UML : Use Case

Una classe di funzionalità fornite dal sistema, cioè una astrazione di un insieme di scenari relazionati tra loro

- Diverse modalità di fare un compito
- Interazione normale e possibili eccezioni

Al suo interno vengono date, in maniera testuale non formalizzata, informazioni circa:

- Pre e post condizioni
- flusso normale di esecuzione
- eccezioni e loro trattamento …

Spesso vengono collegati ad altri diagrammi (Sequence , Activity) che ne spiegano il flusso

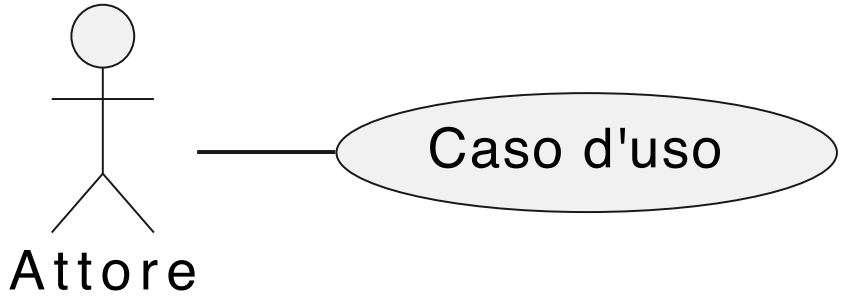
UML: scenari

Gli scenari sono descrizioni di come il sistema è usato in pratica

- Utili nella raccolta dei requisiti perché più semplici da scrivere di affermazioni astratte di ciò che il sistema deve fare
- Possono essere usati anche complementarmente a schede di descrizione dei requisiti (come esempi)

Use Cases

- Identificazione attori
- Denominazione del tipo di interazione
- Collegamenti tra attori e casi d'uso e tra casi d'uso



Chi sono gli attori?

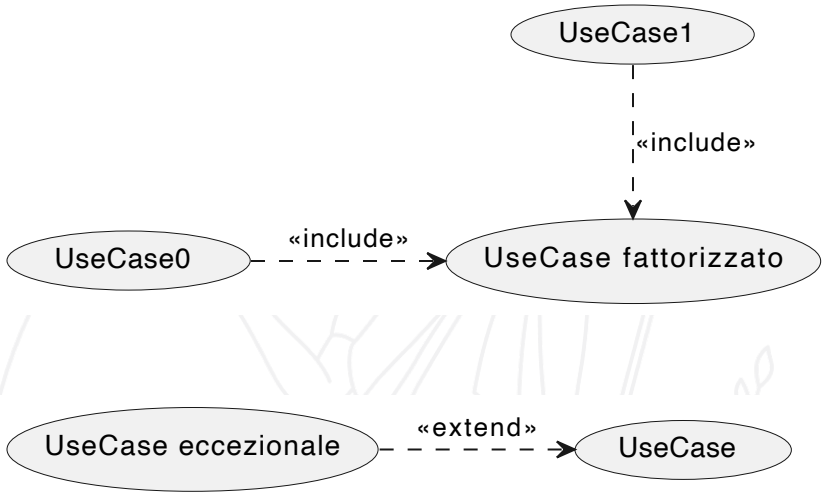
- È entità esterna al sistema
- Interagisce col sistema
  - Fonte o destinatario in scambi di informazioni
- Non sono una “persona” …
  - ma un ruolo che tale persona può coprire
  - Un utente
  - Un altro sistema con cui interfacciarsi
  - Una periferica hardware

Come identifico gli Uses cases

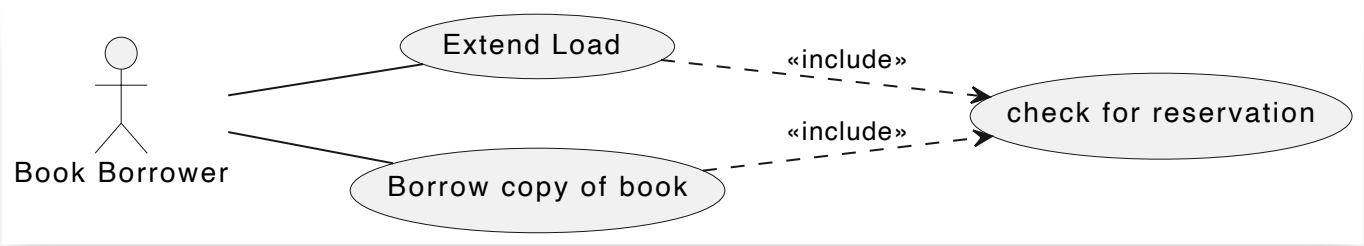
- Posso partire dalle funzionalità del sistema
- Posso partire dagli attori (i beneficiari)
  - Cosa fanno?
  - Cosa vogliono?

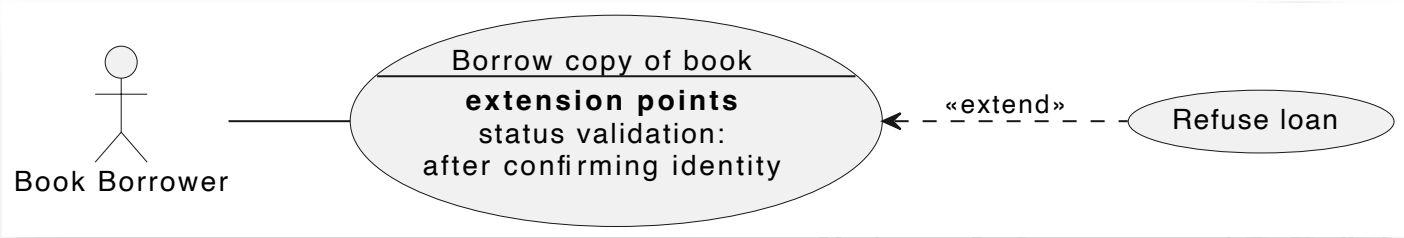
Associazioni

- Use cases / Attori
  - uno use case deve essere associato ad almeno un attore
  - un attore deve essere associato ad almeno uno use case
  - Esiste un attore detto *primario* che ha il compito di far partire lo use case
- Tra use case
  - include
  - extend



Esempi


Extend Load
«include»
check for reservation
«include»
Book Borrower
Borrow copy of book


Borrow copy of book
extension points
status validation:
after confirming identity
«extend»
Refuse loan
Book Borrower

Generalizzazione

Tra attori

Permette di esplicitare relazioni tra ruoli

- Un *StaffMember* è anche un *LibraryMember* nel nostro esempio

Tra Use Cases

Molto simile a **extends**

La differenza è che non ho dei punti di estensione, ma sostituisco alcune parti della descrizione, mentre eredito le altre…

Activity Diagram

In UML hanno molti punti in comune con il diagramma degli stati però:

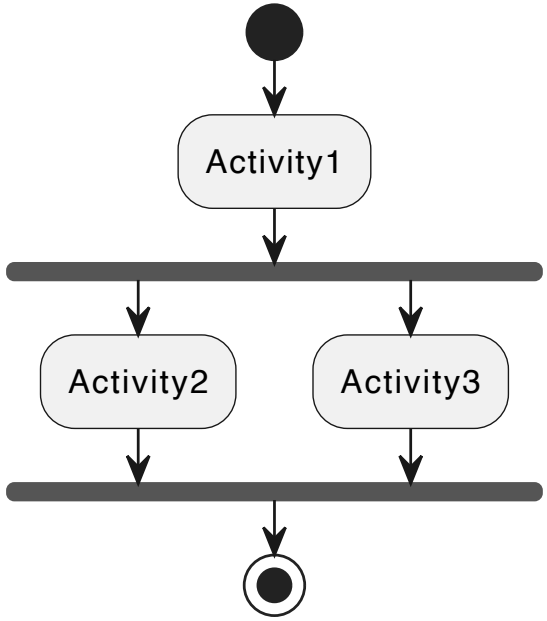
- Gli stati vengono chiamati attività
- Le transizioni di solito non sono etichettate con eventi (sono tutte del tipo “quando è terminata l’attività”)
- Le azioni di solito sono inserite dentro le attività
- Le attività possono essere interne o esterne al sistema
- i blocchi di sincronizzazione non sono "eccezione"

Diversi livelli di astrazione

Posso usare gli Activity Diagram per rappresentare:

- il flusso all'interno di un metodo
  - Con eventuali indicazioni di (pseudo)concorrenza
- il flusso di uno use-case
  - Alternativo (o ortogonale) a sequence
- la logica all'interno di un business process
  - Caso forse più comune

