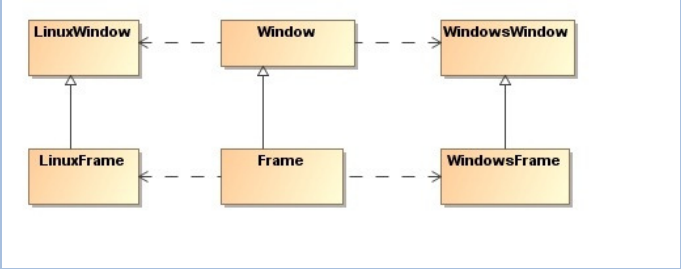
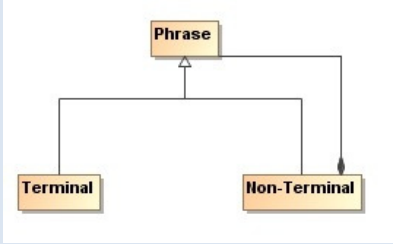


## Fitness for Future of Design Patterns & Architectural Styles

Design patterns are difficult to teach, so we conducted a class collaboration where we all researched and reported on a variety of design patterns and architectural styles. We want to have a sheet where given a problem where specific things need to be changed, we can find which pattern we should use.

### Summary of Design Patterns

	Design Pattern	Future?	Description
Creational	Singleton	No	<p>Restricts the instantiation of a class to a single object or a finite set of objects.</p> <p>What is it about?</p> <ul style="list-style-type: none"> <li>Single global point of access to a single or finite set of objects</li> <li>Controlled instantiation of a single or finite set of objects</li> </ul>
	Adapter	Yes	<p>Translates on interface for a class (adaptee) into a compatible interface (adaptor).</p> <p>What changes?</p> <ul style="list-style-type: none"> <li>The adapter - switch different adapters in and out for different functionality</li> </ul> <p>What doesn't change?</p> <ul style="list-style-type: none"> <li>The adaptee – pre-existing code doesn't change</li> </ul>
Structural	Bridge	Yes	<p>Decouples an abstraction from an implementation so the two can vary independently.</p> <p>What changes?</p> <ul style="list-style-type: none"> <li>Refined abstractions and concrete implementations</li> </ul> <p>What doesn't change?</p> <ul style="list-style-type: none"> <li>Abstraction and abstract implementation classes</li> </ul> <p>Example</p>  <pre> classDiagram     class Window     class Frame     class LinuxWindow     class WindowsWindow     class LinuxFrame     class WindowsFrame      Window &lt; -- LinuxWindow     Window &lt; -- WindowsWindow     Frame &lt; -- LinuxFrame     Frame &lt; -- WindowsFrame     LinuxWindow ..&gt; LinuxFrame     Window ..&gt; Frame     WindowsWindow ..&gt; WindowsFrame     </pre> <ul style="list-style-type: none"> <li>Window is abstraction, Frame is refined abstraction</li> <li>Linux and Windows Window/Frame are Implementers/Concrete Implementers, respectively</li> <li>Window and Frame abstraction can vary independently of Windows and Linux Window and Frame implementations (e.g. new version of Windows: Windows Window/Frame implementations change, new feature of GUI toolkit: Window and Frame abstraction change)</li> </ul>
	Composite	No	<p>Allow uniform treatment of individual objects and compositions of those objects.</p> <p>What is it about?</p> <ul style="list-style-type: none"> <li>Used for iterating through tree structures composed of heterogeneous</li> </ul>

