# 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 | Restricts the instantiation of a class to one object.  What is it about?   * Single global point of access to one or more objects * Controlled instantiation of one or more objects |
| Structural | Adapter | Yes | Translates on interface for a class (adaptee) into a compatible interface (adaptor).  What changes?   * Switch different adapters in and out for different functionality   What doesn’t change?   * Adaptee class interface changes requires the adaptor’s interface to change as well |
| Bridge | Yes | Decouples an abstraction from an implementation so the two can vary independently.  What changes?   * Refined abstractions and concrete implementations   What doesn’t change?   * Abstraction and abstract implementation classes |
| Composite | No | Allow uniform treatment of individual objects and compositions of those objects.  What is it about?   * Used for iterating through tree structures composed of heterogeneous components * Useful for avoiding having to deal with the difference between branch and leaf nodes * Designed for fitness for purpose over future * Difficult to add new functionality to components since it must be added to each existing class * New classes can be easily added as long as they conform to current interface |
| Facade | Yes | Provide a simplified interface to a larger body of code.  What changes?   * Implementation of underlying code can change * Interaction of classes behind the facade can change   What doesn’t change?   * Interface of facade stays the same * Functionality usually can’t change |
| Behavioural | Command | Yes | Allows behaviour of certain operations to be encapsulated within reusable logic.  What changes?   * Implementation of commands can change, for example the actions performed   What doesn’t change?   * Interface stays the same, usually a single method called “execute()” |
| Interpreter | Yes | Specifies how to evaluate sentences in a language using a syntax tree.  What changes?   * Grammar easily extended, though difficult to maintain complex grammars due to large number of tightly-coupled classes   What doesn’t change?   * Existing grammar structure remains relatively the same |
| Observer | Yes | 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.  What changes?   * Producer can be changed without changing the consumers (so long as message format remains the same) * New consumers can be created without change to the producer   What doesn’t change?   * Format of message sent out to consumers cannot change without changing all consumers, as well as the interface producer uses to notify consumer |
| Strategy | Yes | Allows algorithm to be selected at runtime.  What changes?   * Can add any number of algorithms as classes   What doesn’t change?   * Interface of each “class” of algorithms stays the same (e.g. sorting routines all take and return a list) |
| Visitor | Yes | Adds the ability to add new operations to data structures without changing the structures themselves.  What changes?   * Can add as many new visitors as you like * Can change the operations performed the visitor   What doesn’t change?   * Interface of the underlying data structures |

## Summary of Architectural Styles

|  |  |  |
| --- | --- | --- |
| Architectural Style | Future? | Description |
| Pipes n’ Filters | Yes | Allows components to be arranged so that the output of each element is the input of the next.  What changes?   * Reuse/reorder pipes and filters in new arrangements * Add and remove pipes and filters to aid between converstion of input/output format or to add new functionality   What doesn’t change?   * Need to make sure the input stream is in the format expected * Individual pipes and filters typically do not change |
| Layered/Hierarchical | Yes | Abstracts different portions of functionality into separate layers that build on top of each other.  What changes?   * The implementation of a layer can change so long as the interface is held constant * When a layer changes, only the layers above and below it must change   What doesn’t change?   * Relative ordering of each layer not changed easily (e.g. OS memory management usually below networking layer) |
| Repository | No | Provides a centralized data store, with many components.  What is it about?   * Two types of components:   + Datastore – holds and represents the system state/data   + Data-use components – components that operate on the central datastore * Type main classifications of repository   + Database – streams of transactions trigger processes to act on datastore   + Blackboard – current state/state changes trigger processes |
| Implicit Invocation | Yes | Allows components to listen and react to events.  What changes?   * 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 * New components can be added to the bus at will, provided they adhere to the event format on said bus   What doesn’t change?   * 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 |

## 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.**
   1. **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.
   2. **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.
   3. **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 evening modifying the request itself, and then pass it off to the next component.
   4. **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.**
   1. **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).
   2. **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.