40.317 Lecture 7

Dr. Jon Freeman

9 June 2020

Slido Event #Q020

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Agenda

- Object-oriented programming
- Object-oriented design
  - Responsibility-driven design
  - SOLID
- OOP’s forgotten history

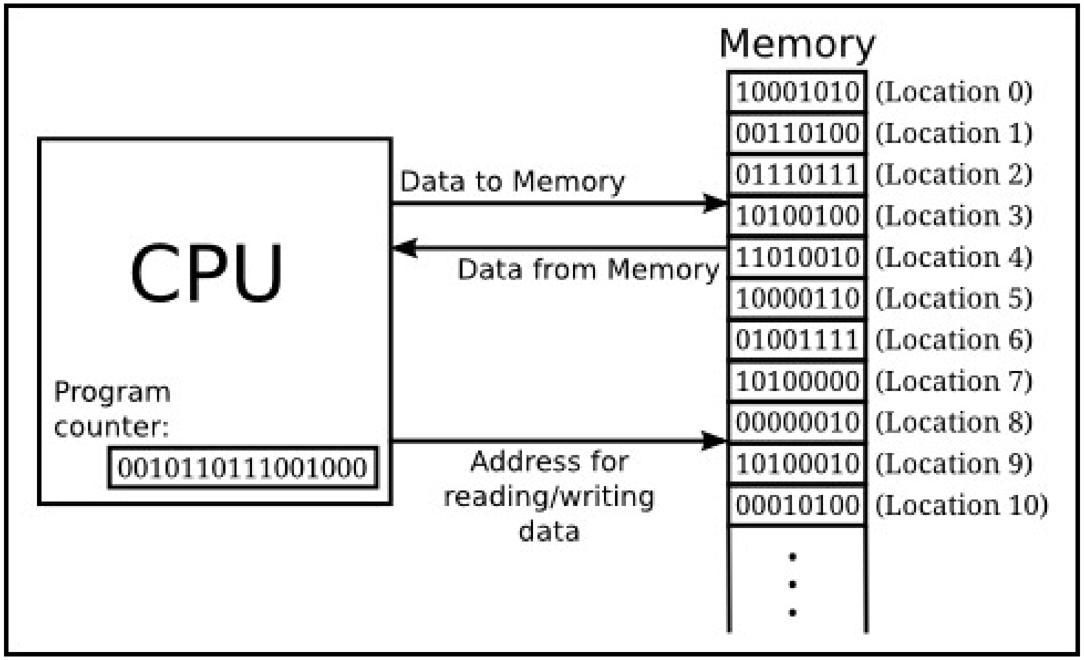
OOP: Motivation

Why are we studying this topic?

- OOP is one of the fundamental programming paradigms.
- *Applied well*, it *can* result in cleaner, more modular, more extensible code compared to “imperative” programming.
- It is the basis of most reusable software libraries in popular languages.
  - You cannot use `unittest` without it

What is *Imperative* Programming?

Imperative (or “*procedural*”) *programming* is an approach to writing software consistent with a model of a digital computer as a “random access machine.”



Source: http://aditio1997.blogspot.sg

A BETTER WORLD BY DESIGN.

Main Benefits of OO Design

The main benefits of a thoughtful OO design are:

- Modularity, via “encapsulation”
- Code reuse, via “inheritance”
- Flexibility, via “polymorphism”
- Closer modeling of the underlying problem domain

Getting Started: Key Artifacts

There are two key artifacts, objects and classes.

- **Object**: a model of the concepts, processes, or things in the real world that are meaningful in your system.

- **Class**: a template for generating a particular kind of object.

More About Objects

An object:

- *has a lifetime*
- *has internal state*
  - Some is private, the rest is public
- *knows how to do things*
  - Again, some of its functionality is private, the rest is public

In-Class Discussion

What are some objects you might expect to find in the trading game from Lecture 1?

(Let’s take a look!)

In-Class Discussion, continued

What are some additional objects you might expect to find in a financial system?

Finance
risk
debt
inflation
bonds
assets
investments
exchange rate
stock market
money
banks
tax
investor
private equity
financial
equity
loan
currency
hedge funds

Source: FreeImages.com

A BETTER WORLD BY DESIGN.

