**Data Architecture**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Type** | | **Description** | | **Pros** | | **Cons** |
| Batch | Components independent programs  Connectors are media  Each step runs to completion | | Discrete Uses | | No Concurrency  Non-interactive | |
| Pipes and Filters | Processed Incrementally  Each step encapsulated in a filter component  Components are filters  Connectors are pipes | | Can connect filters  Easily maintained  Prototyping flexibility  Can be parallel | | Filters independent  Filters disregard upstream and downstream  Sharing data difficult  Difficult to design  Error handling (must do whole process) | |

**Design Patterns**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Description** | **When to use** | **Consequences** |
| Visitor *Behavioural* | Centralize operations so it can change independently, but still apply to different classes | Classes have unrelated operations  Operations change, but relationships don’t | +Flexible – visitor and object independent  +Localized functionality  -Dependent on Element interface  -Brittle |
| Iterator  *Behavioural* | Access elements sequentially without exposing representation | Multiple traversals over something  Requires uniform traversal interface several times  Aggregate classes and travel must be independent | +Flexible – aggregate and traversal independent  +Multiple iterators  -Additional comms between iterator and aggregate |
| Observer  *Behavioural* | Many objects depend on it such that when one object state changes, others are notified and updated | When objects are dependent  Unsure how many objects require changing after an update  When objects need to notify others on changes | +Modular – subject and observers can vary  +Extensible – can add/define many observers  +Customizable – different observers provide different views  -Unexpected updates – observers independent |
| State | Allows object to change behaviour according to its state | Object denotes different behaviours depend on state | +Object state logic concentrated in classes that denote the state ie. that the code is more logically divided |
| Strategy | Family of algorithms called through one interface | Need multiple algorithms & algorithms encapsulated & common interface can be used | +Flexible – Code reuse  +Dynamically change what is called  -Costly to associate algorithm with objects  -Must use common interface |
| Singleton | We can construct a limited number of objects from a class, and provides reference to a unique object | Want to guarantee a certain number of objects created | +Simplifies structure of program  +Allows creation of specific number of objects  +Allows Extension of object using specialization  -If not used properly, works like a global variable  -Implementation more costly than general variable, but more secure  -Parallel usage dangerous |
| Factory Method | We can construct classes without the client code knowing what is going on | General interface to create objects  Do not know in client code which objects need to be built |  |
| Adapter | Allows client code to access an object with different interfaces, and gives the correct interface | Want to replace a class with another one with a different interface, without erasing clients using the first one  Implement a class for others which are not complete (prototyping) | Allows client code to remain same, even though classes used are the same  Useful for large project in which there are different changing interfaces |
| Bridge | Decouples an abstraction from implementation, so the two can vary independently. Can be used to implement layered architectures, and used in conjunction with Factory |  |  |
| Composite | Manipulate a single instance of an object just as you would a group of them  Recursive usage | Fuck it’s in greek | -Must have primitive and composite objects. Processing a primitive object different from composite |
| Proxy | Allows creation of a substitute object that holds a reference to the object. Sub controls access to object | Useful when function takes a long time to execute, as sub can send messages about progress of function |  |
| Template Method | Define a general structure of algorithm, with ability to change certain steps | When basic steps can be implemented in a class, and subclasses implement variations  Bring together basic steps to promote reusability | This bitch esl |
| Chain of Responsibility | Promote low coupling between sender and objects that handle request | Objects that handle a request are chained, so that it passes from one to another as needed  Request handled by multiple objects |  |
| Decorator | Add new features to a single object. AKA Wrapper.  We can augment specific objects of a class, instead of whole class | When not practical to extend functions using inheritance  Dynamically expand functionality delivered by ONE object, and remove them if need be | +Flexible  +Can wrap decorators with others  +Avoid multiple subclasses  +Recursive use and encapsulation  -Restrictions on definition of interfaces  -Identity crisis of object |

**Architectures**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Description** | **Details** | **Consequences** |
| Main program + subroutine | Used from 60s to pre OO  Hierarchical decomposition  Data passed as parameters | Old af don’t use it |  |
| Data Abstraction/OO | Widely used, as it doesn’t grant direct access to objects  DA doesn’t have inheritance | Components are objects or ADTs  Connections – Messages or functions | +Change implementation without affecting clients  +Can break problems into interacting agents  -objects must know each other  -distributed functionality  -side effects |
| Implicit Invocation | Communication between components is key. Use a generalized Observer to communicate | Components and connectors not necessarily objects  Components have a procedure associated with event  Commonly used to integrate tools in a shared environment | +Reusability by simply plugging in new components  +Easily maintained by adding/replacing components  -Loss of control, as we don’t know who will react to what  -Can’t tell when components are finished  - |
| MVC | Composed of 3 components:   1. Core functions and data 2. User display 3. User input   Improved maintainability | Under implicit invocation style  Model components denote domain application data  Viewer: implement different ways data is presented  Controller: implement software logic that alter data |  |
| Broker | Structured systems with decouple components | Broker component coordinates communication between components  Server communicates capabilities to broker, and clients request services from a broker |  |
| BlackBoard | Useful for problems with no solution strategies known. All components can access the blackboard, and solution stored on blackboard | 3 components  Blackboard – global memory containing objects from solution space  Knowledge source – specialized modules  Control – selects and executes modules |  |
| Layered | Used to structure programs that can be made into subtasks. Like an onion. Each layer connected to adjacent | Mix of high and low level issues  Components are like virtual machines  Connectors are interactions between layers | +Increasing levels of abstraction  +Change has minimal effect, so easy maintenance  +Different implementations of a layer can be interchanged  -Not all systems can be structured in such a way  -Efficiency – traverse multiple layers |
| Tiered | 2, 3 or multi tiered | Middle tier can execute logic and rules  Increases performance, flexibility, maintainability etc etc. |  |
| Interpreter | Interprets programs written in a dedicated language, and specifies how to evaluate lines of programs | Used in database languages like SQL  Languages are communications  Components are program being executed |  |
| Heterogenous | Used in practice | Hierarchical – component in one style can be completely diff in another style  Localized – Overall archi at same level is of different styles  Perspective – Different archi in different perspectives |  |

**Software Life Cycle**

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Consequences** |
| Waterfall | Sequential steps  Documentation driven  Follows a strict order | +Maintenance easier  -Frozen specifications up-front not feasible in practice  -Limited customer involvement  -Difficult to control  -Complete and sequential execution of phases  -Product developed very late |
| Incremental | Each release adds more functionality |  |
| Evolutionary | New versions implement new and changing requirements |  |
| Spiral | Much more complex  Useful for new projects  Heavy time usage in determine objectives  4 phases:   1. Determine requirements 2. Evaluate Risks 3. Development 4. Plan next stage |  |