DevNet Associate v1.0

Software Design Patterns

3.2.1

Introduction

Software design patterns are best practice solutions for solving common problems in software development. Design patterns are language-independent. This means that they can be implemented in any contemporary, general-purpose computing language, or in any language that supports object-oriented programming. Often, popular design patterns encourage creation of add-on frameworks that simplify implementation in widely-used languages and paradigms.

Artisans have always shared time-tested methods and techniques for problem-solving. Calling these things "design patterns" was first done systematically in the field of architecture and urban planning. Architectural patterns were organized by abstract class of problem solved, urging designers to recognize common core themes shared across numerous divergent contexts. For example, a bus stop and a hospital waiting room, are both places in which people wait; so both can usefully implement features of the pattern A PLACE TO WAIT.

This way of thinking about patterns was quickly picked up by pioneers in object-oriented coding and Agile software development. In 1994, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (known collectively as the Gang of Four (GoF)) published a book called Design Patterns - Elements of Reusable Object-Oriented Software. We'll offer a broad view of the patterns they identified and documented.

Program to an interface, not an implementation. Tightly coupling mainline program logic with implementations of specific functionality tends to make code hard to understand and maintain. Experience has shown that it works better to loosely-couple logical layers by using abstract interfaces. For instance, the mainline code calls functions and methods in a generic way. Lower-level functions implement matching interfaces for the functionality they provide, ensuring, for example, that all serialization functions used in a program are called in similar fashion.

Object-oriented languages like Java formalize these ideas. They enable explicit declaration of interfaces that classes can implement. An interface definition is basically a collection of function prototypes, defining names and types for functions and parameters that higher-level logic might use to invoke a range of classes. For example, the interface to a range of 'vehicle' classes (e.g., class 'car,' 'motorcycle,' 'tractor') might include start_engine(), stop_engine(), accelerate(), and brake() prototypes.

Favor object composition over class inheritance. Object-oriented languages enable inheritance: more generalized base classes can be inherited by derived classes. Thus a 'duck' class might inherit substantial functionality from a 'bird' class. This requires, however, that the bird class implement a very wide range of methods, most of which may not be used by a specific derived class.

The principle of favoring composition over inheritance suggests that a better idea may be to favor implementing a specific class (class - duck) by creating only required unique subclasses (class - quack) along with abstract interfaces (interface - 'ducklike') to classes (class - fly, class - swim) that can be shared widely in similar fashion (class - penguin implements interface 'penguinlike,' enabling sharing of class 'swim,' but not 'fly'). Organizing software in this way has proven to be most flexible, ultimately easier to maintain, and encourages reuse of code.

Software design patterns have already been proven to be successful, so using them can speed up development because developers don't need to come up with new solutions and go through a proof of concept to make sure they work.

3.2.2

The Original Design Patterns

In their Design Patterns book, the Gang of Four divided patterns into three main categories:

- **Creational** - Patterns used to guide, simplify, and abstract software object creation at scale.
- **Structural** - Patterns describing reliable ways of using objects and classes for different kinds of software projects.
- **Behavioral** - Patterns detailing how objects can communicate and work together to meet familiar challenges in software engineering.

They listed a total of 23 design patterns, which are now considered the foundation of newer design patterns. Most of these patterns, at some level, express basic principles of good object-oriented software design.

Let's dive deeper into two of the most commonly used design patterns: the Observer design pattern (a Behavioral design pattern), and Model-View-Controller (MVC).

