# Software Engineering Design 2016

* + 1. The Law of Demeter states that a class should only be able to talk to its immediate friends. We can identify a violation of the Law of Demeter by looking for long chains of method calls, also know as a ‘train wreck’, for example: obj.getX().getY().getZ().getA();. This is a clear violation since the object talks not only to its immediate collaborators, but also their collaborators, and their collaborators etc.
    2. Violating the Law of Demeter might make code more fragile, since you are exposing internal implementation of distant objects. If these are modified, we may find that objects in the outer part of the chain break unexpectedly.
    3. Tell Don’t Ask.
  1. editorWindow.getTabManager… is the fragile part, as it clearly violates the law of Demeter. Refactoring reduces coupling between classes which subsequently reduces fragility of code.

public class TextEditor {

private final EditorWindow editorWindow;

…

public TextEditor(EditorWindow window) {

this.editorWindow = window;

}

public void openNewFile(String fileName) {

String title = “Untitled”;

if (fileName != null) {

title = fileName;

}

editorWindow.addTabToManager(title);

}

}

public class EditorWindow {

tabManager;

….

public void addTabToManager(title) {

tabManager.addTab(title);

}

public class tabManager {

allTabs;

….

public void addTab(title) {

allTabs.add(0, new Tab(title));

}

}

* 1. Refactor to take a EditorWindow instance as a constructor field, and provide a default constructor that just creates a new EditorWindow to preserve outside functionality.  
     Create a Mock object using the EditorWindow.class, Pass it into the TextEditor using the new constructor. Call the openNewFile and add an expectation for exactly one call to addTab with the correct title. (There should be two tests, one for when fileName is null (and the title is “Untitled”, and one for when it is provided)

@Rule

JunitRuleMockery context = new JunitRuleMockery();

EditorWindow window = context.mock(EditorWindow.class);

TextEditor editor = new TextEditor(window);

@Test

public void nullFileOpenedWithUntitiled() {

context.checking(new Expectations() { {

oneOf(window).addTabToManager(with(“Untitled”));

}});

editor.openNewFile(null);

}

@Test

public void nullFileOpenedWithParameter() {

context.checking(new Expectations() { {

oneOf(window).addTabToManager(with(“File”));

}});

editor.openNewFile(“File”);

}

* 1. Hexagonal Architecture - ‘Ports and Adapters’. Separating core logic of the machine from external code used. We can create adapters which implement the functionality of a third-party and plug them in when needed. This makes it easy to swap between different implementations without impacting the core of our application. This means that we don’t create any direct dependencies between the core logic of our program and any third party libraries.
  2. public void mapToFrames(Function f, int start, int end){

for (Frame frame : frames.sublist(start\*FRAMES\_PER\_SECOND, end\*FRAMES\_PER\_SECOND)) {

f.apply(frame)

}

}

*Don’t think this is right, but don’t think this was taught in 18-19:* External Iterator - Active programm

er controlled e.g. above (although there’s some haziness here when it comes to foreach loops), Internal Iterator is using stream() in Java. See <https://stackoverflow.com/questions/224648/external-iterator-vs-internal-iterator>

* 1. 1. “Edit” class that implements the Command pattern which wraps the behaviour in an object to do later. Thizs takes a VideoClip in a function `execute` and returns an edited VideoClip.
     2. “Editor” class that stores a list of edits, and provides `edit` function to execute the current list of edits and chain these video clips together. Also can update a field for progress per edit. Follows a sequential Publish-Subscribe pattern.f

**public** **class** VideoClip {

**private** **static** **final** int FRAMES\_PER\_SECOND = 25;

**private** **List**<Frame> frames;

**public** **static** applyFilter(int startTime, int endTime, **Filter** filter) {

startFrame = FRAMES\_PER\_SECOND \* startTime;

finishFrame = FRAMES\_PER\_SECOND \* finishTime;

applyFilter(startFrame, endFrame, filter);

}

**private** applyFilter(int startFrame, int endFrame, **Filter** filter) {

framesAfter = frames.subList(finishFrame, frames.size());

frames = frames.subList(0, startFrame);

frames.addAll(applyFilter(frames.subList(startFrame, finishFrame), filter));

frames.addAll(framesAfter);

}

**private** **static** **List**<Frame> applyFilter(**List**<Frame> frames, **Filter** filter) {

**List**<Frame> fitered = **new** **ArrayList**<>();

// Internal Iterator

**for** (iter = frames.iterator(); iter.hasNext(); frame = iter.next()) {

filtered.add(filter.apply(frame));

}

}

}

**interface** Filter {

**public** Frame apply(Frame frame);

}

// Producer/Consumer - Publish/Subscribe Model

**public** **class** ProcessVideoClip {

**private** **Queue**<**Runnable**> queue = **new** **LinkedList**<**Runnable**>();

**private** VideoClip videoClip

**public** ProcessVideoClip(VideoClip videoClip) {

**this**.videoClip = videoClip;

}

**public** void addEdit(VideoEdit videoEdit) {

queue.add(videoEdit);

}

**public** VideoClip getFinishedVideo() {

**return** videoClip;

}

**public** processAllEdits() {

VideoClip editedVideoClip = videoClip;

**for** (**Runnable** edit : queue) {

edit.set(editedVideoClip);

edit.run();

editedVideoClip = edit.get();

}

}

}

// Command Pattern

**public** **class** VideoEdit **implements** Runnable {

**private** VideoClip videoClip;

**private** int startTime;

**private** int endTime;

**private** **Filter** filter;

**public** VideoEdit(int startTime, int endTime, **Filter** filter) {

**this**.startTime = startTime;

**this**.endTime = endTime;

**this**.filter = filter;

}

**public** set(VideoClip videoClip) {

**this**.videoClip = videoClip;

}

**public** get(VideoClip videoClip) {

**return** videoClip;

}

**@Override**

**public** void run() {

videoClip.applyFilter(startTime, endTime, filter);

}

}

* + 1. Sequence UML for a [:DesktopViewer], [:ActionListener] and [:Editor]. Shows a button click on the ActionListener calling to the editor, and the editor notifying the GUI on completion. Used the Builder pattern for turning the button click into an “Edit” Command. Uses JFrame and panels.
  1. Directly relates:
     1. DesktopViewer -> Viewer
     2. ActionListener -> Controller
     3. Editor -> Model
     4. Difference is the Editor directly tells the View to redraw, rather than going through the UI as it has shown in the slides.