Design Patterns

As software engineers, we commonly have to make decisions about how to create complex objects, how to encapsulate some actions, how to allow for the undoing of certain operations, etc. Instead of developing individual solutions every time, it is useful to apply standard design patterns. The patterns have been used and tested by many other software engineers. Moreover, they are known by many software engineers and well documented in literature so that using them will make it easier for other programmers to understand your code.

The Singleton Design Pattern

The singleton design pattern is a simple creational pattern, i.e., it applies to the creation of objects. This pattern can be used when at most one instance of a particular object is allowed to exist at any given time. For example, there should only be one mouse cursor object. The singleton design pattern prevents direct access to the object constructors but provides an instance() method for calling member operations. When called for the first time, the instance() method will create the only instance of the object.

UML Basics

UML is a standard modeling language, particularly for object-oriented modeling. For system development, we use UML to represent three different models of the system:
- the functional model, represented with use case diagrams (system from the user’s perspective),
- the object model, represented with class diagrams (structure of the system),
- the dynamic model, represented with sequence diagrams, statechart diagrams, and activity diagrams (internal behavior of the system).
Use Case Diagrams
Use case diagrams represent the functionality of the system. A use case is a function of the system that yields a visible result for an actor. An actor is any person or object outside the system that interacts with the system. In use case diagrams, the actors are outside the boundary of the system, and the use cases are inside. We indicate use cases with ovals (ellipses).

Example: Use case diagram for a simple watch.

Use Case Diagrams

Actors
An actor models an external entity which communicates with the system:
• User
• External system
• Physical environment
An actor has a unique name and an optional description.
Examples:
• Passenger: A person in the train
• GPS satellite: Provides the system with GPS coordinates

Use Cases
A use case represents a class of functionality provided by the system as an event flow. A use case consists of:
• Unique name
• Participating actors
• Entry conditions
• Flow of events
• Exit conditions
• Special requirements

Use Cases: Example
Name: Purchase ticket
Participating actor: Passenger
Entry conditions:
Passenger standing in front of ticket distributor.
Passenger has sufficient money to purchase ticket.

The <<extend>> Relationship
<<extend>> relationships represent exceptional or rarely invoked cases. The exceptional event flows are factored out of the main event flow for clarity. Use cases representing exceptional flows can extend more than one use case. The direction of a <<extend>> relationship is towards the extended use case.
The <<extend>> Relationship

An <<extend>> relationship represents behavior that is factored out of the use case.
An <<extend>> represents behavior that is factored out for reuse, not because it is an exception.
An <<extend>> relationship arrow starts from the including use case (unlike <<extend>> relationships).

The <<include>> Relationship

An <<include>> relationship represents behavior that is factored out of the use case.
An <<include>> represents behavior that is factored out for reuse, not because it is an exception.
An <<include>> relationship arrow starts from the including use case (unlike <<extend>> relationships).

Data Types and Instances

In programming languages, data types are used as concepts:
• A data types has a name for identification,
• a data type has a purpose (structure and operations on its members),
• a data type has members.

An abstract data type is a data type whose structure and implementation is hidden from the rest of the system.
An instance is any member of a specific data type.

Classes

In object-oriented programming languages, classes are used as concepts (abstractions).
Classes are like abstract data types in terms of encapsulating structure and behavior.
But unlike abstract data types, classes can be derived from other classes by using generalization.
Imagine that you developed a watch that can also be used to surf the web.
Then the class WebWatch would be a refinement of the class Watch.
In UML class diagrams, the triangle points to the superclass, and the other end is attached to the subclass. The second and third compartments can be omitted.

Classes

In UML, names of instances are underlined.

Instances

Associations

An instance represents a phenomenon. The name of an instance is underlined and can contain the class of the instance. The attributes are represented with their values.

Class diagrams describe the system in terms of objects, classes, attributes, operations, and their associations.

A class represents a concept. A class encapsulates states (attributes) and behavior (operations). Each attribute has a signature (the tuple made out of the types of its parameters and the type of the return value). The class name is the only mandatory information.

Associations denote relationships between classes. The multiplicity of an association end denotes how many objects the source object can legitimately reference.
**Class Diagrams**

**Example:** Class diagram for a simple watch.

```
SimpleWatch
PushButton 1
Display 1
Battery 2
Time 1
```

The numbers on the ends of associations indicate the number of links that each object has with an object of a given class.

**Aggregation**

An aggregation is a special case of association denoting a "consists of" hierarchy. The aggregate is the parent class, the components are the children class.

```
Exhaust System
1
Muffler
Tailpipe
```

**Generalization**

Generalization relationships denote inheritance between classes. The child classes inherit the attributes and operations of the parent class. Generalization simplifies the model by eliminating redundancy.

**Sequence Diagrams**

Sequence diagrams show the communication among objects. The objects involved in a use case are called participating objects.

A sequence diagram represents the interactions that take place among these objects. In a sequence diagram, the columns represent timelines of actors and objects, labeled arrows represent stimuli (messages) that are sent from an actor to an object or between objects.
Sequence Diagrams

Used during requirements analysis
- to refine use case descriptions,
- to find additional objects ("participating objects")

Used during system design to refine subsystem interfaces.

Classes are represented by columns
Messages are represented by arrows
Activations are represented by narrow rectangles
Lifelines are represented by dashed lines

Sequence Diagram Observations

- UML sequence diagrams represent behavior in terms of interactions.
- Complement the class diagrams which represent structure.
- Useful to find participating objects.
- Time consuming to build but worth the investment.

Statechart Diagrams

Statechart diagrams describe the behavior of an individual object like a finite state automaton, that is,
- a finite number of states that the object can assume,
- transitions between these states.

Here, a state is a particular set of values for an object. States are represented by ovals.
A transition represents a state change that occurs under a particular condition.
Transitions are represented by arrows that are labeled with the corresponding condition.

Statechart Diagrams

Example: Statechart diagram for SimpleWatch.

Activity Diagrams

An activity diagram describes a system in terms of activities.
Activities are shown as states that represent the execution of a set of operations.
After the completion of one or more activities, other activities are started.
Activity diagrams are similar to flowchart diagrams. They represent activities as ovals and transitions between activities as arrows.

Activity Diagrams

Example: Activity diagram for incident management

Thick bars represent the synchronization of the control flow.