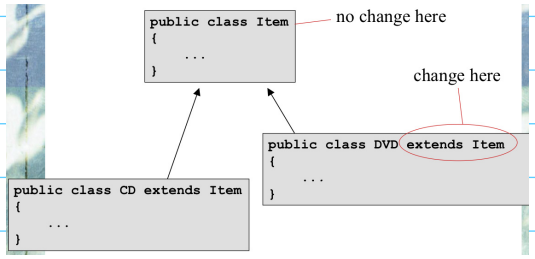
Critique of DoME

- code duplication
  - CD and DVD classes very similar
  - makes maintenance difficult/more work
  - introduces danger of bugs through incorrect maintenance
- code duplication also in Database class

Using Inheritance



- the superclass defines common attributes
- the subclasses inherit the superclass attributes
- the subclasses add own attributes



- subclass constructors must always contain a 'super' call

To conclude, inheritance helps with

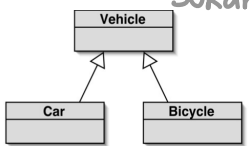
1. avoiding code duplication
2. code reuse
3. easier maintenance
4. extendibility

- Declared type of a variable is its static type
- Type of the object a variable refers to is its dynamic type

Using Subtyping

- subclasses define subtypes.
- objects of subclasses can be used where objects of supertypes are required = substitution

Sukanya Chinwicha
6313050022 8



- subclass objects may be assigned to superclass variables

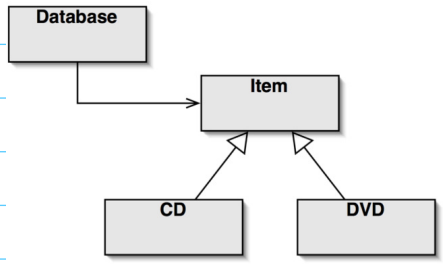
```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```

- subclass objects may be passed to superclass parameters

```
public class Database
{

    public void addItem(Item theItem)
    {
        ...
    }
}

DVD dvd = new DVD(...);
CD cd = new CD(...);

database.addItem(dvd);
database.addItem(cd);
```

subclass objects may be passed to superclass parameters

Class diagram



Using Polymorphic Variables

- polymorphic = hold objects of more than one type

Casting :

- used to overcome 'type loss'
- ClassCastException to ensure object really is of that type
- use it sparingly

- Object variables are polymorphic
- All collections are polymorphic

Wrapper classes

- Primitive types ≠ objects MUST be wrapped into an object
- Wrapper classes exist for all simple types e.g. int-Integer, float-Float, char-Character, ...

```
int i = 18;
Integer iwrap = new Integer(i);   <-- wrap the value
...
int value = iwrap.intValue();     <-- unwrap it
```

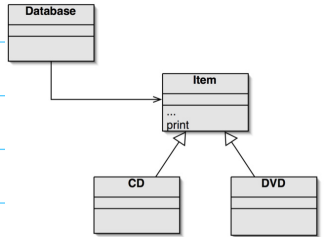
```
private ArrayList<Integer> markList;
...
public void storeMark(int mark)
{
    markList.add(mark);   autoboxing
}
```

```
int firstMark = markList.remove(0);   unboxing
```

→ don't often have to do this

Sukanya Chinwicha
63130500228

Exploring Conflicting output polymorphism



What we want

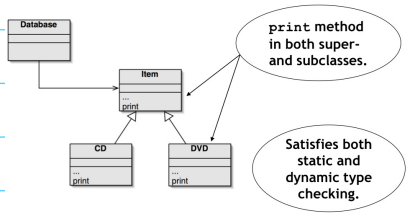
```
CD: A Swingin' Affair (64 mins)*
    Frank Sinatra
    tracks: 16
    my favourite Sinatra album

DVD: O Brother, Where Art Thou? (106 mins)
     Joel & Ethan Coen
     The Coen brothers' best movie!
```

What we have

