Lesson 1 Introduction

Background

1. Challenges of designing *reusable* object-oriented software (from a problem statement to an object model, then from a high-level design to detailed design, etc)
   1) You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them.
   2) Your design should be specific to the problem at hand but also flexible enough to address future change.
   3) You want to avoid redesign to address new problems and requirements.
   4) Difficult if not impossible to get "right" the first time.
   5) It takes time and practice to become an expert.

2. How people become experts and what makes them different from inexperienced ones?

   1) All experts start off as beginners who are keen learners and who do **not** simply 'code without thinking'.
   2) They learn from others and keep acquiring new skills and knowledge.
   3) Expert designers do not try to solve every problem from first principles. Rather, they reuse proven solutions that have worked for them in the past if they see recurring patterns.
   4) For problems that have a more unique nature, use the knowledge you have learned and create your own patterns (and record them for others to use). The so-called 'enterprise patterns' are some of the examples.
   5) This path does not only apply to software designers. Novelists and playwrights rarely design their plots from scratch. Instead, they learn and follow patterns like "Tragically Flawed Hero" (Macbeth, Hamlet, etc.) or "The Romantic Novel" (countless romance novels). In the same way, object-oriented designers follow patterns too. Once you know the pattern, a lot of design decisions follow automatically.

What is a Design Pattern?

A well-known building architect (Christopher Alexander) says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

A typical definition you see on the Internet - "a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design".

My own definition of design patterns is the "common-sense approach to OOSD" meaning common sense in everyday lives applied in Object-oriented Software Design.

The definition of design patterns given in our textbook: *"descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."*

In general, a pattern has four essential elements:
1. The **pattern name**.
2. The **problem:** explains the problem and its context and when to apply the pattern.
3. The **solution:** a 'template' that describes the elements that make up the design - their relationships, responsibilities, and collaborations.
4. The **consequences:** are the results and trade-offs of applying the pattern.

What is a Software Framework and a Software Foundation? What is the difference between Frameworks, Design Patterns and Libraries?

To put simply, design patterns are ideas on how to solve certain problems. They need to be implemented with a programming language.

Frameworks are semi-completed applications that need to be customized according to user requirements to become an application for a particular organization.

Libraries are a lower-level concept. A library can be a collection of classes that you need to call in your application. A library does not have an application structure or architecture.

A software foundation is a complete application. But different from regular applications, a software foundation was built with a fully-developed ‘core’ plus well-designed/documented extension mechanisms provided so other developers can easily add functions/modules to the foundation for their own application(without having to worry about any plumbing issues at all).

Difference between developing applications vs frameworks?

Application development: both technical and business knowledge (without using a framework) or primarily business knowledge (developing based on a framework).

Framework development: mainly technical knowledge with a special focus on providing cross-cutting services, reducing complexity, simplifying development work, and improving productivity.

Users of an application are the people who operate in the business.

Users of a framework are application developers who need to understand how to support the business.

Why use a framework?

Frameworks reduce the software development effort: Software applications = code for business logic + code for the infrastructure that holds the application components together.

Frameworks provide generic services (or plumbing/boiler-plate): Examples include request processing, caching, logging, configuration and database access. They simplify development for event handling, user interface management, data exchange and job

processing. This infrastructure code is usually the most detailed and tedious code to write, requiring deep technical skills.

Frameworks reduce complexity: Many of the design decisions necessary when starting development from scratch are already built into the framework, like the architecture, object creation, performance optimization, etc. Therefore, developers do not spend time reinventing the wheel. Less experienced developers can still build better quality software based on a framework.

Frameworks improve productivity, quality and consistency: Frameworks make the software development cycle a more predictable process by providing a standardized architecture and a standardized development approach. For example, developers work with pre-defined rules for coding and proven methodologies for building parts of an application. They also use a given set of development tools.

The main focus of this course is **Design Patterns within the context of software framework/application design.**

Describing Design Patterns

We are going to use the same template as the textbook does to describe 23 GoF design patterns, which are divided into the following sections:

Pattern Name (and Classification)
The pattern's name conveys the essence of the pattern succinctly.

Intent
A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address? (at a very high level)

Also Known As
Other well-known names for the pattern, if any. (Reality is not many people refer to those patterns by a “secondary” name)

Motivation
A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows. (A lot of times, it also serves as a problem statement)

Applicability
What are the situations in which the design pattern can be applied? What are examples of alternative designs with obvious design flaws but this pattern can address better? How can you recognize these situations?

