Design patterns

UNIVERSITÉ DE LORRAINE

Conception Architectures Distribuées

Horatiu CIRSTEA

Classification GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Problème

- Comment composer des objets dans des structures arborescentes pour représenter des hiérarchies ?
- Le client manipule uniformément les objets simples et les objets au sein de leurs compositions

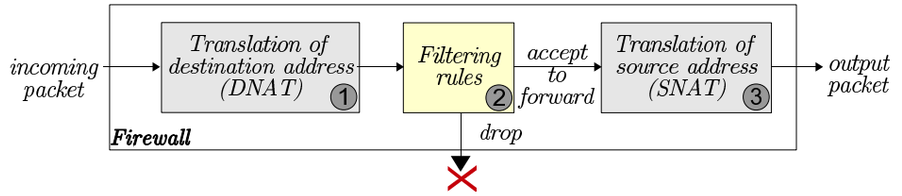


Figure 1. Firewall processing model

destination port. To control packet transmission between different subnetworks, it is common to deploy a network security policy based on a combination of firewalls. A firewall is an application that controls the forwarding of packets which cross it by using a combination of:

- *packet filtering*, which consists in inspecting each packet and either allowing it to continue its traversal or dropping it and
- *network address translation*, which consists in modifying network address information in packet headers.

Firewalls inspect incoming packets and accept or deny to forward them based on a list of decision rules which map the description of a set of packets to a decision. The most common criteria [11, 33] that firewalls use are the packet's source and destination address, its protocol, and, for TCP and UDP traffic, the port number. Moreover, firewalls often offer network address translation (NAT) functionality, which consists in rewriting the source (SNAT) or destination address (DNAT) into another address. The diagram in Figure 1 sums up the behavior of a firewall. At each step (1, 2 and 3), the packet is compared against a list of rules and the action (translation of destination address, drop or forward and translation of source address) corresponding to the first matched rule is performed.

EXAMPLE 1. *We give in Figure 2 a simple example of a firewall consisting of three rules: a filtering rule, a DNAT rule, and an SNAT rule. We use the CIDR notation [20] to denote subnetworks*[2]*. According to the rules of this firewall, any packet of protocol* tcp *whose destination address is* 192.168.5.130:80 *and whose source address is* 192.168.20.1:80 *(notation address:port) is forwarded by the firewall as a packet whose source address is* 121.130.1.1:80 *and whose destination address is* 121.130.1.15:80 *whereas any packet whose destination address is* 121.130.1.30:80 *is dropped by the firewall.*

3.2 Rewrite-based specification of firewalls

In this section, we present a formalization of firewalls based on rewrite systems. The ports and IP addresses are specified as terms and the firewall rules are specified as rewrite rules. Such specifications can be easily and automatically obtained from firewall configurations defined using classical firewall configuration languages such as netfilter [33].

3.2.1 Packets and subnetworks

In our approach, packets are represented as algebraic terms. For readability reasons, we consider that firewalls inspect only addresses and ports. Other information, such as protocols, TCP flags, states, could be considered without difficulty. The selected symbolic representation of packets is based on the following signature:

$$\begin{array}{lll} \mathbf{0}, \mathbf{1} & : & \mathsf{Binary} \to \mathsf{Binary} \\ \# & : & \to \mathsf{Binary} \\ \mathbf{from} & : & \mathsf{Binary} \times \mathsf{Binary} \to \mathsf{SrcAddr} \\ \mathbf{dest} & : & \mathsf{Binary} \times \mathsf{Binary} \to \mathsf{DstAddress} \\ \mathbf{packet} & : & \mathsf{SrcAddr} \times \mathsf{DstAddress} \to \mathsf{Packet} \end{array}$$

We briefly describe in this section the meaning of the above symbols and defer until Section 3.3 the discussion about the consequences of our choices.

IPv4 as well as IPv6 addresses can be equivalently seen as sequences of bits, tuples of hexadecimal numbers or integers. We have thus numerous possibilities to describe addresses as terms. However, we must keep in mind that the packet inspection performed by firewalls strongly relies on checking whether an address belongs to some given address ranges which generally correspond to subnetwork[3] domains. A special feature of subnetwork domains is that all hosts it contains are addressed with a common, identical prefix in their IP address when this address is written as a bit vector. It is thus natural to specify subnetworks as bit sequences of variable length and to use pattern matching for testing if an address belongs to a subnetwork domain.

By representing addresses as words over $\{0, 1\}$, or equivalently as terms built from monadic symbols $\mathbf{0}$ and $\mathbf{1}$, and a constant $\#$, we obtain a representation of subnetworks by linear terms built from $\mathbf{0}$ and $\mathbf{1}$. For example, the term

$$t = \mathbf{11000000\,10101000\,00010100}(x)$$

(we omit parentheses to keep readability) denotes the subnetwork $192.168.20.1/24$ whereas

$$\mathbf{11000000\,10101000\,00010100\,00000001}(\#)$$

denotes the IP address $192.168.20.1$. For convenience, we will use in this paper the dot-decimal notation for addresses, the decimal notation for ports and the CIDR notation for subnetworks. When a variable occurs in the corresponding term, it will be indicated in brackets. For example t will be denoted by $192.168.20.1/24[x]$. As we will see later on, this representation allows us to efficiently build tree automata recognizing addresses that belong to given ranges and consequently to efficiently analyse firewall behavior.

Finally, packets are terms of sort Packet. For example, the term $\mathbf{packet}\left(\begin{array}{l}\mathbf{from}(192.168.1.1, 80),\\ \mathbf{dest}(172.20.3.1, 80)\end{array}\right)$ represents a packet and $\mathbf{packet}\left(\begin{array}{l}\mathbf{from}(192.168.1.1/24[x], y),\\ \mathbf{dest}(172.20.3.1/24[x'], y')\end{array}\right)$ refers to the set of packets whose source address belongs to the subnetwork $192.168.1.1/24$ and whose destination address belongs to $172.20.3.1/24$.

As showed in Section 3.3, explicitly identifying the source and destination addresses (using the symbols $\mathbf{from}$ and $\mathbf{dest}$) allows

[2] The CIDR notation is a compact specification of an IP range of addresses. It consists of an IP address expressed in a dot-decimal notation and of a decimal number representing the size (in bits) of the common prefix of all the addresses in the range. The notation ip/n denotes the range of addresses whose n first bits (when addresses are expressed as binary numbers) are the same as the n first bits of the address ip. For example, $192.168.20.1/24$ refers to the range $[192.168.20.1, 192.168.20.255]$ and $121.130.1.1/28$ refers to the range $[121.130.1.1, 121.130.1.15]$.

[3] A subnetwork is a logically visible subdivision of a network characterized by an IP range (its domain).

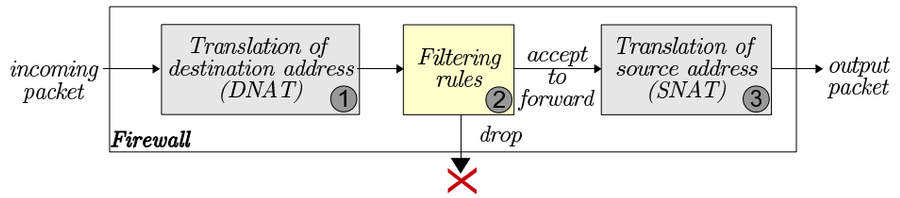


Figure 1. Firewall processing model

destination port. To control packet transmission between different subnetworks, it is common to deploy a network security policy based on a combination of firewalls. A firewall is an application that controls the forwarding of packets which cross it by using a combination of:

- *packet filtering*, which consists in inspecting each packet and either allowing it to continue its traversal or dropping it and
- *network address translation*, which consists in modifying network address information in packet headers.

Firewalls inspect incoming packets and accept or deny to forward them based on a list of decision rules which map the description of a set of packets to a decision. The most common criteria [11, 33] that firewalls use are the packet's source and destination address, its protocol, and, for TCP and UDP traffic, the port number. Moreover, firewalls often offer network address translation (NAT) functionality, which consists in rewriting the source (SNAT) or destination address (DNAT) into another address. The diagram in Figure 1 sums up the behavior of a firewall. At each step (1, 2 and 3), the packet is compared against a list of rules and the action (translation of destination address, drop or forward and translation of source address) corresponding to the first matched rule is performed.

EXAMPLE 1. *We give in Figure 2 a simple example of a firewall consisting of three rules: a filtering rule, a DNAT rule, and an SNAT rule. We use the CIDR notation [20] to denote subnetworks*[2]*. According to the rules of this firewall, any packet of protocol* tcp *whose destination address is* 192.168.5.130:80 *and whose source address is* 192.168.20.1:80 *(notation address:port) is forwarded by the firewall as a packet whose source address is* 121.130.1.1:80 *and whose destination address is* 121.130.1.15:80 *whereas any packet whose destination address is* 121.130.1.30:80 *is dropped by the firewall.*

3.2 Rewrite-based specification of firewalls

In this section, we present a formalization of firewalls based on rewrite systems. The ports and IP addresses are specified as terms and the firewall rules are specified as rewrite rules. Such specifications can be easily and automatically obtained from firewall configurations defined using classical firewall configuration languages such as netfilter [33].