Behavioural			<p>components (e.g. hierarchy)</p> <ul style="list-style-type: none"> <li>• Useful for avoiding having to deal with the difference between branch and leaf nodes</li> <li>• Designed for fitness for purpose over future</li> <li>• Difficult to add new functionality to components since it must be added to each existing class</li> <li>• New classes can be easily added as long as they conform to current interface</li> </ul> <p>Examples</p> <ul style="list-style-type: none"> <li>• Parse tree (below)</li> <li>• File system</li> <li>• Organizational hierarchy</li> </ul> 
	Facade	Yes	<p>Provide a simplified interface to a larger body of code.</p> <p>What changes?</p> <ul style="list-style-type: none"> <li>• Implementation of underlying code can change</li> <li>• Interaction of classes behind the facade can change</li> </ul> <p>What doesn't change?</p> <ul style="list-style-type: none"> <li>• Interface of facade stays the same</li> <li>• Functionality usually can't change</li> </ul> <p>Examples</p> <ul style="list-style-type: none"> <li>• J2EE Web Content (Service Layer)</li> <li>• Linux File System (System Call Interface, Virtual File System)</li> </ul>
	Command	Yes	<p>Encapsulates operations within reusable logic so they may be executed at a different time or on a different thread.</p> <p>What changes?</p> <ul style="list-style-type: none"> <li>• Time and thread of execution</li> </ul> <p>What doesn't change?</p> <ul style="list-style-type: none"> <li>• The implementation – the instance of reusable logic to be executed</li> </ul> <p>Examples</p> <ul style="list-style-type: none"> <li>• GUI Event Loop</li> <li>• Undo/Redo Stack</li> </ul>
	Interpreter	Yes	<p>Specifies how to add functionality to a composite pattern structure.</p> <p>What changes?</p> <ul style="list-style-type: none"> <li>• New methods on composite structure</li> </ul> <p>What doesn't change?</p> <ul style="list-style-type: none"> <li>• The existing composite structure</li> </ul> <p>Example</p> <ul style="list-style-type: none"> <li>• Evaluate mathematical expression – create parse tree composite structure, add method to evaluate</li> </ul>
	Observer	Yes	<p>Defines a one-to-many relationship so that when the state of the single object (producer) changes, all of the dependent objects (consumers) are notified.</p> <p>What changes?</p>

		<ul style="list-style-type: none"> <li>• Producer can be changed without changing the consumers (so long as message format remains the same)</li> <li>• New consumers can be created without change to the producer</li> </ul> <p>What doesn't change?</p> <ul style="list-style-type: none"> <li>• Format of message sent out to consumers cannot change without changing all consumers, as well as the interface producer uses to notify consumer</li> </ul>
Strategy	Yes	<p>Allows algorithm to be selected at runtime.</p> <p>What changes?</p> <ul style="list-style-type: none"> <li>• Can add any number of algorithms as classes</li> </ul> <p>What doesn't change?</p> <ul style="list-style-type: none"> <li>• Interface of each "class" of algorithms stays the same (e.g. sorting routines all take and return a list)</li> </ul>
Visitor	Yes	<p>Adds the ability to add new operations to data structures without changing the structures themselves.</p> <p>What changes?</p> <ul style="list-style-type: none"> <li>• Can add as many new visitors as you like</li> <li>• Can change the operations performed the visitor</li> </ul> <p>What doesn't change?</p> <ul style="list-style-type: none"> <li>• Interface of the underlying data structures</li> <li>• The structures themselves</li> </ul>

