Design Patterns

IF2210 – Semester II 2020/2021

Pattern (1)

- "Pattern is an **idea** that has been useful in one **practical context** and will probably be **useful in others.**" —Martin Fowler, *Analysis Patterns* (1996)
  - **Idea**: pattern can be anything.
  - **Practical context**: patterns are developed out of the practical experience of a real project—patterns are *discovered* rather than *invented*.
  - **Useful in other[context]s**: not all ideas are patterns.
- Dalam OO, *pattern* membantu dalam memodelkan persoalan di sebuah domain ke objek.

Pattern (2)

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." —Christopher Alexander, *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press, 1977)
- Pemikiran arsitek bangunan ini menginspirasi banyak arsitek preangkat lunak.
  - Buku paling populer, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995) ditulis oleh "Gang of Four": E. Gamma, R. Helm, R. Johnson, J. Vlissides.

Pattern ≠ Framework

- Salah satu keunggulan OO adalah *reusability*.
- Alangkah indahnya jika terdapat *component library* yang "lengkap" sehingga membuat *software* apapun dapat dilakukan dengan langsung merangkai objek-objek yang sudah ada.
  - Kenyataannya *reusability* hanya terjadi dengan "matang" di pengembangan GUI dan interaksi dengan basis data.
  - Sementara itu di *business level* tidak terjadi *reusability*, akibat kompleksitas di berbagai domain.
- *Pattern* membantu dalam pemodelan suatu domain dengan mengambil hikmah dari domain lain yang sudah pernah dimodelkan.
  - *Pattern*: alternative ways of modeling a situation,
  - *Framework*: choosing a particular model.

Buku “Gang of Four” (GoF)

- Berisi 23 *design pattern* yang dikelompokkan dalam 3 kategori:
  - **Creational**: mengenai penciptaan objek,
  - **Structural**: mengenai komposisi objek,
  - **Behavioral**: mengenai komunikasi antar objek.
- Setiap *pattern* dalam buku memiliki setidaknya 4 elemen:
  - Pattern name,
  - Problem (when to apply the pattern),
  - Solution (elements that make up the design),
  - Consequences (results, trade-offs).

23 pattern dari GoF

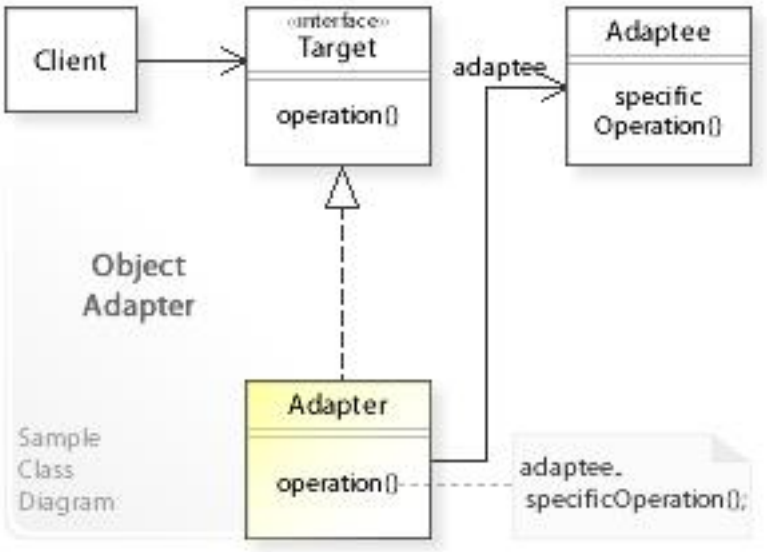
- Creational
  1. Abstract factory
  2. Builder
  3. Factory method
  4. Prototype
  5. Singleton
- Structural
  6. Adapter
  7. Bridge
  8. Composite
  9. Decorator
  10. Façade
  11. Flyweight
  12. Proxy
- Behavioral
  13. Chain of responsibility
  14. Command
  15. Interpreter
  16. Iterator
  17. Mediator
  18. Memento
  19. Observer
  20. State
  21. Strategy
  22. Template method
  23. Visitor

Contoh #1: Adapter (kutipan langsung GoF)

- Intent:
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Also Known As:
  - Wrapper
- Applicability:
  - Use the Adapter pattern when
    - you want to use an existing class, and its interface does not match the one you need.
    - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
    - (object adapter only) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