```
title: A Swingin' Affair (64 mins)*
       my favourite Sinatra album

title: O Brother, Where Art Thou? (106 mins)
       The Coen brothers' best movie!
```

Inheritance is a one-way street: superclass knows nothing about its subclass's fields.

Solution



Overriding

- super- and subclass define methods with same signature.
- superclass satisfies static type check
- subclass method is called at runtime
  → overrides superclass version

Method Polymorphism

- Polymorphic variable can store objects of varying types.
- Method calls are polymorphic

- polymorphism and overriding - the first version found is used
- method lookup - start with dynamic type
  - variable is accessed
  - object stored in the variable is found
  - class of the object is found
  - class is searched for a method match
  - If no match is found, the superclass is searched
  - repeat until a match is found, or the class hierarchy is exhausted

Object class's methods

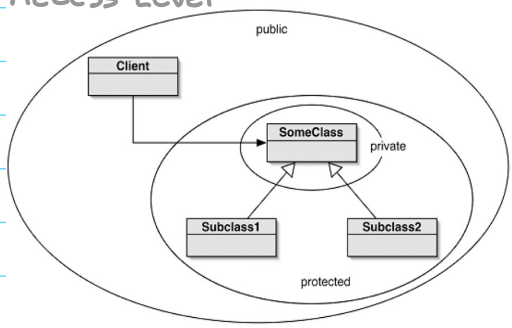
- Methods in Object are inherited by all classes
- Any of these may be overridden e.g. toString() method

- overridden method can be called from the method that overrides it
  - super.method(...)

Private >> too restrictive for a subclass

Protected access >> more restricted than private
>> still keeping fields private

Access Level



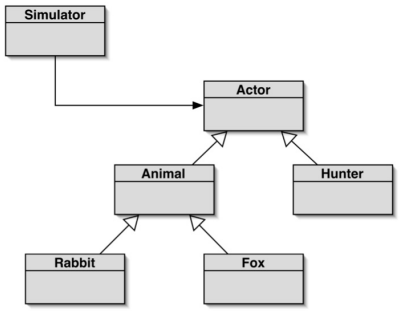
Sukanya Chinwicha
6313050022B

Abstract classes and interfaces

Abstract classes and methods

- abstract methods - have no body
  - make the class abstract
  - allow static type checking without requiring implementation
- support polymorphism
- abstract classes - cannot be instantiated
- concrete subclasses complete the implementation

```
public abstract class Animal
{
    fields omitted

    /**
     * Make this animal act - that is: make it do
     * whatever it wants/needs to do.
     */
    abstract public void act(Field currentField,
                             Field updatedField,
                             List<Animal> newAnimals);

    other methods omitted
}
```



A class inherits directly from multiple ancestors

- Java forbids it for classes
- Java permits it for interfaces

```
public interface Actor
{
    /**
     * Perform the actor's daily behavior.
     * Transfer the actor to updatedField if it is
     * to participate in further steps of the simulation.
     * @param currentField The current state of the field
     * @param updatedField The updated state of the field
     * @param newActors New actors created as a result
     *                  of this actor's actions.
     */
    void act(Field currentField, Field updatedField,
             List<Actor> newActors);
}
```

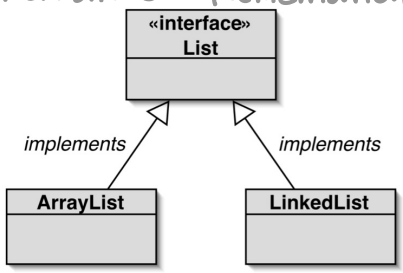
```
public class Fox extends Animal implements Drawable
{
    ...
}