Constructors and Destructors

As we said, an object *has a lifetime*.

- The function which initialises a new object’s contents is called a constructor.
- Once an object is no longer needed, the function which frees its resources is called a destructor.

The language calls these functions *for you*.

- In general, you cannot be certain when a destructor will be called.

Instance Variables

As we said, an object *has internal state*.

This state resides in its *instance variables*.

- Instance variables can be objects themselves, of course.

Instance variables can be either *private* or *public*.

Methods

As we said, an object *knows how to do things*.

Its *methods* are special functions which can access and modify all of its instance variables, private as well as public.

Like instance variables, methods can be either private or public.

Getting Started: IS-A and HAS-A

How do we come up with an OO design? One of the first steps is to look for two simple relationships among the entities in our problem domain:

- **IS-A**
  "*x* is-a *y*" ⇨ *x* could be a subclass of *y*
- **HAS-A**
  "*x* has-a *y*" ⇨ *x* could be an instance variable of *y*

In-Class Exercise

For each of the following, indicate whether the relationship is IS-A, HAS-A, or neither:

- Student ___ person
- Car ___ total distance traveled
- Rainbow ___ list of colours
- Food ___ nutrition
- Temperature ___ humidity

Object-Independent Artifacts

- A *static* instance variable is an instance variable shared among every object of a class (past, present, and future).
- A *static* method is a method which only accesses its class's static instance variables, i.e. which can do nothing object-specific.

Next, the two most important OO concepts: inheritance and polymorphism.

Inheritance

Inheritance enables new objects to take on the properties of existing objects.

A class which serves as the basis for inheritance is called a superclass or base class.

A class which inherits from a superclass is called a subclass or derived class.

Subclasses are Really Similar

Each instance variable in a base class is also an instance variable in every subclass.

- A subclass can have *additional* instance variables.

Each method in a base class is also a method in every subclass.

- A subclass can have *additional* methods or <u>override</u> *existing* base class methods.

Inheritance and Privacy

A subclass's methods can access every instance variable in its base class, private as well as public.

A subclass's methods can call every method in its base class, private as well as public.

Single vs. Multiple Inheritance

A language supports multiple inheritance ("MI") if a subclass can have more than one base class. Otherwise it supports single inheritance ("SI").

Python supports MI. (N.B. Java and C# only support SI.)

Using MI can be problematic. We will not discuss it further, recommending Python's *mixins* instead.

Abstract Base Classes

An abstract base class ("ABC") is a class which cannot generate any objects.

- *Why is this useful?*

ABCs are extremely important for OO design: they serve as a *specification* or *contract* which any compatible class must fulfill, i.e. they describe everything a class must do to be fit for a certain purpose.

  – We will refer to "ABCs" and "interfaces" interchangeably.

An ABC Example in Python 3

To write a class supporting the full Set API, you only need to implement its three abstract methods:

- `__contains__(),`
- `__iter__(),` and
- `__len__().`

The ABC supplies (generic versions of) the remaining methods. Reference: https://docs.python.org/3/library/collections.abc.html

Polymorphism

Polymorphism is the ability to redefine, or “override,” the same method in one or more derived classes.

At run time, the language figures out *for you* which version of such a method to call.