## Summary of Architectural Styles

Architectural Style	Future?	Description
Pipes n' Filters	Yes	<p>Allows components to be arranged so that the output of each element is the input of the next.</p> <p>What changes?</p> <ul style="list-style-type: none"> <li>Reuse/reorder pipes and filters in new arrangements</li> <li>Add and remove pipes and filters to aid between conversion of input/output format or to add new functionality</li> </ul> <p>What doesn't change?</p> <ul style="list-style-type: none"> <li>Need to make sure the input stream is in the format expected</li> <li>Individual pipes and filters typically do not change</li> </ul>
Layered/Hierarchical	Yes	<p>Abstracts different portions of functionality into separate layers that build on top of each other.</p> <p>What changes?</p> <ul style="list-style-type: none"> <li>The implementation of a layer can change so long as the interface is held constant</li> <li>When a layer changes, only the layers above and below it must change</li> </ul> <p>What doesn't change?</p> <ul style="list-style-type: none"> <li>Relative ordering of each layer not changed easily (e.g. OS memory management usually below networking layer)</li> </ul>
Repository	No	<p>Provides a centralized data store, with many components.</p> <p>What is it about?</p> <ul style="list-style-type: none"> <li>Two types of components: <ul style="list-style-type: none"> <li>Datastore – holds and represents the system state/data</li> <li>Data-use components – components that operate on the central datastore</li> </ul> </li> <li>Type main classifications of repository <ul style="list-style-type: none"> <li>Database – streams of transactions trigger processes to act on datastore</li> <li>Blackboard – current state/state changes trigger processes</li> </ul> </li> </ul>
Implicit Invocation	Yes	<p>Allows components to listen and react to events.</p> <p>What changes?</p> <ul style="list-style-type: none"> <li>It allows for easier adding of functionality in the form of a separate component due to low coupling between components, and the only form of communication being indirect</li> <li>New components can be added to the bus at will, provided they adhere to the event format on said bus</li> </ul> <p>What doesn't change?</p> <ul style="list-style-type: none"> <li>Harder to change the way that data is represented in an event itself, since data is possibly used by large number of heterogeneous components which rely on a particular event structure</li> </ul>

## Questions

1. **You are designing a web framework that allows for pre-packaged components for common features provided by web applications, such as session management, CSRF checking, authentication, etc., as well as allowing developers to write their own components. You want to allow these components to be used together to build web applications quickly, while providing the flexibility for allowing developers to add their own components when necessary.**
  - a. **What architectural style would you use to build this web framework?**

Using *Pipes and Filters* would be a suitable approach for this kind of framework.
  - b. **What advantages does this architectural style give you with regards to fitness for future?**

Each component is independent of other components; the only commonality between all of them is the data they operate on (i.e. the request, along with any database resources in response to that request). This allows components to be “dropped in” as needed in the future.
  - c. **What design pattern would go well with this architectural style in implementing the framework? How does it contribute to the framework’s fitness for future?**

Something similar to the *Command* pattern would suit this design well by encapsulating each request as an object describing the request. Each component would read this request object, perform the appropriate operations in response to it, possibly even modifying the request itself, and then pass it off to the next component.
  - d. **Give an example of a currently existing framework that implements this paradigm.**

The Django framework is a Python web framework that utilizes the Web Server Gateway Interface (WSGI), which defines a universal interface between web servers/applications. In this paradigm, a WSGI application is an application that takes a request (also known as an environment) and performs operations on that environment before returning a response or passing it off to the next WSGI application (typically only the last WSGI application returns the response, but preceding applications that received the request first can “short circuit” the chain in exceptional circumstances).

This design is admittedly not a *perfect* example of pipes and filters, but it is easy to see where the advantages of pipes and filters were taken into consideration when designing the Django framework.
2. **You are designing a real-time notifications platform for the new social network Google+. One of the uses will be to support a feature where if one user comments on a photo while another is viewing, that comment appears “immediately” in the viewing user’s browser. However, you expect the new social network will acquire many features requiring this framework over the next year.**
  - a. **Which design pattern would you utilize when designing this platform at a high level?**

The *Observer* pattern works well in this situation, as we can think of any viewing users as consumers and the server as the producer (since it is the first entity to be made aware of the new comment).

- b. **What is a potential problem you might face if you were to open up this platform to third-party developers?**

A significant issue to consider with the *Observer* pattern is the possibility of changing the message format/interface used to notify consumers of events. If third-party developers start developing their applications to use V1 of the notifications platform, and for some reason you have to change the format, all of these applications will break. Thus you will need to version your platform so that existing applications don't break, while still providing a way for you to innovate your platform for use by new clients created by third-party developers.