Structure
A graphical representation of the classes in the pattern using a UML-style notation. Some other times, interaction diagrams are also used to illustrate sequences of requests and collaborations between objects.

Participants
The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations
How the participants collaborate to carry out their responsibilities.

Consequences
How does the pattern support its objectives? What are the trade-offs (fast, simple, cheap but good enough like the 'space-complexity-time' tradeoff) and results of using the pattern?

Implementation
What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues? Sample Code fragments that illustrate how you might implement the pattern. (Sample code given in textbook is Smalltalk, C, or C++.)

Known Uses
Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns
What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

The Catalog of Design Patterns (Gang of Four)

Creational: patterns used to create objects. (Object creation).
Structural: patterns that are concerned with class or object composition to put in place a certain capability or perform a certain task. (Micro-architecture).
Behavioral: patterns that are concerned with communication between objects in a design solution. (Object interaction).

	Intent	Motivation	Examples
Creational			
Singleton(*)	Ensure a class only has one instance, and provide a global point of access to it.	How do we ensure that a class has only one instance and the instance is easily accessible?	java.lang.Runtime#getRuntime() java.awt.Desktop#getDesktop()
Prototype(**)	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.	You need to make new instances by copying template instances. When your application has to create objects without knowing their type or any details of how to create them. When creating a large/complex object that is resource intensive, cloning works better than using the constructor sometimes.	3D printing The clone() method in Java (Object or its subclasses that implement java.lang.Cloneable)
Factory Method(**)	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Client may only know that it needs an object of a certain type but does not know the exact subtype to be selected	java.util.Calendar#getInstance() java.util.ResourceBundle#getBundle() java.text.NumberFormat#getInstance()
Builder(**)	Separate the construction of a complex object from its representation so that the same construction process can create different representations.	You take a step-by-step approach to construct a complex object.	Complex objects created in assembly lines in a factory. Order with multiple order-line items. Java InputStream and Outstream subclasses
Abstract Factory(***)	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.	You need to create different series of products that belong to different inheritance hierarchies	javax.xml.parsers.DocumentBuilderFactory#newInstance() javax.xml.transform.TransformerFactory#newInstance() javax.xml.xpath.XPathFactory#newInstance()

Structural			
Adapter(*)	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.	Class library cannot be used because its interface is incompatible with the interface required by an application	Electricity plug adapter, Stack implemented with an array, java.util.Arrays#asList()
Composite(***)	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.	You want to represent part-whole or parent-child hierarchies of objects	Tree structure java.awt.Container
Façade(*)	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.	To simplify complex interfaces exposed by classes in a set of subsystems	User interface of some home appliances (microwave oven, washer, etc) Company's front desk or telephone switchboard
Bridge(**)	Decouple an abstraction from its implementation so that the two can vary independently.	Avoid permanent binding between abstraction and implementation	JDBC-ODBC bridge, JVM spec and implementation, GC spec and implementation, etc
Flyweight(***)	Use sharing to support large numbers of fine-grained objects efficiently.	Your application deals with a potentially large number of objects of the same type, which may take a huge amount of memory space	DB connection pool, Java String Pool
Proxy(***)	Provide a surrogate or placeholder for another object to control access to it.	a client does not or cannot reference an object directly, but wants to still interact with the object	RMI or CORBA Data source for a DBMS Web Proxy Reference counting module inside Java Garbage Collector
Decorator(**)	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.	Sometimes we want to add responsibilities to individual objects, not to an entire class	FileInputStream fis = new FileInputStream("/objects.gz"); BufferedInputStream bis = new BufferedInputStream(fis); GzipInputStream gis = new GzipInputStream(bis);
Behavioral			
Chain of Responsi-bility(***)	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the	From the handlers' perspective, multiple resources with different skills, different roles, or different responsibilities organized in a chain structure	Java Exception Handling, Template hierarchy in WordPress core, Customer Services Model(rep, senior rep, supervisor, etc)

	chain until an object handles it.		

