# CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2021
Doc 19 Assignment 3
Dec 2, 2021

public class Caretaker {

public class Originator {

```
public class Invoker {
    Command theCommand;

    public Invoker() { }

    public Invoker(Command newCommand) {
        theCommand = newCommand;
    }

    public void execute() throws IOException {
        theCommand.execute();
    }
}
```

```
    public void addBook(Book book) throws IOException {
        invoker = new Invoker(new AddBookCommand(inventory, book));
        invoker.execute();
    }

    public void addBook(Book book) throws IOException {
        command = new AddBookCommand(inventory, book);
        command.execute();
    }
```

```
class CommandInvoker():
    def __init__(self, command):
        self._command = command

    def set_command(self, command):
        self._command = command

    def get_command(self):
        return self._command

    def invoke_command(self, inventory):
        self._command.execute(inventory)

    def save_command(self, directory):
        self._command.serialize(directory)
```

Just use commands directly

```
public static void saveMemento(Inventory inventory) throws IOException {
    ArrayList<Book> books = inventory.getBooks();
    originator.setState(books);
    caretaker.add(originator.createMemento());
```

# In Inventory Class

```
def save_memento(self):
    """
    Saves a serialized memento of the current state of the Inventory object
    """
    # Create a memento and save the memento's state to the memento data file
    memento = Memento()
    memento.set_state(copy.deepcopy(self.inventory))
    memento.set_state(copy.deepcopy(self._memento_file_name))

    memento_output_file = open(self._memento_file_name, "wb")
    pickle.dump(memento.get_state(), memento_output_file)
    memento_output_file.close()
```

```
class Inventory(ABC):
    """Abstract class for defining the methods in an Inventory class for storing the objects."""
    @abstractmethod
    def add_book(self, book_payload):
        pass

   class InventoryImplementation(Inventory):

       def add_book(self, book_payload):
           index = self.find_book(book_payload.name)
           if index:
           etc.

class InventoryDecorator(InventoryImplementation):

    def add_book_command(self, book_payload):
        command = AddBookCommand(self)
        command.execute(book_payload)
        self.write_cmd_to_file(command.serialize(book_payload))
```

Decorator has to implement same methods as real thing

Here we have to call different method on Decorator

# Decorator Subclassing Real Thing

```
class Inventory {
    public boolean addBook(aBook) { stuff }
    etc.
}
```

```
class InventoryFooDecorator extends  Inventory {
    public boolean addBook(aBook) { different stuff }
    etc.
}
```

At runtime one can do either

```
    Inventory books = new Inventory();
```

or

```
    Inventory books = new InventoryFooDecorator();
```

But once we have

```
    Inventory books = new Inventory();
```

We can not decorate it

# Decorator Subclassing Real Thing

```
class Inventory {
    public boolean addBook(aBook) { stuff }
    etc.
}


class InventoryFooDecorator extends  Inventory {
    public boolean addBook(aBook) { different stuff }
    etc.
}

class InventoryBarDecorator extends  Inventory {
    public boolean addBook(aBook) { even different stuff }
    etc.
}
```

This give us three options

But at runtime we want 4 options
And want to be able to change
which option we are using

Inventory
Inventory decorated by Foo
Inventory decorated by Bar
Inventory decorated by Foo & Bar

Name structure -6

class commandSerialize(Command):

class deSerialize(Command):

class addCommand(Command):

class sellCommand(Command):

class changePriceCommand(Command):

class getInfoCommand(Command):

```
class AddBookCommand(Command):

    def __init__(self, receiver):
        self.__receiver = receiver

    def execute(self, *args):
        self.__receiver.add_book(*args)
```

Now have to keep track of the book separately

```
def __init__(self, inventory):
    self.__initialized = False
    self.__method_count = 0
    self.__inventory = inventory
    self.__add_command = cmd.AddBookCommand(self.__inventory)
    self.__command_io = FileIO('command_log')

    ## deserialize all logged commands
    for command_data in self.__command_io.read():
        command = command_data[0]
        args = command_data[1:]

    if command == 'add_book':
        self.add_book(*args)
    elif command == 'add_copy':
        self.add_copy(*args)
    elif command == 'sell_book':
        self.sell_book(*args)
    elif command == 'change_price':
        self.change_price(*args)

self.__initialized = True
```