...dst (baca buku GoF)

Adapter: class diagram



Adapter: contoh kasus

- Ponsel Android memiliki *interface* Micro-USB untuk *charging*, sedangkan iPhone menggunakan lightning.
- iPhone bisa saja di-*charge* menggunakan kabel Micro-USB selama ada konektor yang membuat iPhone seolah memiliki *interface* Micro-USB.

```
interface MicroUsbPhone {
  void recharge();
  void useMicroUsb();
}

interface LightningPhone {
  void recharge();
  void useLightning();
}
```

*Adaptasi dari Wikipedia/Adapter pattern

```java
class Android implements MicroUsbPhone {
  private boolean connector;

  @Override
  public void useMicroUsb() {
    connector = true;
    System.out.println("MicroUsb connected");
  }

  @Override
  public void recharge() {
    if (connector) {
      System.out.println("Recharge started");
      System.out.println("Recharge finished");
    } else {
      throw new IllegalStateException("Connect MicroUsb first");
    }
  }
}
```

```java
class IPhone implements LightningPhone {
  private boolean connector;

  @Override
  public void useLightning() {
    connector = true;
    System.out.println("Lightning connected");
  }

  @Override
  public void recharge() {
    if (connector) {
      System.out.println("Recharge started");
      System.out.println("Recharge finished");
    } else {
      throw new IllegalStateException("Connect Lightning first");
    }
  }
}
```

```java
class LightningPhoneAsMicroUsbPhone implements MicroUsbPhone {
  private final LightningPhone lightningPhone;

  public LightningPhoneAsMicroUsbPhone(LightningPhone lightningPhone) {
    this.lightningPhone = lightningPhone;
  }

  @Override
  public void useMicroUsb() {
    System.out.println("MicroUsb connected");
    lightningPhone.useLightning();
  }

  @Override
  public void recharge() {
    lightningPhone.recharge();
  }
}
```

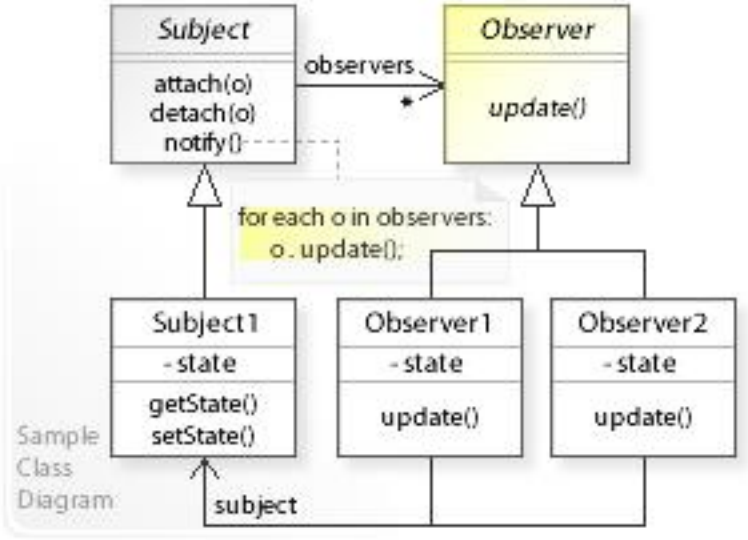
```
public class AdapterDemo {
  static void rechargeMicroUsbPhone(MicroUsbPhone phone) {
    phone.useMicroUsb();
    phone.recharge();
  }

  static void rechargeLightningPhone(LightningPhone phone) {
    phone.useLightning();
    phone.recharge();
  }

  public static void main(String[] args) {
    Android android = new Android();
    IPhone iPhone = new IPhone();

    System.out.println("Recharging android with MicroUsb");
    rechargeMicroUsbPhone(android);

    System.out.println("Recharging iPhone with Lightning");
    rechargeLightningPhone(iPhone);

    System.out.println("Recharging iPhone with MicroUsb");
    rechargeMicroUsbPhone(new LightningPhoneAsMicroUsbPhone(iPhone));
  }
}
```

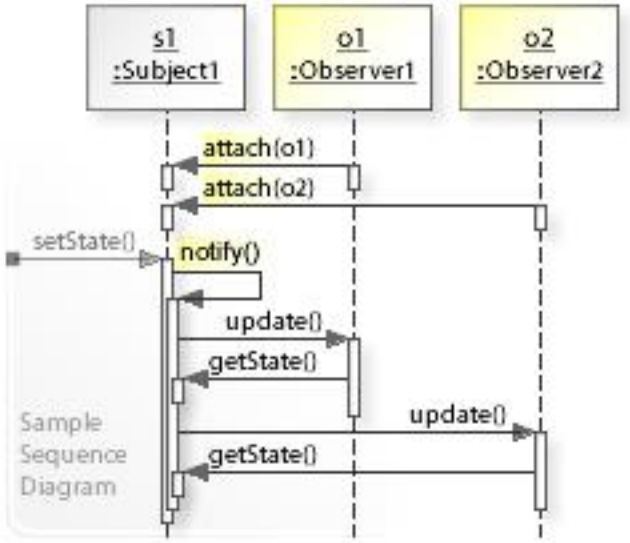
Contoh #2: Observer (kutipan langsung GoF)

- Intent:
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Also Known As:
  - Dependents, Publish-Subscribe[, Source-Target]
- Applicability:
  - Use the Observer pattern in any of the following situations:
    - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
    - When a change to one object requires changing others, and you don't know how many objects need to be changed.
    - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don’t want these objects tightly coupled.

...dst (baca buku GoF)

Observer: class & sequence diagram





```
class Source {
  public interface Target {
    void update(String event);
  }

  private final List<Target> targets = new ArrayList<>();

  private void notifyTargets(String event) {
    targets.forEach(target -> target.update(event));
  }

  public void addTarget(Target target) {
    targets.add(target);
  }

  public void scanSystemIn() {
    Scanner scanner = new Scanner(System.in);
    while (scanner.hasNextLine()) {
      String line = scanner.nextLine();
      notifyTargets(line);
    }
  }
}
```

*Adaptasi dari Wikipedia/Observer pattern

```java
public class SourceTargetDemo {
  public static void main(String[] args) {
    System.out.println("Enter Text: ");
    Source source = new Source();

    source.addTarget(event -> {
      System.out.println("Received response: " + event);
    });

    source.scanSystemIn();
  }
}
```

Kritik terhadap design pattern (1)

- Terdapat beberapa kritik terhadap *pattern* (khususnya GoF):
  - Sebagian *pattern* dianggap hanya *workaround* atas keterbatasan fitur bahasa pemrograman (C++, Java)
  - Orang yang baru belajar *pattern* umumnya melakukan *overusing* → *"If all you have is a hammer, everything looks like a nail."*
  - Sebagian *pattern* dinilai oleh sebagian pegiat OO sebagai "tidak OOP"
- Christopher Alexander dalam *The Timeless Way of Building* (1979) yang merupakan landasan filosofis dari *Pattern Language*, mengatakan bahwa *pattern* hanyalah **gerbang** yang harus dilewati **lalu ditinggalkan** untuk bisa mencapai *the timeless way*.

Kritik terhadap design pattern (2)

"The most disastrous thing about programming — to pick one of the 10 most disastrous things about programming — there's a very popular movement based on pattern languages. When Christopher Alexander first did that in architecture, he was looking at 2,000 years of ways that humans have made themselves comfortable. So there was actually something to it, because he was dealing with a genome that hasn't changed that much. I think he got a few hundred valuable patterns out of it. But the bug in trying to do that in computing is the assumption that we know anything at all about programming. So extracting patterns from today's programming practices ennobles them in a way they don't deserve. It actually gives them more cachet."

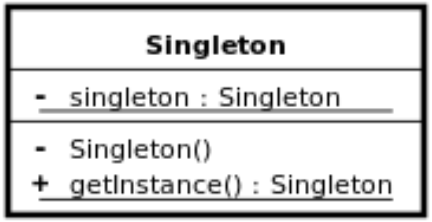
—Alan Kay (pencetus konsep OO, Smalltalk, dan GUI)

Contoh #3: Singleton (kutipan langsung GoF)

- Intent:
  - Ensure a class only has one instance, and provide a global point of access to it.
- Applicability:
  - Use the Singleton pattern when
    - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
    - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

...dst (baca buku GoF)

Singleton: class diagram



Singleton: contoh implementasi

```
public final class Singleton {

  private static final Singleton INSTANCE = new Singleton();

  private Singleton() {}

  public static Singleton getInstance() {
    return INSTANCE;
  }
}
```

*Adaptasi dari Wikipedia/Singleton pattern

Kritik: what is so bad about Singletons?