Sincronizzazione



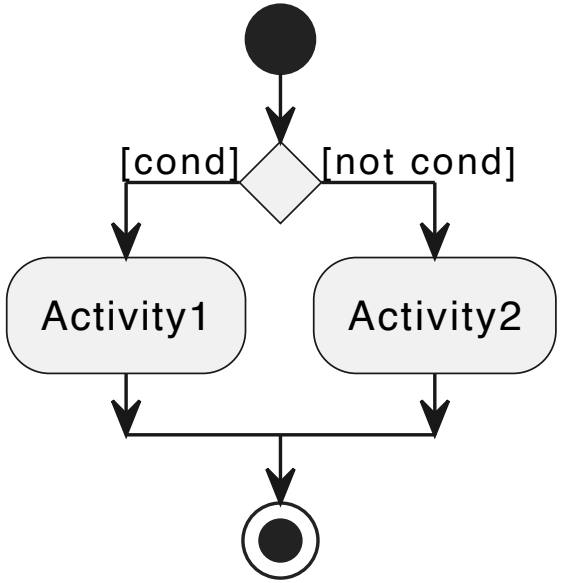
FORK/JOIN di attività

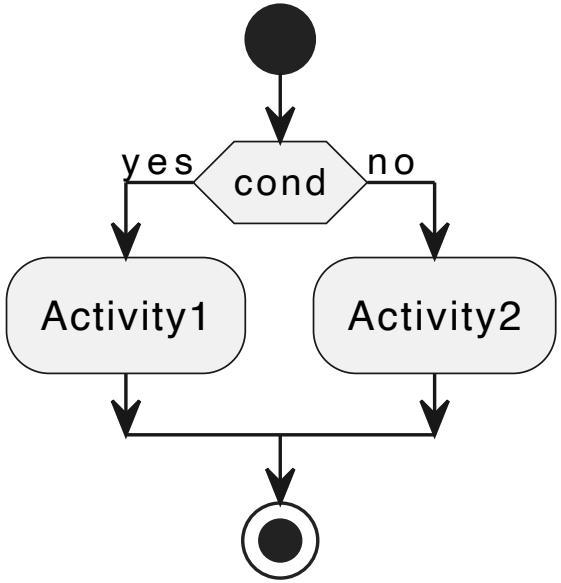
- Attraverso l'uso di barre si possono stabilire dei punti di sincronizzazione
  - se non specificato diversamente i join sono in ***and***

Decisioni

Permettono di evidenziare un momento di decisione

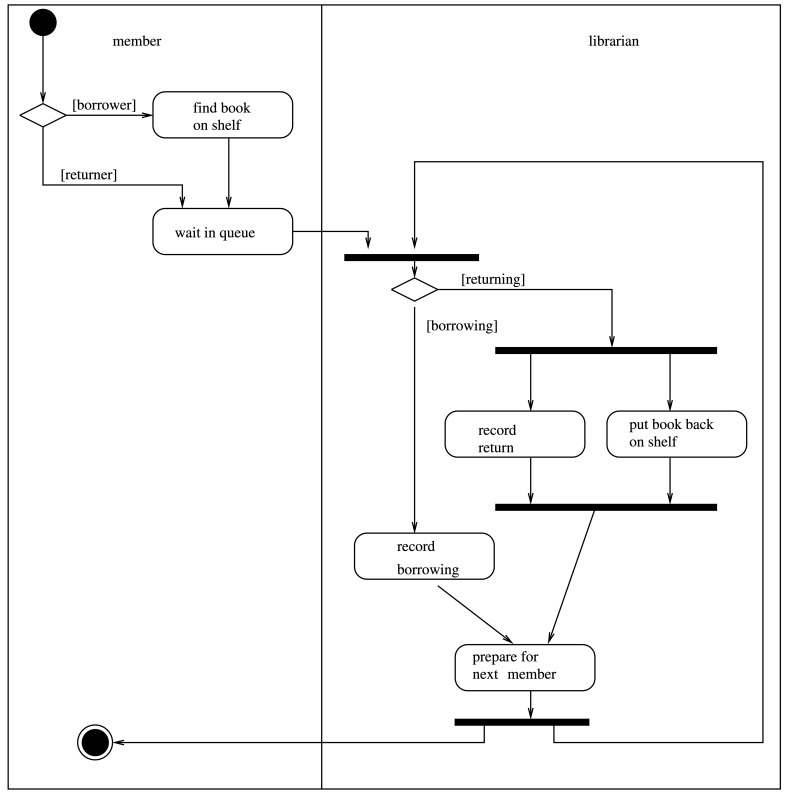
- Sono rappresentate da un piccolo rombo
- Sono necessarie?





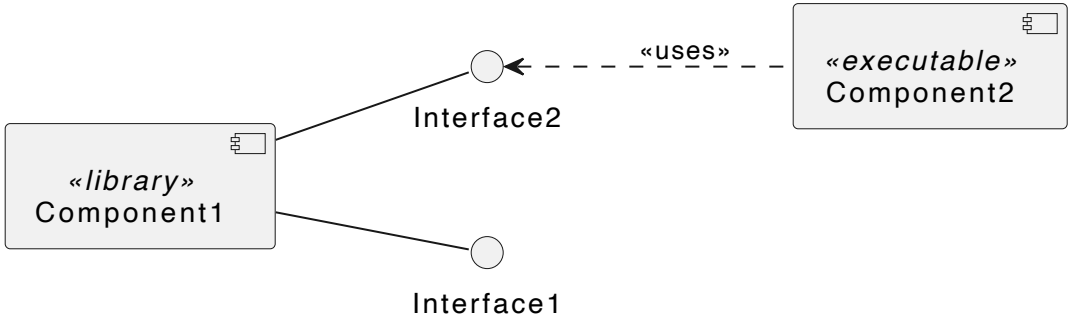
Swim lane

- Sono peculiari degli activity diagram
- Si possono partizionare le attività al fine di rappresentare le responsabilità sulle singole attività.
- Vengono identificate delle “corsie” verticali



Component diagram

- Permette di ragruppare ragionando in termini di *componenti fisici*
  - File (codice sorgente o file dati)
  - Libraries (modulo statico o libreria)
  - Executables (modulo eseguibile)
  - Table (una tabella di un DB)
  - Document (documenti in linguaggio naturale)



Caratteristiche di un componente

- Definisce una parte rimpiazzabile del sistema
- Svolge una funzione ben determinata
- Può essere annidato in altri componenti
- Vengono indicate chiaramente:
  - quali interfacce realizza (supporta)
  - le relazioni di dipendenza e di composizione

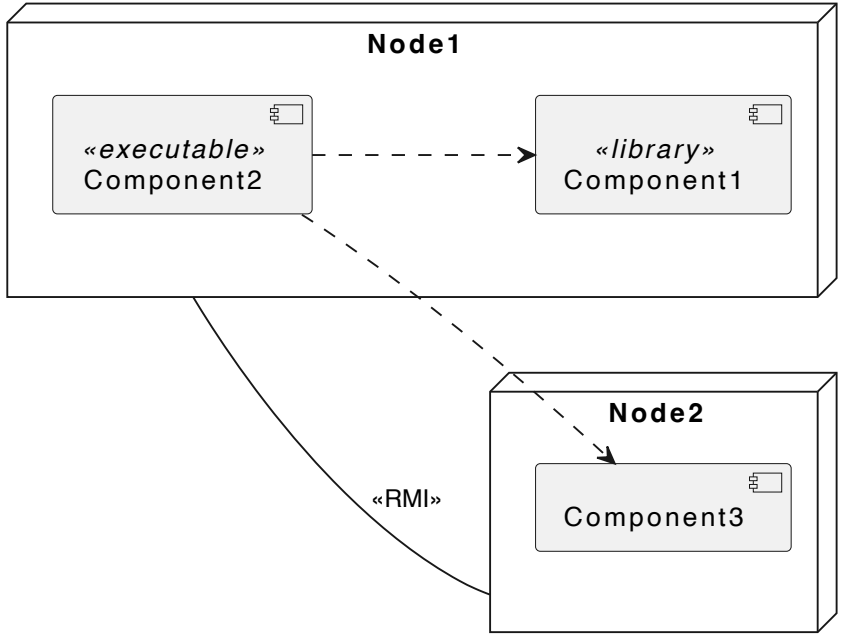
Deployment Diagram

Permette di specificare la dislocazione fisica delle istanze di componenti

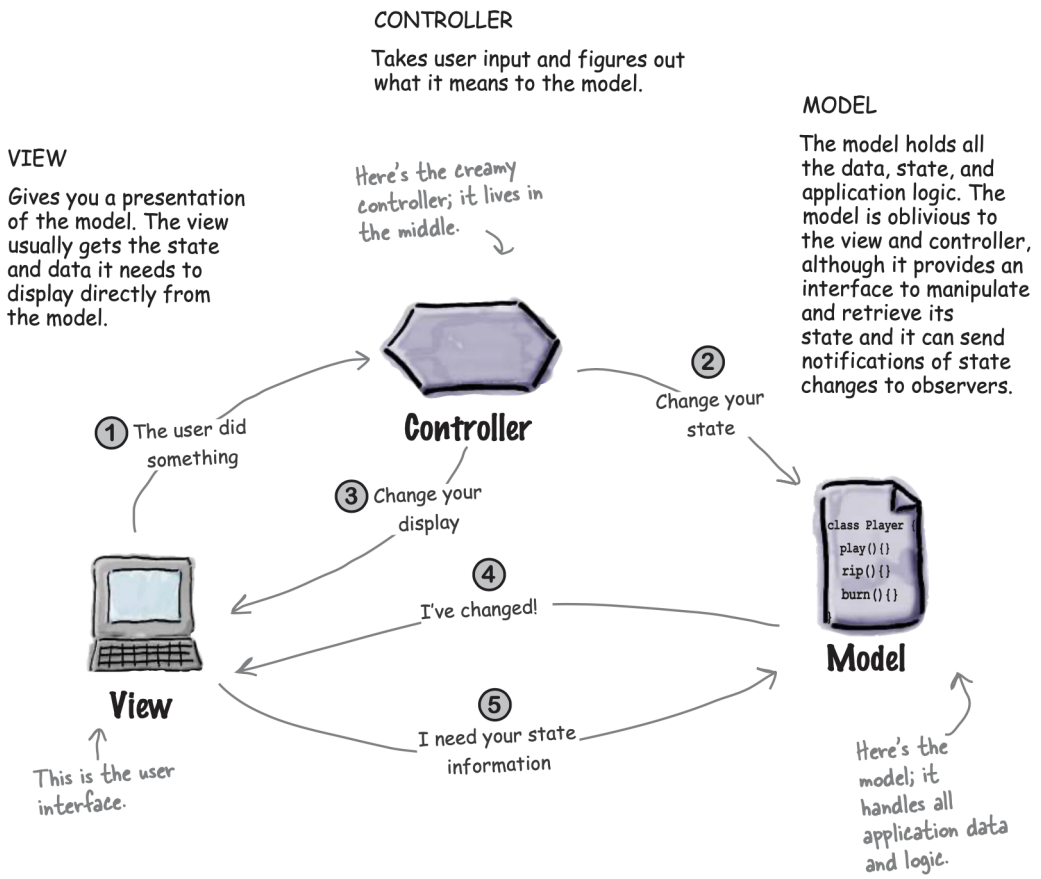
- Una vista statica della configurazione a run-time
- Di aiuto agli installatori

Permette di specificare:

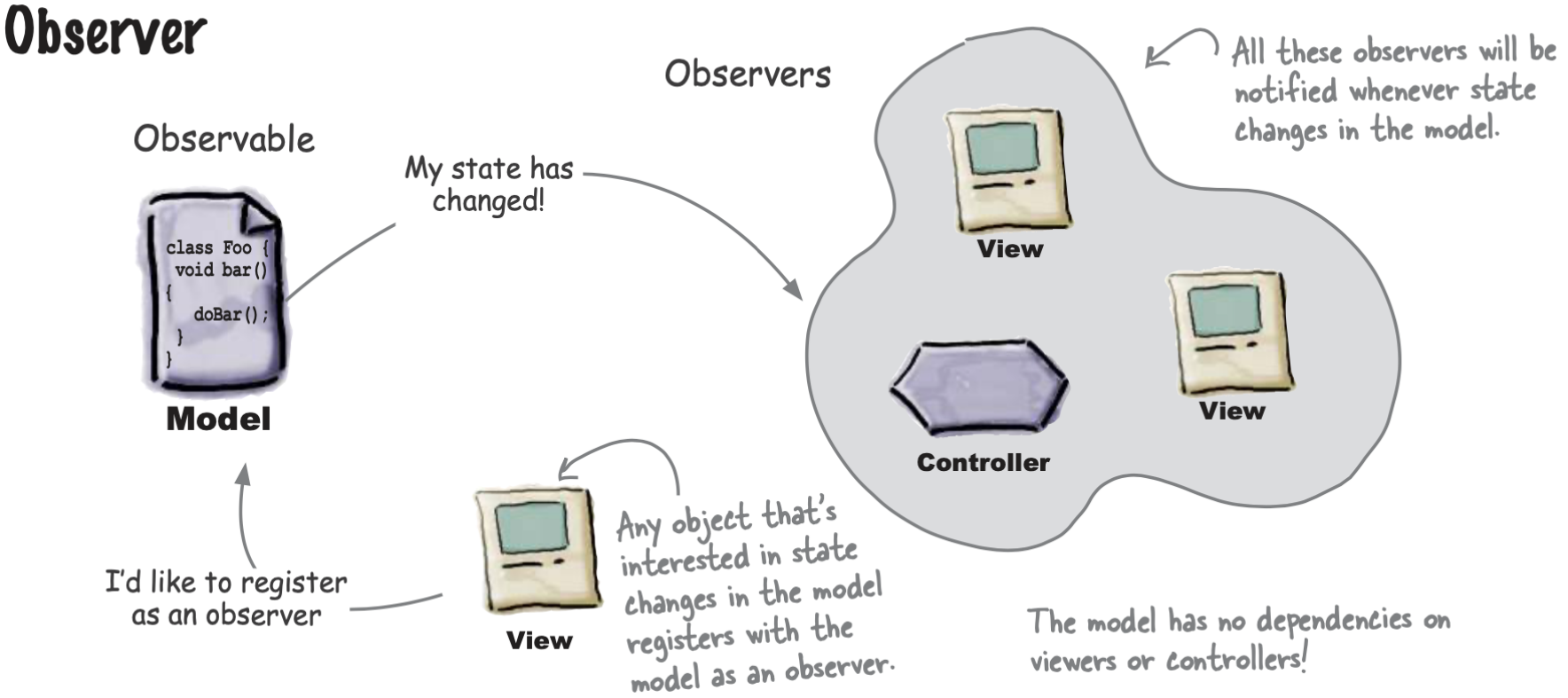
- Nodi del sistema (le macchine fisiche)
- Collegamenti tra nodi (RMI, HTTP,…)
- Dislocazione delle istanze di componenti all'interno dei nodi e le loro relazioni
  - Simili a quelle del component diagram, ma tra istanze