Adding commands now requires editing this case statement

# In InventoryDecorator

```
public void storeInMemento(){
    BookStoreMemento bookStoreMemento = new
          BookStoreMemento(bookStoreInventory.getMapOfBookInventory(),
                                    bookStoreInventory.getNextAvailableBookId());
    bookStoreMemento.StoreStateInMemento();
    clearCommandsFromFile();
}
```

```
public HashMap<Integer, Book> restoreFromMemento() {
    BookStoreMemento bookStoreMemento = new
          BookStoreMemento(bookStoreInventory.getMapOfBookInventory(),
                                    bookStoreInventory.getNextAvailableBookId());
    return bookStoreMemento.restoreFromMementoState(bookStoreInventory);
}
```

```
public boolean addNewBook(Book book) {
    Book currentBook = mapOfBookInventory.get(getNextAvailableBookId());
    if (currentBook == null) {
        book.setUniqueId(getNextAvailableBookId() );
        mapOfBookInventory.put(book.getUniqueId(), book);
        return true;
    } else {
        System.out.println("The Book already Exists, Please enter new ID for the book");
        return false;
    }
 }
```

```
    private Command getCommandInstance(CommandDetails commandDetails,
BookStoreInventoryImpl bookStoreInventory) {

        CommandEnum commandEnum = commandDetails.getCommandEnum();
        Book book = commandDetails.getBook();
        Command command = null;

        switch(commandEnum){
            case ADD_NEW_BOOK:
                command = new AddNewBookCommand(bookStoreInventory, book);
                break;
            case ADD_EXISTING_BOOK:
                command = new AddExistingBookCommand(bookStoreInventory, book, book.getQuant
                break;
            case SELL_BOOK:
                command = new SellBookCommand(bookStoreInventory, book.getUniqueId());
                break;
            case UPDATE_PRICE_OF_BOOK:
                command = new UpdatePriceOfTheBookCommand(bookStoreInventory, book.getPrice(
book.getUniqueId());
```



```
        }
```

```
public class WriteCommandJSONToFile {
```

```
public class DataSaveHelper {

    private static int commandCounter = 0;
    private static ArrayList<Command> commands = new ArrayList<>();
    private static Inventory inventory;

    private static Caretaker caretaker;
    private static Originator originator;
```

```
public void execute(ArrayList<Book> books) throws IOException {
    boolean found = false;
    for (Book bookIterator : books) {
        if (Objects.equals(bookIterator.getName(), name) && bookIterator.getQuantity() > 0) {
            found = true;
            bookIterator.decrementQuantity();
            break;
        }
        else if (Objects.equals(bookIterator.getName(), name) && bookIterator.getQuantity() <= 0) {
            throw new NoSuchElementException("the book you want to sell does not exist");
        }
    }
    DataSaveHelper.saveCommand(this);
}
```

```
public class Inventory implements InventoryInterface {

        ArrayList<Book> bookDatabase = new ArrayList<>();
        String CommandFileName = "Command.ser";
        Integer commandFileSizeMax = 10;
        Integer changesCount = 0;
        Memento memento = new Memento();
        InventoryDecorator inventory;
```

Inventory should not know it is decorated

```
class AddBookCommand extends Command implements java.io.Serializable {
        Book book;
        String fileName = "Command.ser";

        AddBookCommand(Book newBook){ this.book = newBook;}

        @Override
        public void execute(Inventory newInventory) {
        newInventory.addBook(book);
        try
        {
                FileOutputStream fileOut = new FileOutputStream(fileName,true);
                ObjectOutputStream out = new ObjectOutputStream(fileOut);
                out.writeObject(this);
                out.close();
                fileOut.close();

        }catch(IOException i)
        {
                i.printStackTrace();
        }
```