1. They are generally used as a global instance, why is that so bad? Because you hide the dependencies of your application in your code, instead of exposing them through the interfaces. Making something global to avoid passing it around is a code smell.
2. They violate the single responsibility principle: by controlling their own creation and lifecycle.
3. They inherently cause code to be tightly coupled. This makes faking them out under test rather difficult in many cases.
4. They carry state around for the lifetime of the application. Another hit to testing since you can end up with a situation where tests need to be ordered which is a big no no for unit tests. Why? Because each unit test should be independent from the other.

(Bagian dari diskusi di https://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons, *read for more!*)

Alternatif Singleton: Dependency Injection

```
// Contoh dengan singleton
public final class SingletonService { ... }
public class Client {
  public String greet() {
    return "Hello " + SingletonService.getInstance().getName();
  }
}
...
new Client().greet();


// Contoh dengan dependency injection
public class Client {
  private Service service;

  Client(Service service) { this.service = service; }

  public String greet() { return "Hello " + service.getName(); }
}
...
new Client(new SomeService()).greet();
```

Memandang design pattern (1)

- Kembali ke definisi awal, “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that ***you can use this solution a million times over, without ever doing it the same way twice.***”
- A pattern is a form of seed. It contains a **reflection of current work and thinking**, as well as the **vision of a future** in which the seeds all have been successfully cultivated
- Patterns are a **starting point**, not a destination (Fowler, 1996)

Memandang design pattern (2)

- Once you have identified a potentially useful pattern, **try it out.**
- **Read the full text** of the pattern so you get a sense of its **limitations** and **important features.**
- Try to make the pattern fit, but **don't try too hard.**
- Even if the pattern wasn't the right one, you **have not** wasted your time
  - You have **learned something** about the pattern and/or about the problem.
- If a pattern does not fit exactly, **modify it.**
  - Patterns are **suggestions**, not prescriptions.

(Fowler, 1996)

Design pattern: referensi

- Beberapa referensi lain tentang *pattern*:
  - Martin Fowler, *Analysis Patterns* (1996)
  - Kent Beck, *Smalltalk Best Practice Patterns* (1997)
  - Martin Fowler, *Patterns of Enterprise Application Architecture* (2002)
  - Prosiding konferensi-konferensi tahunan Pattern Languages of Programs (PLoP)

More examples!

Contoh #4: Decorator (GoF)

What problems can it solve?

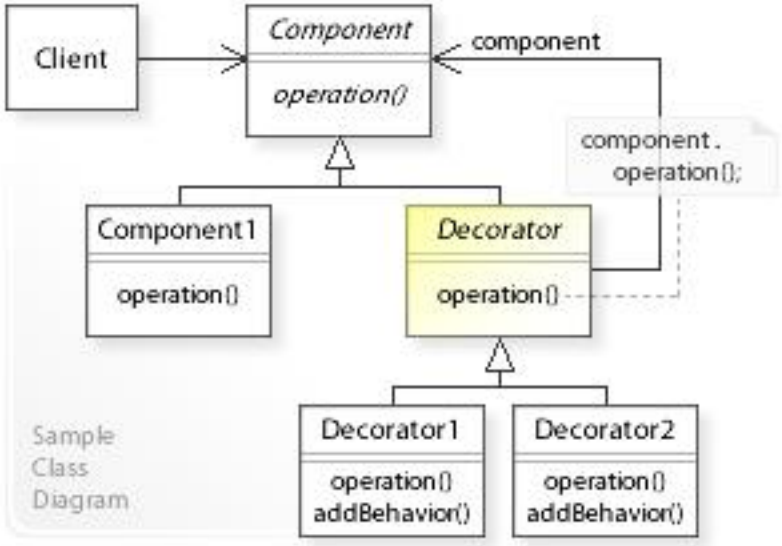
- Responsibilities should be added to (and removed from) an object dynamically at run-time.
- A flexible alternative to subclassing for extending functionality should be provided.

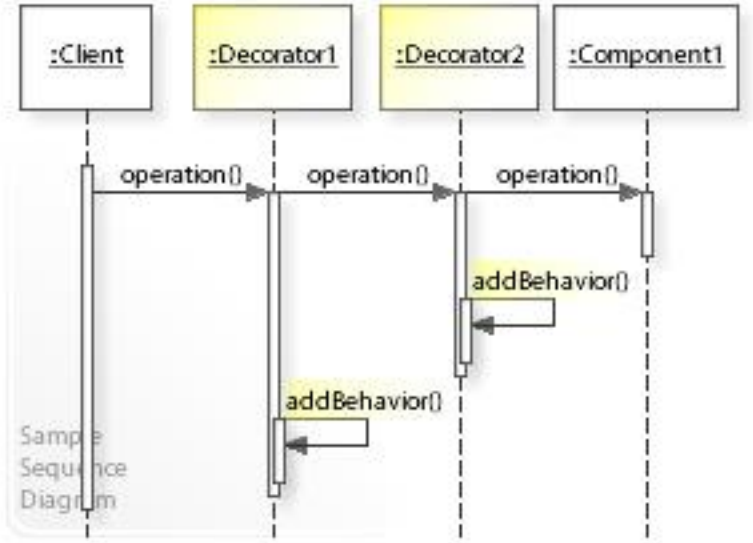
- When using subclassing, different subclasses extend a class in different ways. But an extension is bound to the class at compile-time and can't be changed at run-time.

What solution does it describe?

- Define Decorator objects that
  - implement the interface of the extended (decorated) object (Component) transparently by forwarding all requests to it
  - perform additional functionality before/after forwarding a request.
- This allows working with different Decorator objects to extend the functionality of an object dynamically at run-time.

Decorator: class & sequence diagram





Decorator: contoh implementasi

```
// The interface Coffee defines the functionality of Coffee implemented by
// decorator
public interface Coffee {
  public double cost(); // Returns the cost of the coffee
  public String ingredients(); // Returns the ingredients of the coffee
}

// Extension of a simple coffee without any extra ingredients
public class SimpleCoffee implements Coffee {
  @Override
  public double cost() {
    return 1;
  }

  @Override
  public String ingredients() {
    return "Coffee";
  }
}
```

*Adaptasi dari Wikipedia/Decorator pattern

```java
// Abstract decorator class - note that it implements Coffee interface
public abstract class CoffeeDecorator implements Coffee {
  private final Coffee decoratedCoffee;

  public CoffeeDecorator(Coffee c) {
    this.decoratedCoffee = c;
  }

  @Override
  public double cost() { // Implementing methods of the interface
    return decoratedCoffee.cost();
  }

  @Override
  public String ingredients() {
    return decoratedCoffee.ingredients();
  }
}
```

```java
// Decorator WithMilk mixes milk into coffee.
// Note it extends CoffeeDecorator.
class WithMilk extends CoffeeDecorator {
  public WithMilk(Coffee c) { super(c); }

  @Override
  public double cost() { return super.cost() + 0.5; }

  @Override
  public String ingredients() { return super.ingredients() + ", Milk"; }
}

// Decorator WithSprinkles mixes sprinkles onto coffee.
// Note it extends CoffeeDecorator.
class WithSprinkles extends CoffeeDecorator {
  public WithSprinkles(Coffee c) { super(c); }

  @Override
  public double cost() { return super.cost() + 0.2; }

  @Override
  public String ingredients() {
    return super.ingredients() + ", Sprinkles";
  }
}
```

```java
public class Main {
  public static void printInfo(Coffee c) {
    System.out.println("Cost: $" + c.cost() +
                       "; Ingredients: "  + c.ingredients());
  }

  public static void main(String[] args) {
    Coffee c = new SimpleCoffee();
    printInfo(c);

    c = new WithMilk(c);
    printInfo(c);

    c = new WithSprinkles(c);
    printInfo(c);
  }
}
```

```
Output:
Cost: $1.0; Ingredients: Coffee
Cost: $1.5; Ingredients: Coffee, Milk
Cost: $1.7; Ingredients: Coffee, Milk, Sprinkles
```

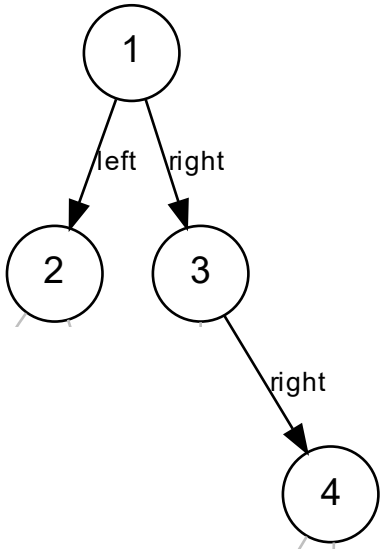
Contoh #5: Null Object (PLoP)

- *Object reference* bisa mengacu ke `null` sehingga sebelum objek digunakan, harus dilakukan *null-checking*.
- Misal, *node* sebuah pohon biner memiliki dua anak *left* dan *right*, masing-masing bisa `null`.
  - `nodeCount()` secara rekursif:
    ```
    return 1 + left.nodeCount() + right.nodeCount();
    ```
  - Karena left/right bisa `null`, harus:
    ```
    int sum=1;
    if (left!=null) sum+=left.nodeCount();
    if (right!=null) sum+=right.nodeCount();
    return sum;
    ```
- Untuk mengatasinya, dapat diciptakan objek yang berperan sebagai objek kosong.

Null Object: contoh implementasi

```
public class Node {

  public static Node EMPTY = new Node(0, null, null) {
    @Override
    public int nodeCount() { return 0; }
  };

  private int value;
  private Node left, right;

  public Node(int v, Node l, Node r) {
    value = v; left = l; right = r;
  }
  public Node(int v) {
    value = v; left = Node.EMPTY; right = Node.EMPTY;
  }
  public int nodeCount() {
    return 1 + left.nodeCount() + right.nodeCount();
  }
}
```

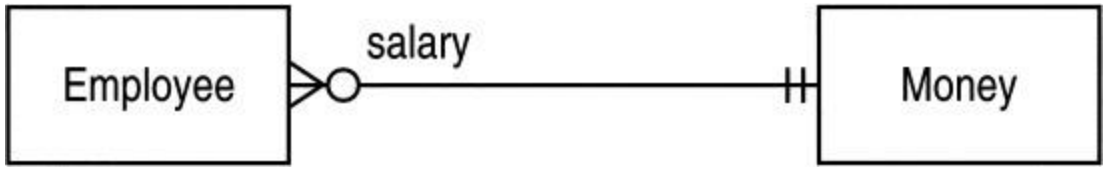
```
public class NodeDemo {
  public static void main(String[] args) {
    Node n =
      new Node(1,
        new Node(2),
        new Node(3,
          Node.EMPTY,
          new Node(4)));
    System.out.println(n.nodeCount());
  }
}
```



Contoh #6: Historic Mapping (Fowler)

- Objects do not just represent objects that exist in the real world;
  - they often represent the memories of objects that once existed but have since disappeared.

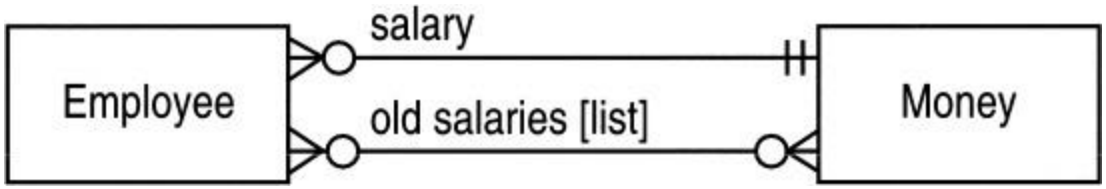
- Case: At any single moment a person has a single salary.



Model #1: At any point in time a person has one salary.

  - As time passes that salary may change.