A Working Example

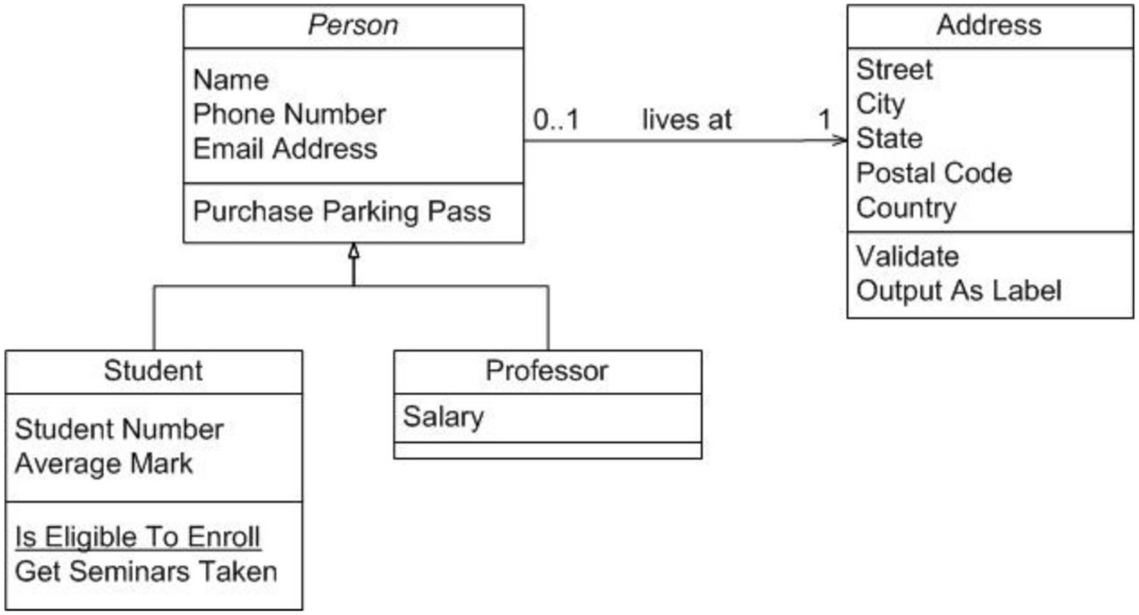
Demo: `shape_polymorphism.py`

We have come a long way from “peeking and poking” (i.e. the RAM model)!

1. Imperative: “Evaluate this formula for the area of a triangle, using these inputs.”
2. With encapsulation, via classes:
   “Hey `triangle461`, what is your area?”
3. With code reuse and flexibility, via inheritance and polymorphism:
   “Hey `shape283`, what is your area?”

UML Class Diagrams

There is a formal graphical notation for describing classes and their relationships, called UML (“Unified Modeling Language”) class diagrams.



Person

Name
Phone Number
Email Address

Purchase Parking Pass

0..1 lives at 1

Address

Street
City
State
Postal Code
Country

Validate
Output As Label

Student

Student Number
Average Mark

Is Eligible To Enroll
Get Seminars Taken

Professor

Salary

Source: http://www.agilemodeling.com/artifacts/classDiagram.htm

Object-Oriented Design: Motivation

Why are we studying this topic?

“[T]he critical design tool for software development is a mind well educated in design principles.” –Craig Larman

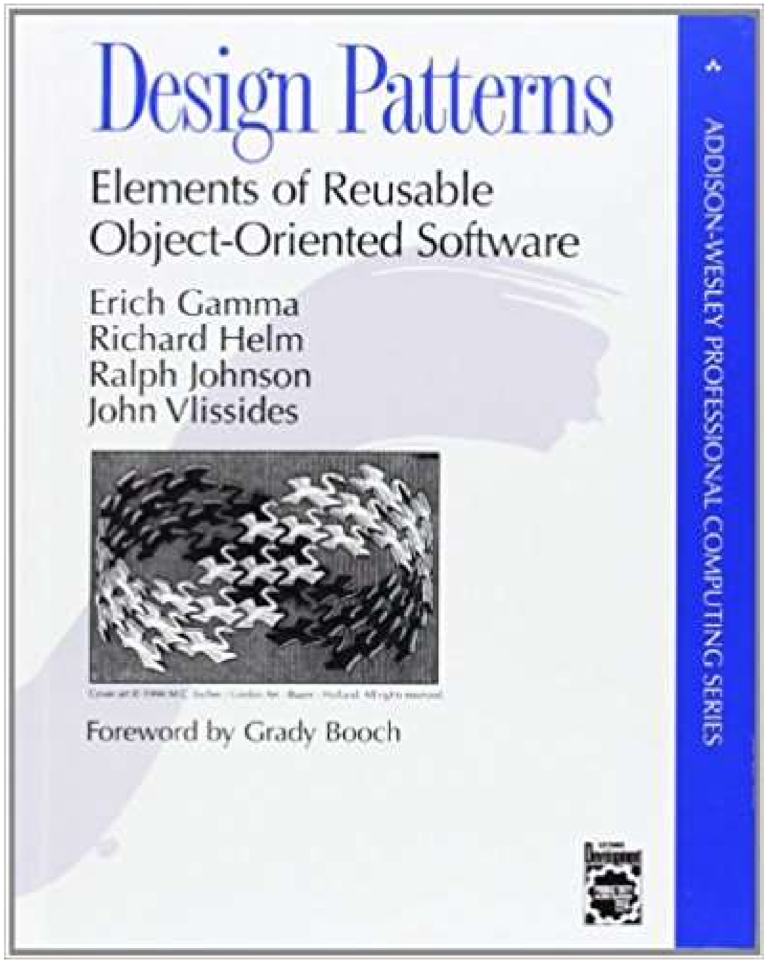
There are well-known OOD principles, and collections of principles, you should be familiar with.

