Exploring Inheritance part 2

Dr. Isaac Griffith

Idaho State University

Outcomes

After today's lecture you will:

- Have a deeper understanding of inheritance
- Understand the OCP
- Understand the SDP
- Understand the Factory and Visitor patterns
- Learn about the issues with multiple inheritance

Exploring Inheritance

CS 2263

ROAR

Drawbacks of the Approach

- In general, we have two goals:
  - The system should be easy to build and test
  - The system should be adaptable

Easy to Build and Test

- As we store more and more complexity in `Book` we decrease the ease of build and test
- With two categories of objects we have
  - 10 outcomes to test
- Furthermore, as we increase the amount of combinations in a class we increase the likelihood of human error

Adaptability

Idaho State University | Computer Science

Change is inevitable, and in our system changes takes two forms;

1. Procedures for performing library operations may change

2. We may add new categories of items to the library

- Our current structure hurts us for both of these situations
  - each time we add a new item type, we must modify each method
  - This means we must add new tests for each method

- This approach violates the **OCP**
  - A module must be open for extension but closed for modification
    - **Extension** -> adding new features
    - **Modification** -> converting design to code

OCP

- We want to be able to update the system
  - This will require modifying existing classes
  - But, we want to ensure that classes not directly involved in the change are unaffected
- Often, we can encapsulate changes into a separate class, which must be related to existing classes
  - Inheritance allows this without requiring much modification of other classes

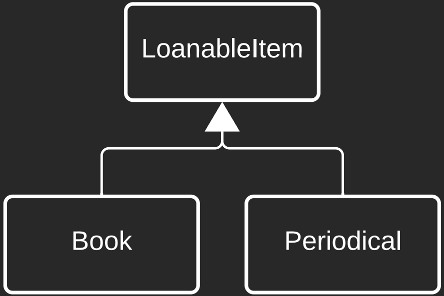
Improving Design

- What classes must change to accomodate the new kinds of items?
  - `Library` -> it is a `Facade` after all
  - `UserInterface`
- Is there a set of guiding principles that can be employed when we introduce inheritance to incorporate the new items?
- Is there a systematic procedure that we can employ when introducing inheritance?

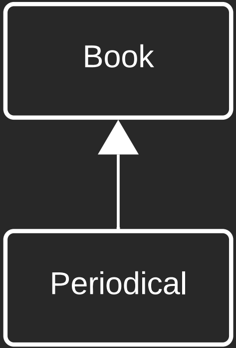
Designing the Hierarchy

- We have two classes: `Book` and `Periodical`
  - Now we need to determine how they are related

Option 1: Common Ancestor

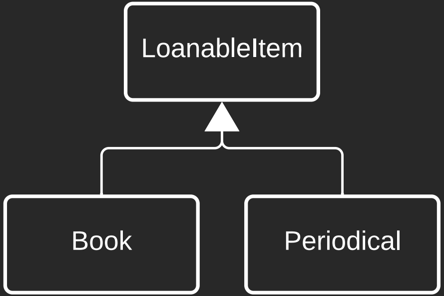


Option 2: One Inherits from the Other



Designing the Hierarchy

Option 1: Common Ancestor



- Create abstract class `LoanableItem`
- Update `Catalog` to be collection of `LoanableItem`s
- Update other classes to use `LoanableItem` rather than `Book`

Advantages

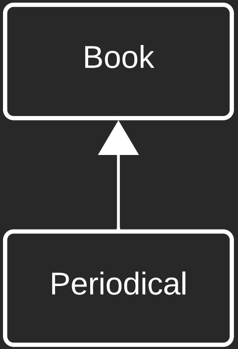
- Treats all items uniformly

Disadvantages

- Updates are tedious

Designing the Hierarchy

Option 2: One Inherits from the Other



- `Book` already exists so we don't need to change other classes

Advantages

- Quick to implement

Disadvantages

- In the long run this approach will be detrimental to the system

Stability

Idaho State University | Computer Science

Stable Dependencies Principle (SDP)

Depend in the direction of stability

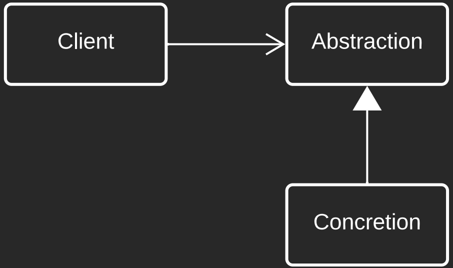
- **Stability:** the amount of work required to be put in to disturb the existing equilibrium
- An unstable system, if left undisturbed, may remain stable for a very long time, but it only takes minimal effort to make it unstable

Stability

- In our system, we want to ensure that `LoanableItem` is very stable, as the hierarchy depends upon it.

