Problems Of Design Patterns

Some patterns to showing problems for us. For make to decision; We can see somethings in here.

Design Patterns

A. Creational patterns

In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

1. Abstract Factory Design Pattern — Problem

If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulation is not engineered in advance, and lots of `#ifdef` case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

2. Builder Design Pattern — Problem

An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

3. Factory Method Design Pattern — Problem

A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

4. Object Pool Design Pattern — Problem

Object pools (otherwise known as resource pools) are used to manage the object caching. A client with access to a Object pool can avoid creating a new Objects by simply asking the pool for one that has already been instantiated instead. Generally the pool will be a growing pool, i.e. the pool itself will create new objects if the pool is empty, or we can have a pool, which restricts the number of objects created.

It is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the Reusable Pool class is designed to be a singleton class.

5. Prototype Design Pattern — Problem

Application “hard wires” the class of object to create in each “new” expression.

6. Singleton Design Pattern — Problem

Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

B. Structural patterns

In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.

1. Adapter Design Pattern

An “off the shelf” component offers compelling functionality that you would like to reuse, but its “view of the world” is not compatible with the philosophy and architecture of the system currently being developed.

2. Bridge Design Pattern

“Hardening of the software arteries” has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be indepenently extended or composed.

3. Composite Design Pattern

Application needs to manipulate a hierarchical collection of “primitive” and “composite” objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the “type” of each object before attempting to process it is not desirable.

4. Decorator Design Pattern

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

5. Facade Design Pattern

A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

6. Flyweight Design Pattern

Designing objects down to the lowest levels of system “granularity” provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

7. Private Class Data

A class may expose its attributes (class variables) to manipulation when manipulation is no longer desirable, e.g. after construction. Using the private class data design pattern prevents that undesirable manipulation. A class may have one-time mutable attributes that cannot be declared final.

Using this design pattern allows one-time setting of those class attributes. The motivation for this design pattern comes from the design goal of protecting class state by minimizing the visibility of its attributes (data).

8. Proxy Design Pattern

You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

C. Behavioral patterns

In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

1. Chain of Responsibility

There is a potentially variable number of “handler” or “processing element” or “node” objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

2. Command Design Pattern

Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

3. Interpreter Design Pattern

A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a “language”, then problems could be easily solved with an interpretation “engine”.

4. Iterator Design Pattern

Need to “abstract” the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

5. Mediator Design Pattern

We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the “spaghetti code” phenomenon (trying to scoop a single serving results in an “all or nothing clump”).

6. Memento Design Pattern

Need to restore an object back to its previous state (e.g. “undo” or “rollback” operations).

7. Null Object Design Pattern

Given that an object reference may be optionally null, and that the result of a null check is to do nothing or use some default value, how can the absence of an object — the presence of a null reference — be treated transparently?

8. Observer Design Pattern

A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

9. State Design Pattern

A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterized by large and numerous case statements that vector flow of control based on the state of the application.

10. Strategy Design Pattern

One of the dominant strategies of object-oriented design is the "open-closed principle". How this is routinely achieved — encapsulate interface details in a base class, and bury implementation details in derived classes. Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.

11. Template Method Design Pattern

Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

12. Visitor Design Pattern

Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid “polluting” the node classes with these operations. And, you don’t want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.