3.2.3

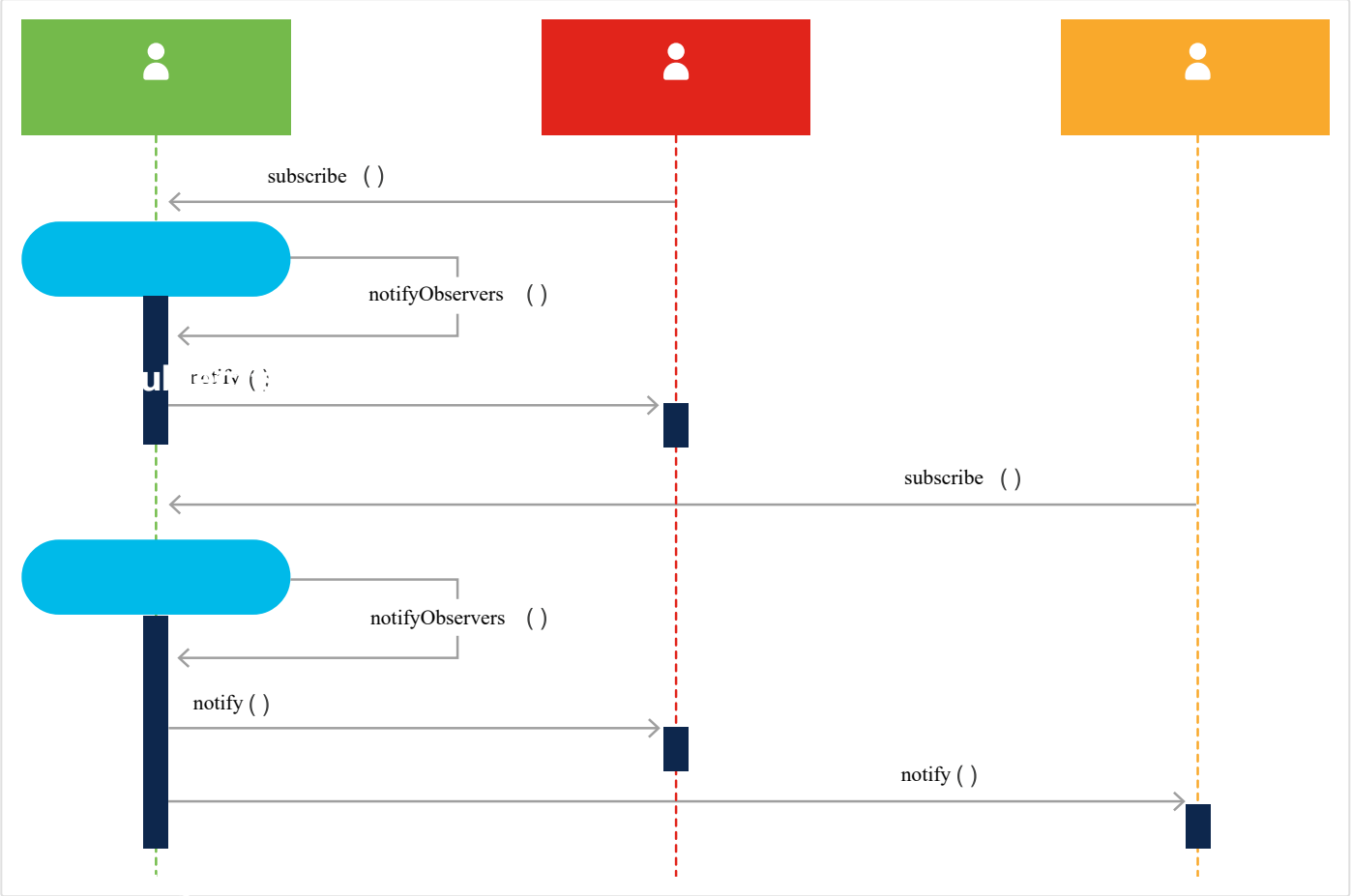
Observer Design Pattern

The observer design pattern is a subscription notification design that lets objects (observers or

subscribers) receive events when there are changes to an object (subject or publisher) they are observing. Examples of the observer design pattern abound in today's applications. Think about social media. Users (observers) follow other users (subjects). When the subject posts something on their social media, it notifies all of the observers that there is a post, and the observers look at that update.

To implement this subscription mechanism:

1. The subject must have the ability to store a list of all of its observers.
2. The subject must have methods to add and remove observers.
3. All observers must implement a callback to invoke when the publisher sends a notification, preferably using a standard interface to simplify matters for the publisher. This interface needs to declare the notification mechanism, and it needs to have parameters for the publisher to send the necessary data to the observer.



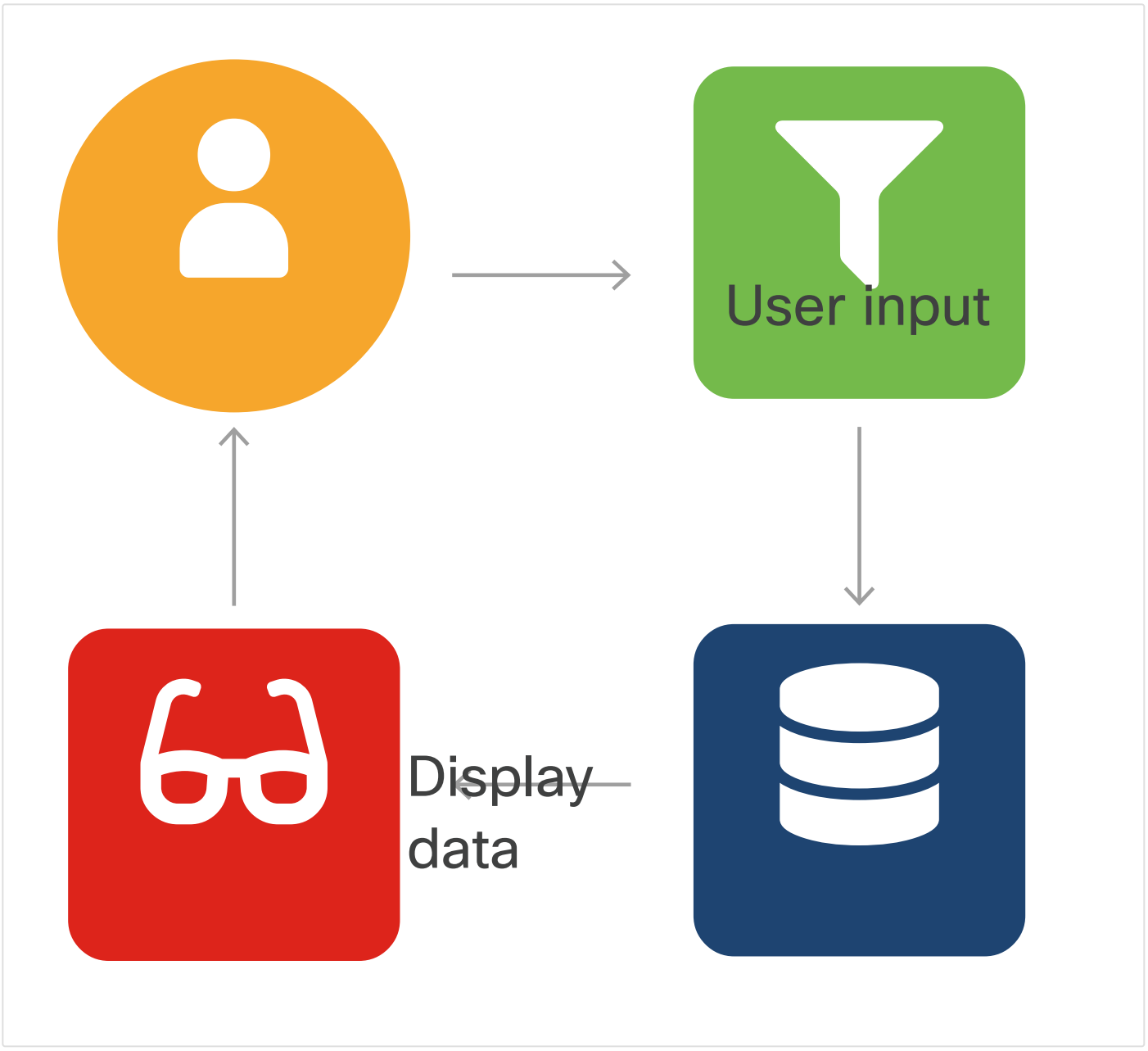
The execution of this design pattern looks like this:

1. An observer adds itself to the subject's list of observers by invoking the subject's method to add an observer.
2. When there is a change to the subject, the subject notifies all of the observers on the list by invoking each observer's callback and passing in the necessary data.
3. The observer's callback is triggered, and therefore executed, to process the notification.
4. Steps 2 and 3 continue whenever there is a change to the subject.
5. When the observer is done receiving notifications, it removes itself from the subject's list of observers by invoking the subject's method to remove an observer.

The benefit of the observer design pattern is that observers can get real time data from the subject when a change occurs. Subscription mechanisms always provide better performance than other options, such as polling.

3.2.4

Model-View-Controller (MVC)



The Model-View-Controller (MVC) design pattern is sometimes considered an architectural design pattern. Its goal is to simplify development of applications that depend on graphic user interfaces. MVC abstracts code and responsibility into three different components: model, view, and controller. Each component

communicates with each other in one direction. This design pattern is commonly used in user interfaces and web application.

Components

- **Model -** The model is the application's data structure and is responsible for managing the data, logic and rules of the application. It gets input from the controller.
- **View -** The view is the visual representation of the data. There can be multiple representations of the same data.
- **Controller -** The controller is like the middleman between the model and view. It takes in user input and manipulates it to fit the format for the model or view.

The execution of the Model-View-Controller looks like this:

1. The user provides input.
2. The controller accepts the input and manipulates the data.
3. The controller sends the manipulated data to the model.
4. The model accepts the manipulated data, processes it, and sends the selected data (in the strictest forms of MVC, via the controller) to the view.
5. The view accepts the selected data and displays it to the user.
6. The user sees the updated data as a result of their input.

The benefit of the Model-View-Controller design pattern is that each component can be built in parallel. Because each component is abstracted, the only information each component needs is the input and output interface for the other two components. Components don't need to know about the implementation within the other components. What's more, because each component is only dependent on the input it receives, components can be reused as long as the other components provide the data according to the correct interface.

3.1
Software Development

3.3
Version Control Systems