- The top-down design of procedural approaches tend to lead to unstable systems, but the bottom-up approach of OO design will tend to lead to stable systems

  - In OO we turn the dependency around such that we specify only the abstraction for which concretions satisfy



Designing the Hierarchy

Rule of Thumb: Depend upon abstractions; avoid depending upon concrete implementations

- Thus, we choose Option 2:

```
public abstract class LoanableItem implements Matchable<String> {
  // code common to all types of items that the library lends
}

public class Book extends LoanableItem {
  // code specific to books
}

public class Periodical extends LoanableItem {
  // code specific to periodicals
}
```

- Which implies that we need to update several classes, including:
  - `Library`
  - `Catalog`
  - `Member`

Invoking the Constructors

- We move our calls to constructors into `Library` under method `addLoanableItem()`
  - here we need to add switching for each type
  - as we add new types, we would need to update this method
  - this does shift us to refer to the abstract type and lets **dynamic binding** take care of the rest
  - **Note:** the conditionals for calling the constructors cannot be eliminated
- We now want to protect `Library` from being affected by changes to the hierarchy

Invoking Constructors

Idaho State University | Computer Science

```
private static final int BOOK = 1;
private static final int PERIODICAL = 2;
public void addLoanableItem() {
  LoanableItem result;
  do {
    String typeString = getToken("Enter type: " +
                                BOOK + " for books\n" +
                                PERIODICAL + " for periodicals\n");
    int type = Integer.parseInt(typeString);
    String title = getToken("Enter title");
    String author = null;
    if (type == BOOK) {
      author = getToken("Enter author");
    }
    String id = getToken("Enter id");
    result = library.addLoanableItem(type, title, author, id);
    if (result != null) {
      System.out.println(result);
    } esle {
      System.out.println("Item could not be added");
    }
    if (!yesOrNo("Add more items?")) {
      break;
    }
  } while (true):
}
```

Calling Constructors

Idaho State University | Computer Science

- We have three options to construct out items in the `Library` class
    1. We could extend `Library` to have subclasses specific to each type we want to create
    2. We could move the creation logic to `LoanableItem` (the abstract superclass)
    3. We could create a class to take care of creating items

Calling Constructors

- We have three options to construct out items in the `Library` class
    1. We could extend `Library` to have subclasses specific to each type we want to create
        - This doesn't make sense as all classes which depend on `Library` would need to change. Since this is a facade that is a lot of classes.
    2. We could move the creation logic to `LoanableItem` (the abstract superclass)
    3. We could create a class to take care of creating items

ROAR

Calling Constructors

- We have three options to construct out items in the `Library` class
  1. We could extend `Library` to have subclasses specific to each type we want to create
  2. We could move the creation logic to `LoanableItem` (the abstract superclass)
     - Seems logical, but it defeats the purpose of having an inheritance hierarchy in the first place as the superclass depends on its children.
  3. We could create a class to take care of creating items

Calling Constructors

- We have three options to construct out items in the `Library` class
  1. We could extend `Library` to have subclasses specific to each type we want to create
  2. We could move the creation logic to `LoanableItem` (the abstract superclass)
  3. We could create a class to take care of creating items
     - This is the best choice as we will create a separate class to encapsulate the changes that occur for calling constructors

Creating Inheritance Hierarchies

Some things to remember when creating inheritance hierarchies:

- Do not rush in too soon
  - have a clear data abstraction in mind before creating the hierarchy
- Allow for future expansion
  - define methods to be as general as possible at each level of the hierarchy
  - be generous in defining data types and storage to avoid difficult changes later on
- Make sure the construction is secure
  - choose the right access modifiers for your attributes
  - only expose items that are needed by derived classes
  - the functionality provided by the methods of the base class should not depend on features that can be overridden

A Simple Factory

- A **Factory** is employed when we wish to make a system independent of how its products are created, composed, and represented

- Here, we want `Library` to be independent of the construction of items.

- Thus, we create the `LoanableItemFactory`

A Simple Factory

Idaho State University | Computer Science

```java
public class LoanableItemFactory {
  private static final int BOOK = 1;
  private static final int PERIODICAL = 2;
  private static LoanableItemFactory lFactory;
  private LoanableItemFactory() { }

  public static LoanableItemFactory instance() {
    if (lFactory == null)
      return lFactory = new LoanableItemFactory();
    else
      return lFactory
  }

  public LoanableItem createLoanableItem(int type, String title,
                                         String author, String id) {
    switch (type) {
      case BOOK:
        return new Book(title, author, id);
      case PERIODICAL:
        return new Periodical(title, id);
      default:
        return null;
    }
  }
}
```

```java
public LoanableItem addLoanableItem(int type, String title,
                                    String author, String id) {
  LoanableItemFactory factory = LoanableItemFactory.instance();
  LoanableItem item = factory.createLoanableItem(type, title, author,

  if (item != null) {
    if (catalog.insertLoanableItem(item))
      return item;
  }
  return null;
}
```

Distributing Responsibilities

Idaho State University Computer Science

- This perhaps is the most difficult part of hierarchy changes

- **Goal:** to keep client classes unaware of the structure of the hierarchy
  - Thus, any method called by the client must exist in the abstract superclass (`LoanableItem`)
  - We also need to store fields and assign access modifiers based on these considerations

```
public abstract class LoanableItem implements Serializable, Matchable<String> {
  private String title;
  private String id;
  private Member borrowedBy;
  protected Calendar dueDate;

  public boolean matches(String other) { return this.id.equals(id);  }

  public String getTitle() { return title; }

  public String getId() { return id; }

  public Member getBorrower() { return borrowedBy; }

  public String getDueDate() { return (dueDate.getTime().toString()); }
  // other fields and methods
}
```

Distributing Responsibilities

Idaho State University | Computer Science

- Fields `title` and `id` are immutable -> `private`
- Other fields are `protected` so they can be accessed by subclasses
- Methods like `getAuthor()` are a specialization of `Book` -> left out
- Methods for processing holds are similar for all items -> implemented in superclass

```
protected List holds = new LinkedList();
protected String author;

public String getAuthor() { return ""; }
public Iterator getHolds() { return holds.iterator(); }
public void placeHold(Hold hold) { holds.add(hold); }
public void removeHold(String memberId) {
  for (ListIterator iterator = holds.listIterator();
                  iterator.hasNext(); ) {
    Hold hold = (Hold) iterator.next();
    String id = hold.getMember().getId();
    if (id.equals(memberId)) {
      iterator.remove();
    }
  }
}
```

```
public Hold getNextHold() {
  for (ListIterator iterator = holds.listIterator();
                  iterator.hasNext(); ) {
    Hold hold = (Hold) iterator.next();
    iterator.remove();
    if (hold.isValid()) {
      return hold;
    }
  }
  return null;
}
public boolean hasHold() {
  ListIterator iterator = holds.listIterator();
  if (iterator.hasNext()) {
    return true;
  }
  return false;
}
```

Consequences of Introducing Inheritance

- **Exception Handling in Inheritance Hierarchies**
  - **Rule:** A subclass that overrides a method of a superclass may not throw an exception that is not thrown by the superclass method
  - This is a consequence of the **LSP**
  - We can get around this limitation by using an **Exception Hierarchy**

Introducing Inheritance by Refactoring

- In the first attempt at changing the library system we introduced the code smell **Switch Statements** where we were selecting the behavior of the `Book` class based on the value of a field.

- Whenever we see code like this we should immediately be thinking that we can replace this with polymorphism. Luckily, there is such a refactoring that can guides us in this process.

Replace Conditional with Polymorphism

If you have a conditional that chooses different behavior depending on some feature of the object, move each leg of the conditional to an overriding method in a (possibly newly defined) subclass and make the original method abstract.

Steps in this Refactoring

1. Identify a conditional statement in a method that changes its behavior based on the value stored in a particular field.
   - **Note:** in a large class there will be several methods that switch on a field
2. If the conditional statement is part of a larger method, the conditional may have to be extracted using the **Extract Method** refactoring.
   - **Note:** if the extract cannot be easily done, you may need to re-evaluate the class more closely
3. Define an inheritance hierarchy where the subclasses reflect the variations in the field on which we are switching
4. Create a subclass method that overrides the conditional statement method
   - Copy one leg of the conditional into each of the subclass methods
   - Adjust the code so it fits
5. Remove the conditional from the superclass method and make the method abstract.
   - If appropriate remove the field on which the switching was done

Adding New Functionality

- Replacing a class with a hierarchy poses additional problems when new functionality needs to be added

- Problems that can occur are as follows:
  - The design may be tailored too specifically to one kind of operation
    - Making it difficult to add new ones
  - The design is tailored for one class and cannot accommodate a hierarchy
    - Does not allow for subclass specific operations

Visitor Pattern

Idaho State University | Computer Science

- The standard solution for dealing with these issues is the **Visitor Design Pattern**

- **Pattern Intent:** to represent an operation to be performed on the elements of an object structure, and is employed to define a new operation without changing the classes of elements on which it operates

- The base principle is to **Encapsulate What Varies** by creating a separate structure that accommodates the changes and shields the classes that must be kept stable

Visitor Pattern

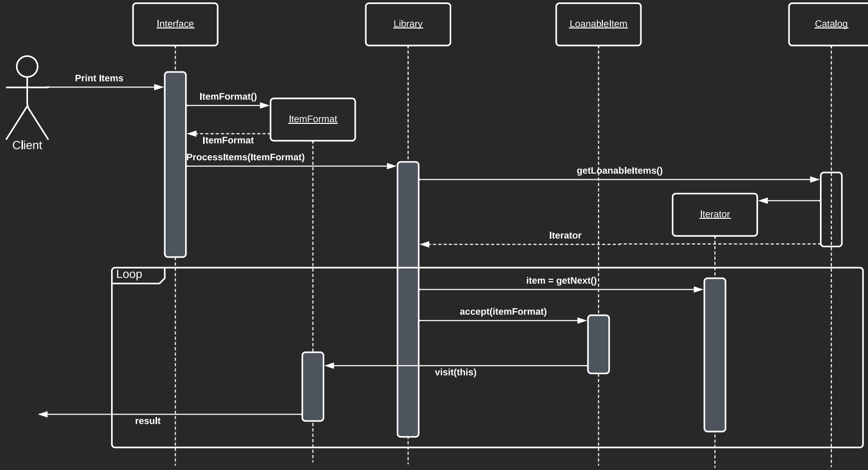
Description

- **Visitor Interface:** encapsulates the variability in the object structure

- `accept` **method**: required in each visitee
  `public void accept(Visitor v)`

- Concrete class for the required functionality that implements the visitor interface

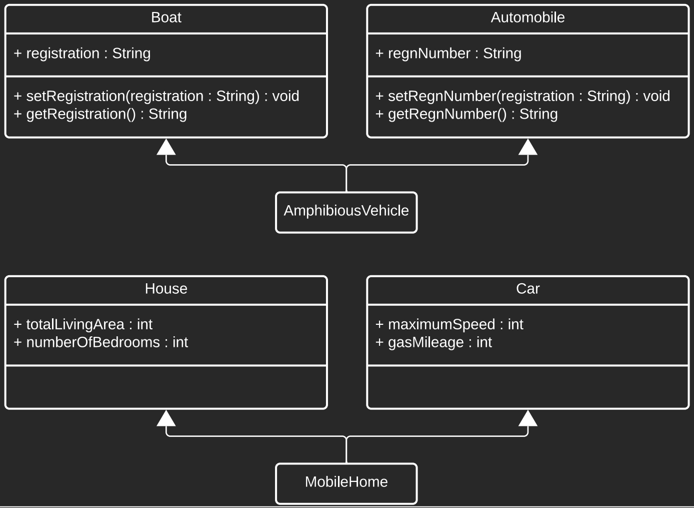
Visitor Behavior



Multiple Inheritance

- **Multiple Inheritance:** ability of a class to subclass multiple classes

Multiple Inheritance



Resolving Conflicts

Idaho State University | Computer Science

AmphibiousVehicle

Boat
+ registration : String = "xyz"
+ setRegistration(registration : String) : void + getRegistration() : String

Automobile
+ regnNumber : String = "abc"
+ setRegnNumber(registration : String) : void + getRegnNumber() : String

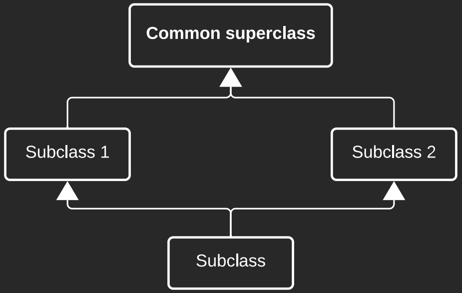
Since we have redundant concepts we need a way to resolve this

1. “un-inherit” specific components to resolve the ambiguity
2. declare unwanted components abstract

ROAR

Repeated Inheritance

- Also called the diamond problem
  - This could allow for multiple invocations of ancestor method
  - Requires deep knowledge of the hierarchy
    - This may not be available (as in the case of an API or library)
  - No real good way of handling issues this creates



Multiple Inheritance in Java

- Java does not allow multiple inheritance in the traditional sense
- Allows only inheriting an implementation from a single class
- We can implement an interface, and as these are types, we have a form of multiple inheritance

For Next Time

- Review this Lecture
- Review Chapter 9.5 - 9.9

Are there any questions?