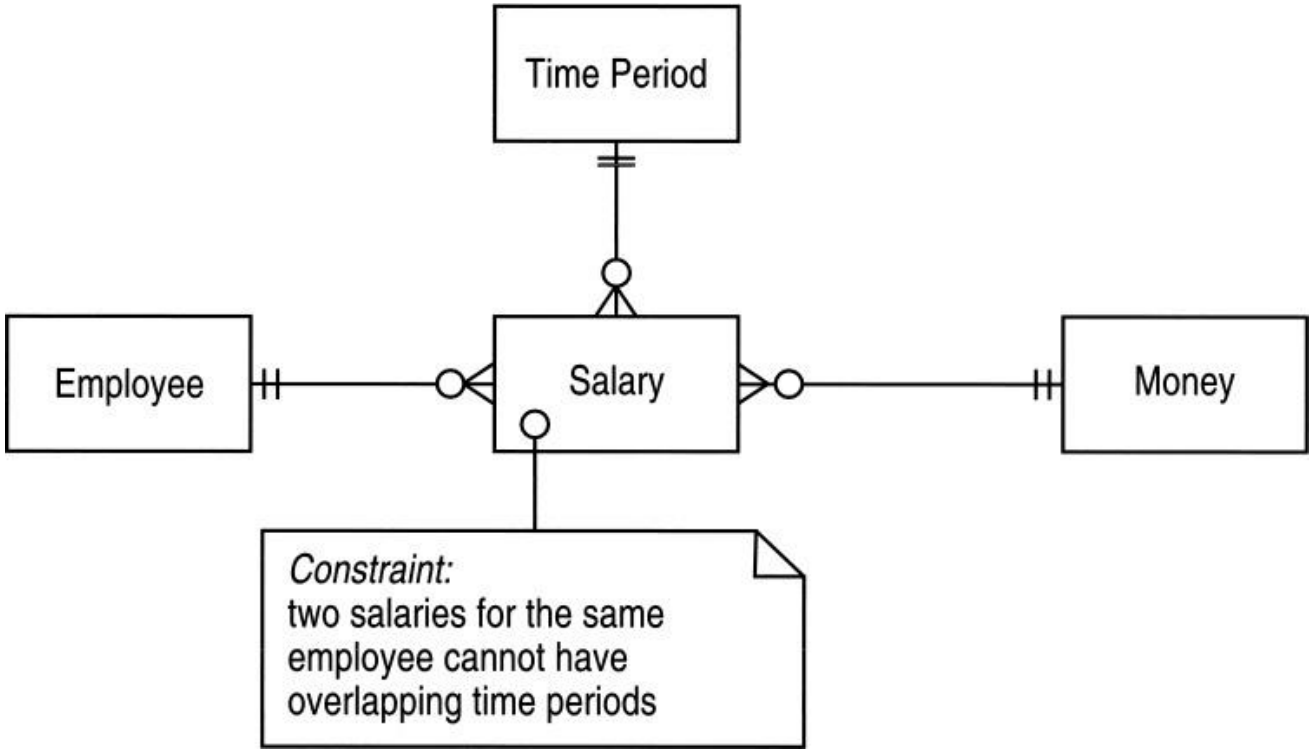
Historic Mapping: contoh kasus – model #2



Model #2: A model that remembers past salaries.

- Using this model, we remember past salaries.
  - Remember to add ability to append the old salary to an old salaries list.
  - Using a list ⟶ not only record previous salaries but also preserve the order.
- Doesn't help us answer the question "What was John Smith's salary on January 2, 1997?"

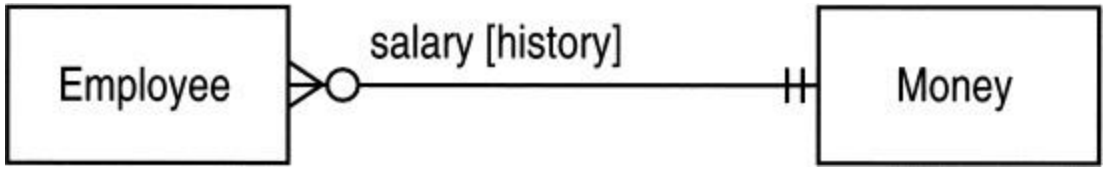
Historic Mapping: contoh kasus – model #3



Model #3: A full record of salary history.

- Provides the power we need, but too complex and hides the point that an employee can have only one salary at a time.

Historic Mapping: contoh kasus – model #4



Model #4: Representing the power of Model #3 with a simpler notation.

- Getting – method `salary()` returns current salary but actually calls `salary(Date d)` by supplying `Date.now()`
- Setting – more complex because of the rule that an employee must have one salary at any one date.
  - Must include taking the complete record out, changing it, and replacing it all at once.
- **Principle:** *If you come across a repetitive situation that is difficult to model, then define a notation. However, define a notation only if the resulting simplification outweighs the difficulty of remembering the extra notation.*