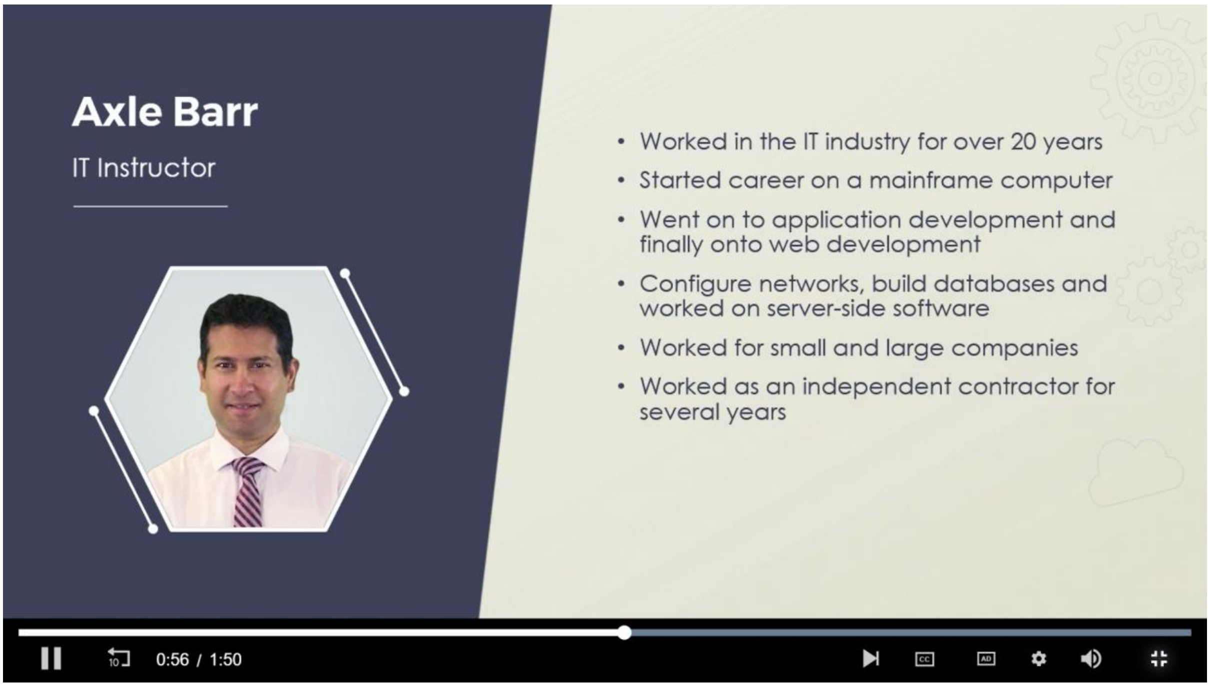
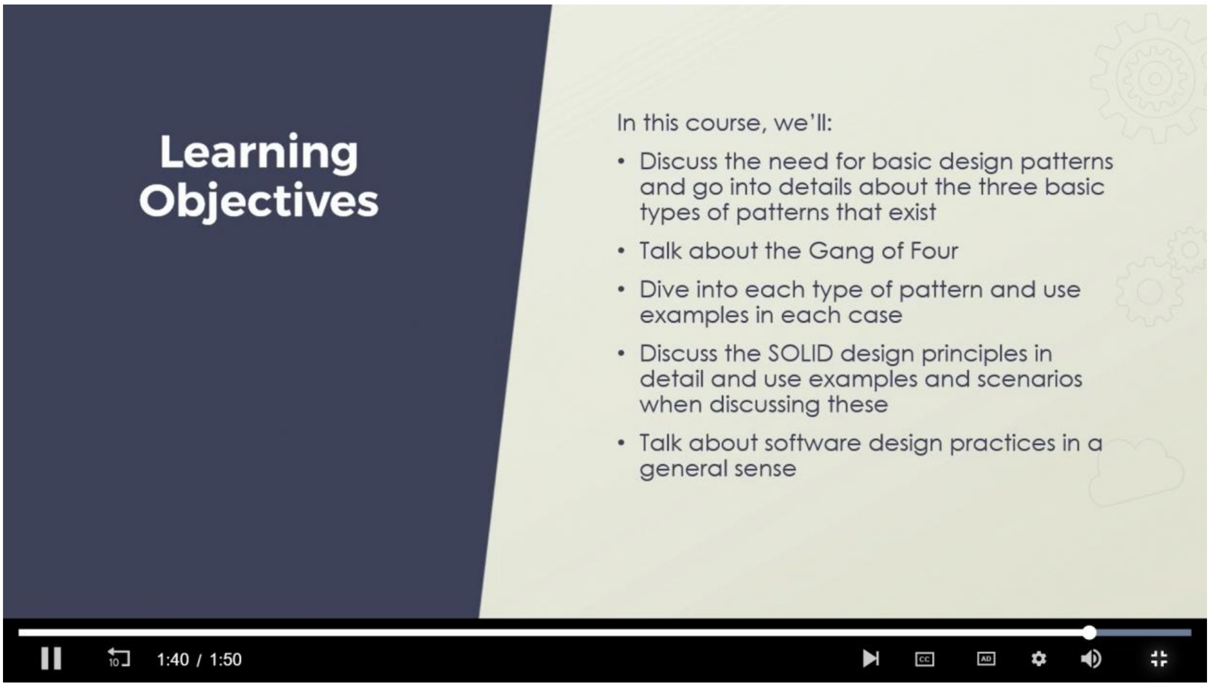


Axle Barr

IT Instructor

- Worked in the IT industry for over 20 years
- Started career on a mainframe computer
- Went on to application development and finally onto web development
- Configure networks, build databases and worked on server-side software
- Worked for small and large companies
- Worked as an independent contractor for several years

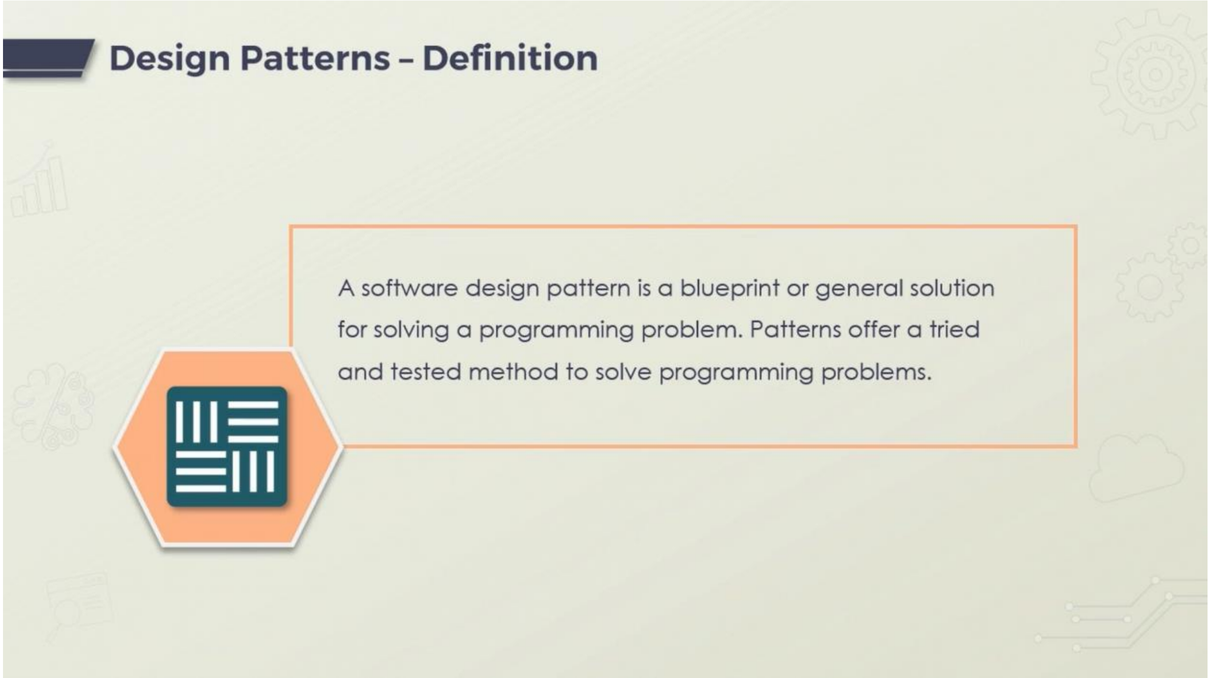


Learning Objectives

In this course, we'll:

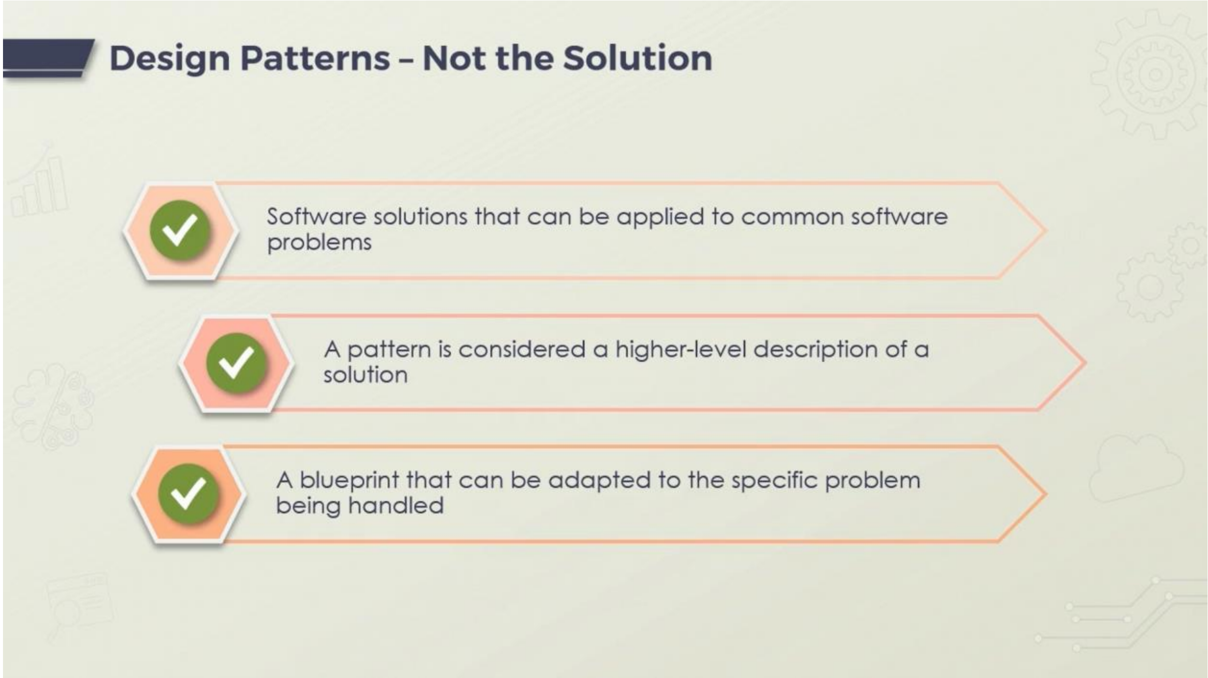
- Discuss the need for basic design patterns and go into details about the three basic types of patterns that exist
- Talk about the Gang of Four
- Dive into each type of pattern and use examples in each case
- Discuss the SOLID design principles in detail and use examples and scenarios when discussing these
- Talk about software design practices in a general sense

Need for Design pattern.



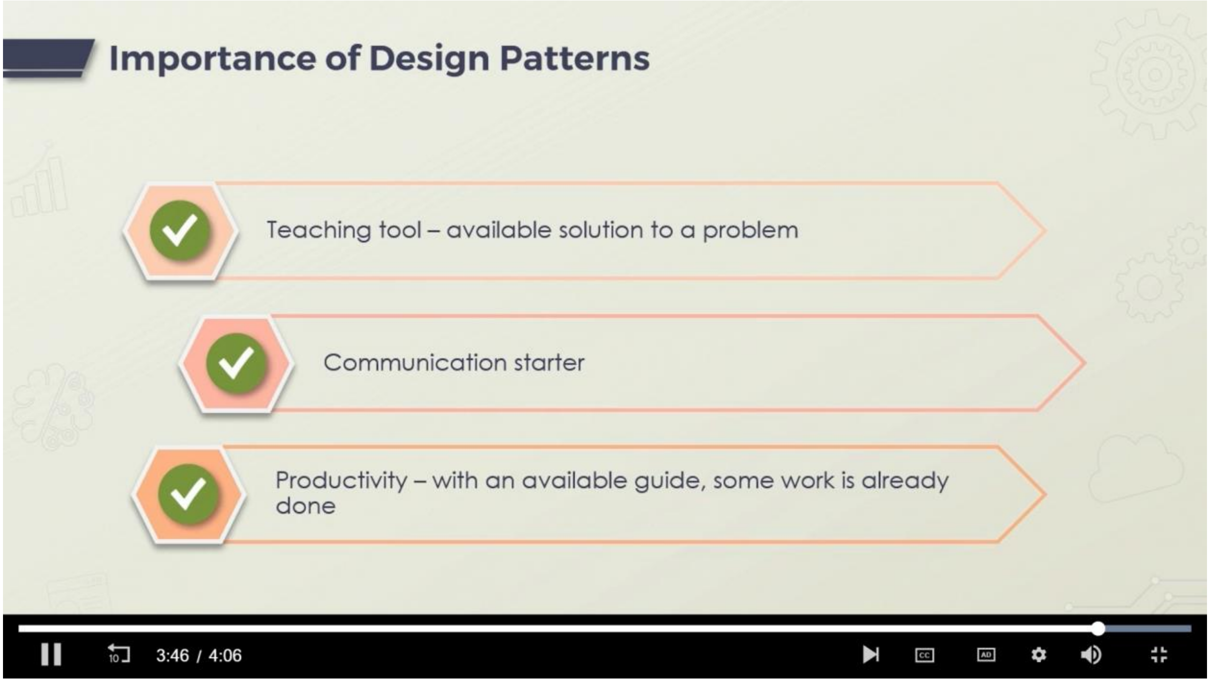
Design Patterns – Definition

A software design pattern is a blueprint or general solution for solving a programming problem. Patterns offer a tried and tested method to solve programming problems.