OOD: Design Patterns

Object-oriented *design* really got going with the publication of this seminal 1994 book.

The original inspiration: a 1977 book on architecture and urban design (!).

DPs are language-independent, and make heavy use of ABCs.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Why We Need DPs, Briefly

- A class and its subclasses are closely related, i.e. tightly coupled.
- But in a typical software system, only a few classes are coupled this tightly; the rest are more loosely coupled.
- So we need much more than IS-A to model most problem domains well.

DPs are a *catalog* of *typical ways loosely coupled classes interact*.

So What Does a DP Look Like?

Let’s walk through one of them, namely Singleton, to get an idea:

https://www.oodesign.com/singleton-pattern.html

Can you think of a use for this pattern in the Lecture 1 trading game?

Further Study of DPs is out of scope

Work on DPs has progressed well beyond the original 23 patterns in the 1994 book.

Study these 23, then study patterns for "Enterprise Application Architecture."

I recommend studying *message-based* EAA patterns, specifically 65 messaging patterns collected by Hohpe and Woolf (2003).

- Don't miss their bond pricer case study.

Getting Started with OO Design

The most straightforward way to design an OO system is to think about the responsibility of each component.

What does it mean for a software component to be *responsible*?

- Such a component has an *obligation* to *perform a task* or *know information*.
- I.e. responsibility involves either behaviour (*doing*) or data (*knowing*).

Responsibility-Driven Design

"*Doing* responsibility" involves:

- Direct action, e.g. creating an object, processing data, calculating something
- Indirect action, i.e. initiating and coordinating actions with other objects

"*Knowing* responsibility" involves:

- Public and private object data
- References to related objects
- Derived data or objects

In-Class Exercise

Consider the central exchange in the Lecture 1 trading game.

- What does it have to *know*?
- What does it have to *do*?

SOLID

The most widely known list of OO design principles is the following five, known collectively as “SOLID”:

- **S**ingle responsibility
- **O**pen/closed
- **L**iskov substitution
- **I**nterface segregation
- **D**ependency inversion

SOLID: **S**ingle Responsibility

A class should only be responsible for *a single aspect* of the program’s functionality.

I.e. a class should only have a single reason to be changed.

Example: A class which compiles *and* prints a report has *two* reasons to be changed:

- Changes to report content, and
- Changes to report formatting

SOLID: **O**pen/Closed

A class should be *open for extension*, but *closed for modification*.

“Extending without modifying” typically means either of two things:

- Creating a *subclass* of a base class
- Creating an *implementation* of an ABC

SOLID: Liskov Substitution

A derived class must have compatible pre-conditions, post-conditions, and invariants vis-à-vis its base class.

Functions that use objects from a base class must be able to use objects from any derived class without knowing it.

“Don’t make your clients care about which concrete class is in use.”

SOLID: Interface Segregation

Clients should not be forced to depend on methods they don’t use.

- Try to group your clients and create one dedicated interface (ABC) for each.

Example: Consider a big class for an ATM with a “polluted interface” containing all of:

- `requestDeposit()`
- `requestWithdrawal()`
- `requestTransfer()`
- `informInsufficientFunds()`

SOLID: Dependency Inversion

Don’t allow tight coupling between high-level and low-level classes. Break the tight coupling by adding interfaces in between.

Done right, this approach lets you reuse the high-level classes as well as the low-level ones.