public class Hunter implements Actor, Drawable
{
    ...
}
```

Implenting classes - are subtypes of the interface type.
fully abstract
- provide specification without implementation

Features of interfaces

- all methods are abstract
- no constructors
- all methods are public
- all fields are public, static and final

Alternative implementations

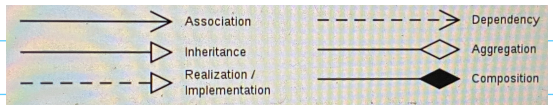


Concrete and abstract classes - inheritance can provide shared implementation

Classes and interfaces - inheritance provides shared type information

Code smells - discernable evidences on code that will likely cause issues and lower overall code quality [opposite code quality]

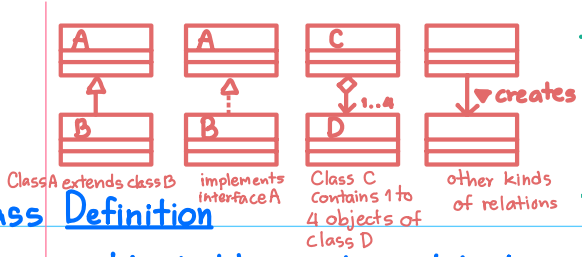
Anti-patterns - opposite "design patterns"
- focus "bad designs" should be avoided



Refactoring - process improving code quality
- "Code freeze" → refactoring → regression tests

Static Code Analysis (SCA) tools - analyze source code, based on rules, looking for code smells, security flaws, and other issue

e.g. Selenium, Sonar toolsuite



Sukanya Chinwicha 6313050022

Class Diagram Primer

Definition

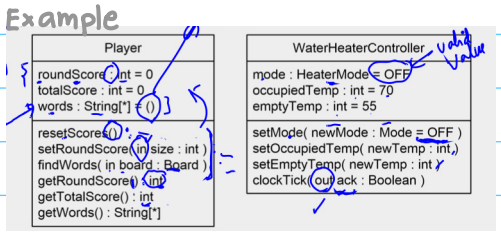
- object (blue-print) and instance (virtual copy) – entity that holds data and exhibits behavior
- class – abstraction of a set of objects with "common operations and attributes"
- attribute – data item held by an object or class
- operation – object or class behavior
- association – connection between classes representing a relation on the sets of instances of the connected classes

UML Class Symbol

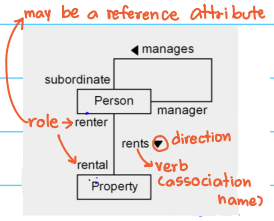
UML Modeling Tool: CASE, Visual Paradigm, Astah, Smartdraw

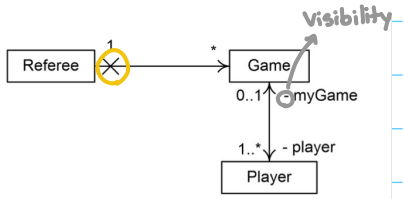
- Compartments
  - class name
  - attributes
  - operations
  - others

Example



- Association Lines → Association Navigability
  - properties:
    1. labeled or unlabeled lines
    2. readable in two directions
    3. direction arrows
    4. rolenames

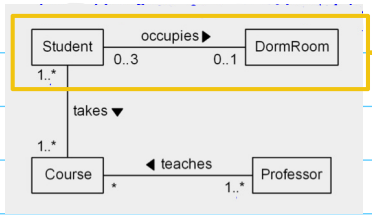




A can access B = non-navigible (get service from)

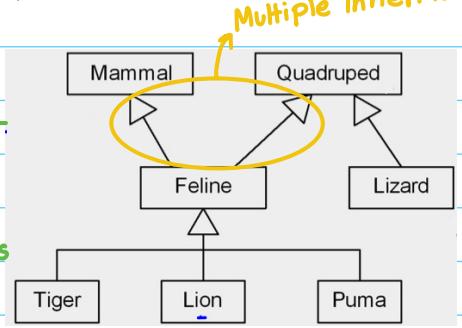
if it cannot access

- Association Multiplicity: multiplicity at the target class end of an association is the number of instances of the target class that can be associated with a single instance of the source class
  - 0..1 = optional
  - 0..3
  - 1..* = 1 or more
  - 0..* = 0 or more



Each students can occupy 0..1 dorm room
Each dorm rooms can be occupied by 0..3 students

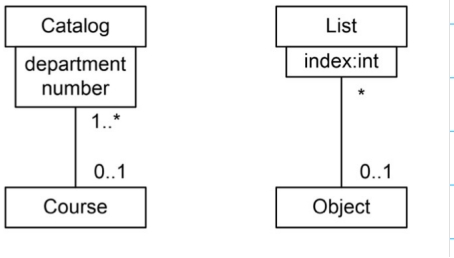
- Generalization Is-a or kind-of relationship
  - child → Parent (child is a special type of the parent)
    - child: More specialized
    - Parent: More generalized
  - used in UML class diagrams to model inheritance



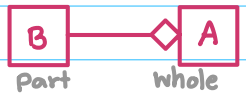
Generalization	vs	Association
• relation btw. classes		• relation on sets of class instances designated by the associated classes
• Never have multiplicities		
• Never have rolenames		
• Never have names		

Sukanya Chinwicha
6313050022 8

- **Association class** relation on instances of the classes it connects, and it holds data and behavior
  - connector is ------- dashed line
  - only one instance of an association class for each pair
  - if more instances are needed, interpose a new class (reified association)
- **Association Qualifier**
  - one or more association attributes that, together with instances of the qualified class, pick out instances of an associated target class
  - represent by box with association attributes and a line
    - attached to qualified class. line runs to target class + can have association adornments

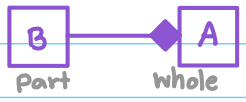


- **Aggregation association**
  - the part-whole relation between classes

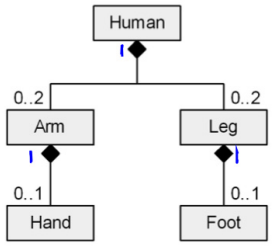


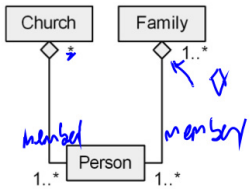
A has B = B is a part of A

- **Composition association**
  - an aggregation association in which each part can be related to only one whole at a time

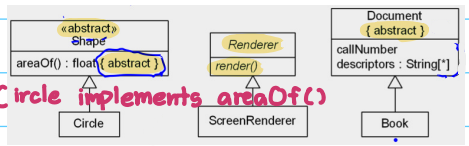


A owns B





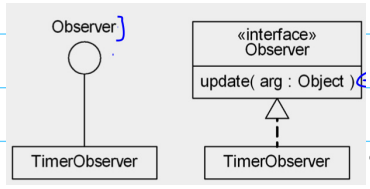
- **Abstract operation**
  - properties:
    1. an operation without body – concrete operation has
    2. a class that cannot be instantiated – concrete class does
    3. a class MUST be abstract ⟷ has an abstract operation
    4. a class may be abstract even if it has no abstract operation
    5. abstract class forces its subclasses to implement certain operations.
  - in UML, represents by *classname*, «abstract» classname, classname {abstract}



Circle implements areaOf()

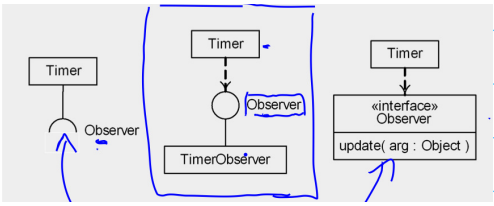
Sukanya Chinwicha
6313050022B

- Interfaces
  - collection of public attributes and abstract operations
  - Types of notation:
    1. provided interfaces - realized by class or component [ball symbol/stereotyped class icon]

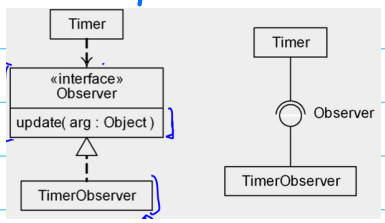


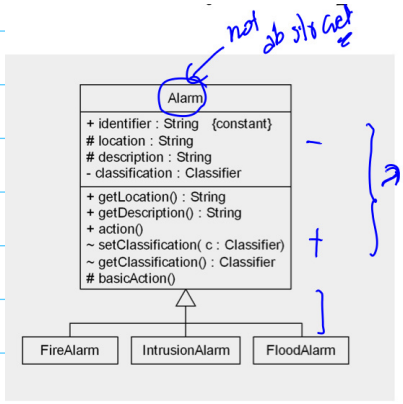
TimeObserver is implementing the Observer interface.

    2. required interfaces - needed by class or component [socket symbol/dependency arrow to ball/or to stereotyped class icon]

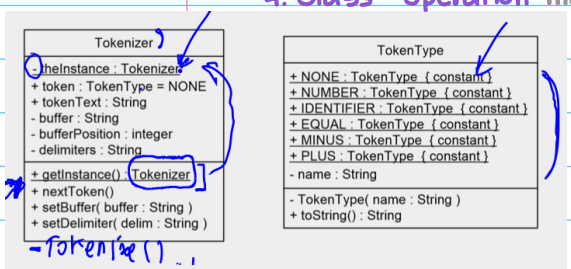


    3. Assembly notations





- Feature Visibility
  - Public
    - + : visible anywhere that the class in which it appears is visible
  - Package
    - ~ : visible anywhere in the package containing the class in which it appears
  - Protected
    - # : visible in the class in which it appears and all its sub-classes
  - Private
    - − : visible only in the class in which it appears
- Static
  - indicated by underlining
  - Types:
    1. instance variable attribute whose value is stored by each instance of a class
    2. class variable attribute whose value is stored only once and shared by all instances
    3. instance operation must be called through an instance
    4. class operation may be called through the class

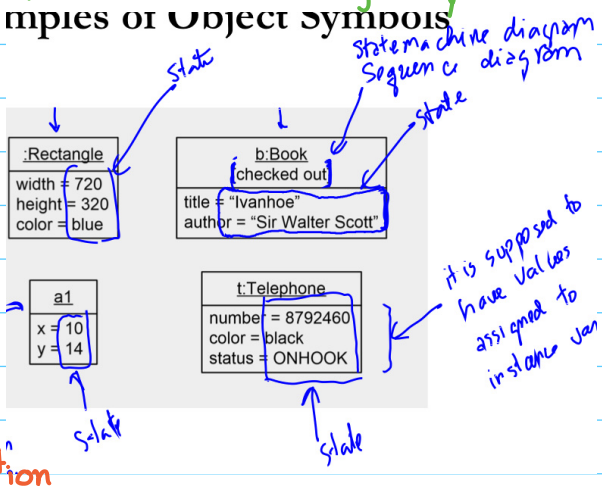


Sukanya Chinwicha
6313050022 8

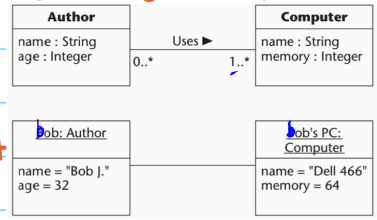
- Class Diagram Rules
  - Class symbols must have a name compartment
  - compartments must be in order
  - attributes and operation specifications must be syntactically correct
- Class Diagram Heuristics 1.1
  - name classes, attributes, and roles with noun phrases
  - name operations and associations with verb phrases
  - capitalize class names only
  - center class and compartment names but left-justify other compartment contents
- Class Diagram Heuristics 1.2
  - never place a name, rolenames, or multiplicities on a generalization connector
  - «abstact» or {abstract} to indicate when drawing by hand; italic font when draw on computer
  - interface ball and socket symbols to abstract interface details and a stereotyped class symbol to show details
- Class Diagram Heuristics 2
  - Don't italicize interface or operation names
  - avoid aggregation and composition
- Class Diagram Heuristics 3
  - if aggregation and composition, check is the represented relation is transitive

- Object Diagram
  - compartments
    1. Object name
    2. attributes
  - syntax:

    objName:ClassName
  - object links
    - instances of association
    - use solid line (no arrow heads), underlined association name



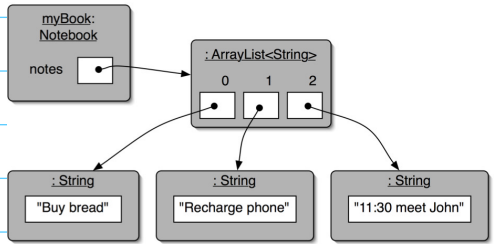
class / object



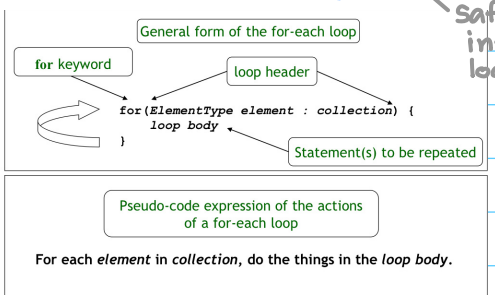
Sukanya Chinwicha
6313050022B

Collections Framework — Collection aka. parameterized or generic types

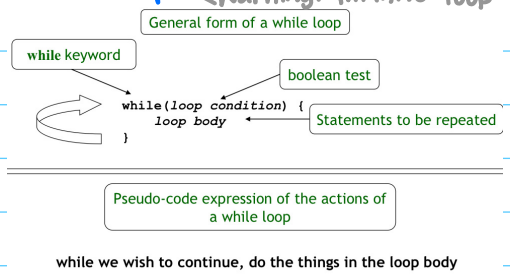
- features
  1. it increases its capacity as necessary
  2. it keeps a private count
  3. it keeps the object in order
  4. details of how all this is done are hidden



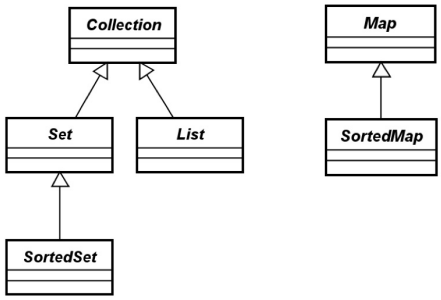
- for each element in collection (easier, safer from infinite loop)



- while-loop (No need to process whole collection, Not use with collection, Warning: infinite loop)



- JAVA Collection Framework



Elements of documentation

The documentation for each constructor and method should include:

- the name of the method
- the return type
- the parameter names and types
- a description of the purpose and function of the method
- a description of each parameter
- a description of the value returned

Elements of documentation

Documentation for a class should include:

- the class name
- a comment describing the overall purpose and characteristics of the class
- a version number
- the authors' names
- documentation for each constructor and each method

Interface of the class

- documentation includes
  - class name
  - general description for class
  - a list of constructors and methods
  - return values and parameters for constructors and methods
  - a description of the purpose of each constructors and methods

Implementation of the class

- documentation does not include
  - private fields
  - private methods
  - the bodies (source code) for each method

Sukanya Chinwicha
6313050022

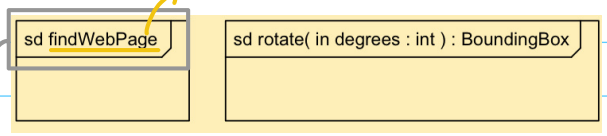
UML Sequence Diagram

Interaction diagram for modeling communication behavior of individuals exchanging information to accomplish some task

- **sequence diagram** – interacting individuals along the top and msg. exchange down the page
- **communication diagram** – msg. exchanged on a form of object diagram
- **interaction overview diagram** – a kind of activity diagram whose nodes are sequence diagram fragments
- **timing diagram** – individual state changes over time

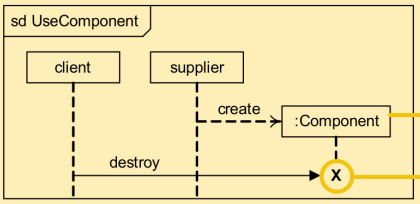
Sequence Diagram – product design / architectural design / mid-level design

- **Frame** — interaction Identifier



name compartment

- **Lifelines** : shows period when an individual exists

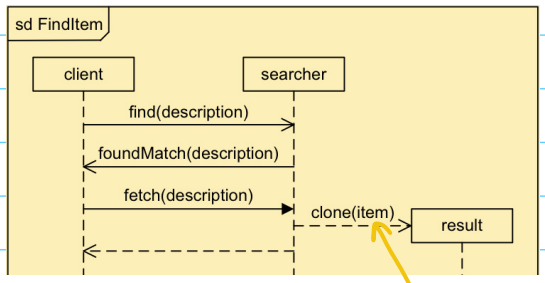


→ new obj. appears at the point it is created
→ destroyed obj. has a truncated lifeline

– lifeline identifier format: name [selector] : typeName
(or 'self'; either name, typeName, or both must appear !!)

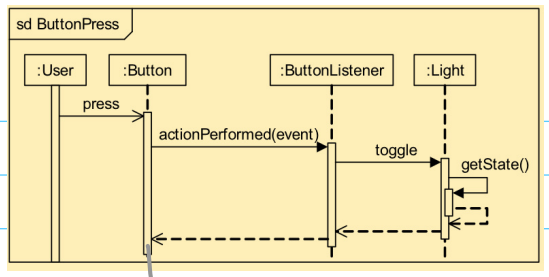
Status:
active = suspend
= execute
execute = running

- **Message Arrows**
  - ——→ : synchronous – execute until the msg. is complete : suspended
  - ——→ : asynchronous – continues execution after sending msg.
  - - - - - → : synchronous message return or instance creation



- **Message Specification Format:** variable = name argumenList
  - → varName = paramName
  - → paramName = argumentValue
  - → comma-seperated list of arguments in parentheses

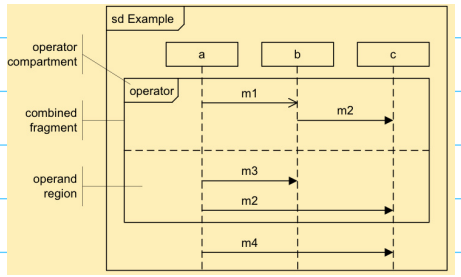
Sukanya Chinwicha
6313050022 8



execution occurence
=period when object is active

• Combined Fragment

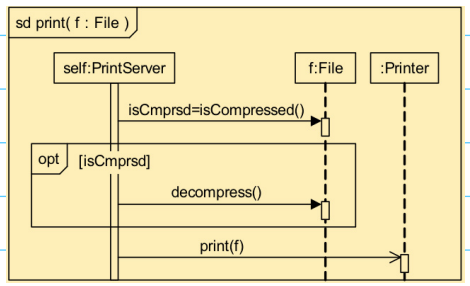
- Concurrent execution, loops, branching



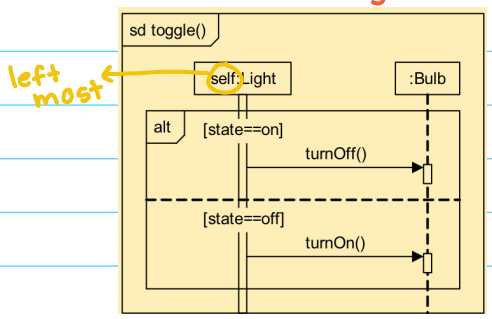
opt = operator

guard = [boolean]

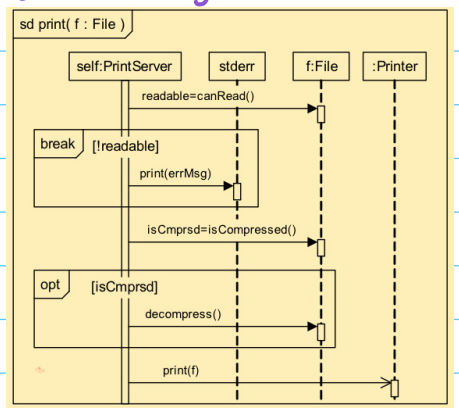
• Optional Fragment



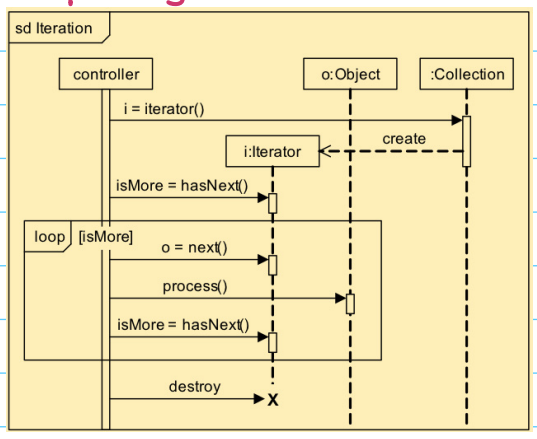
• Alternative Fragment



• Break Fragment



• Loop Fragment



Design Patterns = problem/solution pairs for recurring OO problems

- Conceptual frame of approach to solve recurring problems
- language-independent
- favor delegation over inheritance
- often favor some design quality over the others

Sukanya Chinwicha
6313050022

→B create an object of another class, A, if
1. B contains A
2. B aggregates A
3. B has the initializing data for A
4. B records A
5. B closely uses A

Design Principle =for designing better software

GRASP - General Responsibility Assignment Software Principles by Craig Larman

1. Information expert assign a responsibility to class which has info. necess. to fulfill the resp.
2. Creator
3. Controller (aim) — Encapsulate a system operation; Provide a layer between UI and domain model
4. Indirection give resp. of interaction to an intermediate object so that the coupling among diffe. components remains low
5. Low coupling
6. High cohesion
7. Polymorphism
8. Protected variations
9. Pure fabrication a class not represent a concept in the problem domain

SOLID

1. S: Single Responsibility Principle (SRP)

- A class should have one, and only one, reason to change
- Pros:
  - easier to implement, understand, reuse
  - prevent unexpected side effects of future changes
- Check: ask "what the responsibity of the class/component is?" If answer includes "and", then SRP is violated

2. O: Open/Closed Principle (OCP)