Design Patterns – Not the Solution

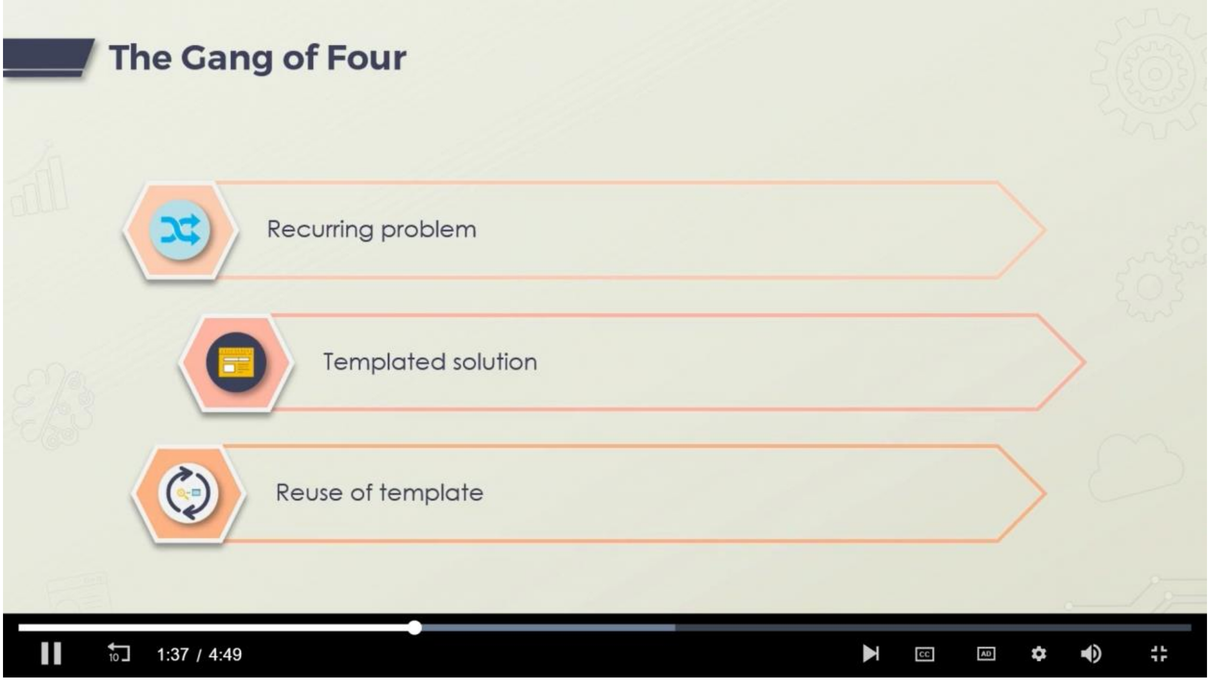
- ✓ Software solutions that can be applied to common software problems
- ✓ A pattern is considered a higher-level description of a solution
- ✓ A blueprint that can be adapted to the specific problem being handled



Importance of Design Patterns

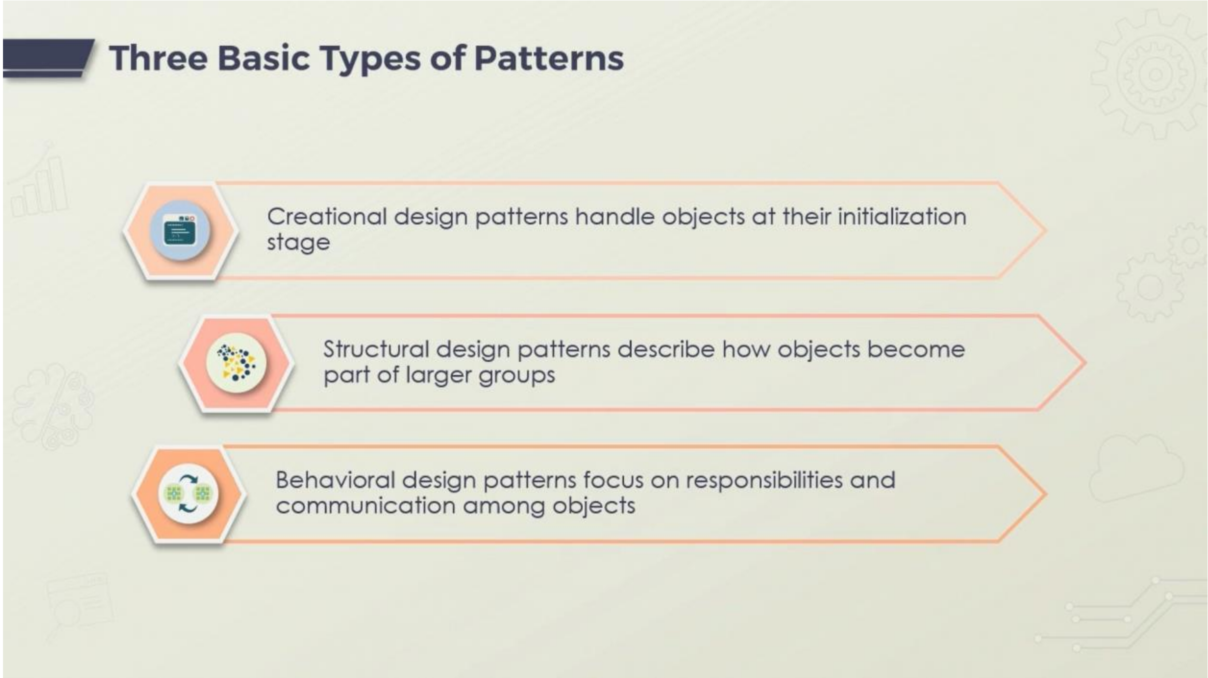
- ✓ Teaching tool – available solution to a problem
- ✓ Communication starter
- ✓ Productivity – with an available guide, some work is already done

Three basic types of design pattern



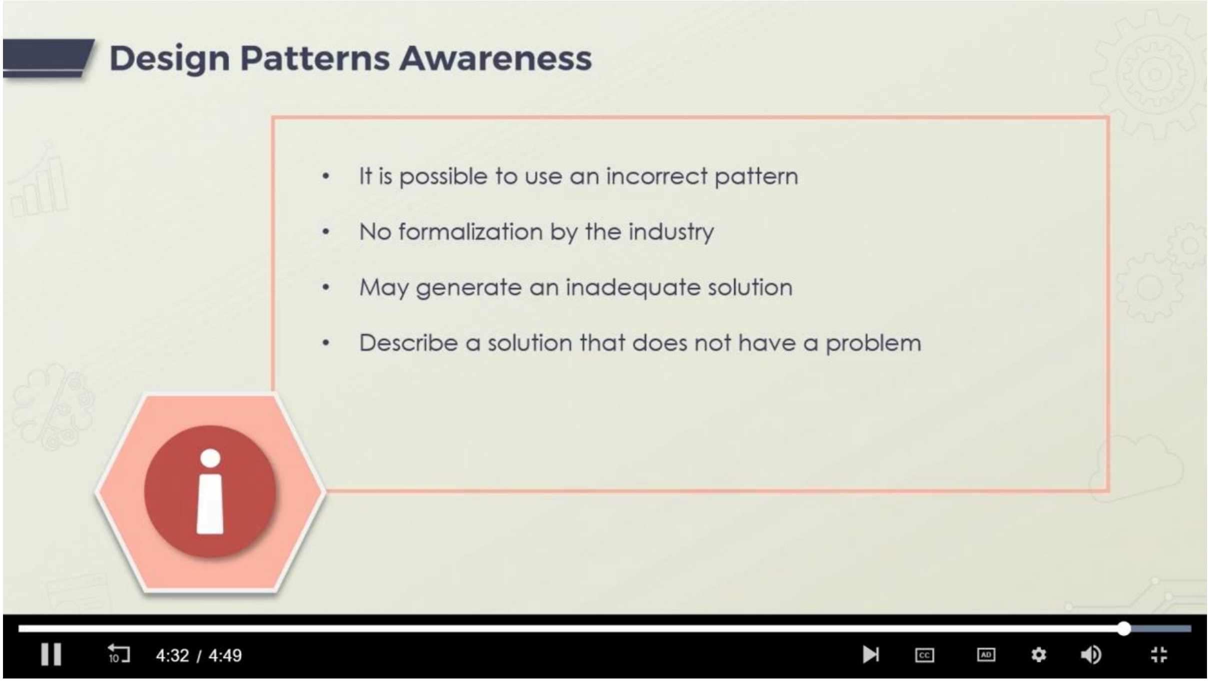
The Gang of Four

- Recurring problem
- Templated solution
- Reuse of template



Three Basic Types of Patterns

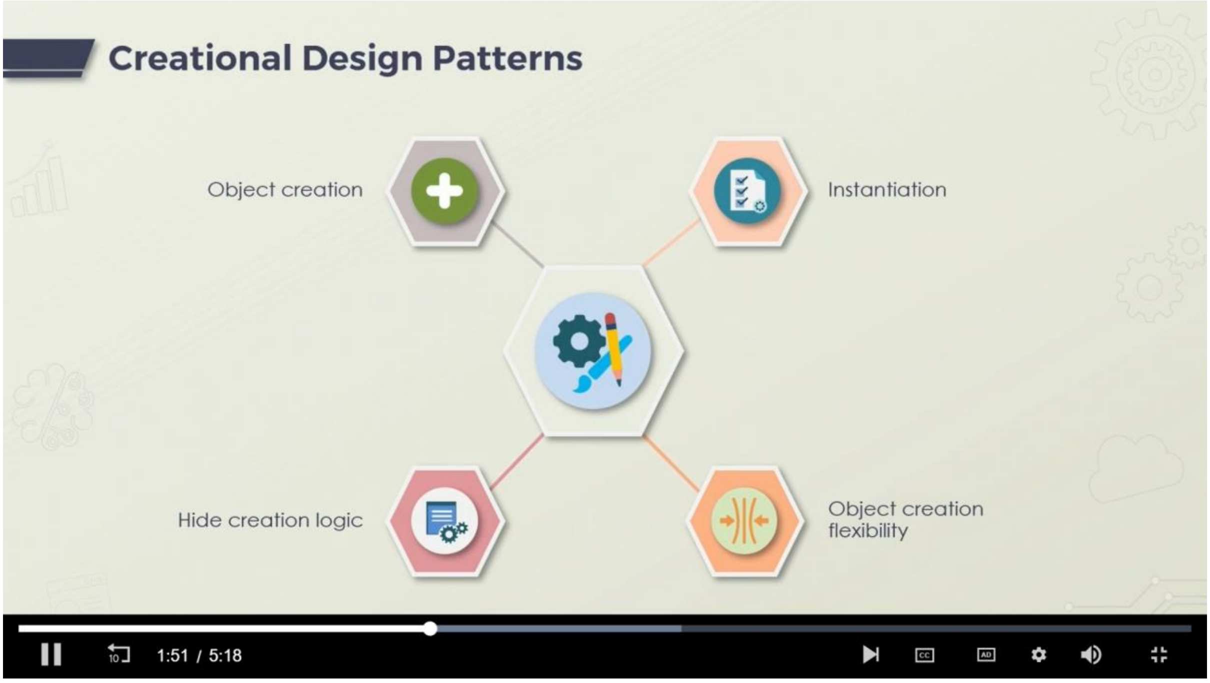
- Creational design patterns handle objects at their initialization stage
- Structural design patterns describe how objects become part of larger groups
- Behavioral design patterns focus on responsibilities and communication among objects



Design Patterns Awareness

- It is possible to use an incorrect pattern
- No formalization by the industry
- May generate an inadequate solution
- Describe a solution that does not have a problem

Creation pattern



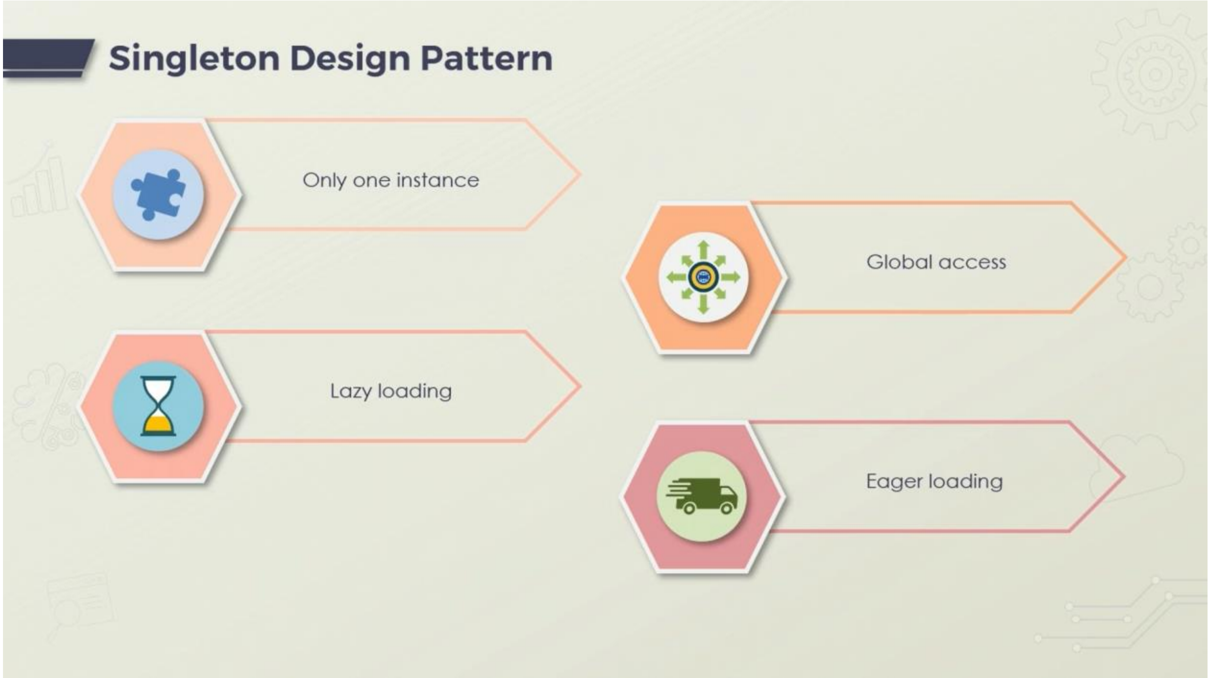
Creational Design Patterns

Object creation

Instantiation

Hide creation logic

Object creation flexibility



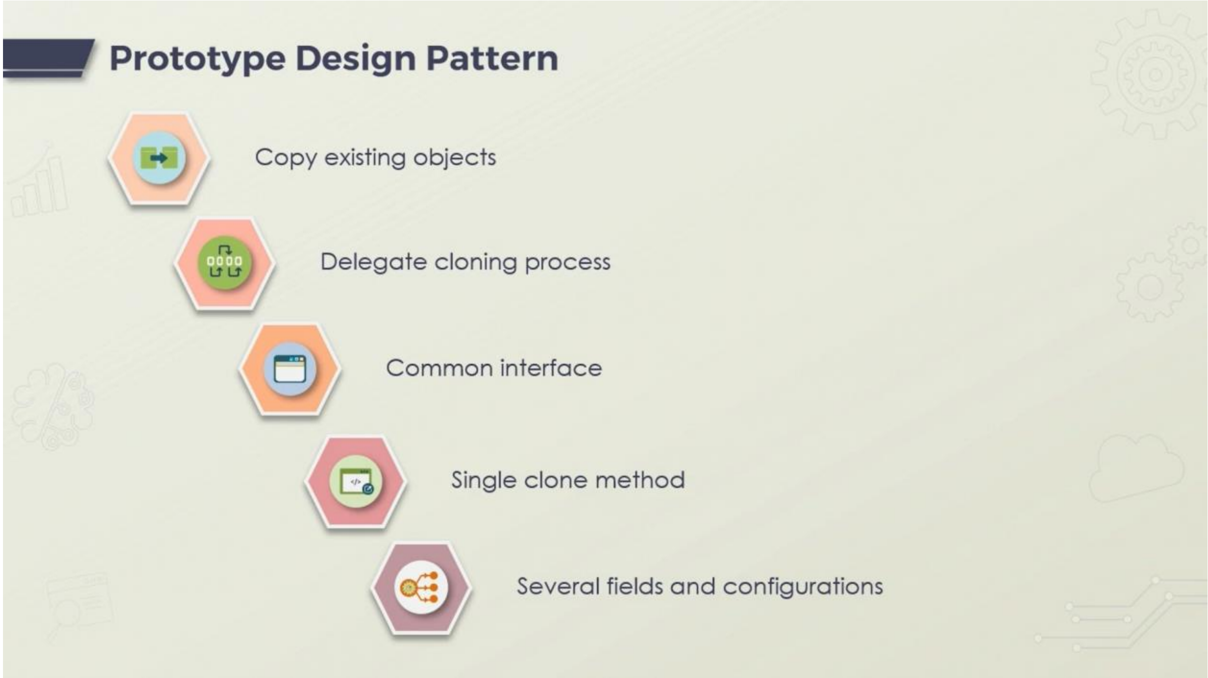
Singleton Design Pattern

Only one instance

Global access

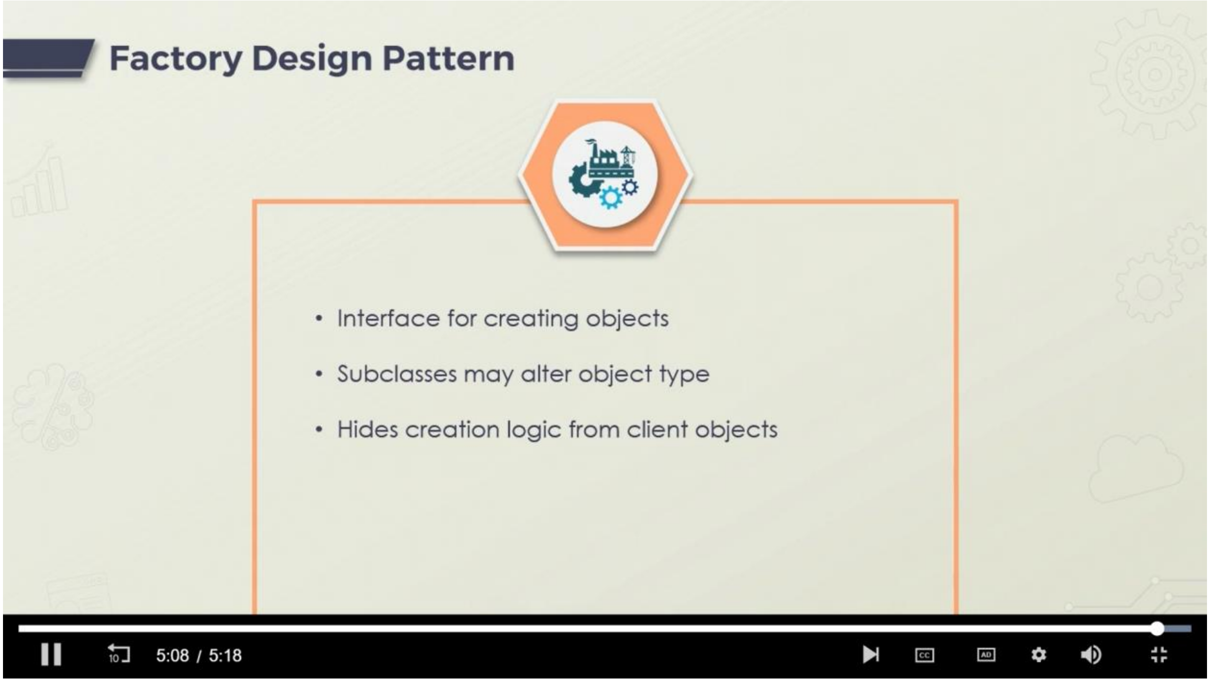
Lazy loading

Eager loading



Prototype Design Pattern

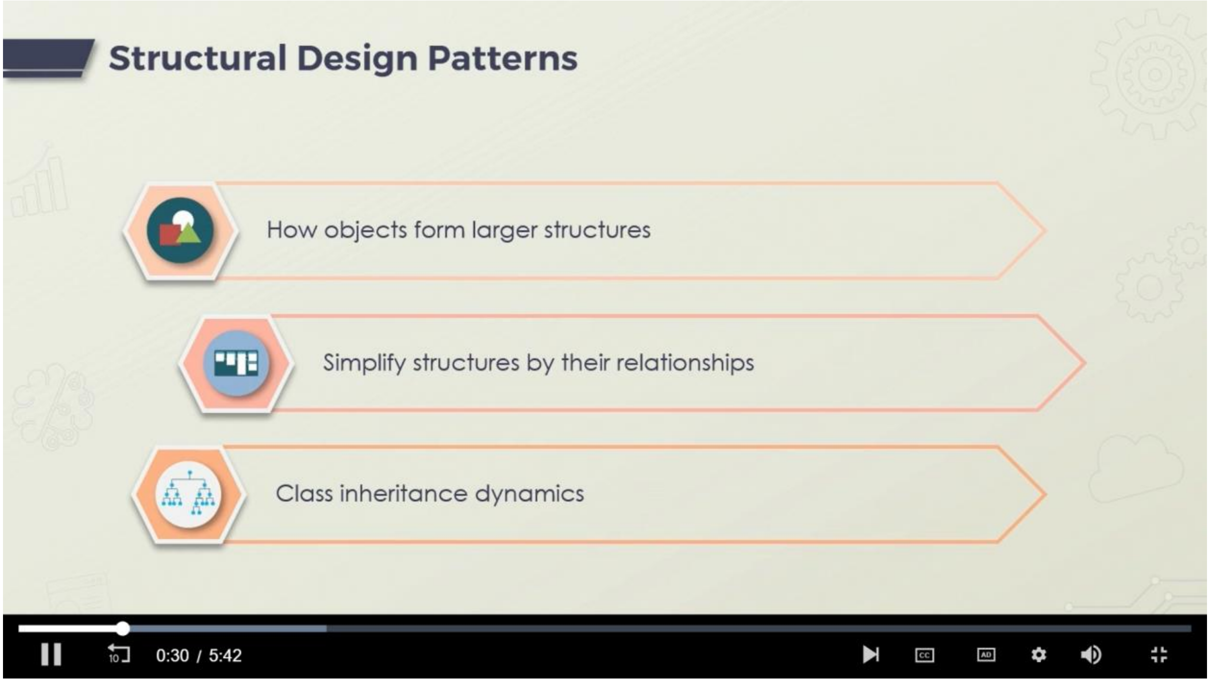
- Copy existing objects
- Delegate cloning process
- Common interface
- Single clone method
- Several fields and configurations



Factory Design Pattern

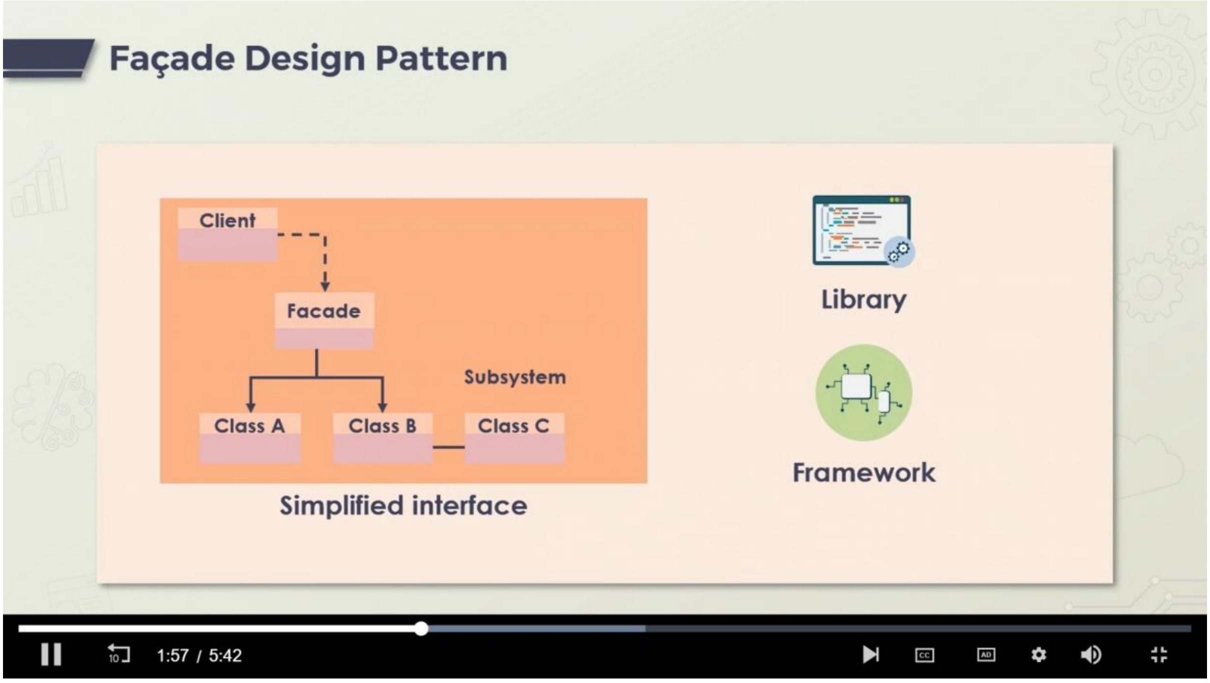
- Interface for creating objects
- Subclasses may alter object type
- Hides creation logic from client objects

Structural Patterns



Structural Design Patterns

- How objects form larger structures
- Simplify structures by their relationships
- Class inheritance dynamics

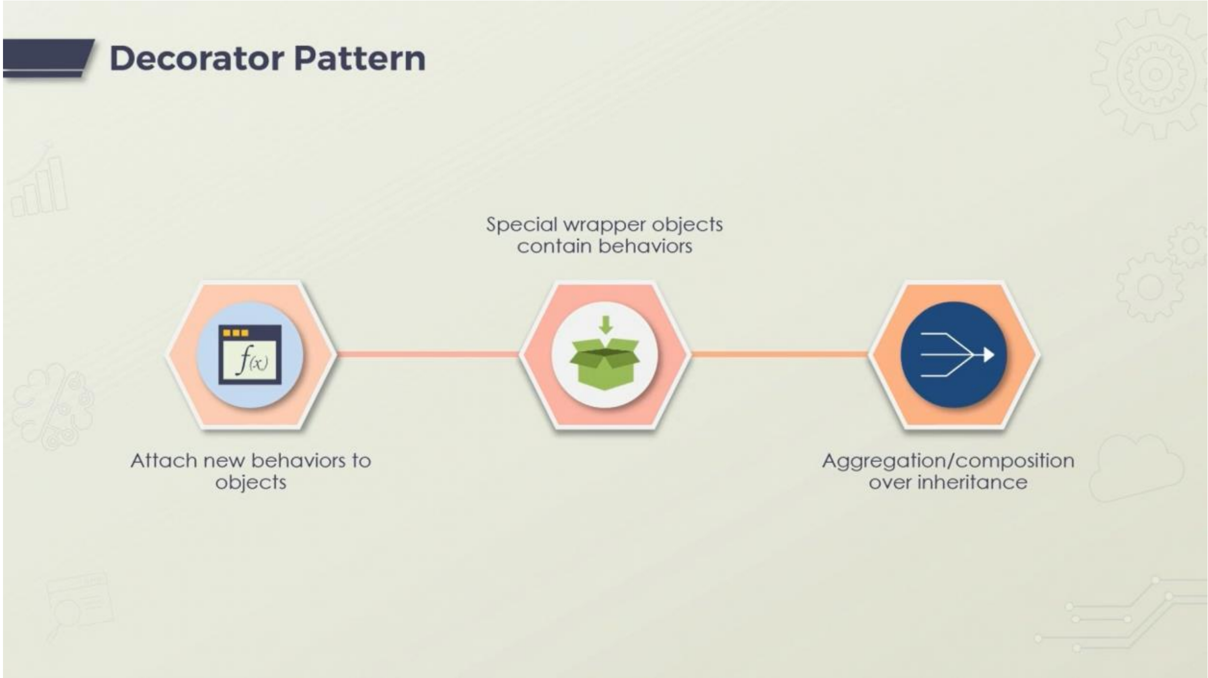


Façade Design Pattern

Simplified interface

Library

Framework

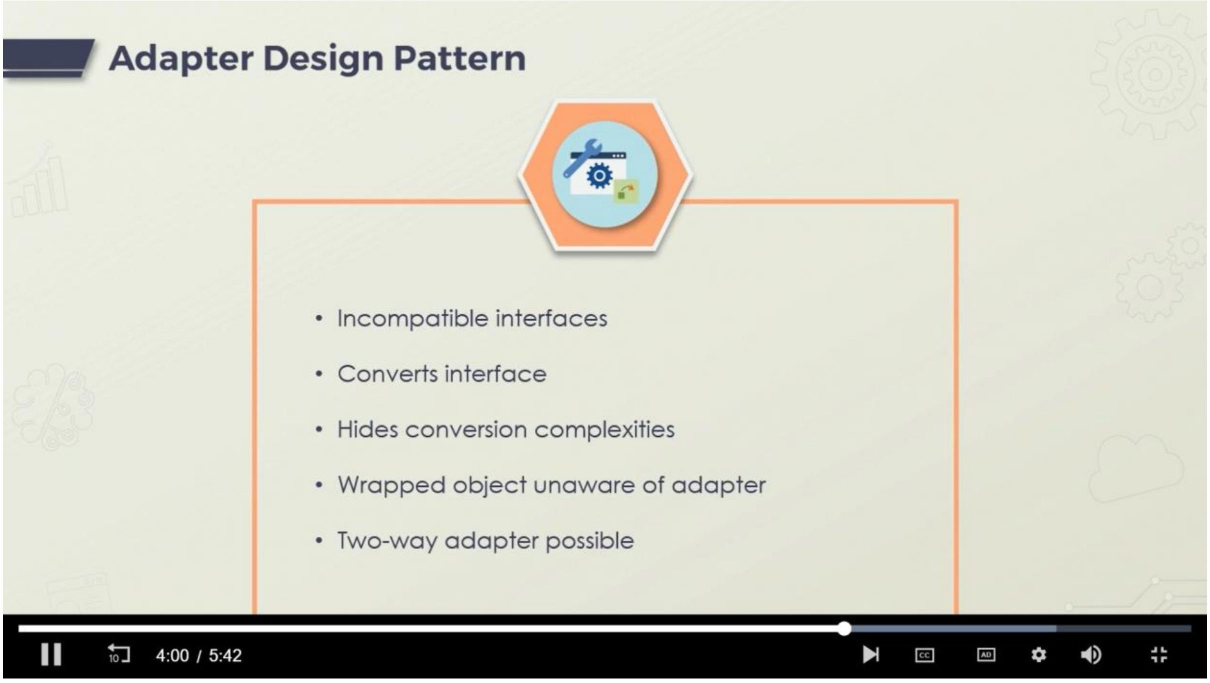


Decorator Pattern

Special wrapper objects contain behaviors

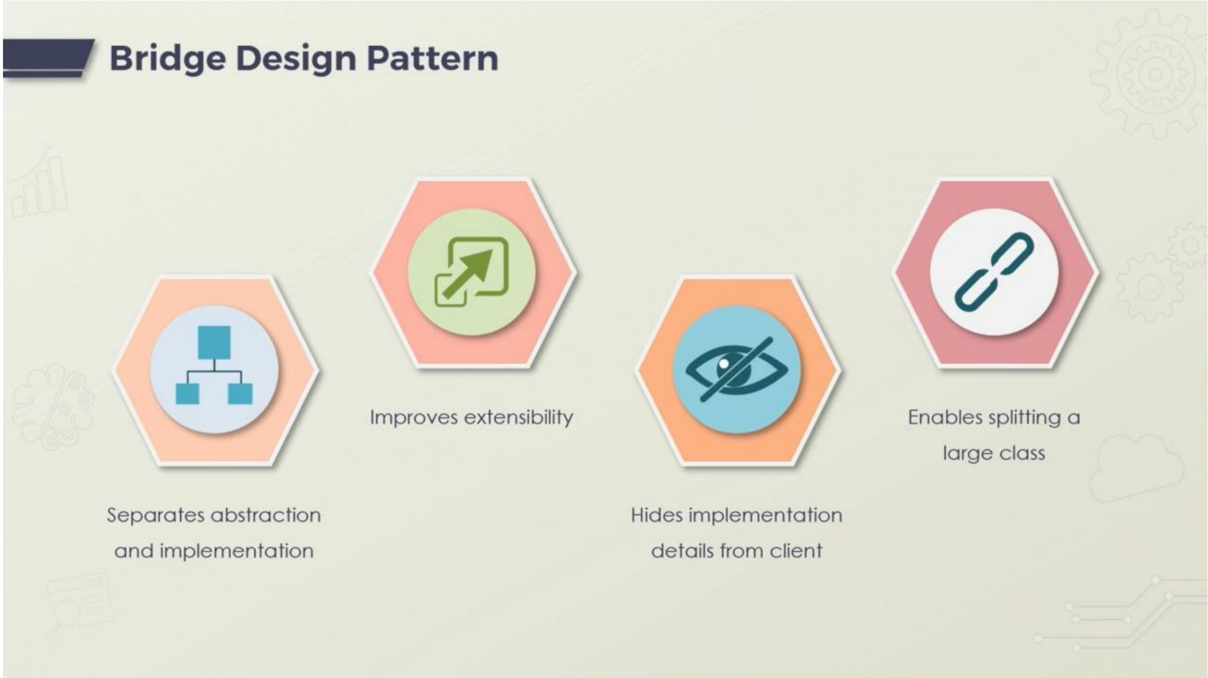
Attach new behaviors to objects

Aggregation/composition over inheritance



Adapter Design Pattern

- Incompatible interfaces
- Converts interface
- Hides conversion complexities
- Wrapped object unaware of adapter
- Two-way adapter possible



Bridge Design Pattern

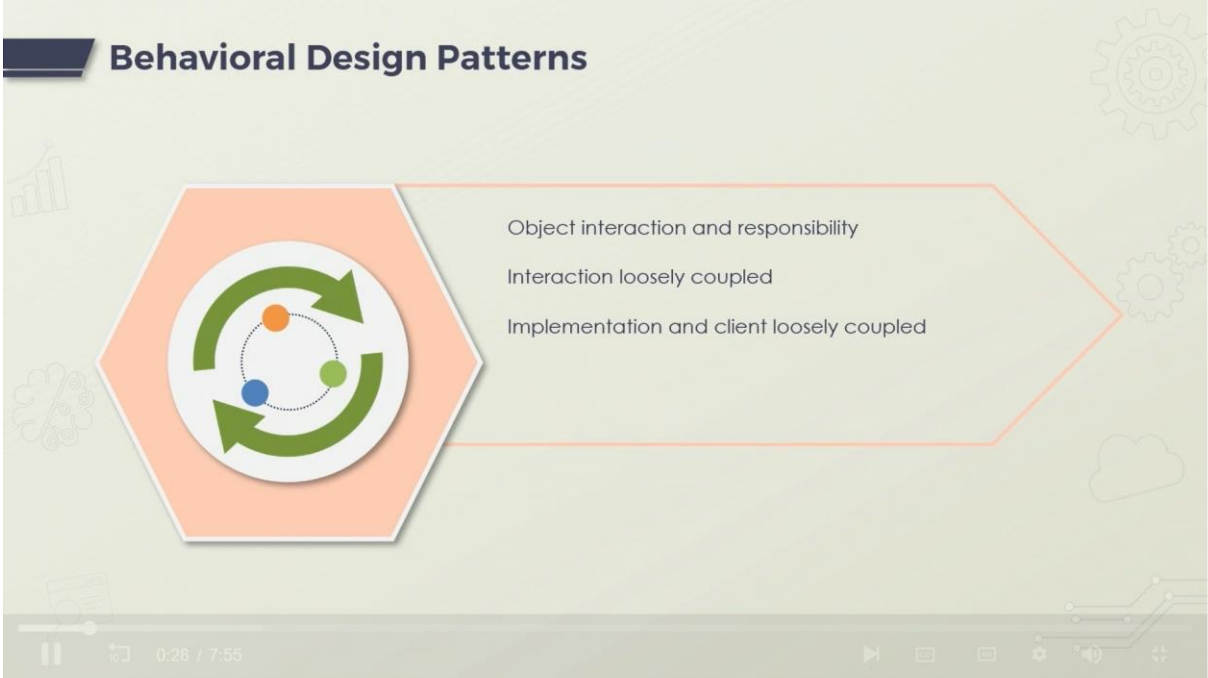
Separates abstraction and implementation

Improves extensibility

Hides implementation details from client

Enables splitting a large class

Behavioral Patterns

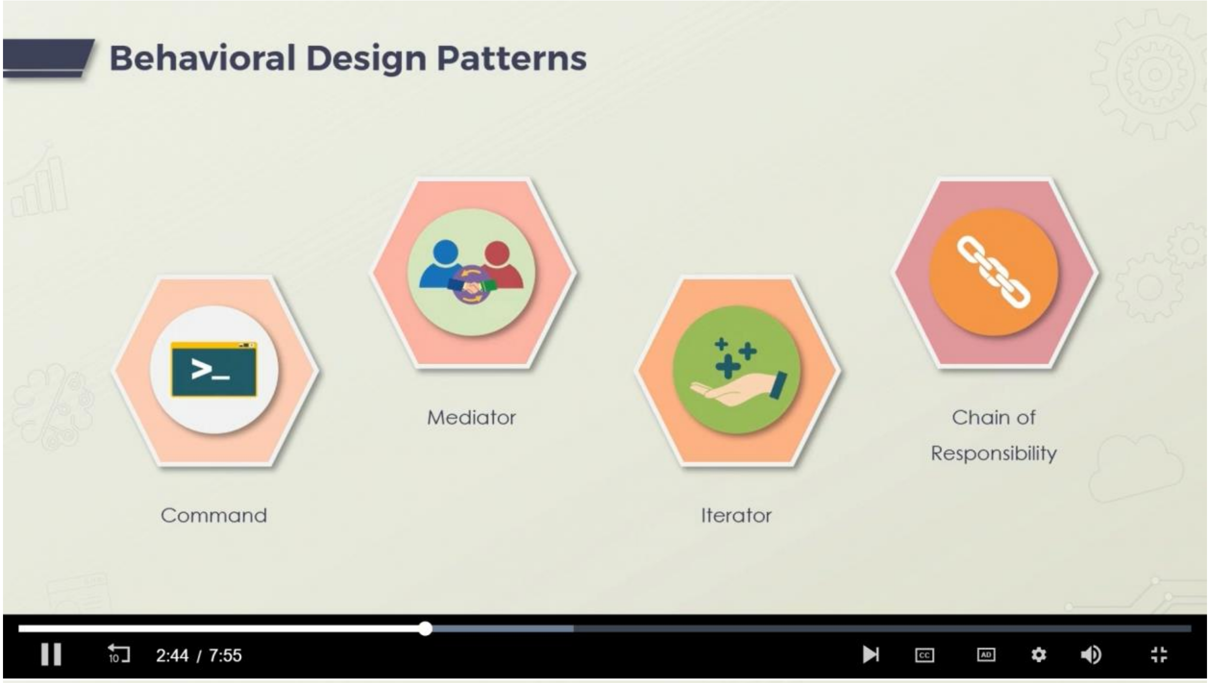


Behavioral Design Patterns

Object interaction and responsibility

Interaction loosely coupled

Implementation and client loosely coupled



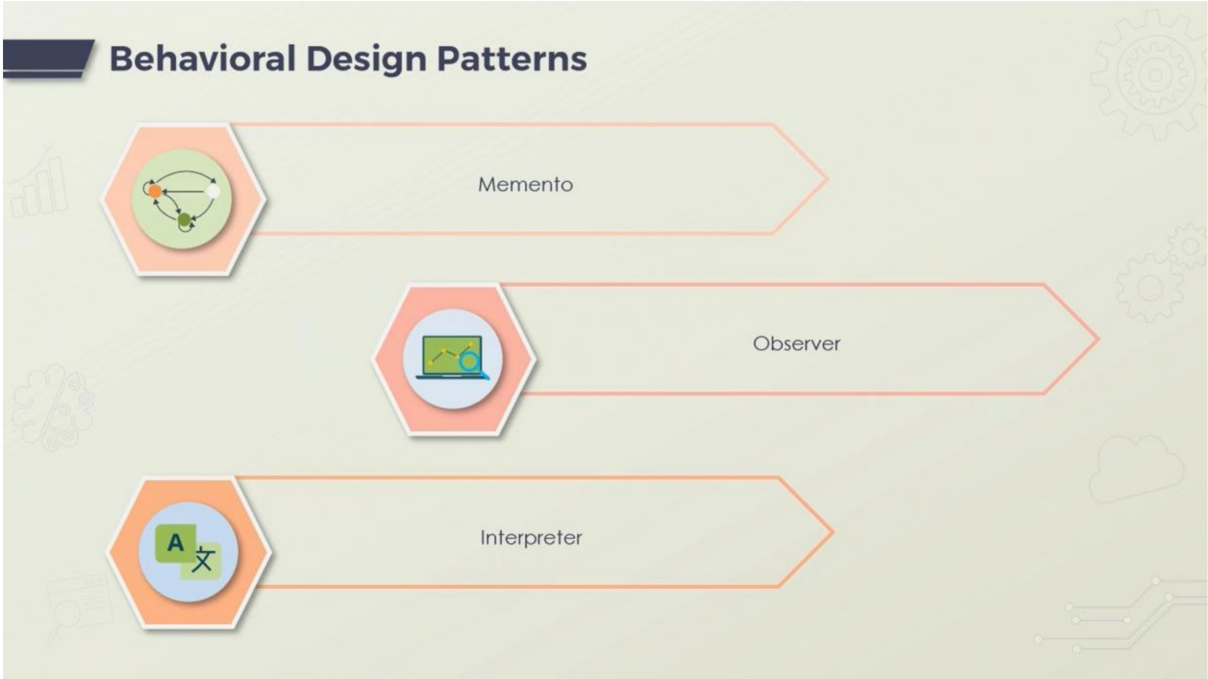
Behavioral Design Patterns

Command

Mediator

Iterator

Chain of Responsibility

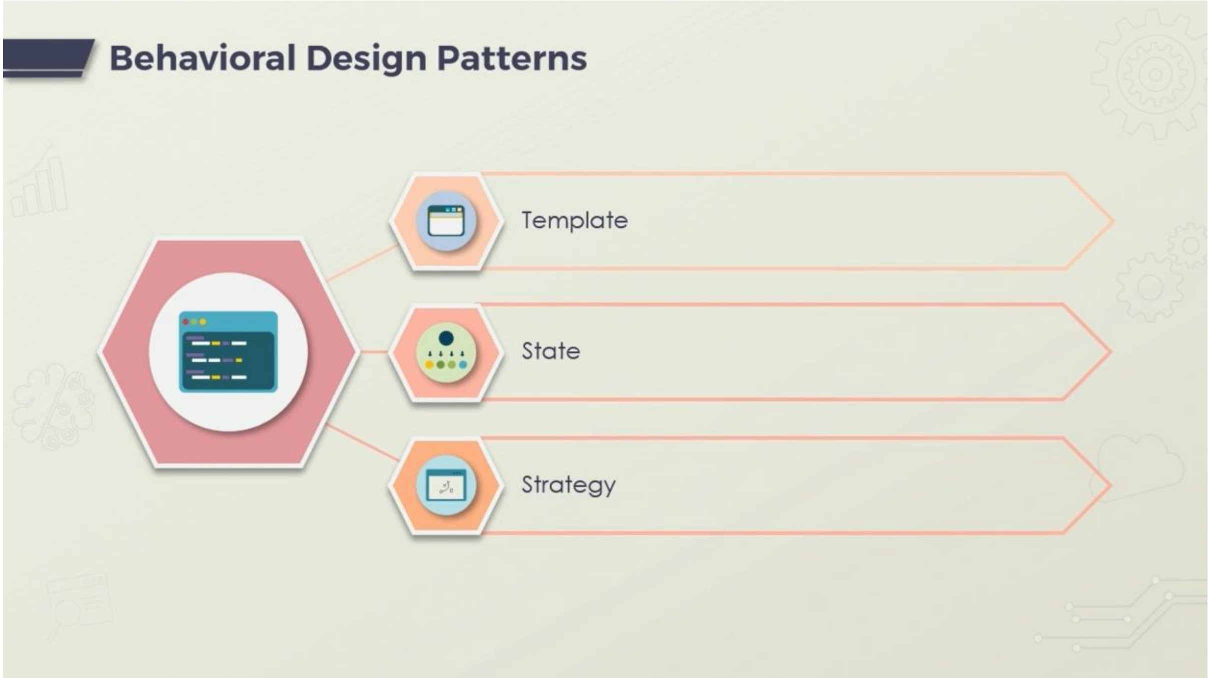


Behavioral Design Patterns

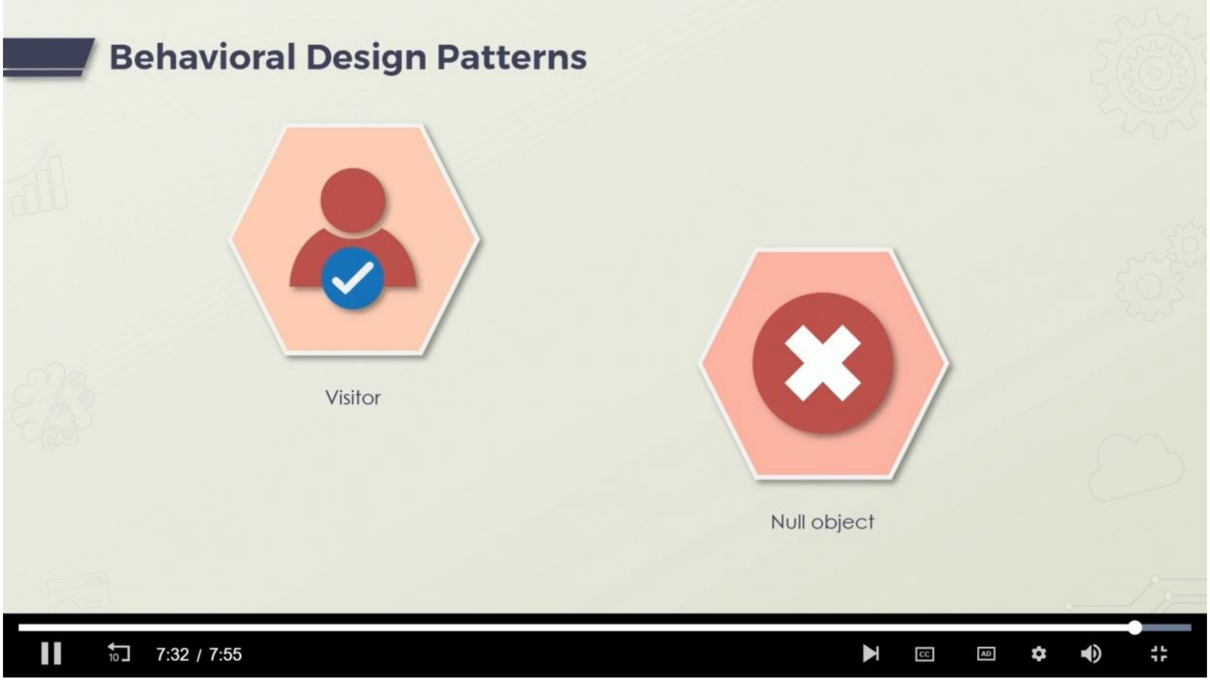
Memento

Observer

Interpreter

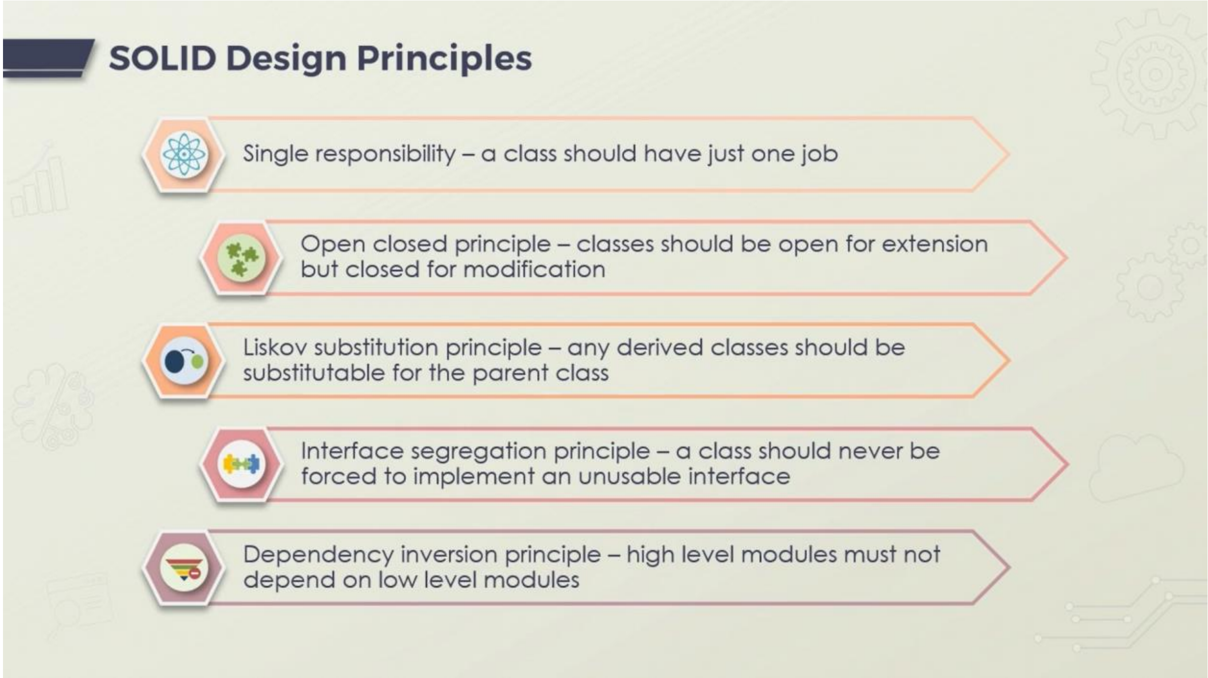


Behavioral Design Patterns



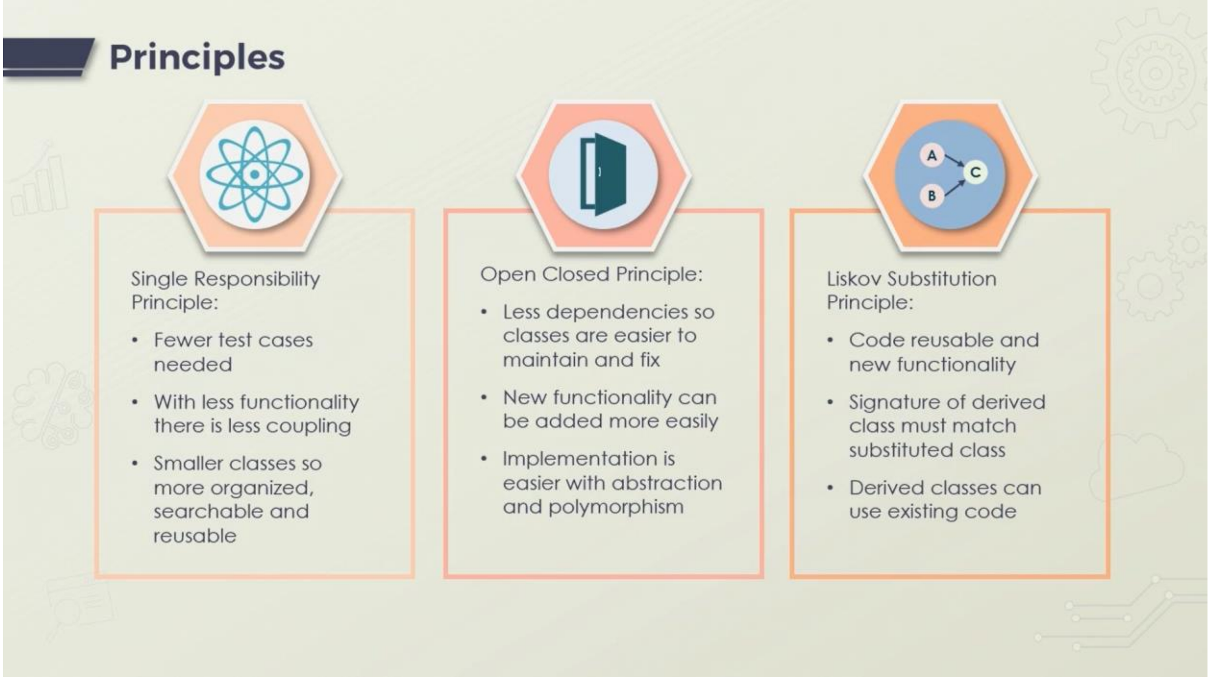
Behavioral Design Patterns

SOLID Design Principles



SOLID Design Principles

- Single responsibility – a class should have just one job
- Open closed principle – classes should be open for extension but closed for modification
- Liskov substitution principle – any derived classes should be substitutable for the parent class
- Interface segregation principle – a class should never be forced to implement an unusable interface
- Dependency inversion principle – high level modules must not depend on low level modules



Principles

Single Responsibility Principle:

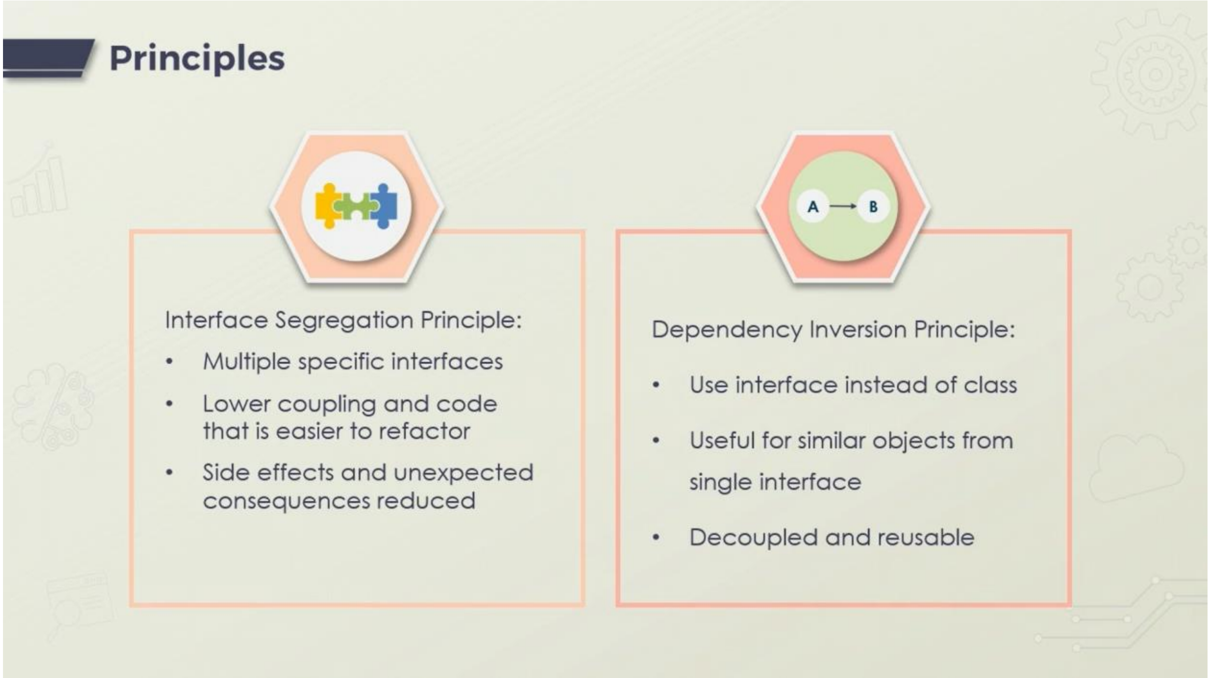
- Fewer test cases needed
- With less functionality there is less coupling
- Smaller classes so more organized, searchable and reusable

Open Closed Principle:

- Less dependencies so classes are easier to maintain and fix
- New functionality can be added more easily
- Implementation is easier with abstraction and polymorphism

Liskov Substitution Principle:

- Code reusable and new functionality
- Signature of derived class must match substituted class
- Derived classes can use existing code



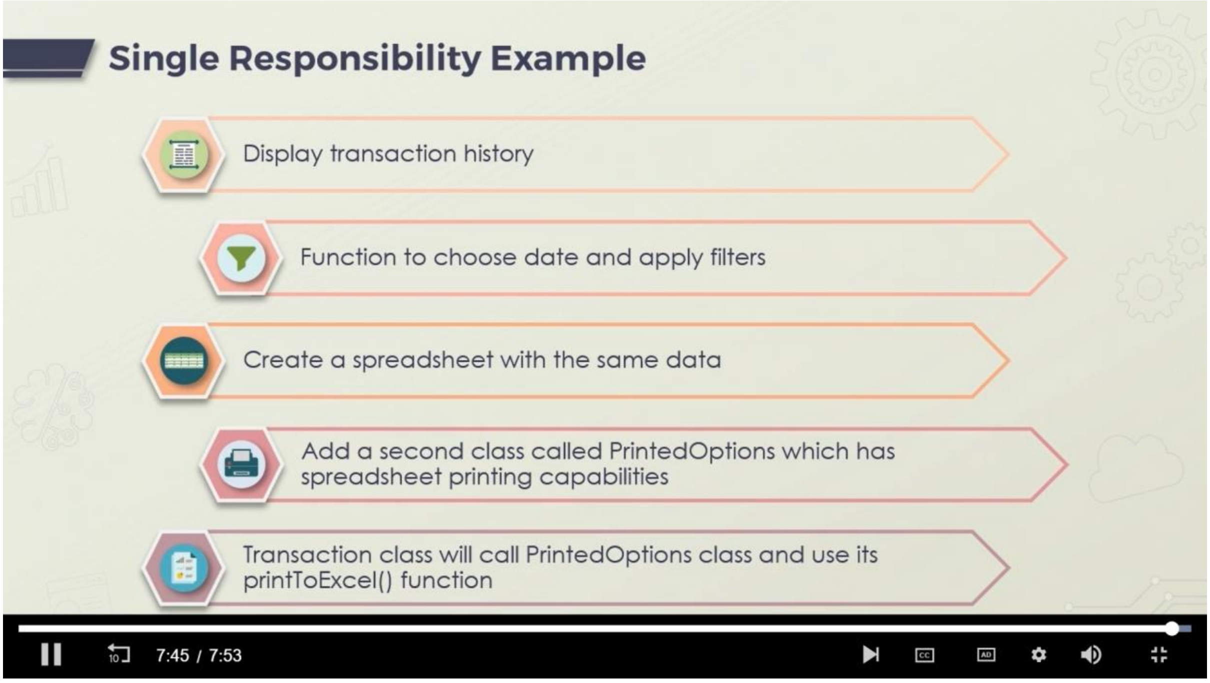
Principles

Interface Segregation Principle:

- Multiple specific interfaces
- Lower coupling and code that is easier to refactor
- Side effects and unexpected consequences reduced

Dependency Inversion Principle:

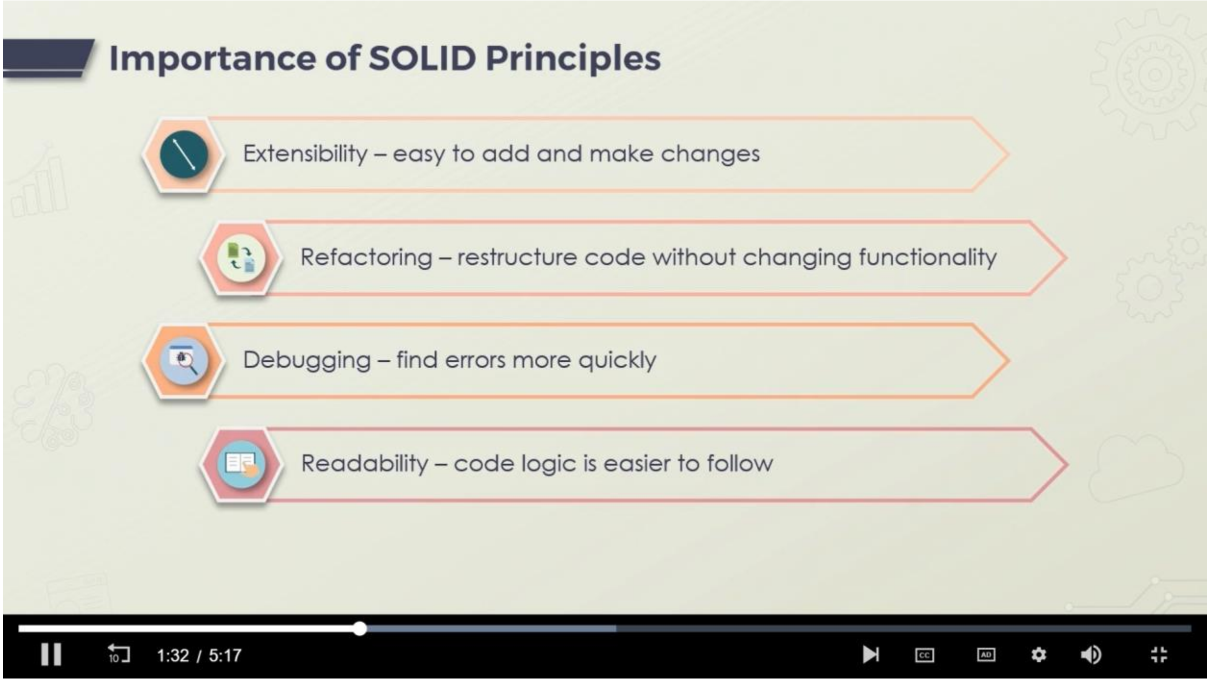
- Use interface instead of class
- Useful for similar objects from single interface
- Decoupled and reusable



Single Responsibility Example

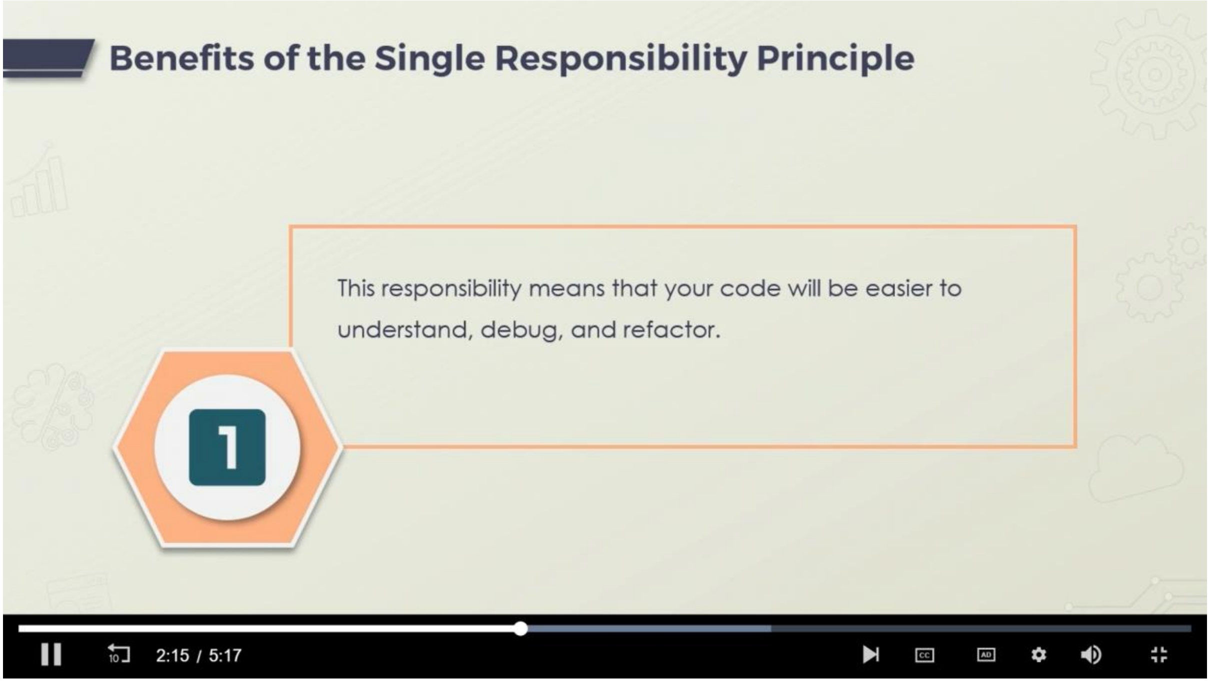
- Display transaction history
- Function to choose date and apply filters
- Create a spreadsheet with the same data
- Add a second class called PrintedOptions which has spreadsheet printing capabilities
- Transaction class will call PrintedOptions class and use its printToExcel() function

Need for solid design principles



Importance of SOLID Principles

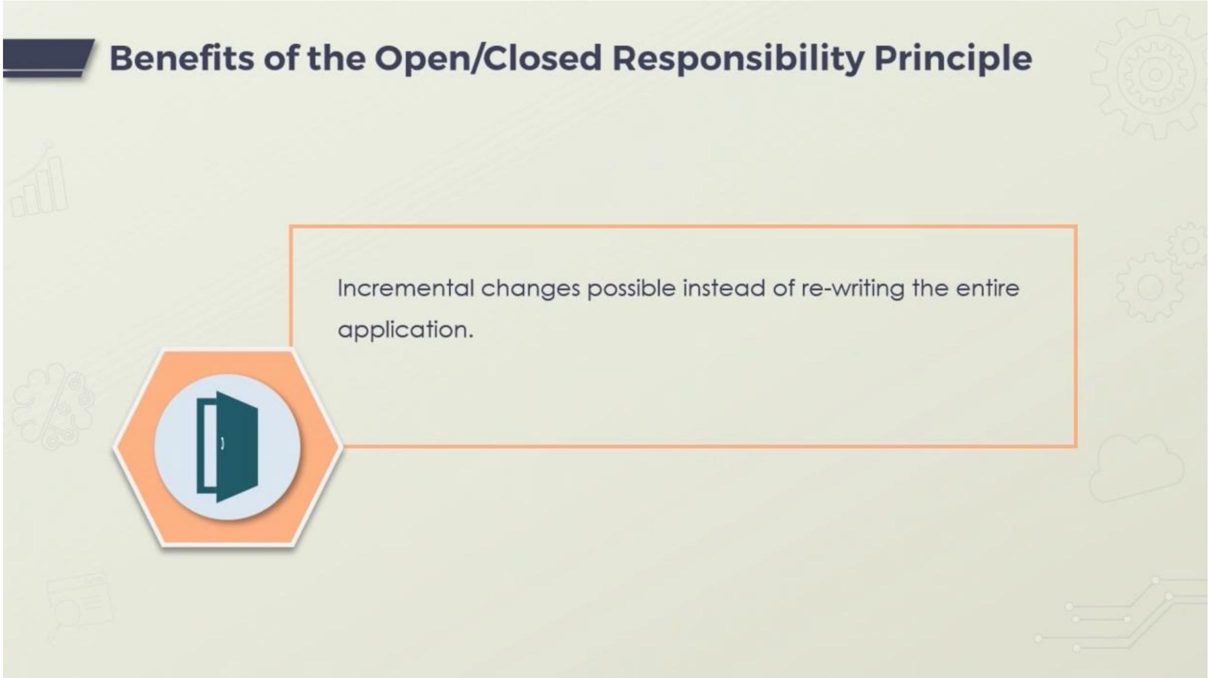
- Extensibility – easy to add and make changes
- Refactoring – restructure code without changing functionality
- Debugging – find errors more quickly
- Readability – code logic is easier to follow



Benefits of the Single Responsibility Principle

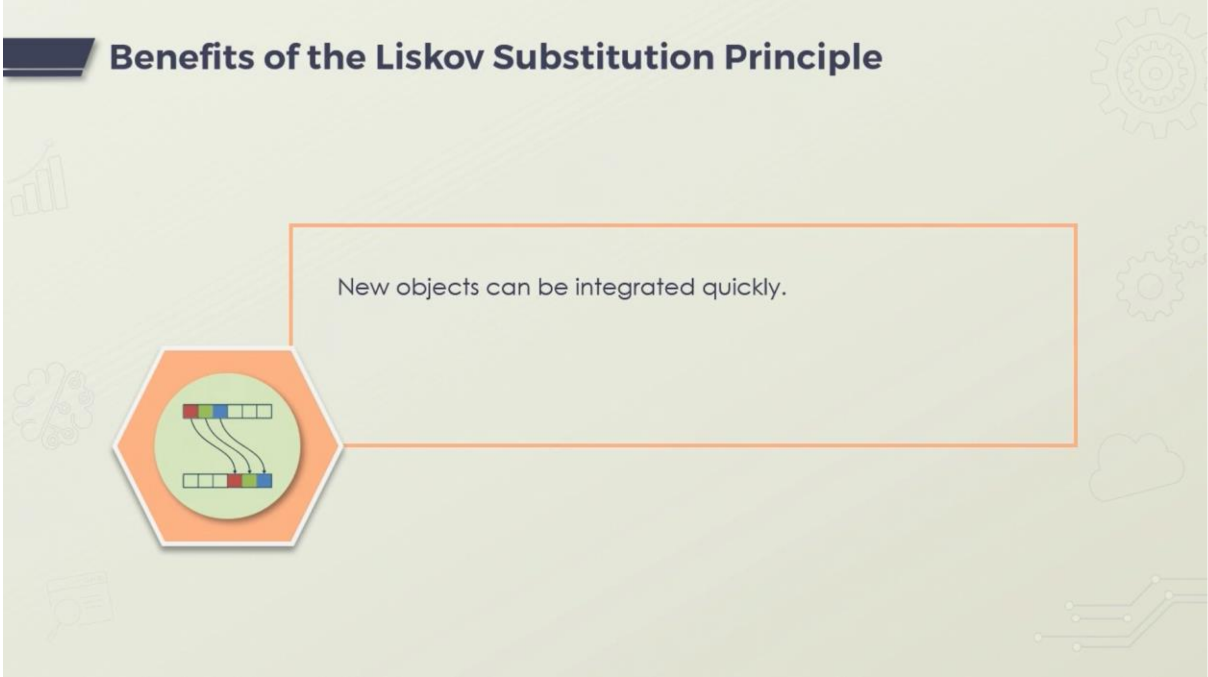
1

This responsibility means that your code will be easier to understand, debug, and refactor.



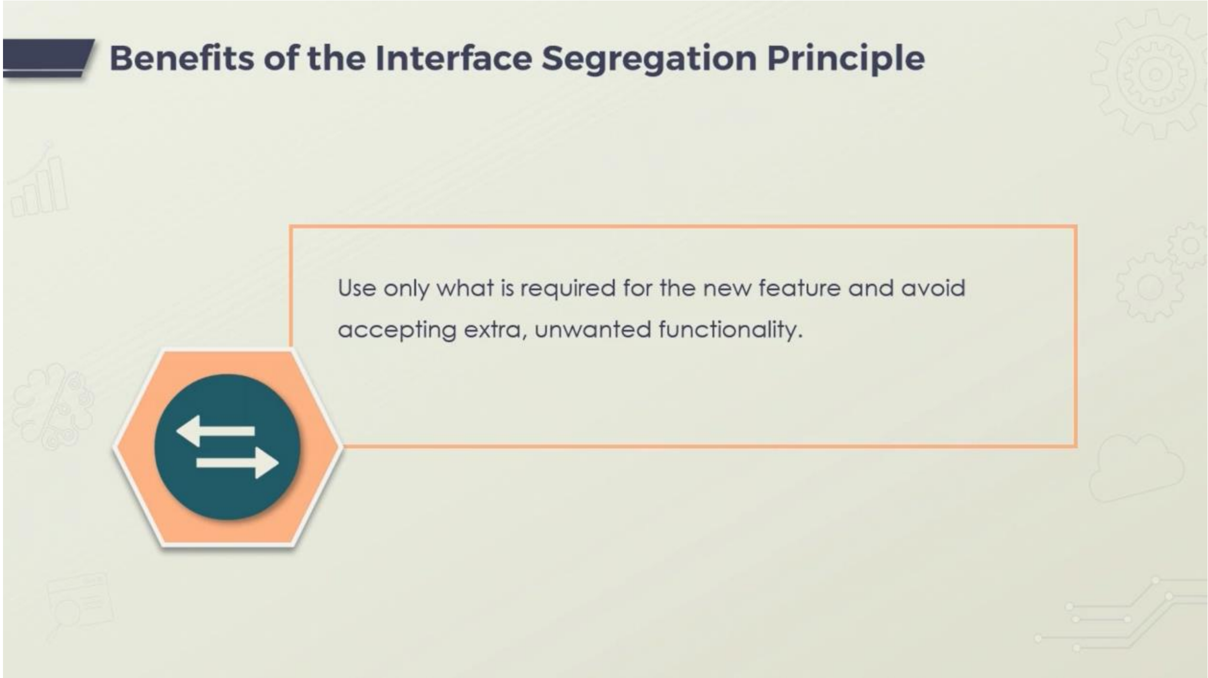
Benefits of the Open/Closed Responsibility Principle

Incremental changes possible instead of re-writing the entire application.



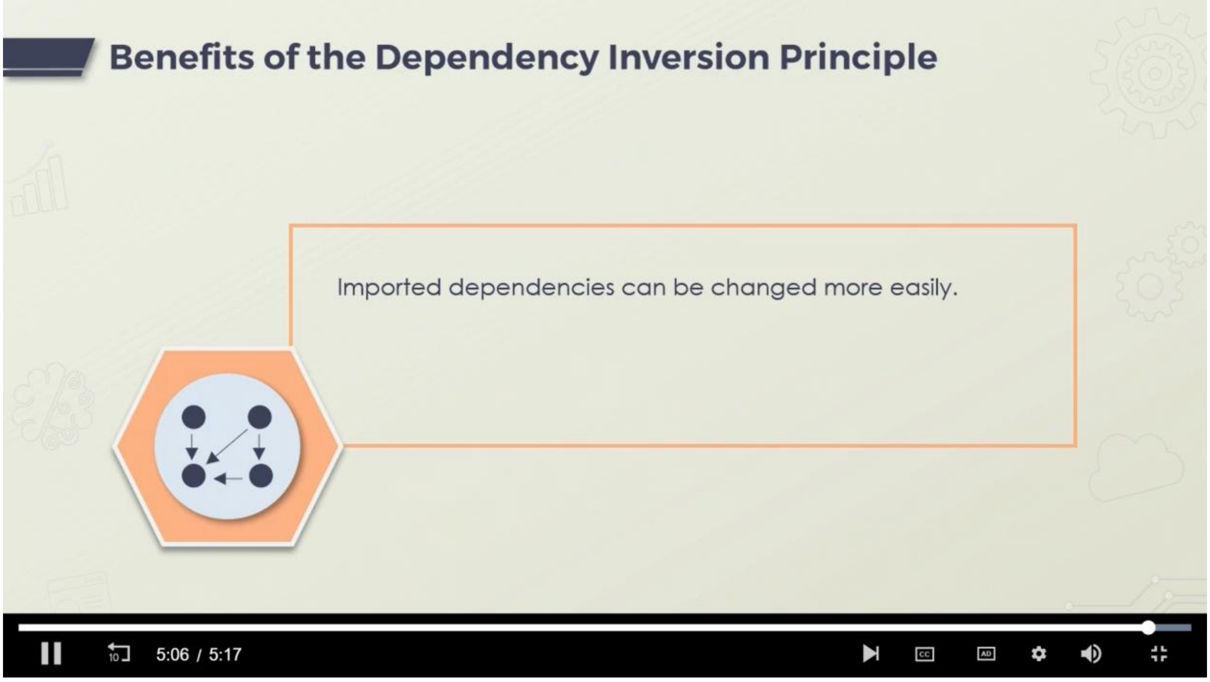
Benefits of the Liskov Substitution Principle

New objects can be integrated quickly.



Benefits of the Interface Segregation Principle

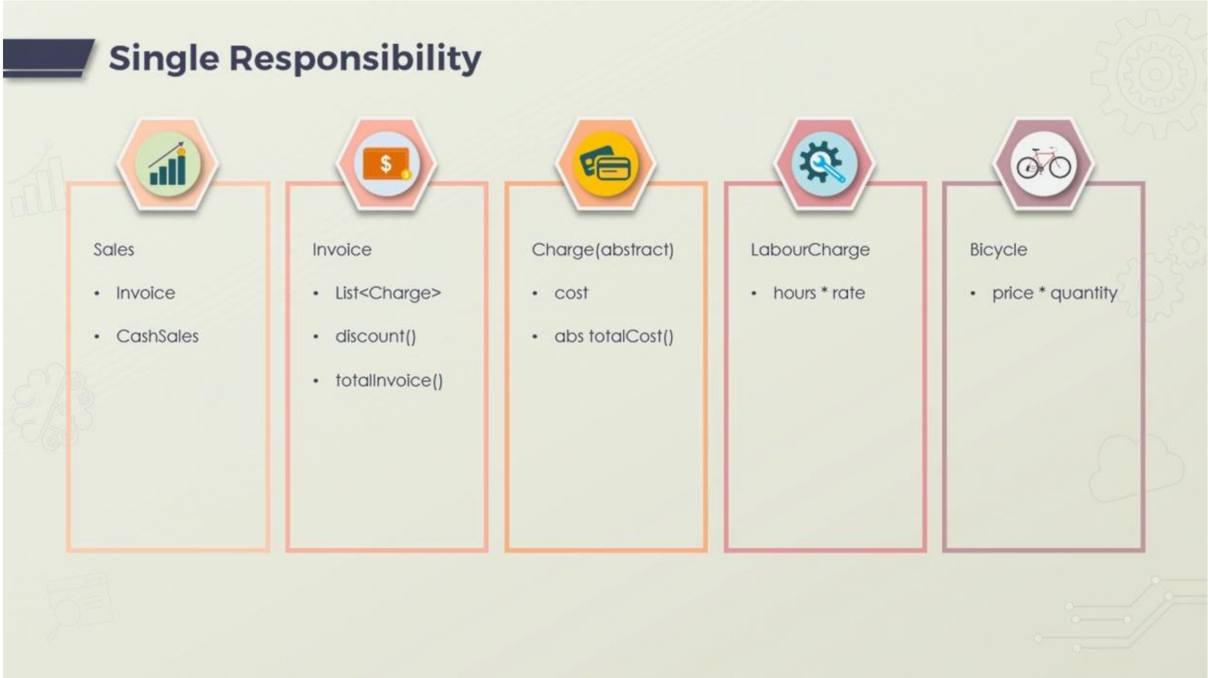
Use only what is required for the new feature and avoid accepting extra, unwanted functionality.



Benefits of the Dependency Inversion Principle

Imported dependencies can be changed more easily.

SOLID design principles in practice



Single Responsibility

Sales

- Invoice
- CashSales

Invoice

- List<Charge>
- discount()
- totalInvoice()

Charge(abstract)

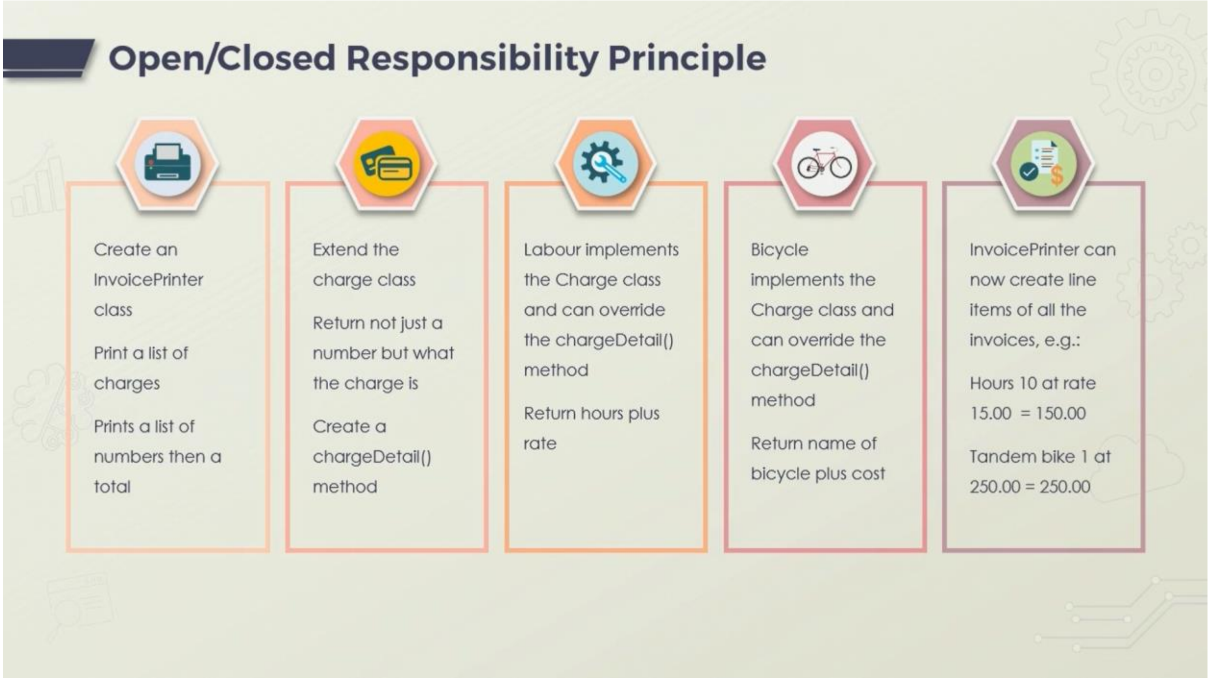
- cost
- abs totalCost()

LabourCharge

- hours * rate

Bicycle

- price * quantity



Open/Closed Responsibility Principle

Create an InvoicePrinter class

Print a list of charges

Prints a list of numbers then a total

Extend the charge class

Return not just a number but what the charge is

Create a chargeDetail() method

Labour implements the Charge class and can override the chargeDetail() method

Return hours plus rate

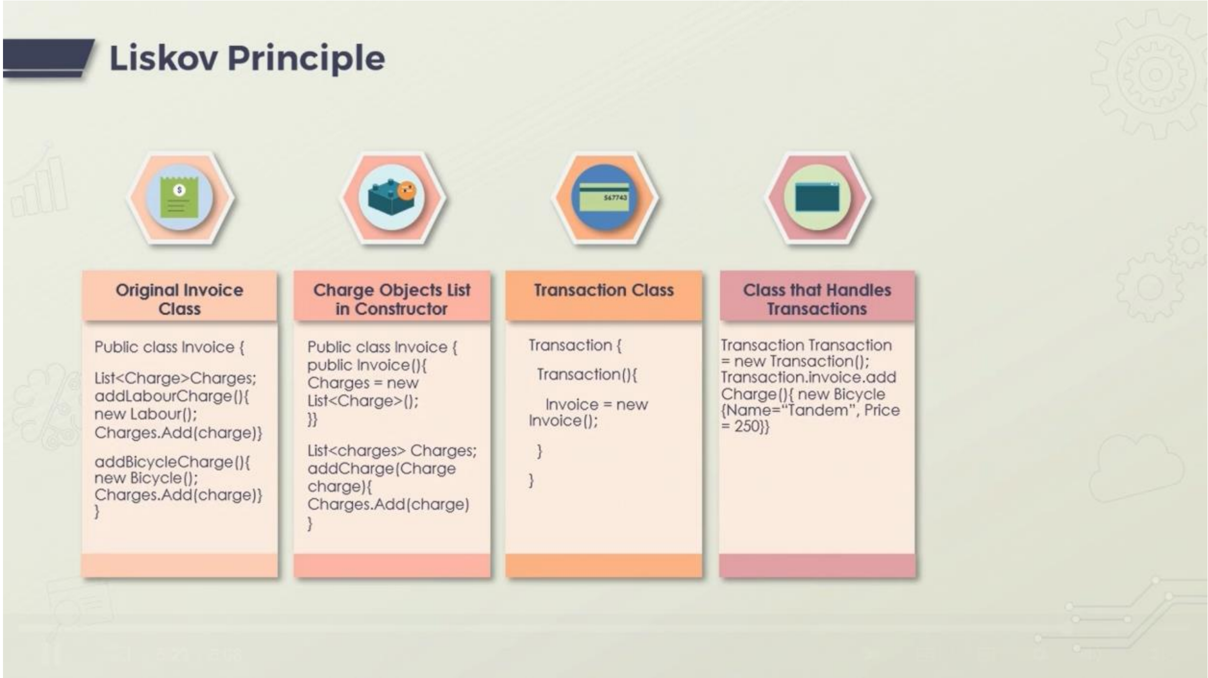
Bicycle implements the Charge class and can override the chargeDetail() method

Return name of bicycle plus cost

InvoicePrinter can now create line items of all the invoices, e.g.:

Hours 10 at rate 15.00 = 150.00

Tandem bike 1 at 250.00 = 250.00



Liskov Principle

Original Invoice Class

```
Public class Invoice {

List<Charge>Charges;
addLabourCharge(){
new Labour();
Charges.Add(charge)}

addBicycleCharge(){
new Bicycle();
Charges.Add(charge)}
}
```

Charge Objects List in Constructor

```
Public class Invoice {
public Invoice(){
Charges = new
List<Charge>();
}}

List<charges> Charges;
addCharge(Charge
charge){
Charges.Add(charge)
}
```

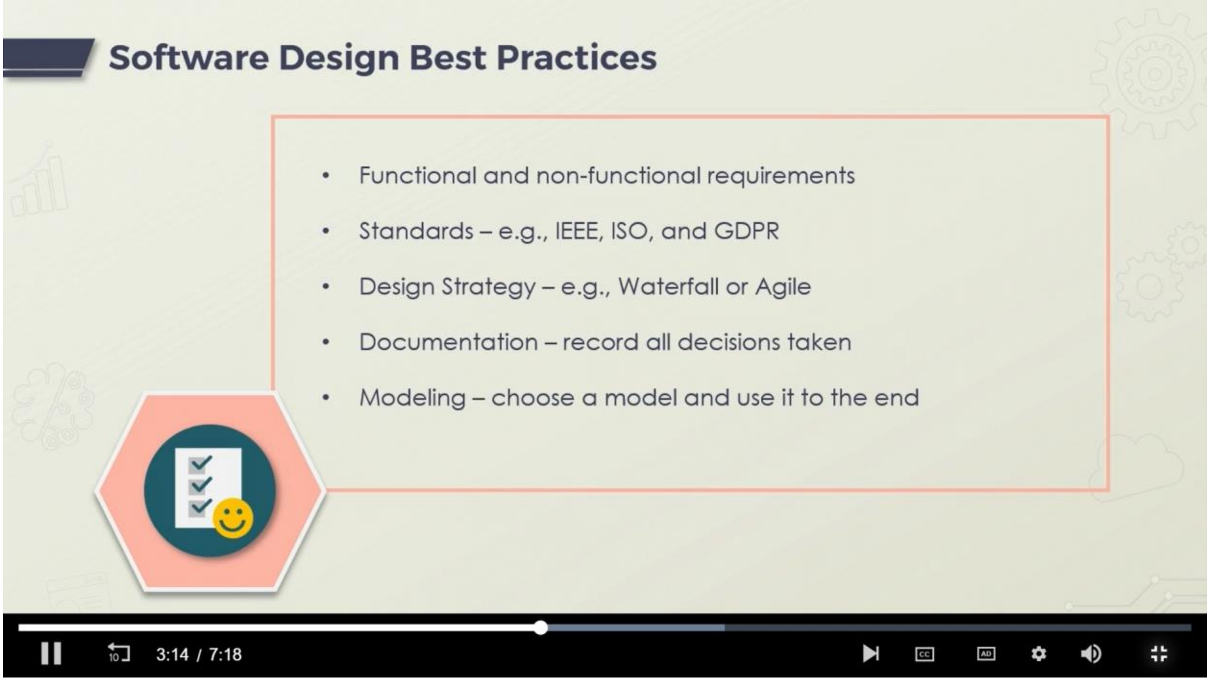
Transaction Class

```
Transaction {

 Transaction(){

   Invoice = new
Invoice();

  }

}
```

Class that Handles Transactions

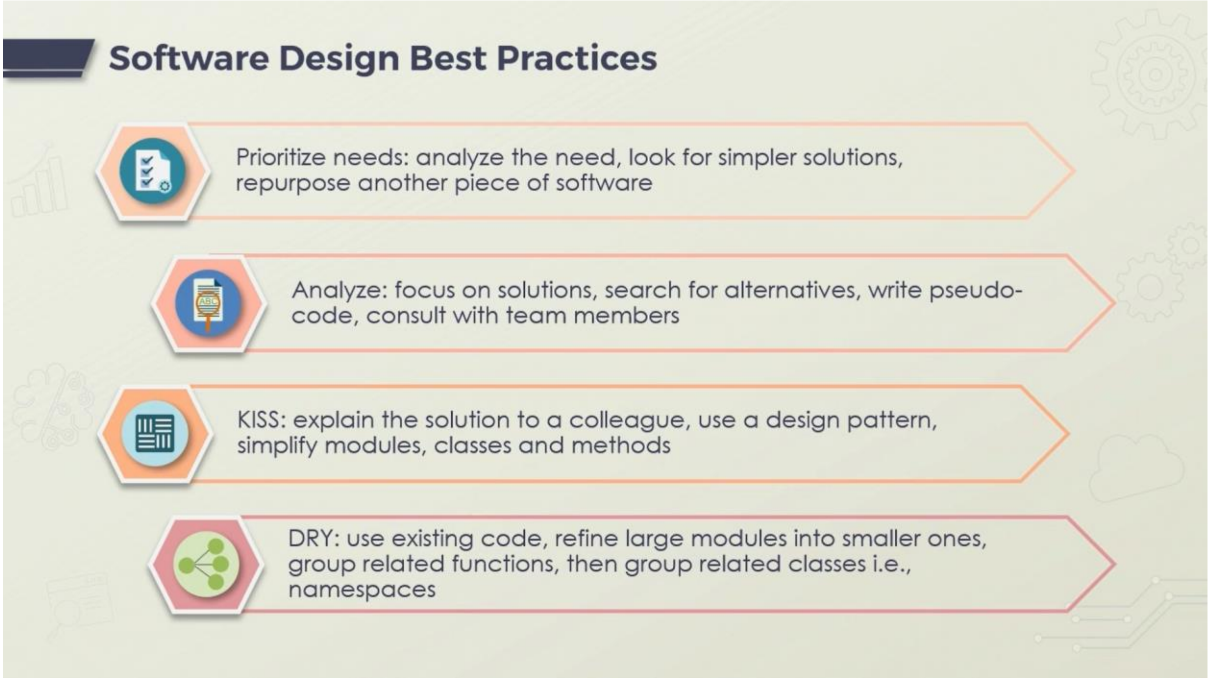
```
Transaction Transaction
= new Transaction();
Transaction.invoice.add
Charge(){ new Bicycle
{Name="Tandem", Price
= 250}}
```

Software design best practices



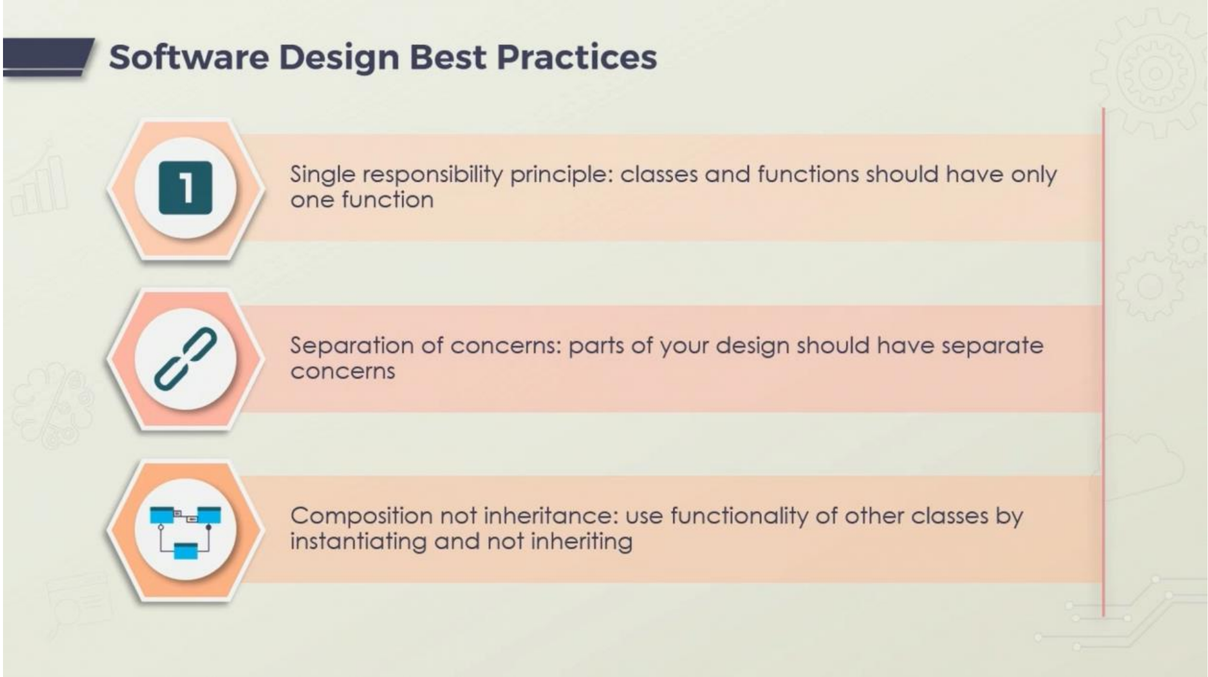
Software Design Best Practices

- Functional and non-functional requirements
- Standards – e.g., IEEE, ISO, and GDPR
- Design Strategy – e.g., Waterfall or Agile
- Documentation – record all decisions taken
- Modeling – choose a model and use it to the end



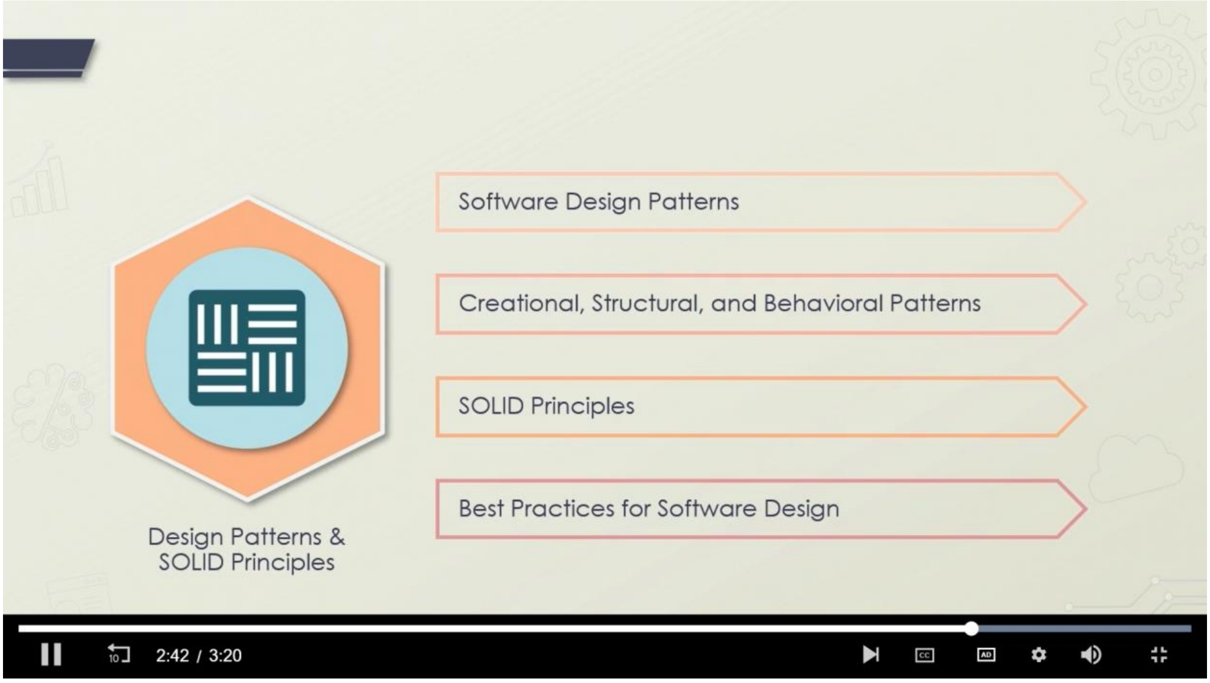
Software Design Best Practices

- Prioritize needs: analyze the need, look for simpler solutions, repurpose another piece of software
- Analyze: focus on solutions, search for alternatives, write pseudo-code, consult with team members
- KISS: explain the solution to a colleague, use a design pattern, simplify modules, classes and methods
- DRY: use existing code, refine large modules into smaller ones, group related functions, then group related classes i.e., namespaces



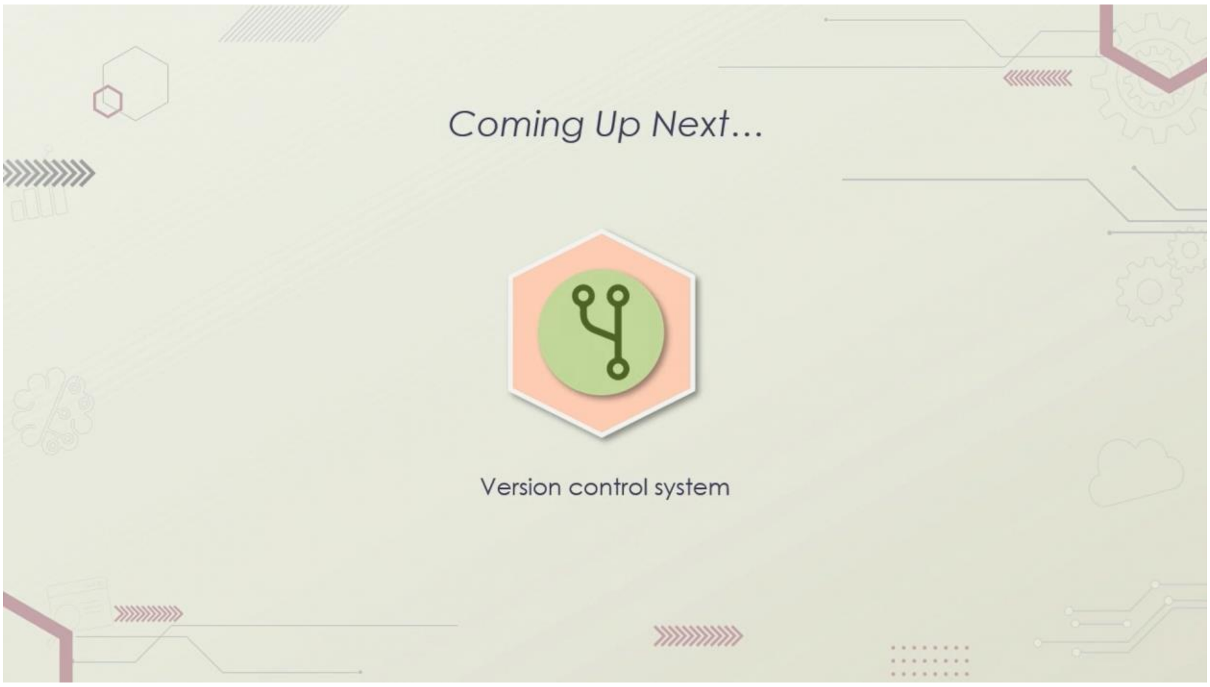
Software Design Best Practices

- Single responsibility principle: classes and functions should have only one function
- Separation of concerns: parts of your design should have separate concerns
- Composition not inheritance: use functionality of other classes by instantiating and not inheriting



Design Patterns & SOLID Principles

- Software Design Patterns
- Creational, Structural, and Behavioral Patterns
- SOLID Principles
- Best Practices for Software Design



Coming Up Next...

Version control system