3.2.1 Packets and subnetworks

In our approach, packets are represented as algebraic terms. For readability reasons, we consider that firewalls inspect only addresses and ports. Other information, such as protocols, TCP flags, states, could be considered without difficulty. The selected symbolic representation of packets is based on the following signature:

$$\begin{array}{lll} \mathbf{0}, \mathbf{1} & : & \mathsf{Binary} \to \mathsf{Binary} \\ \# & : & \to \mathsf{Binary} \\ \mathbf{from} & : & \mathsf{Binary} \times \mathsf{Binary} \to \mathsf{SrcAddr} \\ \mathbf{dest} & : & \mathsf{Binary} \times \mathsf{Binary} \to \mathsf{DstAddress} \\ \mathbf{packet} & : & \mathsf{SrcAddr} \times \mathsf{DstAddress} \to \mathsf{Packet} \end{array}$$

We briefly describe in this section the meaning of the above symbols and defer until Section 3.3 the discussion about the consequences of our choices.

IPv4 as well as IPv6 addresses can be equivalently seen as sequences of bits, tuples of hexadecimal numbers or integers. We have thus numerous possibilities to describe addresses as terms. However, we must keep in mind that the packet inspection performed by firewalls strongly relies on checking whether an address belongs to some given address ranges which generally correspond to subnetwork[3] domains. A special feature of subnetwork domains is that all hosts it contains are addressed with a common, identical prefix in their IP address when this address is written as a bit vector. It is thus natural to specify subnetworks as bit sequences of variable length and to use pattern matching for testing if an address belongs to a subnetwork domain.

By representing addresses as words over $\{0, 1\}$, or equivalently as terms built from monadic symbols $\mathbf{0}$ and $\mathbf{1}$, and a constant $\#$, we obtain a representation of subnetworks by linear terms built from $\mathbf{0}$ and $\mathbf{1}$. For example, the term

$$t = \mathbf{11000000\,10101000\,00010100}(x)$$

(we omit parentheses to keep readability) denotes the subnetwork $192.168.20.1/24$ whereas

$$\mathbf{11000000\,10101000\,00010100\,00000001}(\#)$$

denotes the IP address $192.168.20.1$. For convenience, we will use in this paper the dot-decimal notation for addresses, the decimal notation for ports and the CIDR notation for subnetworks. When a variable occurs in the corresponding term, it will be indicated in brackets. For example t will be denoted by $192.168.20.1/24[x]$. As we will see later on, this representation allows us to efficiently build tree automata recognizing addresses that belong to given ranges and consequently to efficiently analyse firewall behavior.

Finally, packets are terms of sort Packet. For example, the term $\mathbf{packet}\left(\begin{array}{l}\mathbf{from}(192.168.1.1, 80),\\ \mathbf{dest}(172.20.3.1, 80)\end{array}\right)$ represents a packet and $\mathbf{packet}\left(\begin{array}{l}\mathbf{from}(192.168.1.1/24[x], y),\\ \mathbf{dest}(172.20.3.1/24[x'], y')\end{array}\right)$ refers to the set of packets whose source address belongs to the subnetwork $192.168.1.1/24$ and whose destination address belongs to $172.20.3.1/24$.

As showed in Section 3.3, explicitly identifying the source and destination addresses (using the symbols **from** and **dest**) allows

[2] The CIDR notation is a compact specification of an IP range of addresses. It consists of an IP address expressed in a dot-decimal notation and of a decimal number representing the size (in bits) of the common prefix of all the addresses in the range. The notation ip/n denotes the range of addresses whose n first bits (when addresses are expressed as binary numbers) are the same as the n first bits of the address ip. For example, $192.168.20.1/24$ refers to the range $[192.168.20.1, 192.168.20.255]$ and $121.130.1.1/28$ refers to the range $[121.130.1.1, 121.130.1.15]$.

[3] A subnetwork is a logically visible subdivision of a network characterized by an IP range (its domain).

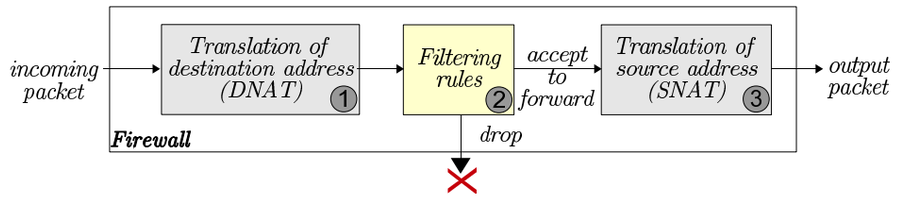


Figure 1. Firewall processing model

destination port. To control packet transmission between different subnetworks, it is common to deploy a network security policy based on a combination of firewalls. A firewall is an application that controls the forwarding of packets which cross it by using a combination of:

- *packet filtering*, which consists in inspecting each packet and either allowing it to continue its traversal or dropping it and
- *network address translation*, which consists in modifying network address information in packet headers.

Firewalls inspect incoming packets and accept or deny to forward them based on a list of decision rules which map the description of a set of packets to a decision. The most common criteria [11, 33] that firewalls use are the packet's source and destination address, its protocol, and, for TCP and UDP traffic, the port number. Moreover, firewalls often offer network address translation (NAT) functionality, which consists in rewriting the source (SNAT) or destination address (DNAT) into another address. The diagram in Figure 1 sums up the behavior of a firewall. At each step (1, 2 and 3), the packet is compared against a list of rules and the action (translation of destination address, drop or forward and translation of source address) corresponding to the first matched rule is performed.

EXAMPLE 1. *We give in Figure 2 a simple example of a firewall consisting of three rules: a filtering rule, a DNAT rule, and an SNAT rule. We use the CIDR notation [20] to denote subnetworks*[2]*. According to the rules of this firewall, any packet of protocol* tcp *whose destination address is* 192.168.5.130:80 *and whose source address is* 192.168.20.1:80 *(notation address:port) is forwarded by the firewall as a packet whose source address is* 121.130.1.1:80 *and whose destination address is* 121.130.1.15:80 *whereas any packet whose destination address is* 121.130.1.30:80 *is dropped by the firewall.*

3.2 Rewrite-based specification of firewalls

In this section, we present a formalization of firewalls based on rewrite systems. The ports and IP addresses are specified as terms and the firewall rules are specified as rewrite rules. Such specifications can be easily and automatically obtained from firewall configurations defined using classical firewall configuration languages such as netfilter [33].

3.2.1 Packets and subnetworks

In our approach, packets are represented as algebraic terms. For readability reasons, we consider that firewalls inspect only addresses and ports. Other information, such as protocols, TCP flags

[2] The CIDR notation is a compact specification of an IP range of addresses. It consists of an IP address expressed in a dot-decimal notation and of a decimal number representing the size (in bits) of the common prefix of all the addresses in the range. The notation ip/n denotes the range of addresses whose n first bits (when addresses are expressed as binary numbers) are the same as the n first bits of the address ip. For example, 192.168.20.1/24 refers to the range [192.168.20.1, 192.168.20.255] and 121.130.1.1/28 refers to the range [121.130.1.1, 121.130.1.15].

states, could be considered without difficulty. The selected symbolic representation of packets is based on the following signature:

$$\begin{array}{lll} \mathbf{0}, \mathbf{1} & : & \mathsf{Binary} \to \mathsf{Binary} \\ \# & : & \to \mathsf{Binary} \\ \mathbf{from} & : & \mathsf{Binary} \times \mathsf{Binary} \to \mathsf{SrcAddr} \\ \mathbf{dest} & : & \mathsf{Binary} \times \mathsf{Binary} \to \mathsf{DstAddress} \\ \mathbf{packet} & : & \mathsf{SrcAddr} \times \mathsf{DstAddress} \to \mathsf{Packet} \end{array}$$

We briefly describe in this section the meaning of the above symbols and defer until Section 3.3 the discussion about the consequences of our choices.

IPv4 as well as IPv6 addresses can be equivalently seen as sequences of bits, tuples of hexadecimal numbers or integers. We have thus numerous possibilities to describe addresses as terms. However, we must keep in mind that the packet inspection performed by firewalls strongly relies on checking whether an address belongs to some given address ranges which generally correspond to subnetwork[3] domains. A special feature of subnetwork domains is that all hosts it contains are addressed with a common, identical prefix in their IP address when this address is written as a bit vector. It is thus natural to specify subnetworks as bit sequences of variable length and to use pattern matching for testing if an address belongs to a subnetwork domain.

By representing addresses as words over $\{0, 1\}$, or equivalently as terms built from monadic symbols $\mathbf{0}$ and $\mathbf{1}$, and a constant $\#$, we obtain a representation of subnetworks by linear terms built from $\mathbf{0}$ and $\mathbf{1}$. For example, the term

$$t = \mathbf{11000000\,10101000\,00010100}(x)$$

(we omit parentheses to keep readability) denotes the subnetwork 192.168.20.1/24 whereas

$$\mathbf{11000000\,10101000\,00010100\,00000001}(\#)$$

denotes the IP address 192.168.20.1. For convenience, we will use in this paper the dot-decimal notation for addresses, the decimal notation for ports and the CIDR notation for subnetworks. When a variable occurs in the corresponding term, it will be indicated in brackets. For example t will be denoted by $192.168.20.1/24[x]$. As we will see later on, this representation allows us to efficiently build tree automata recognizing addresses that belong to given ranges and consequently to efficiently analyse firewall behavior.

Finally, packets are terms of sort Packet. For example, the term $\mathbf{packet}\left(\begin{array}{l}\mathbf{from}(192.168.1.1, 80),\\ \mathbf{dest}(172.20.3.1, 80)\end{array}\right)$ represents a packet $\left(\begin{array}{l}68.1.1/24[x], y),\\ 0.3.1/24[x'], y')\end{array}\right)$ refers to the address belongs to the subnetwork destination address belongs to 172.20.3.1/24.

As showed in Section 3.3, explicitly identifying the source and destination addresses (using the symbols **from** and **dest**) allows

[3] A subnetwork is a logically visible subdivision of a network characterized by an IP range (its domain).

section

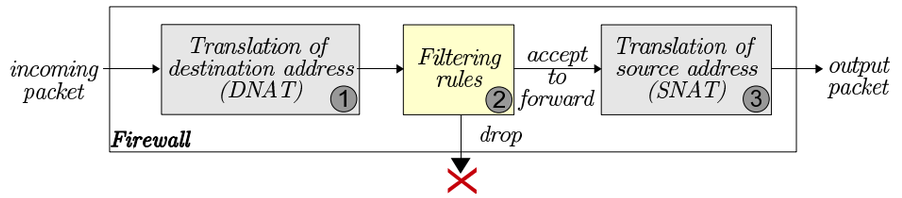


Figure 1. Firewall processing model

destination port. To control packet transmission between different subnetworks, it is common to deploy a network security policy based on a combination of firewalls. A firewall is an application that controls the forwarding of packets which cross it by using a combination of:

- *packet filtering*, which consists in inspecting each packet and either allowing it to continue its traversal or dropping it and
- *network address translation*, which consists in modifying network address information in packet headers.

Firewalls inspect incoming packets and accept or deny to forward them based on a list of decision rules which map the description of a set of packets to a decision. The most common criteria [11, 33] that firewalls use are the packet's source and destination address, its protocol, and, for TCP and UDP traffic, the port number. Moreover, firewalls often offer network address translation (NAT) functionality, which consists in rewriting the source (SNAT) or destination address (DNAT) into another address. The diagram in Figure 1 sums up the behavior of a firewall. At each step (1, 2 and 3), the packet is compared against a list of rules and the action (translation of destination address, drop or forward and translation of source address) corresponding to the first matched rule is performed.

EXAMPLE 1. *We give in Figure 2 a simple example of a firewall consisting of three rules: a filtering rule, a DNAT rule, and an SNAT rule. We use the CIDR notation [20] to denote subnetworks*[2]*. According to the rules of this firewall, any packet of protocol* tcp *whose destination address is* 192.168.5.130:80 *and whose source address is* 192.168.20.1:80 *(notation address:port) is forwarded by the firewall as a packet whose source address is* 121.130.1.1:80 *and whose destination address is* 121.130.1.15:80 *whereas any packet whose destination address is* 121.130.1.30:80 *is dropped by the firewall.*

3.2 Rewrite-based specification of firewalls

In this section, we present a formalization of firewalls based on rewrite systems. The ports and IP addresses are specified as terms and the firewall rules are specified as rewrite rules. Such specifications can be easily and automatically obtained from firewall configurations defined using classical firewall configuration languages such as netfilter [33].

sous-section

3.2.1 Packets and subnetworks

In our approach, packets are represented as algebraic terms. For readability reasons, we consider that firewalls inspect only addresses and ports. Other information, such as protocols, TCP flags, states, could be considered without difficulty. The selected symbolic representation of packets is based on the following signature:

$$\begin{array}{lll} \mathbf{0}, \mathbf{1} & : & \mathsf{Binary} \to \mathsf{Binary} \\ \# & : & \to \mathsf{Binary} \\ \mathbf{from} & : & \mathsf{Binary} \times \mathsf{Binary} \to \mathsf{SrcAddr} \\ \mathbf{dest} & : & \mathsf{Binary} \times \mathsf{Binary} \to \mathsf{DstAddress} \\ \mathbf{packet} & : & \mathsf{SrcAddr} \times \mathsf{DstAddress} \to \mathsf{Packet} \end{array}$$

We briefly describe in this section the meaning of the above symbols and defer until Section 3.3 the discussion about the consequences of our choices.

IPv4 as well as IPv6 addresses can be equivalently seen as sequences of bits, tuples of hexadecimal numbers or integers. We have thus numerous possibilities to describe addresses as terms. However, we must keep in mind that the packet inspection performed by firewalls strongly relies on checking whether an address belongs to some given address ranges which generally correspond to subnetwork[3] domains. A special feature of subnetwork domains is that all hosts it contains are addressed with a common, identical prefix in their IP address when this address is written as a bit vector. It is thus natural to specify subnetworks as bit sequences of variable length and to use pattern matching for testing if an address belongs to a subnetwork domain.

By representing addresses as words over $\{0, 1\}$, or equivalently as terms built from monadic symbols $\mathbf{0}$ and $\mathbf{1}$, and a constant $\#$, we obtain a representation of subnetworks by linear terms built from $\mathbf{0}$ and $\mathbf{1}$. For example, the term

$$t = \mathbf{11000000\,10101000\,00010100}(x)$$

(we omit parentheses to keep readability) denotes the subnetwork $192.168.20.1/24$ whereas

$$\mathbf{11000000\,10101000\,00010100\,00000001}(\#)$$

denotes the IP address $192.168.20.1$. For convenience, we will use in this paper the dot-decimal notation for addresses, the decimal notation for ports and the CIDR notation for subnetworks. When a variable occurs in the corresponding term, it will be indicated in brackets. For example t will be denoted by $192.168.20.1/24[x]$. As we will see later on, this representation allows us to efficiently build tree automata recognizing addresses that belong to given ranges and consequently to efficiently analyse firewall behavior.

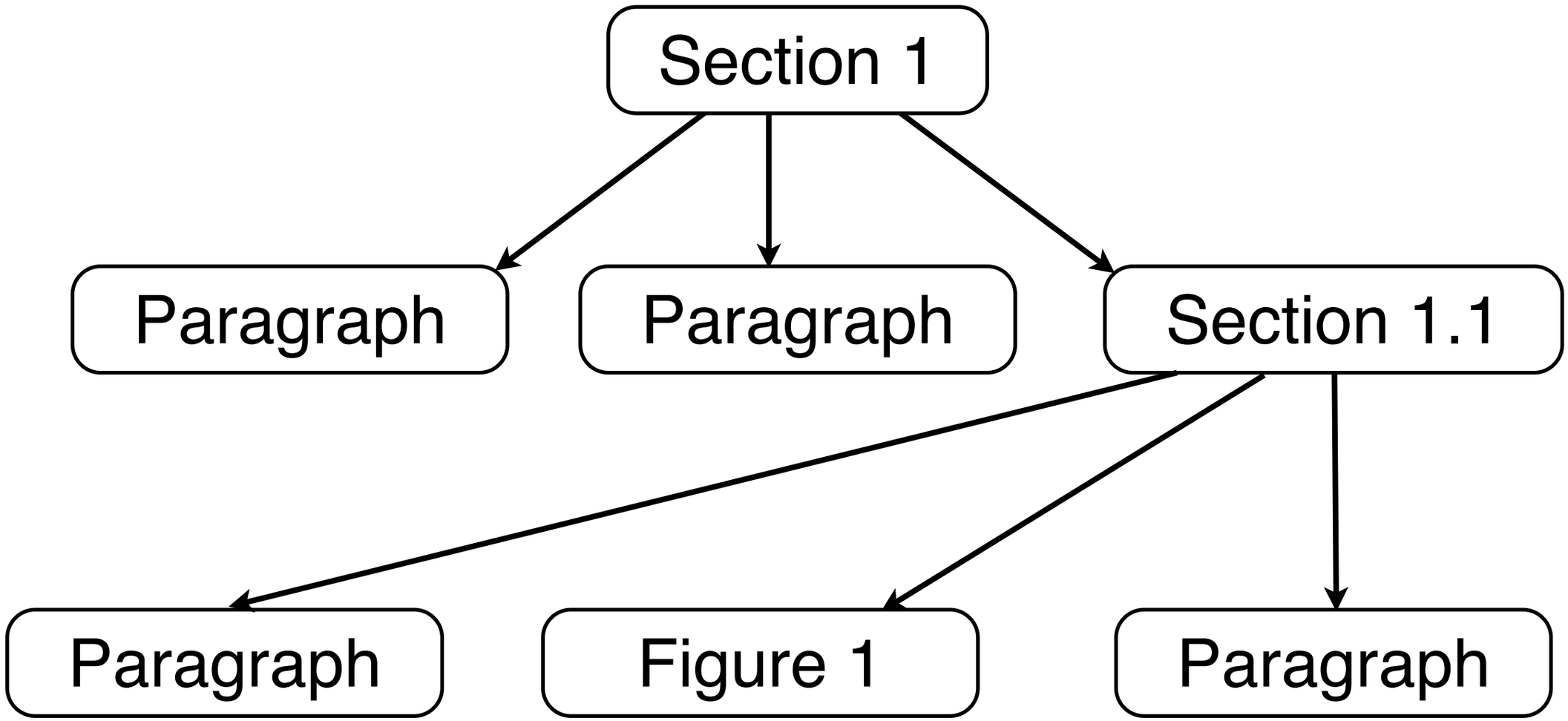
Finally, packets are terms of sort Packet. For example, the term $\mathbf{packet}\left(\begin{array}{l}\mathbf{from}(192.168.1.1, 80),\\ \mathbf{dest}(172.20.3.1, 80)\end{array}\right)$ represents a packet and $\mathbf{packet}\left(\begin{array}{l}\mathbf{from}(192.168.1.1/24[x], y),\\ \mathbf{dest}(172.20.3.1/24[x'], y')\end{array}\right)$ refers to the set of packets whose source address belongs to the subnetwork $192.168.1.1/24$ and whose destination address belongs to $172.20.3.1/24$.