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification – cannot update code, but add new code
- 2 approaches:
  - through Inheritance
  - through Composition

3. L: Liskov Substitution Principle (LSP)

- Derived classes must be substitutable for their base classes
- An overridden method (belongs to superclass) of a subclass must accept the same input parameter values as the method of superclass
- Similarly for return values

4. I: Interface Segregation Principle

- Clients should not be forced to depend upon interfaces that they don't use
- Easy to violate ISP
- splitting Sftw. into multiple, independent parts

Sukanya Chinwicha
6313050022 8

5. D: Dependency Inversion Principle (DIP)

- high-level modules should not depend upon low-level modules. Both should depend upon abstractions
- Each module should program to an interface

Hollywood Principle: Don't call us, we will call you

- similar to DIP
- observer pattern uses this principle
- prevent dependency rot (depends on in back-forth, upward-downward, etc.)

Design Pattern

Patterns = problems + solution pairs in a context

Design Pattern Parts

1. Name
2. Problem
3. Solution
4. Consequences and trade-offs of application

Goals

- Codify good design
  - aid to novices and experts alike
  - abstract how to think about design
- Give design structures explicit names
  - common vocabularly
  - reduced complexity
  - greater expressiveness
- Capture and preserve design info.
  - articulate design decisions succinctly
  - improve documentation
- Facilitate restructuring/refactoring
  - Additional flexibility
  - Patterns are interrelated

Architectures - model software structure at the highest possible level

Pattern - small-scale/local architectures description of a solution

Framework - partially completed software systems (employ several)

more primitive than ↘

Sukanya Chinwicha
6313050028

Pattern Template

1. **Name** text reflects the problem
2. **Problem addressed** intent of the pattern, objective achieved
3. **Context** circumstances under which it can occur
4. **Forces** constraints or issues solution should be addressed
5. **Solution** UML-like structure, abstract code [solve all forces]
6. **Consequences** results and tradeoffs

Stategy

- decouples interface from implementation
- shields client from implementations
- Context is not aware which strategy is being used; Client configures the Context
- strategies can be substituted at runtime

Singleton Pattern Template

Structure: