LECTURE 7

Structural Design Patterns

Adapter, Composite, Decorator, Facade & Proxy

4 Hours | LO4, LO5 | SmartShelf v0.7

Today's Agenda

Hour 1

Adapter & Facade

Interface adaptation: converting external APIs, simplifying subsystems

Hour 2

Decorator & Composite

Runtime behavior: loan decorators, tree-structured library sections

Hour 3

Proxy & Pattern Comparison

Access control, logging, lazy loading. Decision framework.

Hour 4

Workshop: SmartShelf v0.7

Tasks A-E: implement all 5 patterns in SmartShelf

What Are Structural Patterns?

How classes and objects are composed to form larger structures

> Structural patterns deal with object composition. They describe how to assemble objects and classes into larger structures while keeping those structures flexible and efficient.

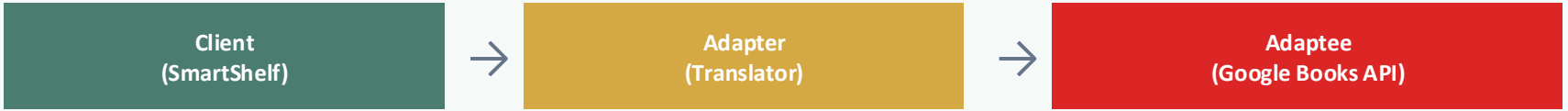
Adapter	Convert one interface to another
Composite	Tree structure of items + groups
Decorator	Add behavior at runtime
Facade	Simplify complex subsystems
Proxy	Control access to an object

Hour 1

Adapter & Facade

Adapter Pattern

Convert an incompatible interface into one the client expects




```
// External API has different method names
class GoogleBooksRecord {
  String getVolumeTitle();    // not getTitle()
  String getAuthorFullName(); // not getAuthor()
  String getIsbn13();         // not getIsbn()
}

// Adapter translates to our interface
class GoogleBooksAdapter extends Book {
  GoogleBooksAdapter(GoogleBooksRecord rec) {
    super(new ISBN(rec.getIsbn13()),
          rec.getVolumeTitle(),
          new Author(rec.getAuthorFullName(), ...),
          rec.getTotalPages(), "EXTERNAL");
  }
}
```

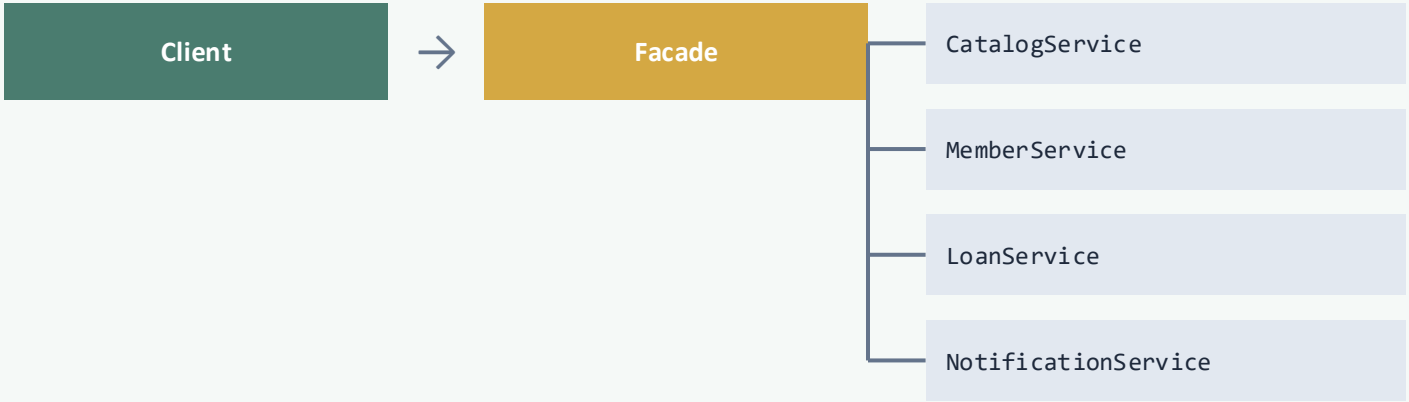
Adapter - When to Use

Use When	Don't Use When
Integrating external/legacy APIs	You control both interfaces
Class has right behavior, wrong interface	Interfaces are already compatible
Need to work with existing code you can't modify	You can modify the target class instead
Multiple adapters for same target (Google, OpenLibrary)	Only one implementation exists

SmartShelf: GoogleBooksAdapter and OpenLibraryAdapter both extend Book — different adaptees, same target.

Facade Pattern

One simple interface to a complex set of subsystems




```
// Client calls ONE method:
facade.borrow("STU001", isbn);

// Behind the scenes: finds member, finds item,
// checks availability, records loan, sends email
// Client doesn't know about 4 subsystems.
```

Hour 2

Decorator & Composite

Decorator Pattern

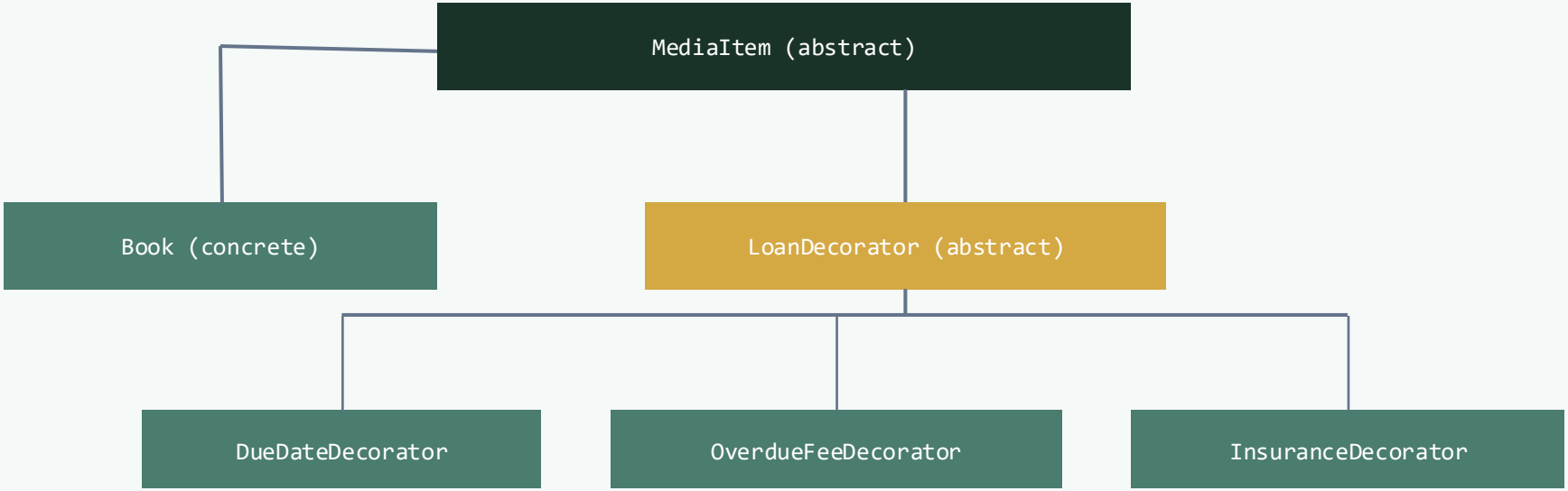
Add behavior at runtime by wrapping objects — stackable layers

```
Book book = new Book(isbn, "Effective Java", ...);

// Wrap with due date
MediaItem withDue = new DueDateDecorator(book, 14);
// "Effective Java..." | Due: 2026-04-11

// Stack: due date + overdue fee + insurance
MediaItem full = new InsuranceDecorator(
    new OverdueFeeDecorator(
        new DueDateDecorator(book, 14), 0.50),
    75.00);
// "Effective Java..." | Due: ... | Fee: $1.50 | Insured: $75

// Still a MediaItem — polymorphism preserved!
full.getTitle();  // works
full.display();   // shows all decorations
```

Each decorator wraps the previous one. Order matters. All remain MediaItem.

Decorator - Structure

Abstract decorator delegates to wrapped object, concrete decorators add behavior



Key insight: LoanDecorator extends MediaItem AND wraps a MediaItem. It delegates all calls to the wrapped object, then adds its own behavior. Concrete decorators override display() to append their info.

Composite Pattern

Treat individual items and collections uniformly through a tree structure

```
[Campus Library] (5 items)
  [Fiction] (1 items)
    - To Kill a Mockingbird
  [Programming] (3 items)
    - Effective Java
    - Clean Code
    [Java] (1 items)
      - Head First Java
  [Media] (1 items)
    - Clean Code Video
```

Participants

Component: LibraryComponent
getName(), getItemCount()

Leaf: SingleItem (one Book)

Composite: Section (has children)

Key benefit

library.getItemCount() works the same whether it's one book or an entire library tree. Client doesn't know or care.