public class customExceptions extends Exception

# Inventory Class

```
public void saveState(){

    memento.saveState(bookDatabase);
    File file = new File(CommandFileName);

        try
        {
                file.createNewFile();
        }
        catch (IOException e) {
                e.printStackTrace();
        }
 }
```

```
public boolean execute() {
  return Inventory.getInstance().addBook(name, price, quantity);
}
```

```
public class RecoverInventory {

  public boolean recover() {
    try {
      InventoryCareTaker careTaker = new InventoryCareTaker();
      InventoryMomento momento = careTaker.getMomento();
      Inventory inventory = Inventory.getInstance();
      inventory.restoreMomento(momento);

      CommandLogger commandLogger = new CommandLogger();
      commandLogger.runCommands();
    } catch (Exception e) {
      e.printStackTrace();
      return false;
    }
    return true;
  }
}
```

# How a Decorator?

```
public interface InventoryManager {
   boolean invoke(InventoryCommand command);
}


public abstract class InventoryDecorator implements InventoryManager {
  private InventoryManager decoratee;

  public InventoryDecorator(InventoryManager decoratee) {
    this.decoratee = decoratee;
  }

  @Override
  public boolean invoke(InventoryCommand command) {
    return decoratee.invoke(command);
  }
}
```

What is the point?

Why not just use command?

Why two levels of indirection?

```
public void getState() {
        bookDB = memento.getState();
}
```

# Inventory Class

```
public void getState() {
	bookDB = memento.getState();
}

public void setState() {
	memento.setState(bookDB);
	File file = new File(COMMAND_FILENAME);
	file.delete();
	try {
		file.createNewFile();
	}
	catch (Exception e) {
		e.printStackTrace();
	}
}
```

```
public Memento getState() {
	return new Memento(foo, bar);
}

public void setState(Memento previousState) {
	foo = previousState.foo();
	bar = previousState.bar();
}
```

```
public class AddBookCommand implements InventoryCommand {
    private final Book book;

    public AddBookCommand(Book book) {
        this.book = book;
    }

    @Override
    public boolean execute(Inventory inventory) {
        return inventory.add(this.book);
    }
}
```

```
public class InventoryOperationExecutor {

    public boolean executeOperation( Inventory bookInventory, InventoryOperation inventoryOperation) {
        return inventoryOperation.execute(bookInventory);
    }
}
```

# BookInventory

```
@Override
public void restoreToPreviousState() {
    //do nothing, decorator will take care
}
```

```
public class BookInventoryDecorator implements Inventory {

    final Logger logger = LogManager.getLogger(BookInventoryDecorator.class);
    Inventory decoratedInventory;
    int commandCounter = 0;
    private Command command;
        CommandCareTaker cmdCareTaker = new CommandCareTaker();
```

command

Field being used to pass data from one method to another

## BookInventoryDecorator

```
private void saveState(Book book) throws IOException {
    cmdCareTaker.saveCommand(command, book);
    if(commandCounter == InventoryConstants.COMMAND_LIMIT) {
        decoratedInventory.createMemento();
        cmdCareTaker.deleteCommandFile();
        commandCounter=0;
        logger.info("Saved new memento");
    }
}
```

3 classes deal with saving files

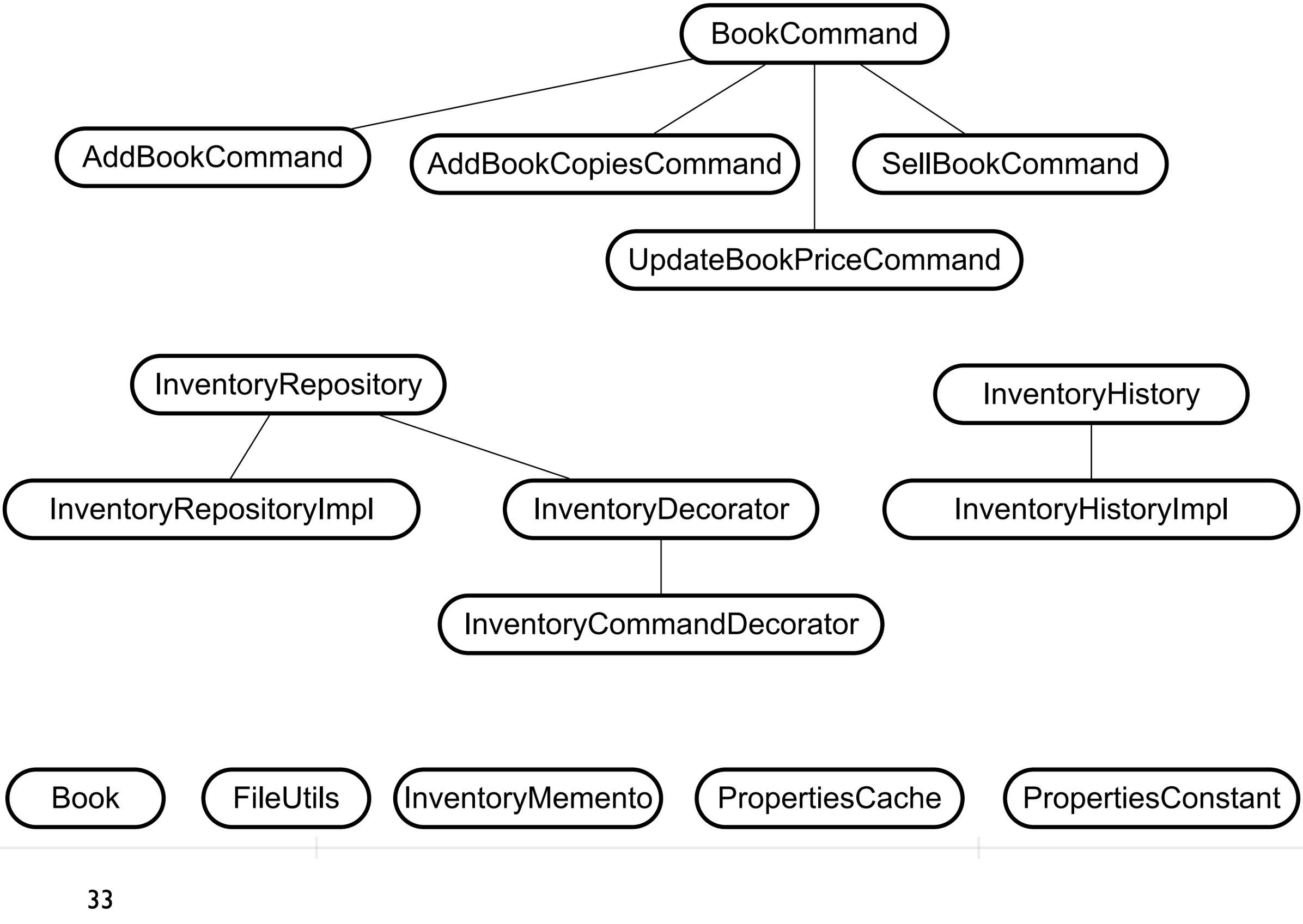
Just put it all in Decorator

## BookInventory

```
public void createMemento() throws IOException {
    caretaker.saveMemento(new InventoryMemento(this.bookStore));
}
```

## InventoryCareTaker

```
public void saveMemento(InventoryMemento memento) throws IOException {
    serialize(memento);
}
```


BookCommand
AddBookCommand
AddBookCopiesCommand
SellBookCommand
UpdateBookPriceCommand
InventoryRepository
InventoryHistory
InventoryRepositoryImpl
InventoryDecorator
InventoryHistoryImpl
InventoryCommandDecorator
Book
FileUtils
InventoryMemento
PropertiesCache
PropertiesConstant

```
public class FileHandler implements Serializable {
  private static final long serialVersionUID = -56419702421747256695L;

  public FileHandler() { }

  public void saveInventoryData(InventoryMemento inventoryObject, String filename) {
    try {
      FileOutputStream file = new FileOutputStream(filename);
      ObjectOutputStream out = new ObjectOutputStream(file);
      out.writeObject(inventoryObject);
    } catch (IOException e) {  e.printStackTrace(); }
  }

  public Object readInventoryData(String filename) {
    try {
     FileInputStream file = new FileInputStream(filename);
      ObjectInputStream in = new ObjectInputStream(file);
     Object mementoObject = in.readObject();
      in.close();
      file.close();
      return mementoObject;
    } catch (EOFException e) {   return null;
    } catch (Exception ex) {  ex.printStackTrace();   }
    return null;
```



```
}
```