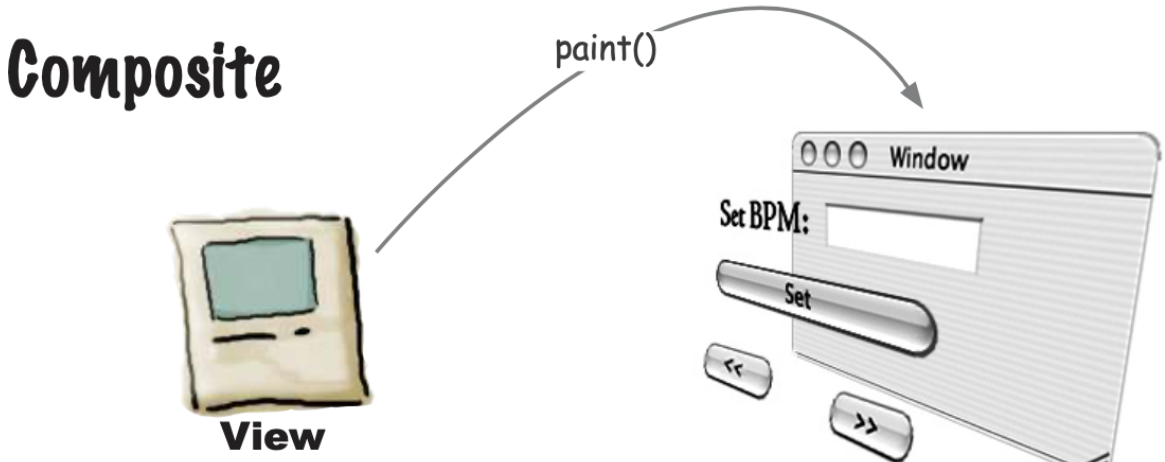
Model View Controller pattern



Observer part

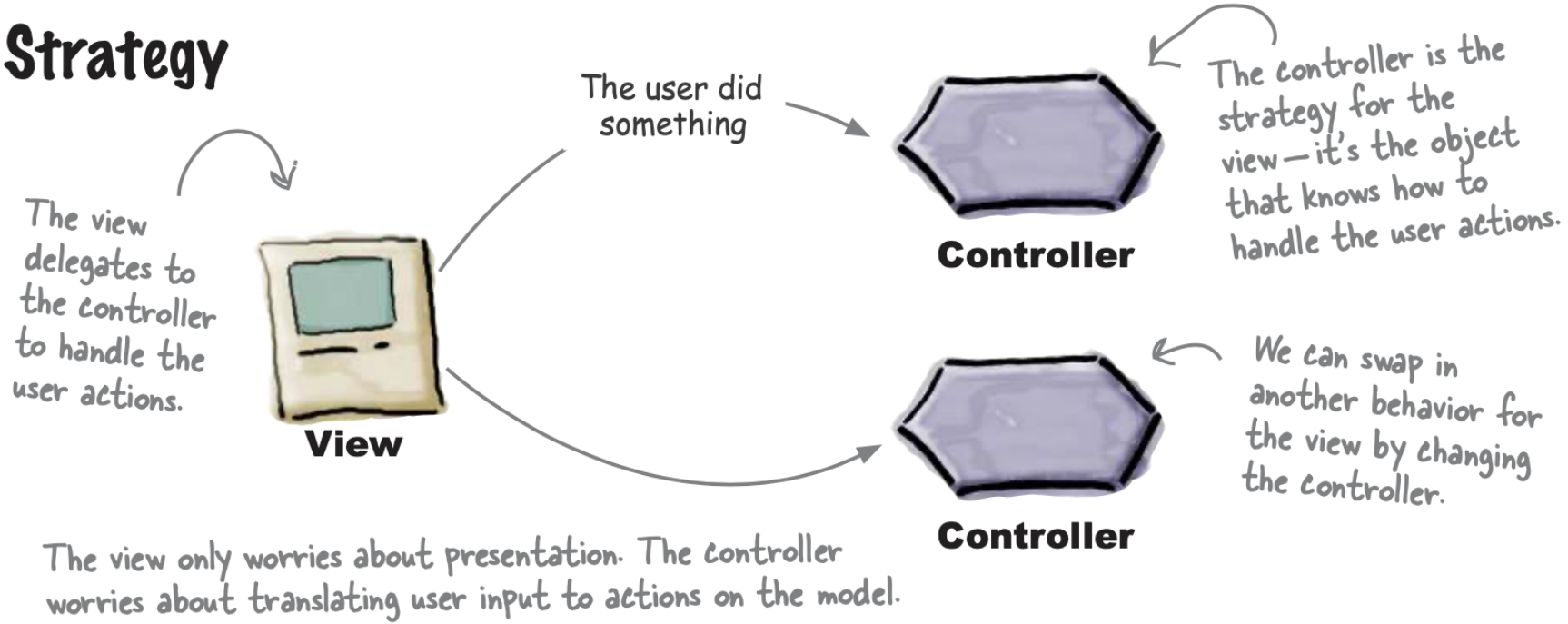


Composite part

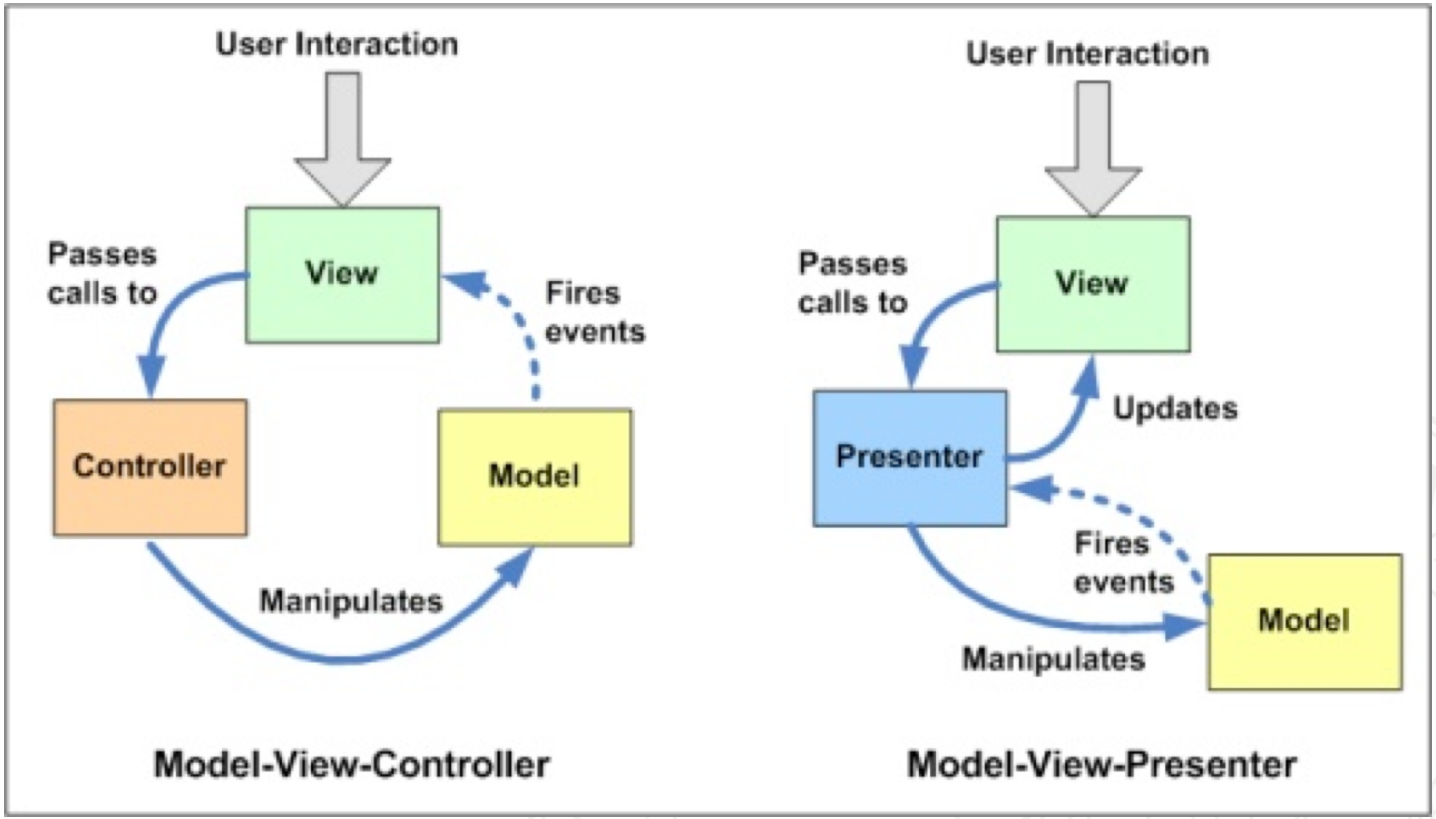


The view is a composite of GUI components (labels, buttons, text entry, etc.). The top-level component contains other components, which contain other components, and so on until you get to the leaf nodes.