Hour 3

Proxy & Pattern Comparison

Proxy Pattern

Control access to an object — same interface, guarded behavior

```
// Access Control Proxy — checks role before allowing borrow
class AccessControlProxy implements BorrowableItem {
  private BorrowableItem real;  // delegates to real object

  void borrow(String memberId) {
    String role = memberRoles.get(memberId);
    if (!allowedRoles.contains(role))
      throw new SecurityException("ACCESS DENIED");
    real.borrow(memberId);  // delegate if allowed
  }
}

// Logging Proxy — audits all operations (stackable)
class LoggingProxy implements BorrowableItem { ... }
```

Protection Proxy	Access control (roles/permissions)
Logging Proxy	Audit trail for all operations
Virtual Proxy	Lazy loading of expensive objects

Decorator vs Proxy

Same wrapper structure, different intent

	Decorator	Proxy
Intent	Add new behavior	Control existing behavior
SmartShelf	Due dates, fees, insurance	Access control, logging
Client knows?	Usually yes (client wraps)	Often hidden (factory provides)
Stacking	Common (due+fee+insurance)	Less common
Lifecycle	Created by client	Created by framework/factory
Example	+DueDateDecorator(book,14)	AccessControlProxy(book, roles)

Adapter vs Facade

	Adapter	Facade
Intent	Convert ONE interface	Simplify MANY subsystems
Direction	External -> internal	Internal complexity -> simple API
SmartShelf	GoogleBooks -> MediaItem	CatalogService+MemberService+... -> borrow()
Wraps	One class	Multiple classes/services
Changes interface?	Yes (adapts methods)	Yes (creates new simple one)

Pattern Selection Decision Framework

Ask yourself these questions to choose the right pattern

Incompatible external API?	**ADAPTER**
Need to add behavior at runtime without changing the class?	**DECORATOR**
Tree structure where items and groups are treated the same?	**COMPOSITE**
Complex subsystems that need a simple unified interface?	**FACADE**
Need to control access, log, cache, or lazy-load?	**PROXY**

Multiple patterns often work together — Facade uses Adapter internally, Decorator chains use Proxy for access control.

Critical Evaluation - LO4, LO5

Pattern	Utility	Pitfall
Adapter	Integrates legacy/external code	Extra indirection layer
Decorator	Runtime flexibility, stackable	Deep nesting hard to debug
Composite	Uniform tree operations	Overly general interface
Facade	Reduces coupling to subsystems	Can become god object
Proxy	Transparent access control	Hidden behavior surprises caller

Assessment tip: Always argue both sides. Decorator adds flexibility but deep chains are unreadable. Facade simplifies but can hide too much. Show you understand the trade-off with SmartShelf examples.

Hour 4

Workshop: SmartShelf v0.7

Workshop Tasks A - C

Build the patterns - 45 min

Task A

Implement Adapter for external APIs

GoogleBooksAdapter and OpenLibraryAdapter. Both extend Book. Foreign records translated to SmartShelf interface.

Task B

Implement Decorator chain for loans

LoanDecorator abstract wrapper. DueDateDecorator, OverdueFeeDecorator, InsuranceDecorator. Stackable.

Task C

Implement Composite for library sections

LibraryComponent interface. SingleItem (leaf) and Section (composite). getItemCount() works recursively.

Workshop Tasks D - E

Complete patterns + integration - 45 min

Task D

Implement Facade and Proxy

LibraryFacade wraps 4 subsystems. AccessControlProxy and LoggingProxy for borrow operations. Stack proxies.

Task E

Full integration demo

Main.java runs all 5 patterns. Decision framework printed. Compare Decorator vs Proxy, Adapter vs Facade.

Stretch

Refactor poorly structured code

Take a god-class Library and extract Facade, add Decorator where if/else chains exist, Adapter for external calls.

SmartShelf v0.7 - Project Structure

```
src/smartshelf/structural/
  adapter/
    GoogleBooksAdapter.java
  decorator/
    LoanDecorator.java
    DueDateDecorator, OverdueFee, Insurance
  composite/
    LibraryComponent.java
    SingleItem, Section
  facade/
    LibraryFacade.java
    CatalogService, MemberService, ...
  proxy/
    ProxyDemo.java
    AccessControlProxy, LoggingProxy
```

Run

```
chmod +x run.sh
./run.sh

Requires: Java 17+
```

What's Coming Next

Week 8: Behavioral Design Patterns

Observer, Strategy, Command, Template Method in SmartShelf. Notification subscriptions (Observer), search algorithms (Strategy), undo-able operations (Command).

Preparation: Read Head First Design Patterns Ch. 1 (Strategy), Ch. 2 (Observer). Complete Tasks A-E.

Lecture 7 Summary

🔌 Adapter: translate external APIs to your interface

🎀 Decorator: stack runtime behaviors (due dates, fees)

🌳 Composite: tree of items + groups, treated uniformly

🏛️ Facade: one call hides many subsystems

🛡️ Proxy: control access, log, cache transparently

⚖️ Decorator adds behavior, Proxy controls access

🔄 Adapter converts one interface, Facade simplifies many

SmartShelf v0.7 — structurally sound

Questions?

Next Week: Behavioral Design Patterns