Command(***)	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.	Sometimes it's necessary to issue requests without knowing anything about the operation being requested or the receiver of the request.	Database operations modelled as commands that are part of a transaction (which also support roll-back)
			Implement actionPerformed(ActionEvent e) with Command Pattern.
			Replace
			public void actionPerformed(ActionEvent e){
			  Object o = e.getSource();
			  if (o == fileNewMenuItem)
			  doFileNewAction();
			  else if (o == fileOpenMenuItem)
			  doFileOpenAction();
			  else if (o == fileOpenRecentMenuItem)
			  doFileOpenRecentAction();
			  else if (o == fileSaveMenuItem)
			  doFileSaveAction();
			  // and more ...
			}
			With
			public void actionPerformed(ActionEvent e){
			  Command command = (Command)e.getSource();
			  command.execute();
			}

Strategy(*)	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.	The client and a strategy it uses to perform a function should not be hard-wired together -to keep the client lean; -or change of strategy easier; -or add new strategies without affecting client code.	Different shipping methods (UPS, FedEx, USPS, etc) Different file compression methods (zip, rar, gzip, etc)
Iterator(**)	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	An aggregate object such as a list or hash map should allow a way to traverse its elements without exposing its internal structure	All implementations of java.util.Iterator All implementations of java.util.Enumeration
Template Method(*)	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the	Sometimes you want to specify the order of operations that a method uses, but allow subclasses to provide their own implementations of some of these operations	java.io.InputStream java.io.OutputStream java.io.Reader java.io.Writer. java.util.AbstractList java.util.AbstractSet java.util.AbstractMap. javax.servlet.http.HttpServlet

	algorithm's structure.		
Mediator(**)	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.	Need to centralize complex communications and control between related objects	Airport control tower; A Chatroom software application for multiple users; Auction;
Observer(**)	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.	The multiple subscribers are dependent on the state of the publisher; therefore they need to be notified of any change of state with the publisher.	Publisher-subscriber java.util.Observer java.util.Observable All implementations of java.util.EventListener
Memento(**)	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.	You must save state information somewhere so that you can restore objects to their previous states	"Undo" button in Microsoft Word. "Restore Data" button on a GUI. Having snapshots when repairing/dismantling a machine (with an assistant taking the snapshots for you)
State(**)	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.	Object behavior varies for different states the object is in.	State machine, vending machine All-in-one remote control for Fan, Light, TV, etc
Interpreter (***)	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.	Sometimes we need to create a rule-based language to represent expressions or statements or sentences, we want to describe how to define the grammar (the rules) and how to interpret the language.	Evaluating prefix-operator expressions like "- + 10 5 + - 8 2 9" which is equivalent to (10 + 5) - ((8 - 2) + 9) Roman Numerals Convertor XIV = 10 + (5 - 1)
Visitor(***)	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.	Visitor allows us to be even more flexible to perform different functions on a collection of objects (that do not need to be part of the same hierarchy) with "double dispatch".	-People do tax return by themselves with a method - doTaxReturn(); -Or they delegate it to a professional tax preparer by passing all information over with taxProparer.doTaxReturn(incomeStatements); - Or they accept a visit by a professional preparer to do tax return, not having to pass any information(you come and visit me). This is like a "double dispatch" in Software Engineering, which sits at the core of the Visitor pattern.

How to Select/Use a Design Pattern

1. You need to know all of them fairly well before you can possibly make a good choice. It is hard to imagine you have to go through the pattern catalog one-by-one to study them on the spot in order to find a solution (different from using a dictionary).
2. You must have a clear understanding of what the problem is, what is important for your solution or trade-offs (response time, need of space, overhead, complexity…), and possibilities of future extension, etc.
3. Most of the times, things in the problem statement will suggest a certain pattern to you if you associate some keywords with the pattern. (Keywords for patterns can be something you see in the name, intent, motivation, and participants, like invoker, caretaker, mediator, proxy, strategy, publisher, subscriber, sharing, decoupling, etc.)
4. Once a pattern is selected, many other decisions will follow.
5. Sometimes more than 1 pattern pop up as candidate solutions. Don’t be confused or surprised. There is no hard solid line that makes one ‘correct’ and the others ‘incorrect’.

Why Design Patterns and what it requires for us to understand them better?

There are two substantial reasons for the existence of design patterns:

First, the use of design patterns facilitates communication between developers.

The second benefit of using design patterns is that they are market proven solutions so we do not have to ‘reinvent the wheel’.

Whenever you feel confused with anything regarding a design pattern, try looking at it from a different perspective.