Strategy part



MVC vs MVP


User Interaction
Passes calls to
View
Fires events
Controller
Model
Manipulates
Model-View-Controller
User Interaction
Passes calls to
View
Updates
Presenter
Fires events
Model
Manipulates
Model-View-Presenter

TEST getPunti()

```java
@ParameterizedTest
@CsvSource({
    "AC AS, 12",
    "2C AS, 13"
})
void testGetPunti(String cards, int points) {
  GiocatoreBJ SUT = mock(GiocatoreBJ.class);
  when(SUT.getPunti()).thenCallRealMethod();
  when(SUT.getCards()).thenAnswer(invocation -> of(cards).iterator());
  assertThat(SUT.getPunti()).isEqualTo(points);
}
```

```java
@ParameterizedTest
@CsvSource({
    "AC AS, 12",
    "2C AS, 13"
})
void testgePuntiMazziere(String cards, int points) {
  Mazziere SUT = spy(new Mazziere());
  doAnswer(invocation -> of(cards).iterator()).when(SUT).getCards();
  assertThat(SUT.getPunti()).isEqualTo(points);
}
```

TEST Sfidante.gioca() con DepInjection

```java
public class Sfidante implements GiocatoreBJ {
  final private String name;
  final private Mazziere banco;
  private List<Card> mano = new ArrayList<>();
  private Strategia strategia;

  public Sfidante(@NotNull String name, @NotNull Mazziere banco) {
    this.name = name;
    this.banco = banco;
  }

  public void setStrategia(@NotNull Strategia strategia) {
    this.strategia = strategia;
  }

  @Override
  public void gioca() {
    while (getPunti() < 21 && strategia.chiediCarta())
      mano.add(banco.draw());
  }
}
```

```java
@ExtendWith(MockitoExtension.class)
class MazziereTest {
  @Mock Mazziere banco;
  @InjectMocks Sfidante SUT;

  @Test
  void checkSfidanteGioca() {
    Strategia strat = mock();
    when(banco.draw())
        .thenReturn(Card.get(Rank.ACE,Suit.CLUBS));
    when(strat.chiediCarta())
        .thenReturn(true,true,false);

    SUT.carteIniziali();
    SUT.setStrategia(strat);
    SUT.gioca();

    verify(banco, times(4)).draw();
  }
}
```

Altra possibilità

```java
@ExtendWith(MockitoExtension.class)
public class SfidanteTest {
  @Mock Strategia strategia;
  Mazziere banco = mock(Mazziere.class);

  @InjectMocks
  Sfidante SUT = new Sfidante("carlo", banco);

  @Test
  public void testGioca() {
    when(strategia.chiediCarta()).thenReturn(true, true, false);
    when(banco.draw()).thenReturn(Card.get(Rank.ACE,Suit.CLUBS));

    SUT.carteIniziali();
    SUT.gioca();
    assertThatIterator(SUT.getCards()).toIterable().hasSize(4);
  }
}
```

```java
@ExtendWith(MockitoExtension.class)
class MazziereTest {
  @Mock Mazziere banco;
  @InjectMocks Sfidante SUT;

  @Test
  void checkSfidanteGioca(@Mock Strategia strat) {
    when(banco.draw())
        .thenReturn(Card.get(Rank.ACE,Suit.CLUBS));
    when(strat.chiediCarta())
        .thenReturn(true,true,false);

    SUT.carteIniziali();
    SUT.setStrategia(strat);
    SUT.gioca();

    verify(banco, times(4)).draw();
  }
}
```

TEST Sfidante.gioca() con DepInjection (cont.)

```java
@Test
void checkSfidanteGiocaSballando() {
  when(banco.draw()).thenReturn(Card.get(Rank.KING,Suit.CLUBS));
  when(strat.chiediCarta()).thenReturn(true);

  SUT.carteIniziali();
  SUT.setStrategia(strat);
  SUT.gioca();

  verify(banco, times(3)).draw();
}
```

```java
@Test
void checkSfidanteGioca21() {
  when(banco.draw()).thenReturn(Card.get(Rank.KING,Suit.CLUBS), Card.get(Rank.ACE, Suit.CLUBS));
  //when(strat.chiediCarta()).thenReturn(true,true,false);

  SUT.carteIniziali();
  SUT.setStrategia(strat);
  SUT.gioca();

  verify(banco, times(2)).draw();
}
```