# Problem - When save Command Save Inventory

```
public class AddCommand extends Command implements Serializable {
    private Inventory inventory;
    private Book book;

    public AddCommand(Inventory inventory, Book book) {
        this.inventory = inventory;
        this.book = book;
    }

    public boolean execute() {
        inventory.add(book);
        return true;
    }
}
```

When read command from file has reference to wrong inventory

Purpose of Commands was to avoid frequent saving of the inventory

# One Solution

```
public class AddCommand extends Command implements Serializable {
    private Book book;

    public AddCommand(Book book) {
        this.book = book;
    }

    public boolean execute(Inventory inventory) {
        inventory.add(book);
        return true;
    }
}
```

```
Command replay = readNextCommand();
replay.execute(inventory);
```

# Solution 2

```
public class AddCommand extends Command implements Serializable {
    private transient Inventory inventory;
    private Book book;

    public AddCommand(Inventory inventory, Book book) {
        this.inventory = inventory;
        this.book = book;
    }

    public boolean execute() {
        inventory.add(book);
        return true;
    }

    public void setInventory(Inventory inventory) {
        this.inventory = inventory;
    }
}
```

```
Command replay = readNextCommand();
replay.setInventory(inventory);
replay.execute();
```

Two methods have to called together

```
public class AddCommand extends Command implements Serializable {
    private transient Inventory inventory;
    private Book book;

    public AddCommand(Inventory inventory, Book book) {
        this.inventory = inventory;
        this.book = book;
    }

    public boolean execute() {
        inventory.add(book);
        return true;
    }

    public Book getBook() {
        return book;
    }
}
```

```
Command replay = readNextCommand();
String commandName = replay.getClass().getSimpleName();
    switch (commandName) {
        case "AddCommand": {
            Command addCmd = new AddCommand(inventory, ((AddCommand) cmd).getBook());
             addCmd.execute();
              break;
          }
          case "SellCommand": {
            Command addCmd = new SellCommand(inventory, ((SellCommand) cmd).getId(),
((SellCommand) cmd).getQuantity());
             addCmd.execute();
             break;
       }
              etc.
```

Adding a new command requires editing this switch statement

```
public class SaveToFileCommandDecorator extends CommandDecorator {
    private String fileName;

    public SaveToFileCommandDecorator(Command cmd, String fileName) {
        super(cmd);
        this.fileName = fileName;
    }

   public boolean execute() {
       boolean didExecute = super.execute();
       if (!didExecute) { return false;  }
       try {
           FileOutputStream file = new FileOutputStream(fileName, true);
           ObjectOutputStream out = new ObjectOutputStream(file);
           out.writeObject(cmd);
       } catch (IOException e) {
           e.printStackTrace();
       }
       return true;
   }
}
```

# In Decorator

```
public boolean sell(int bookIdToSell){
    try{
        b.takeCommands(new SellBook(this, bookIdToSell));
    }catch (Exception e){
        return false;
    }
    return true;
}
```

```
public class Broker implements java.io.Serializable {
    private final List<Command> commandList = new ArrayList<>();
    private final List<Command> commandsToFile = new ArrayList<>();

    public void takeCommands(Command command){
        commandList.add(command);
    }
```

```
public class BookInventory implements Inventory<Book> {

    private final Memento memento = new Memento();
    private Map<Integer, Book> books = new HashMap<>();
    private BookDecorator concreteBookDecorator;
```

# These should not be fields

```
public class BookInventoryDecorator implements BookInventory {

    private Command command; - should be local variable
    private Memento memento; - not used for anything
    private Object object;  - never used
    private FileInputStream inputFile ; - not a field, just local variable
```