From a technical perspective -

1. **Separate change from non-change**: if there has to be change, do it by extension not by modification.
2. **Communication, Learning and Enhanced Insight**: Over the last decade design patterns have become part of every developer's vocabulary. This really helps in communication. One can easy tell another developer, "I've used Command pattern here" and the other person understands the design.
3. **Decomposing System into Objects**: The hard part about OO Design is finding the appropriate classes/objects in a system. Think about what we did in MPP - encapsulation, granularity, dependencies, interfaces, flexibility, performance, evolution, reusability and so on. Design Patterns really help identify them + the less obvious abstractions.
4. **Ensuring right reuse mechanism**: When to use Inheritance, when to use Composition, when to use Parameterized Types? Is delegation the right design decision in this context? There are various questions that come to a programmer's mind when they are trying to design highly reusable and maintainable code. Knowledge of design patterns can really come handy when making such decisions.
5. **Relating run-time and compile-time structures**: An object oriented program's run-time structure often bares little resemblance to code structure. Sometimes looking at the code does not give us the insights into a run-time structure. Knowledge of design patterns can make some of the hidden structure obvious.

All it boils down to are more cost savings and shorter time-to-market (because all businesses want to win during competition).

Basic assumptions (from employers' perspectives) FYI:

1. Change means costs/money. Developing software that changes less saves money. So separating change from non-change as much as possible is a huge factor.

2. Faster/easier maintenance and extension means less time (thus less spending). Who loves a smaller budget the most? (business owner or manager whose promotion depends on budget management)

3. Well-structured/layered software is being loved by both technical and business people –
   Technical people love doing things the "right" way.
   Business people love spending less, relying less on certain technical resources, easier hiring, lower pays.

Employers are constantly looking for cheaper replacements of all needed resources to run the business and technical people are happily helping them achieve the goal. Why robots are so hot now? Fortunately, robots are not smart enough to write software yet.(Once they are, employers will not think twice to restructure/reorg their business!)

4. There is a step by step approach – to make application developers' jobs easier. Framework developers are considered "smarter" and harder to find. The unstoppable progressive developments are making it closer day by day.

5. What do we do? – make it harder to get rid of you from a job and become smarter by CONSTANT learning and expanding your awareness!

Design Patterns and Your Day-to-day Job

Design patterns solve many of the day-to-day problems object-oriented designers face, and in many different ways. However, keep these in mind when you work on a software project for your job.

1. Most (OO) software development jobs are done on an application level where chances of applying patterns are a lot less compared with the framework layer. (Because the non-change part tends to stay on lower levels. A joke - if you cannot get rid of a huge if/else or switch/case, hide it somewhere – break it or push it down below).
2. Most projects have tight deadlines, which put you under great stress (often times). You may not be able to work the way you like or you may not have the mood to do it the 'right way'. Just get it done ASAP. However, if you speak to the manager and convince him/her of doing it the 'right way', you may make a difference (and stand out in the crowd too).
3. As you use patterns more in your projects, you should also stay alert for opportunities to discover/create new patterns. There should be more possible fundamental patterns than the 23 we are learning in this course. (like Josekis, for example)
4. Regardless of all kinds of comments from the online communities, some of the GoF patterns are really brilliant

ideas. And it is always good to think in patterns when solving a design problem.
5. Working on a project, people discuss design solutions in jargons. You need the vocabulary to communicate with others.
6. OTOH, however, you should never try to overkill. Do not apply patterns for the sake of it. A straight-forward, easy-to-understand design that does the job is always good.
7. As you become a pattern expert, you also have to avoid seeing a solution before the problem.
8. Design patterns are among the most frequently asked questions during job interviews (for software developers)- both GoF and enterprise patterns.

Essential Nature of 23 GoF Design Patterns

The 23 GoF Design patterns are all about the following:

1. Decoupling for easy extension and change (Bridge, Builder, Command, Factory, Abstract Factory, Iterator, State, Strategy, Mediator, Observer, Visitor).
2. Reusing both code and objects by converting interfaces, filling functional gaps, or sharing objects(Adapter, Bridge, Flyweight).
3. Simplifying for easier use/maintenance, profiling/debugging and understanding source code(Decorator, Façade, Prototype, Iterator, Composite).
4. Structuring for clearly defined layers and better architectures(Façade, Mediator, Observer, CoR, Proxy).
5. Defining/Reinforcing rules (Singleton, Template Method, Interpreter).
6. Improving user experiences(Proxy, Memento).