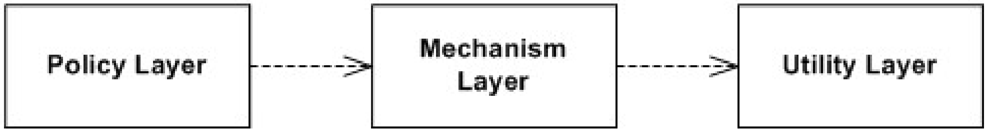
Dependency Inversion, continued

Before: “We require an instance of a particular class.”
After: “We require *any* object which implements a particular interface.”

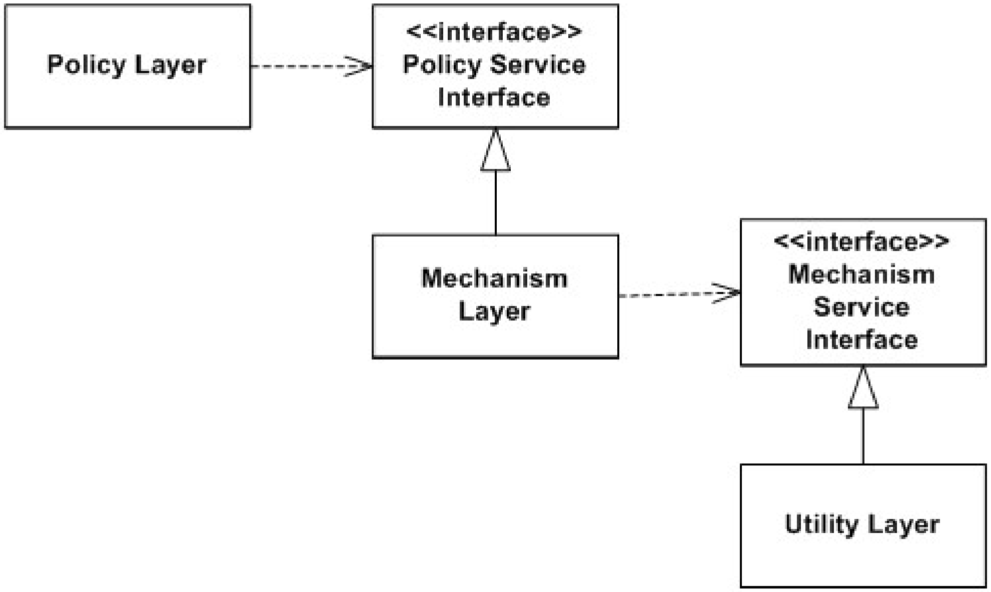
E.g.:
~~“We require an instance of a Microsoft Outlook automation object.”~~
“We require any object which knows how to send an email.”

Dependency Inversion, continued

Before:



After:


Policy Layer
<<interface>> Policy Service Interface
Mechanism Layer
<<interface>> Mechanism Service Interface
Utility Layer

Source: https://en.wikipedia.org

In-Class Discussion

Compare and contrast the expected benefits of object-oriented design vs. the expected benefits of the service-oriented architecture we discussed two weeks ago.

OOP’s Forgotten History

Alan Kay invented the term “object-oriented programming” in 1966 or 1967.

However he later wrote:

“I’m sorry that I long ago coined the term ‘objects’ for this topic because it gets many people to focus on the lesser idea. The big idea is messaging.”

OOP's Forgotten History

In 2003, he elaborated on this:

> "OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."

So his essential ingredients of OOP were:

- Message passing
- Encapsulation
- Dynamic binding

… whereas inheritance and polymorphism were *both non-essential* to him (!).

Concluding Observations

- Python provides many helpful *non-OO* idioms for building smaller apps, such as:
  - Comprehensions
  - Generators
  - Closures
  - Decorators
  - Modules, and packages of modules

- Patterns are hard to appreciate until *after* you missed the chance to apply one!

Recommended Reading

Object Design: Roles, Responsibilities, and Collaborations by Rebecca Wirfs-Brock and Alan McKean (Addison-Wesley, 2002).

Applying UML and Patterns by Craig Larman (3rd Edition, Prentice Hall, 2004).

Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma et al (Addison-Wesley, 1994).

Thank you

A BETTER WORLD BY DESIGN.

SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN