Tasarım Şablonları 4. Hafta

17.03.2025

SOLID Principles

GRASP Principles

SOLID

- SOLID is the acronym for a set of practices
  - When a set of practices are implemented together, it is important to make code adaptive to change.

S : single responsibility principle

O : open/closed principle

L : Liskov substitution principle

I: Interface segregation

D: Dependency inversion

Single Responsibility Principle (SRP)

```
public class GameResult {    // GameResult has a single responsibility clearly and concisely
  int guesses;
  int magicNumber;
  GameResult( int numberOfGuesses, int theNumber) {
    guesses = numberOfGuesses;
    magicNumber = theNumber;
  }              }
```

- A simple example of a Java class.
- The class correctly observes the single responsibility principle.
- For a number guessing game, each time it is played the game tracks two things:
  the magic number & the number of attempts to guess it.

SRP Problem with GameResult

- The developer team wants to add the results of all games played (to keep track of games) .
- They add a List to the GameResult class and add to it with every creation of a new GameResult object.

```
public class GameResult {
static List<GameResult> history = new ArrayList();
 int guesses;
 int magicNumber;
 GameResult( int numberOfGuesses, int theNumber) {
  guesses = numberOfGuesses;
  magicNumber = theNumber;
  history.add(this);
 }     }
```

- The code does not henceforth hold the SRP
- The GameResult component is now also tracking history.
  - But, the GameResult component should do nothing more than keep track of a single game

Refactoring with SOLID

```
public class GameResult {
  static List<GameResult> history = new ArrayList();
  int guesses;
  int magicNumber;
  GameResult( int numberOfGuesses, int theNumber) {
    guesses = numberOfGuesses;
    magicNumber = theNumber;
    history.add(this);        }
  public static void printHistory()   { …… }
  public static void updateRow()    { …… }
  public static void deleteRow()    {……..}
  public static void editRow()       {……..}
                                    }
```

- They number-guessing-game code can be manageable.
- It is required several methods such as: Print the history, Edit the history, Delete elements from the history.
- The initial aim of the GameResult class is changed
- The added methods don't serve the class's primary purpose

Challenges with Fine-Grained Components with SRP

- As the number of components gets too large, it becomes important to increase overall complexity.
- As the coupling between classes that call each other increased,
  - Coupling makes modifications harder.
- As the cohesion within the code decreased,
  - Cohesion makes harder to understand how the integrated system works.
- As an overengineered system created,
  - It becomes more difficult to maintain.

GRASP (General Responsibility Assignment Software Principles) firstly defined by Craig Larman

High Cohesion (GRASP)

Low Coupling (GRASP)

Polymorphism (GRASP)

Information Expert (GRASP)

Creator (GRASP)

Pure Fabrication (GRASP)

Controller (GRASP)

Indirection (GRASP)

Other Basic Object-Oriented Design Principles

Program to an Interface, not to an Implementation

Hollywood Principle (“Don’t Call Us, We’ll Call You” principle)

Favor composition over inheritance

Hollywood Principle

- Define a clear separation of responsibilities between high-level and low-level components in a system.
- Low-Level Components Are Passive: Low-level components, often referred to as "observers," should not actively call or depend on high-level components, known as "subjects."
  - They remain passive and wait for the high-level components to notify them when an action is required.
- High-Level Components Control the Flow: High-level components have the authority to decide when and how low-level components should act.
  - They orchestrate the system's behavior, ensuring that low-level components respond appropriately to changes or events.
- Observer Design Pattern is the fundamental example of Hollywood principle
- Observer Design Pattern is a Behavioral and Object Scope Design Pattern in GOF classification
- This processing results as Event-Driven Architecture

```
class Subject {      // High-level component (subject)
    private List<Observer> observers = new ArrayList<>();
    public void registerObserver(Observer observer) {
        observers.add(observer);                                    }
    public void unregisterObserver(Observer observer) {
        observers.remove(observer);                                  }
    public void doSomethingImportant() {   // Some important logic ……
        System.out.println("Subject is doing something important...");
            notifyObservers(); // Notify observers when needed  }
   private void notifyObservers() {
            for (Observer observer : observers) {
                observer.update();      }     }
                            }
```

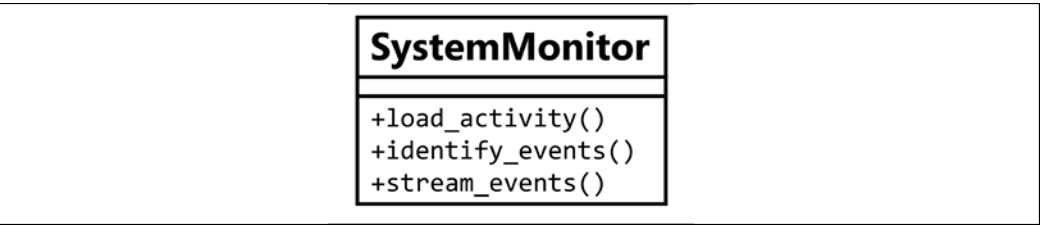
This behaviour pattern developes a **decoupled and flexible** system.

```
class Observer {      // Low-level component (observer)
   private String name;
   public Observer(String name) {
      this.name = name;                }
   public void update() {
      System.out.println(name + " received notification and is acting upon it.");      }          }
   public class ObserverPatternExample {
   public static void main(String[] args) {
       Subject subject = new Subject();
       Observer observer1 = new Observer("Observer 1");
       Observer observer2 = new Observer("Observer 2");
       subject.registerObserver(observer1);
      subject.registerObserver(observer2);
       subject.doSomethingImportant();  // High-level component (subject) decides when to act
                                    }  }
```

- A subject (high-level component) maintains a list of observers (low-level components) notifies them when a change in state occurs.
- Observers remain passive, waiting for notifications and responding accordingly.

This behaviour pattern developes a **decoupled and flexible** system.

An Example with Python violating SRP

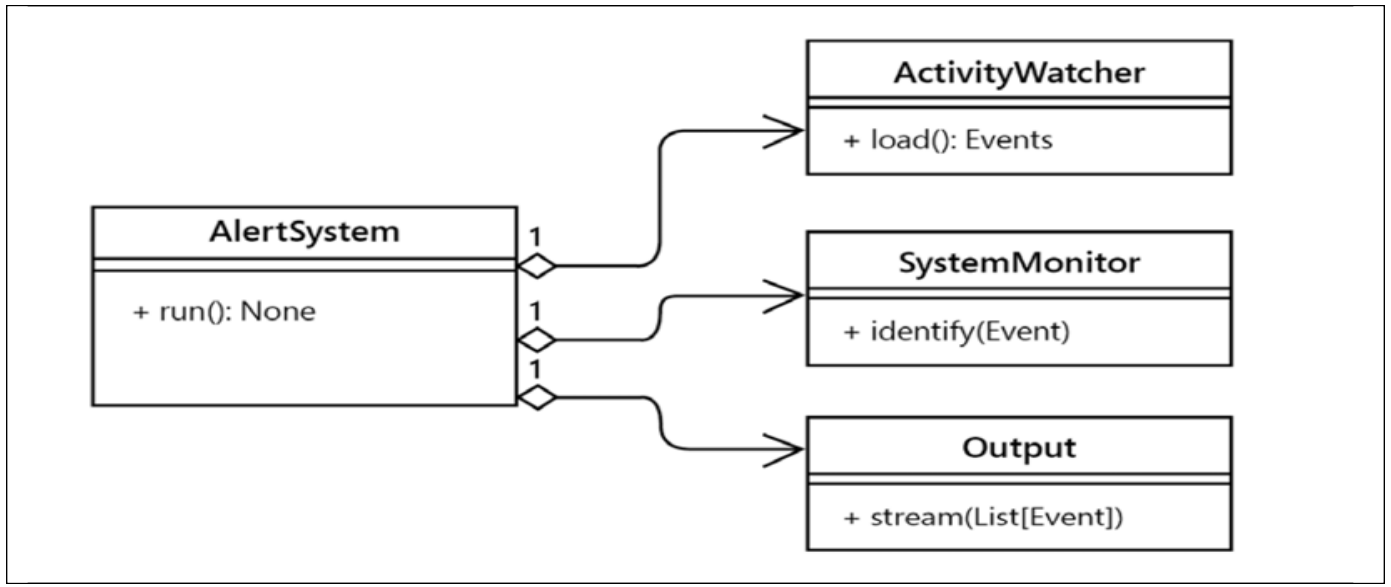


Çok fazla fonksiyonu olduğu için SRP ilkelerini gerçekleştirmeyen bir sınıf

- Bu tasarım hatası (flaw) SystemMonitor sınıfını «rigid», «inflexible» ve «error-prone» yapar. «Maintainability» güçleşir.
- Dış faktörler kodu fazlasıyla etkiler.
  - Çözüm ise daha küçük ve uyumlu soyutlamalar (cohesive abstractions) yaratmaktır.

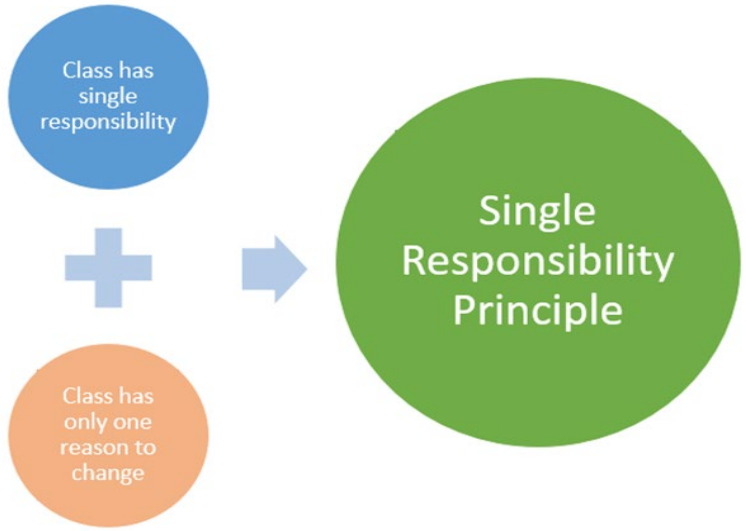
```
# srp_1.py
class  SystemMonitor:
  def load_activity(self):
          """Get the events from a source, to be processed."""
   def identify_events(self):
          """Parse the source raw data into events (domain objects)."""
  def stream_events(self):
          """Send the parsed events to an external agent."""
```

SRP Çözümü: Sorumlulukları Sınıflara Ayırmak



Sorumlulukları sınıflara dağıtarak doğru mu yapılmıştır?

- Bir sınıfın değişimi için birden fazla neden varsa, bu sınıfın birden fazla sorumluluğu vardır.
- Tek bir sorumluluğu olmayan sınıflar daha küçük sınıflara parçalanmalıdır.




```
public class TicketReservation
  {
 public void createTicketReservation
       (ReservationDetails   reservation)
{     new
OnlineReservationDAO.createReservation
                                (reservation);
       }              }
```

Open-Closed Principle

```
class Square() {
  int height;
  int area() { return height * height; }
                        }
public class OpenOpenExample {
  public int compareArea(Square a, Square b) {
    return a.area() - b.area();       }
                        }
```

- This program has two instances of a square and needs a custom component to compare their area.
- The OpenOpenExample isn't necessarily obvious.
- The OpenOpenExample works perfectly well.
  - It returns zero or a positive or a negative number
- But a problem arises when a circle is brought into the mix.

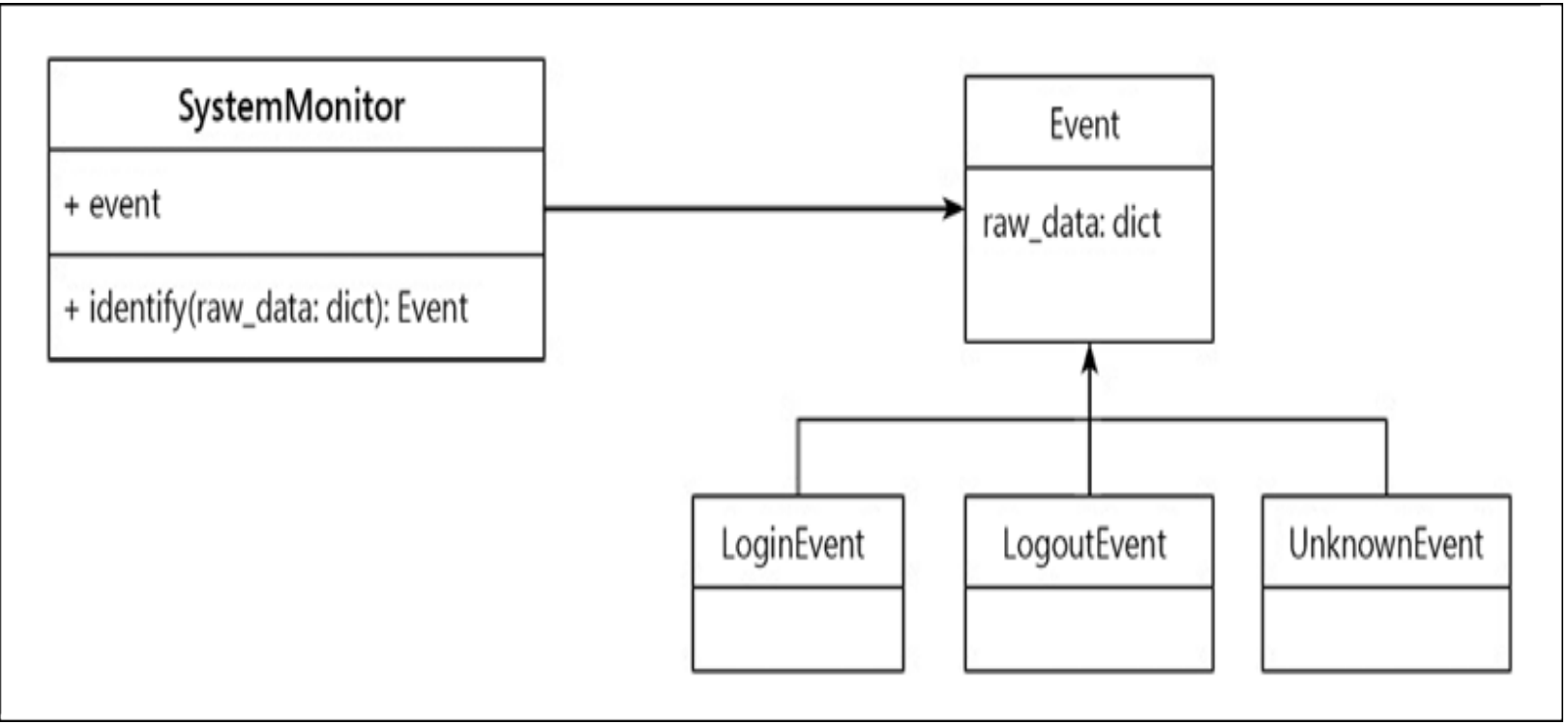
The extension problem of the OpenOpenExample class

```
class Circle {
  int r;
  int area() { return Math.PI*r*r*;}
}

class OpenOpenExample {
  public int compareArea(Square a, Square b) {
    return a.area() - b.area();
                                          }
  public int compareArea(Circle x, Circle y) {
   return x.area() - y.area();
  }      }
```

Applying the open-closed principle to OpenOpenExample class

```
interface Shape {
  int area();        }
class Circle implements Shape {
  int r;
  int area() { return Math.PI*r*r*;  }    }
class Square() implements Shape {
  int height;
  int area()      { return height * height; }    }


public class OpenClosedExample { //  New class   for OpenClosedExample is created.    A single
compareArea method uses the Shape interface as arguments
      public int compareArea    (Shape a, Shape b) {
        return a.area() - b.area();         }              }
```

OCP :Open/Closed Principle



Değişikliğe açık olmayan bir sistem

İlk bakışta bu genişletilebilir bir tasarım gibi görünebilir
Yeni bir olay (event) eklemek, Event sınıfının yeni bir alt sınıfı oluşturmaktır; ardından sistem monitörü bunlarla çalışabilmelidir.

Fakat tümü SystemMonitor sınıfındaki metot içindeki gerçek uygulamaya bağlı olduğundan, bu tam olarak doğru olmayacaktır.

```
from dataclasses  import  dataclass
@dataclass
class Event:   raw_data: dict
class UnknownEvent(Event):       """A type of event that cannot be identified from its data.""
class  LoginEvent(Event):              """An event representing a user that has just entered the system."""
class LogoutEvent(Event):           """An event representing a user that has just left the system."""
class SystemMonitor:                    """ Identify events that occurred in the system
            >>> l1 = SystemMonitor({"before": {"session": 0}, "after": {"session": 1}})
             >>> l1.identify_event().__class__.__name__    'LoginEvent'
            >>> l2 = SystemMonitor({"before": {"session": 1}, "after": {"session": 0}})
             >>> l2.identify_event().__class__.__name__    'LogoutEvent'
             >>> l3 = SystemMonitor({"before": {"session": 1}, "after": {"session": 1}})
             >>> l3.identify_event().__class__.__name__    'UnknownEvent'                """
  def __init__(self, event_data):
    self.event_data = event_data
  def identify_event(self):
    if (
      self.event_data["before"]["session"] == 0
     and self.event_data["after"]["session"] == 1
    ):
     return LoginEvent(self.event_data)
   elif (
     self.event_data["before"]["session"] == 1
     and self.event_data["after"]["session"] == 0       ):
    return LogoutEvent(self.event_data)
    return UnknownEvent(self.event_data)
```

- Örneğin, bir oturum için önceden flag yoksa ancak şimdi varsa , bu kayıt bir oturum açma olayıdır.
- Tersi olduğunda, bu bir çıkış olayıdır.
- Bir olayın tanımlanması mümkün değilse, bilinmeyen tipte bir olay döndürür.
- Burada «null nesne şablonu» ile polimorfizmi korunur.
- Böylece None değer olarak döndürmek yerine, «null nesne şablonu» kullanılmıştır.

Sorunlu tasarımdır.

A. Olay (event) türlerini belirleme mantığının monolitik bir yöntem içinde merkezileştirilmiştir.

Olayların sayısı arttıkça, bu yöntem de artacak ve çok uzun bir yöntem haline gelecektir.

Amacının dışına çıkacaktır, çünkü sadece tek bir şeyi yapmayacaktır.

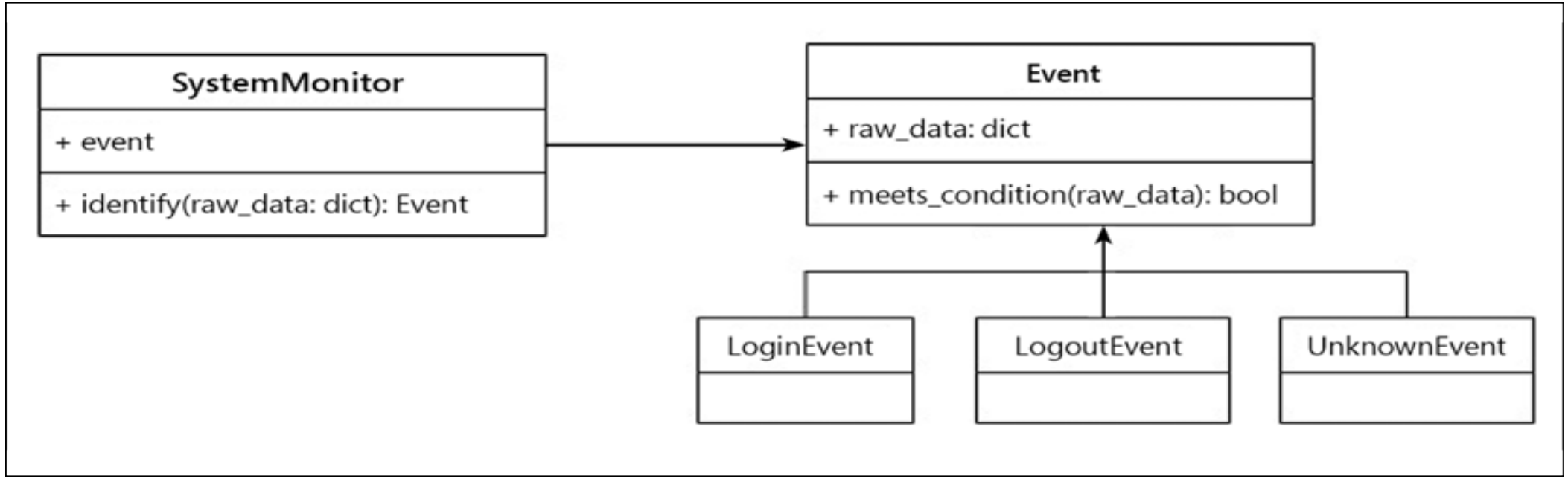
B. Yöntemin **modifikasyona (değişime) kapalı değildir.**

Sisteme her yeni tip olay eklendiğinde, identify-event yönteminde bir şeylerin değişmesi gerekecektir.

Refactoring the Events System for Extensibility

- Bu örneğin sorunu, SystemMonitor sınıfının alacağı somut sınıflarla doğrudan etkileşime girmesidir.
- Open/ Closed ilkesinin uygulandığı bir tasarıma ulaşmak için soyutlamalara yönelik tasarım yapılması gerekir.
  - SystemMonitor sınıfı olaylarla işbirliğindedir (collaboration) ve her bir olay türü için mantıksal olarak karşılık gelen sınıfa yönlendirecektir.
- **Bu süreç şöyle de gerçekleştirilebilir:**
  - Her olay türüne yeni bir (polimorfik) yöntem eklenir.
    - ✓ Bu yöntem, iletilen veriye karşılık gelip gelinmediğini belirler.
  - Böylece tüm olayların (events) kontrolü (if komutu ile) ve doğru olanın bulunması mantığı değiştirilmiş olur.

OCP :Open/Closed Principle



OCP kurallarına uygun sistem

```
from dataclasses import dataclass
@dataclass     class Event:    raw_data: dict
@staticmethod     def meets_condition(event_data: dict) -> bool:     return False
class UnknownEvent(Event):      """A type of event that cannot be identified from its data"""
class LoginEvent(Event):
  @staticmethod         def meets_condition(event_data: dict):
     return (        event_data["before"]["session"] == 0      and event_data["after"]["session"] == 1       )
class LogoutEvent(Event):
   @staticmethod     def meets_condition(event_data: dict):
     return (       event_data["before"]["session"] == 1          and event_data["after"]["session"] == 0      )
class SystemMonitor:          """ Identify events that occurred in the system
        >>> l1 = SystemMonitor({"before": {"session": 0}, "after": {"session": 1}})
         >>> l1.identify_event().__class__.__name__     'LoginEvent'
         >>> l2 = SystemMonitor({"before": {"session": 1}, "after": {"session": 0}})
         >>> l2.identify_event().__class__.__name__   'LogoutEvent'
          >>> l3 = SystemMonitor({"before": {"session": 1}, "after": {"session": 1}})
          >>> l3.identify_event().__class__.__name__      'UnknownEvent‘          """
  def __init__(self, event_data):       self.event_data = event_data
  def identify_event(self):
    for event_cls in   Event.__subclasses__():
      try:
        if event_cls.meets_condition(self.event_data):        return event_cls(self.event_data)
      except KeyError:
        continue
    return UnknownEvent(self.event_data)
```