As showed in Section 3.3, explicitly identifying the source and destination addresses (using the symbols **from** and **dest**) allows

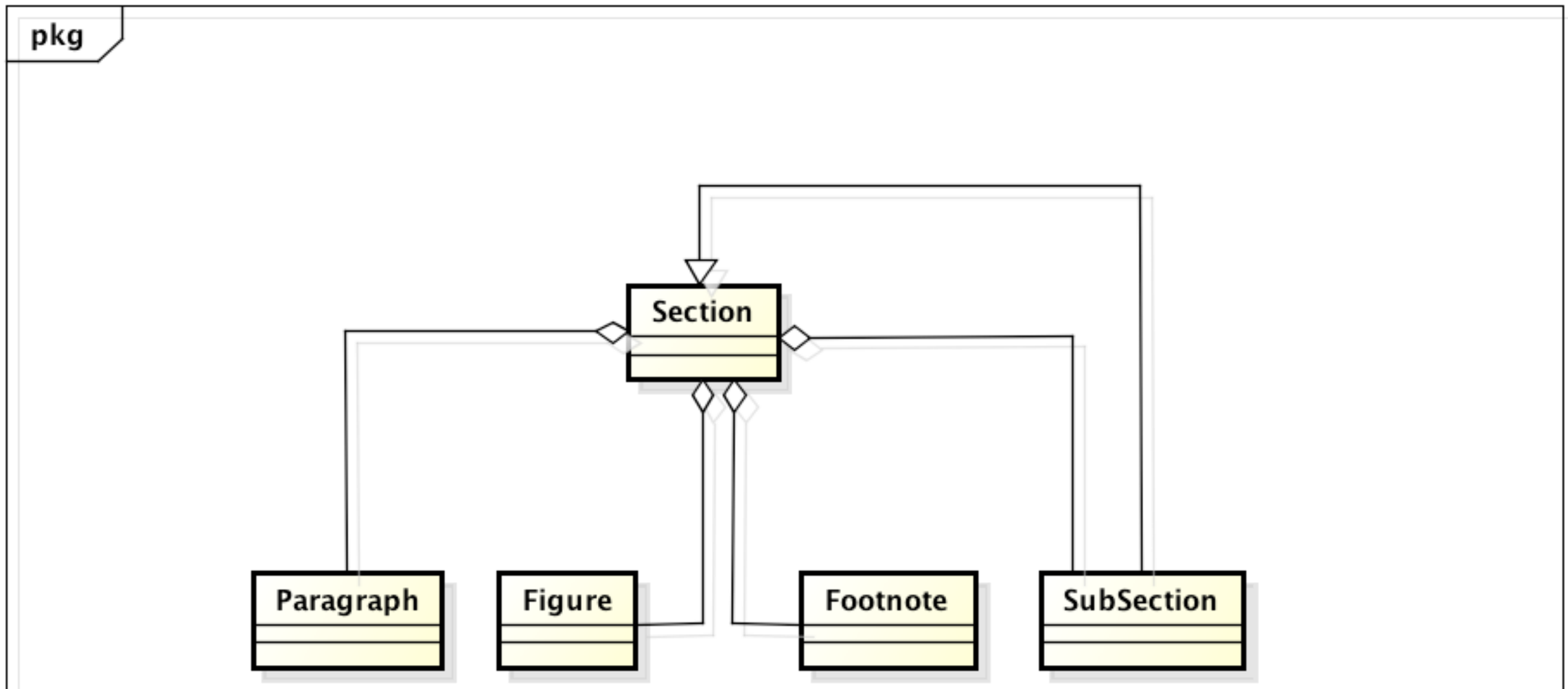
[2] The CIDR notation is a compact specification of an IP range of addresses. It consists of an IP address expressed in a dot-decimal notation and of a decimal number representing the size (in bits) of the common prefix of all the addresses in the range. The notation ip/n denotes the range of addresses whose n first bits (when addresses are expressed as binary numbers) are the same as the n first bits of the address ip. For example, $192.168.20.1/24$ refers to the range $[192.168.20.1, 192.168.20.255]$ and $121.130.1.1/28$ refers to the range $[121.130.1.1, 121.130.1.15]$.

[3] A subnetwork is a logically visible subdivision of a network characterized by an IP range (its domain).

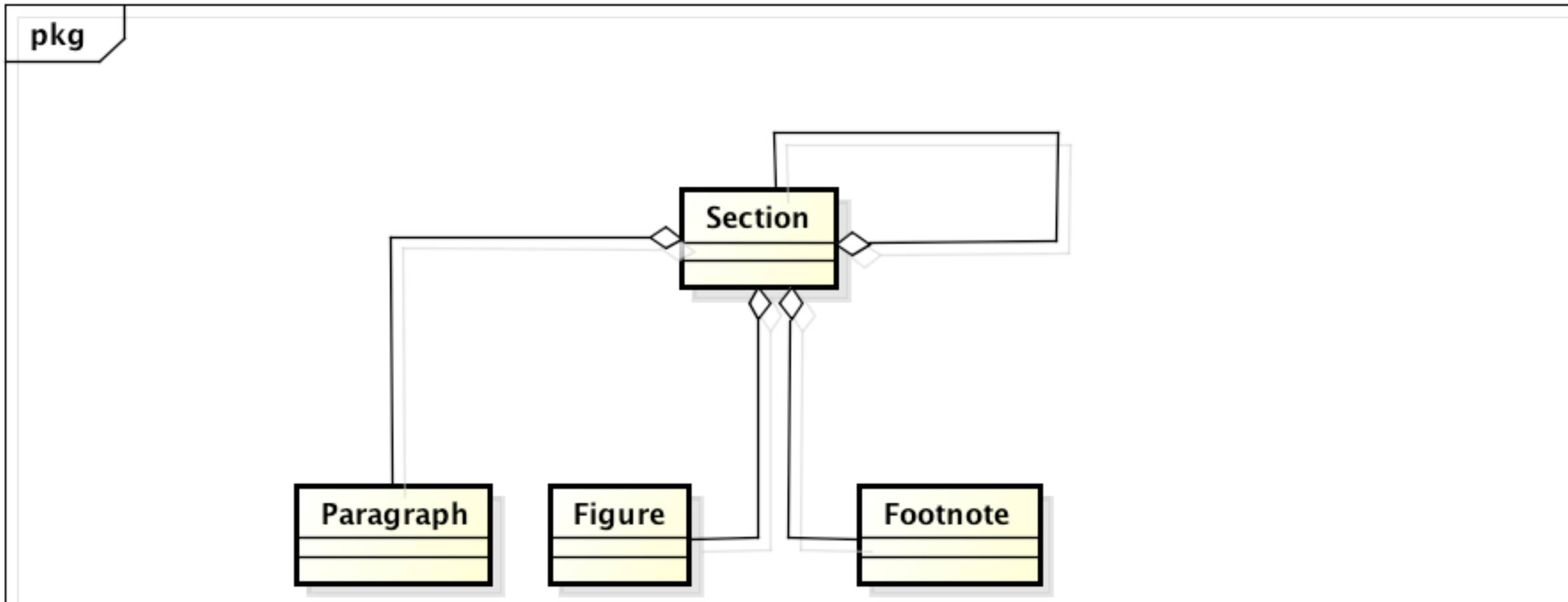
Hiérarchie

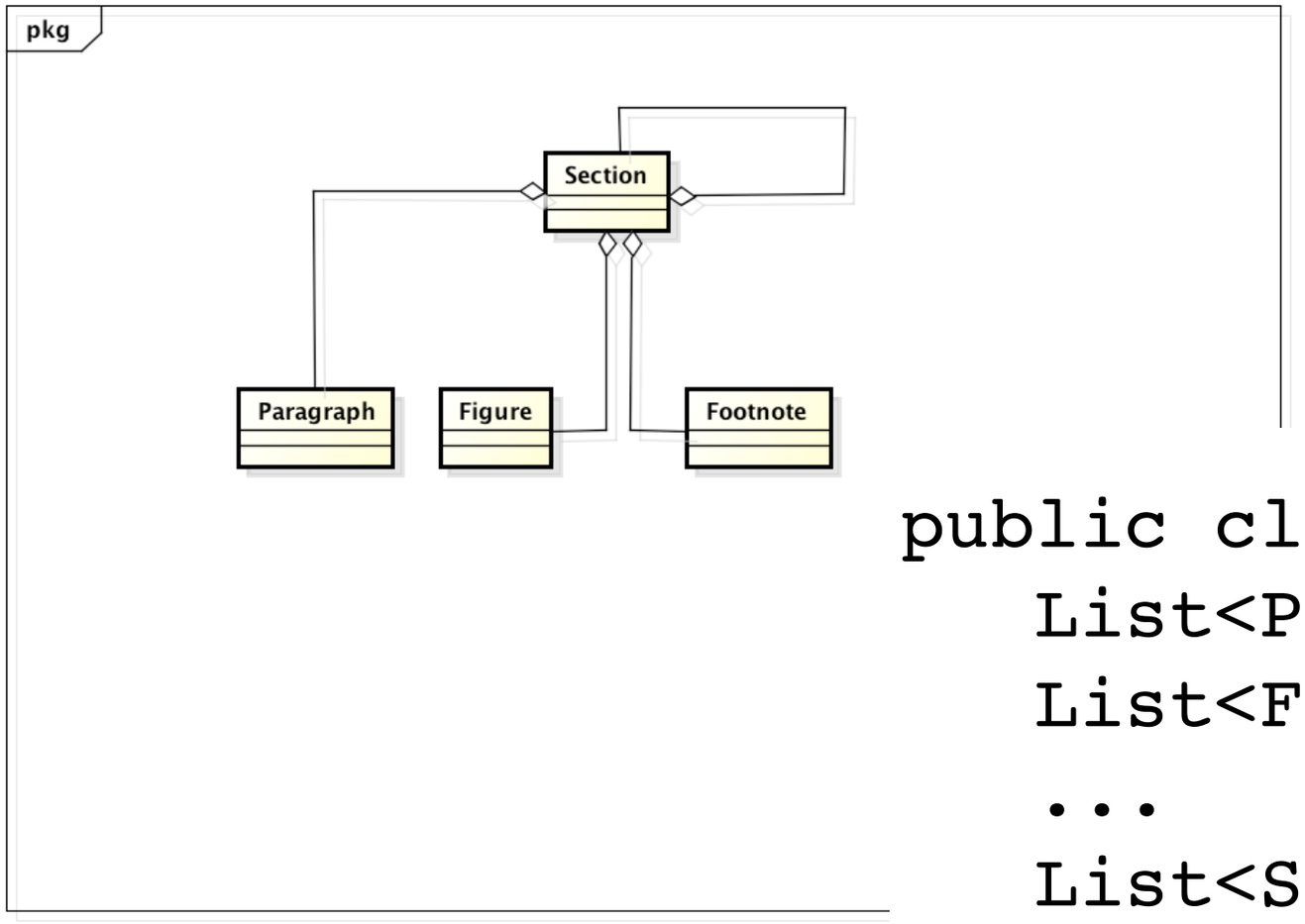

Section 1
Paragraph
Paragraph
Section 1.1
Paragraph
Figure 1
Paragraph

Solution 1

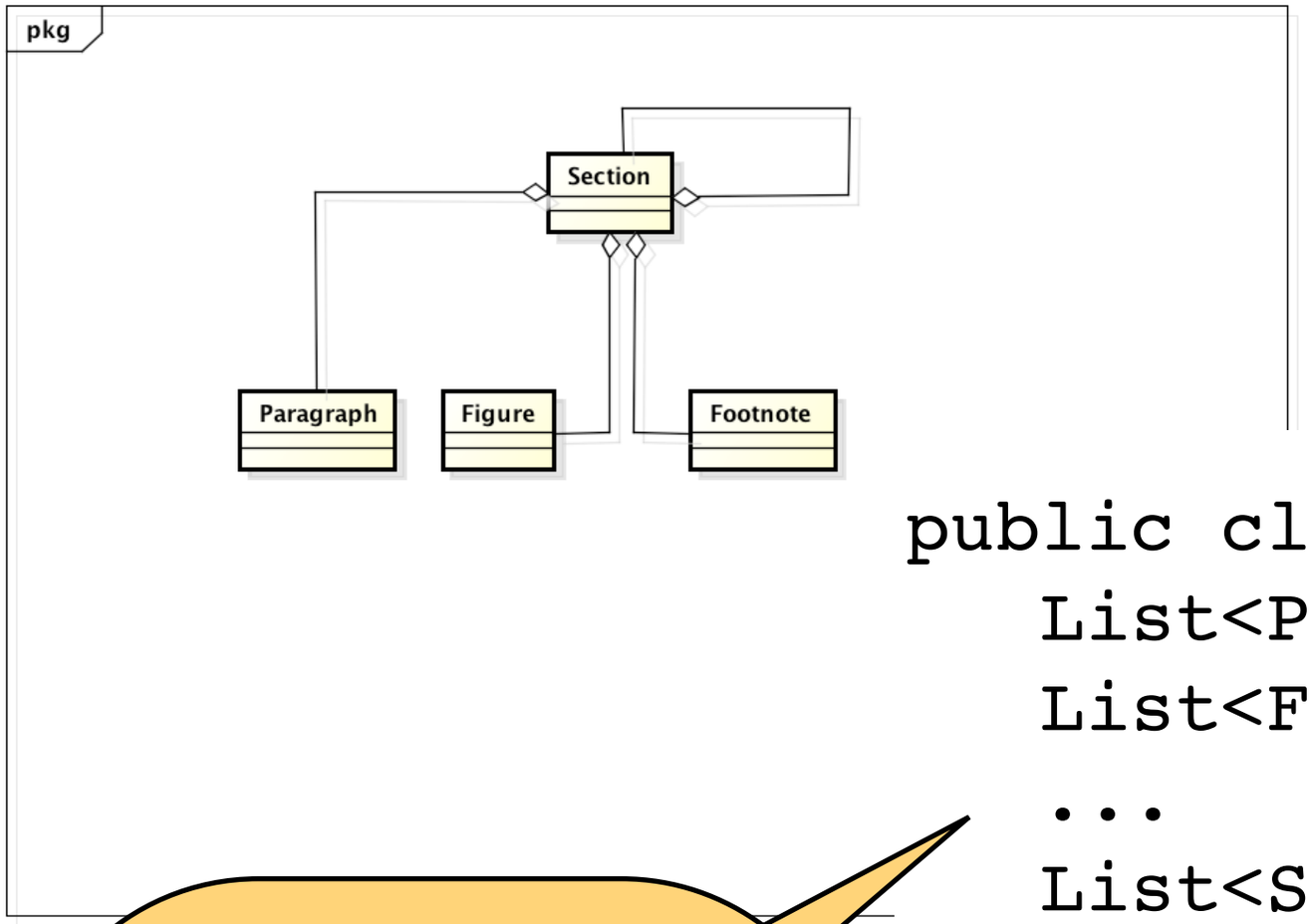


Solution 1.1





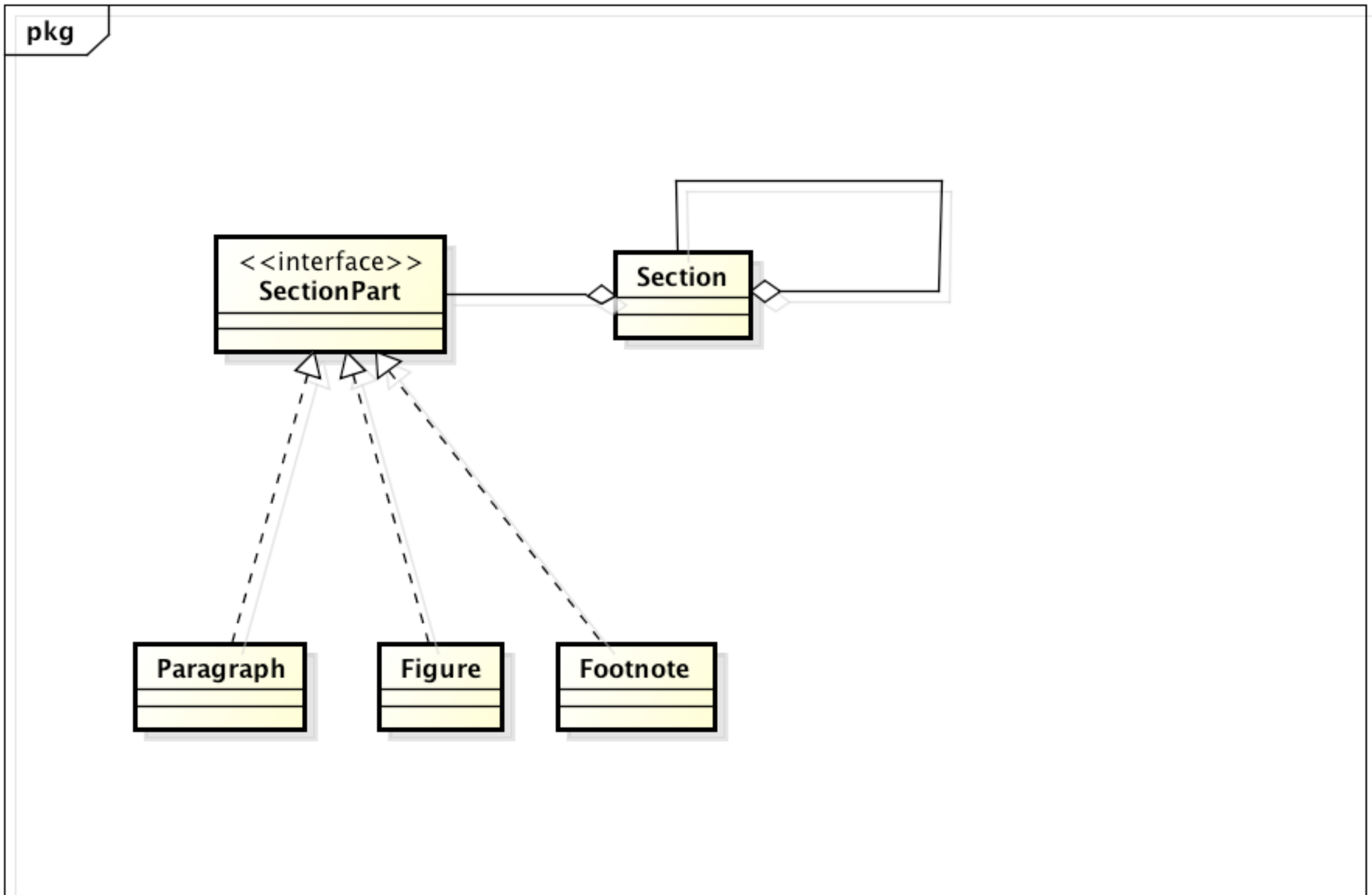

```
public class Book {
    List<Paragraph> paras;
    List<Figure> figures;
    ...
    List<Section> sections;
  public void setColor(Color c) {
    for(Paragraph p: paras)
          p.setBackgroung(c);
    for(Figure f: figures)
          f.setForegroung(c);
    for(Section s: sections)
          s.setColorSection(c);
    ...
  }
```

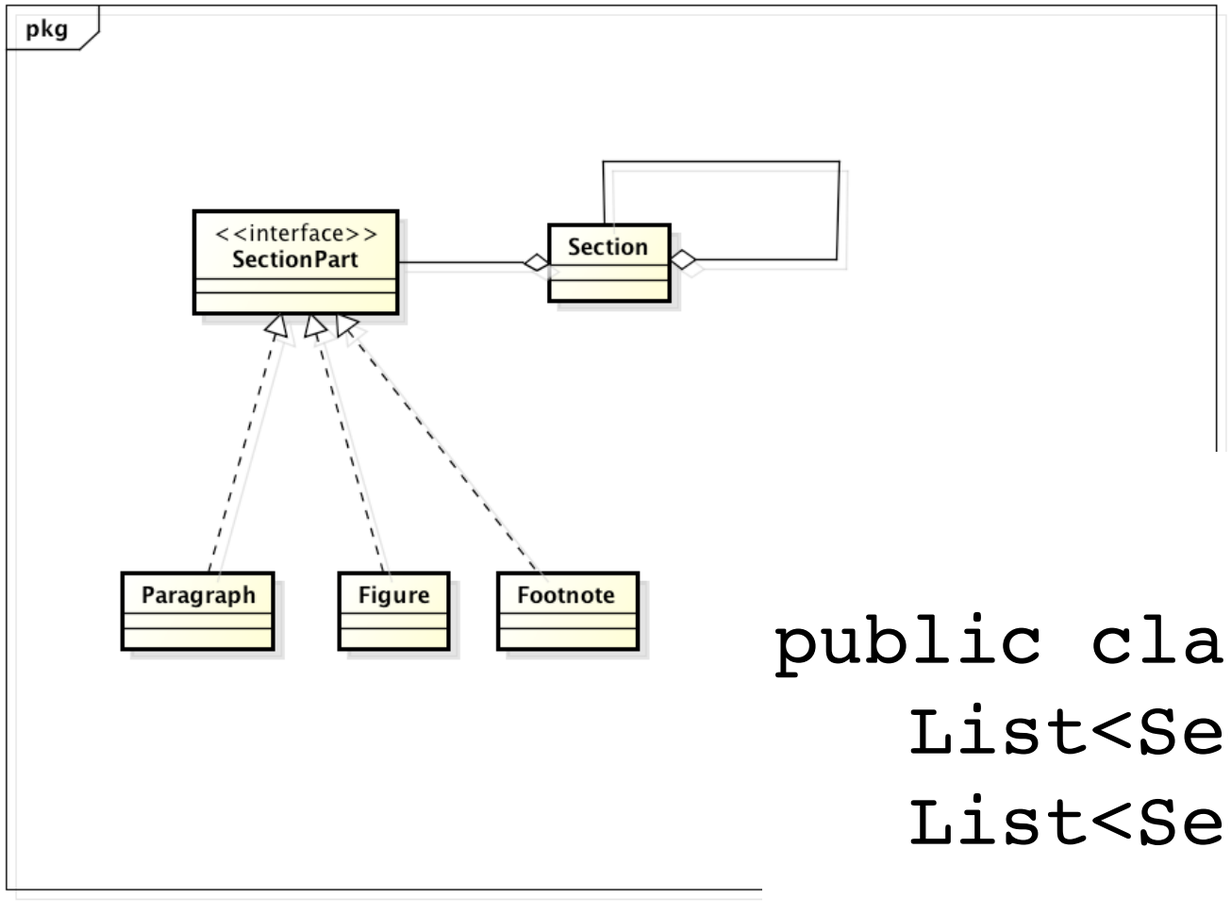


et si on veut ajouter des Tables dans les sections ?

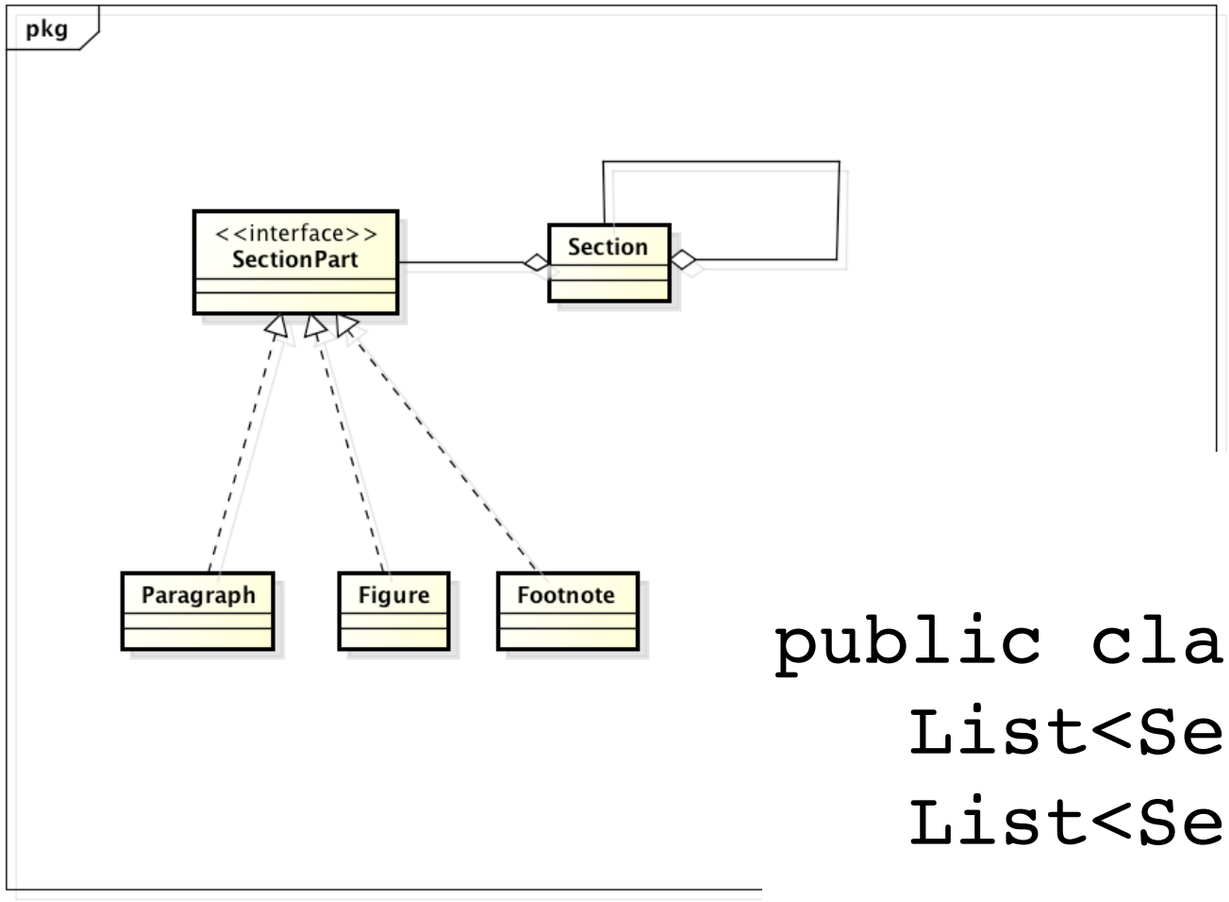
```
public class Book {
   List<Paragraph> paras;
   List<Figure> figures;
   ...
   List<Section> sections;
   public void setColor(Color c) {
     for(Paragraph p: paras)
           p.setBackgroung(c);
     for(Figure f: figures)
           f.setForegroung(c);
     for(Section s: sections)
           s.setColorSection(c);
     ...
}
```

Solution 2


pkg
<<interface>>
SectionPart
Section
Paragraph
Figure
Footnote



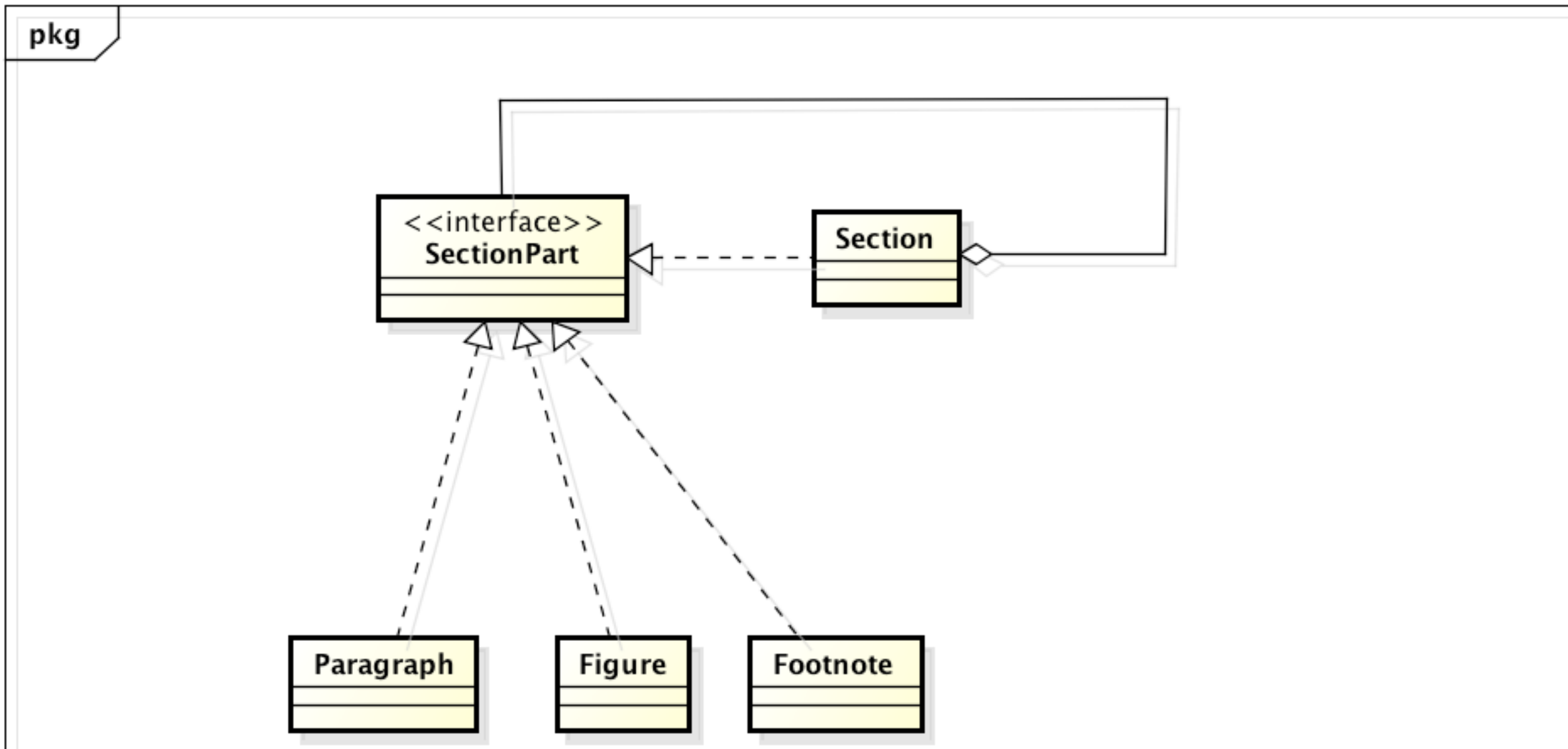

```
public class Book {
    List<SectionPart> parts;
    List<Section> sections;

    public void setColor(Color c) {
      for(SectionPart sp: parts)
            p.setColor(c);
      for(Section s: sections)
            s.setColor(c);

      ...
    }
```

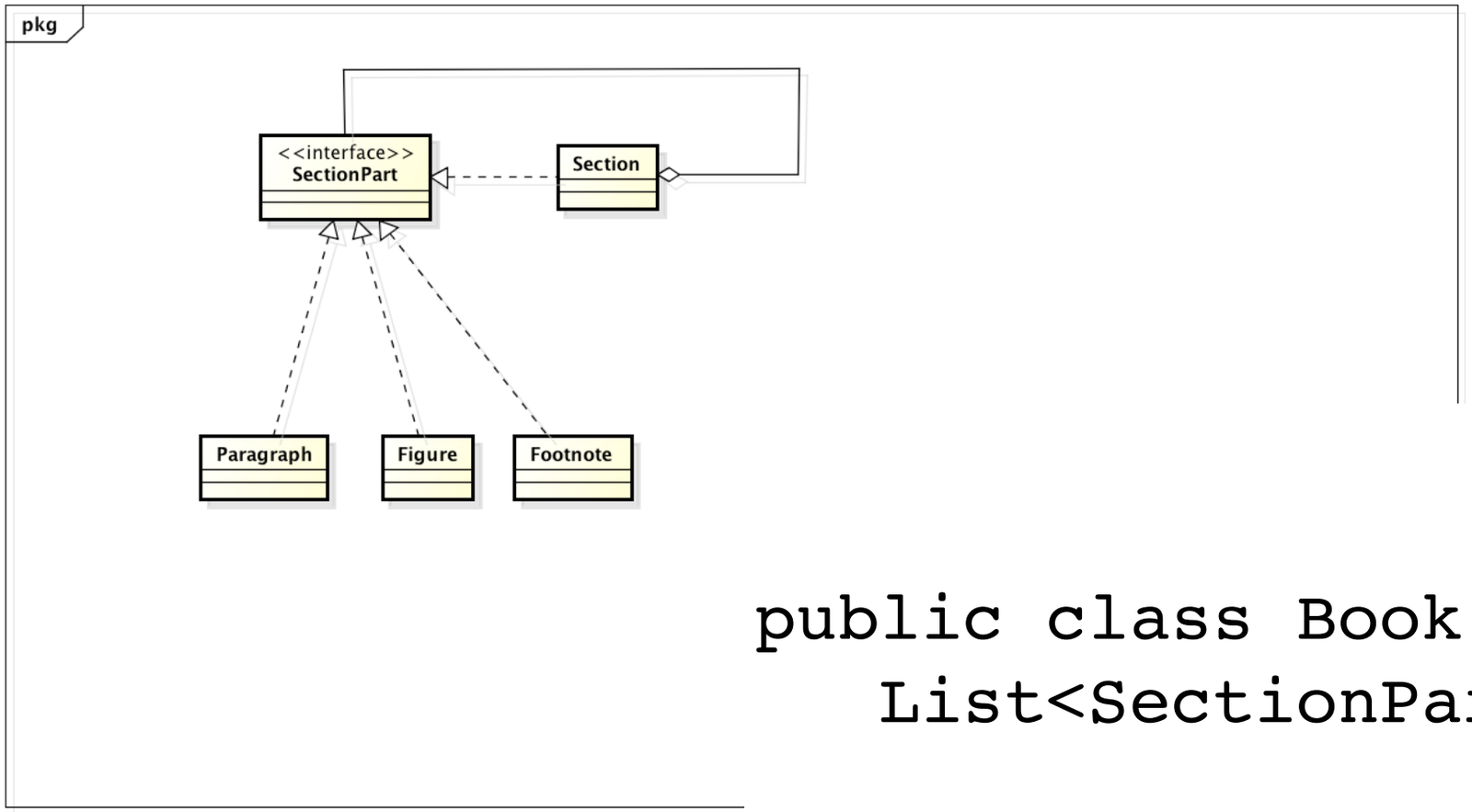



```
public class Book {
    List<SectionPart> parts;
    List<Section> sections;

    public void setColor(Color c) {
      for(SectionPart sp: parts)
        p.setColor(c);
      for(Section s: sections)
        s.setColor(c);

      ...
    }
```