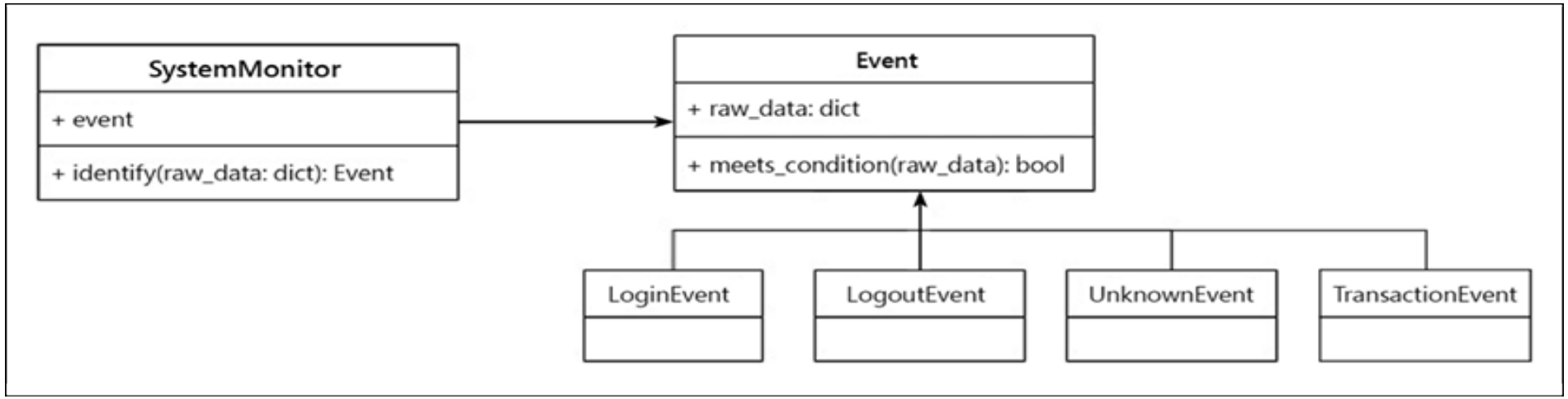
- Event sınıfı soyut ya da arayüz olabilecek bir temel sınıftır.
- Bu örnekte somut bir temel sınıf alınmıştır.
- Yöntem artık belirli olay tipine göre değil, yalnızca ortak arayüzün izlediği olaylara göre çalışır
  - Tümü **«meet_condition»** yöntemine göre polimorfizm özelliğine sahiptir.

- **__subclasses __ ()** yöntemi ile olaylara erişilir.

Yeni olay türlerinin eklenmesi durumunda, belirli kriterlere göre Event sınıfı genişletilecektir.
Bu sınıfın meet_condition () yöntemi yeni olay için çalıştırılacaktır ve bu da yeni bir sınıfın oluşturulmasıdır.

  - **__subclasses __ ()** yöntemi genişletilebilir tasarımı gösterir.
  - **identify_event** yöntemi kapalıdır. Etki alanına yeni bir etkinlik türü eklediğinde değiştirilmesi gerekmez.

«Event» Sistemlerinin Genişletilmesi



meets_condition yöntemi ile yeni bir davranış (behavior) eklenir.

```
class TransactionEvent(Event):
    """Represents a transaction that has just occurred on the system."""
 @staticmethod
  def meets_condition(event_data: dict):
        return event_data["after"].get("transaction")    is not None
```

```
class SystemMonitor:    """Identify events that occurred in the system
    >>> l1 = SystemMonitor({"before": {"session": 0}, "after": {"session": 1}})
    >>> l1.identify_event().__class__.__name__   'LoginEvent'
    >>> l2 = SystemMonitor({"before": {"session": 1}, "after": {"session": 0}})
    >>> l2.identify_event().__class__.__name__   'LogoutEvent'
    >>> l3 = SystemMonitor({"before": {"session": 1}, "after": {"session": 1}})
    >>> l3.identify_event().__class__.__name__   'UnknownEvent'
    >>> l4 = SystemMonitor({"after": {"transaction": "Tx001"}})
    >>> l4.identify_event().__class__.__name__   'TransactionEvent'   ""«

def __init__(self, event_data):
      self.event_data = event_data

   def identify_event(self):
      for event_cls in Event.__subclasses__():
         try:
            if event_cls.meets_condition(self.event_data):
               return event_cls(self.event_data)
         except KeyError:
            continue
      return UnknownEvent(self.event_data)
```

- Yeni olay türü eklendiğinde **SystemMonitor.identify_event()** yöntemi değişmez.

Bu yöntem yeni olay tipi eklense bile değişime kapalıdır.

- Event sınıfı, gerektiğinde yeni bir olay türü eklemeye izin verir.
  - ❖ yeni tiplere göre olaylar (events) bir uzantıya açıktır.
- Open/closed ilkesinin özüdür.
- Problemde bir yenilik gerektiğinde, sadece yeni kod ekleniyor, herhangi mevcut bir kod değişmiyor.