distinction entre sections et leurs composants

Solution 3


pkg
<<interface>>
SectionPart
Section
Paragraph
Figure
Footnote

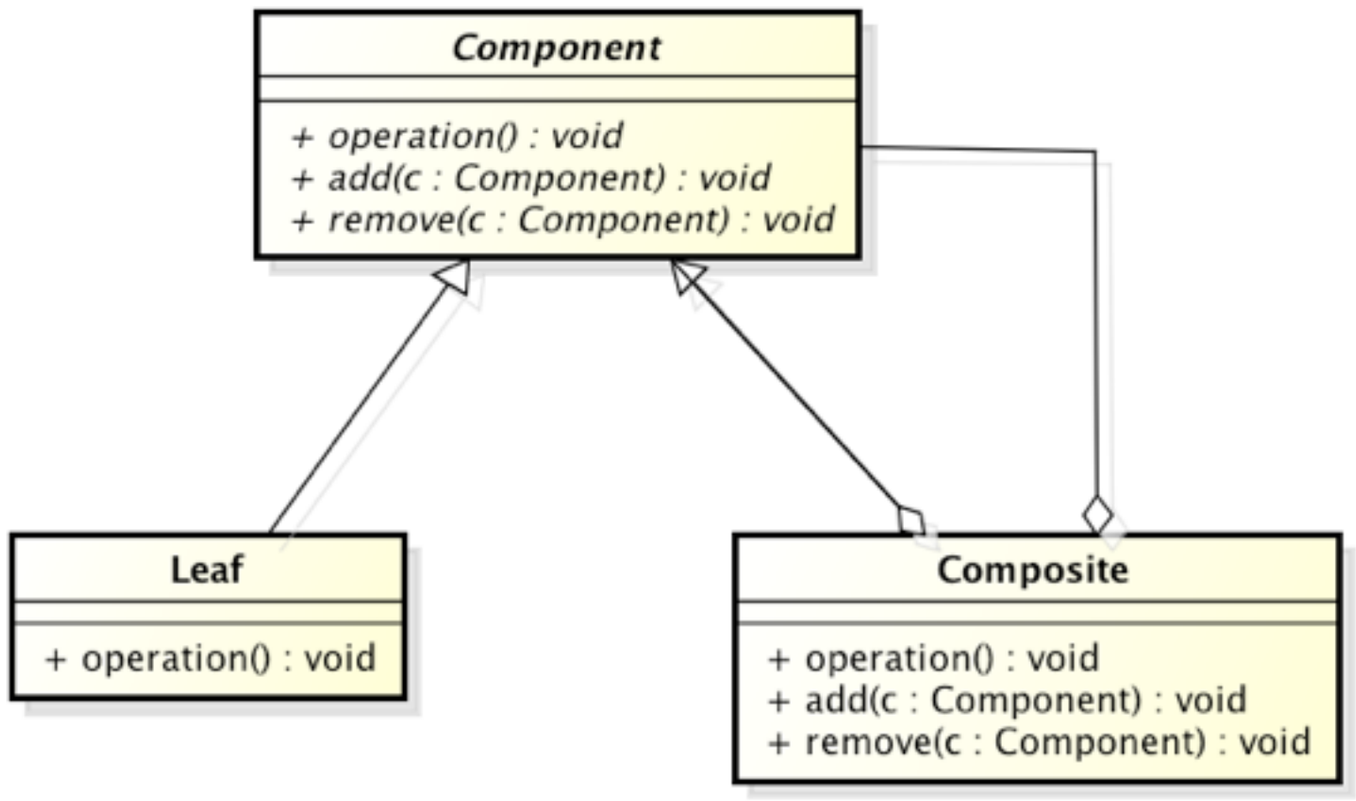



```
public class Book {
   List<SectionPart> parts;
   public void setColor(Color c) {
     for(SectionPart sp: parts)
          p.setColor(c);

     ...
   }

}
```

Composite

pkg

Structure



Structure

Component

- déclare l'interface commune à tous les objets
- **implémente le comportement par défaut**
- **déclare l'interface pour gérer les composants fils**
- définit l'interface pour accéder au composant parent (optionnel)

Structure

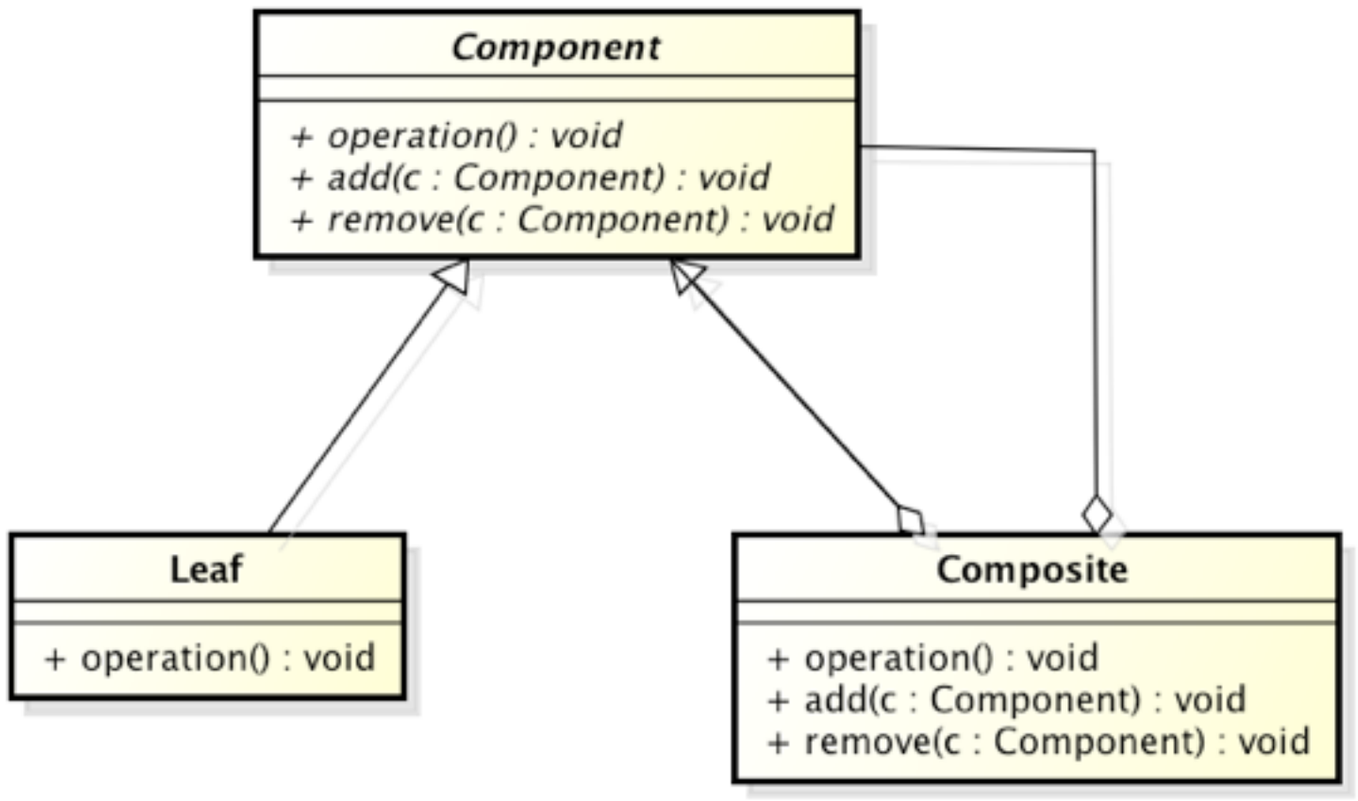
Leaf

- représente un composant élémentaire
- implantation du comportement élémentaire

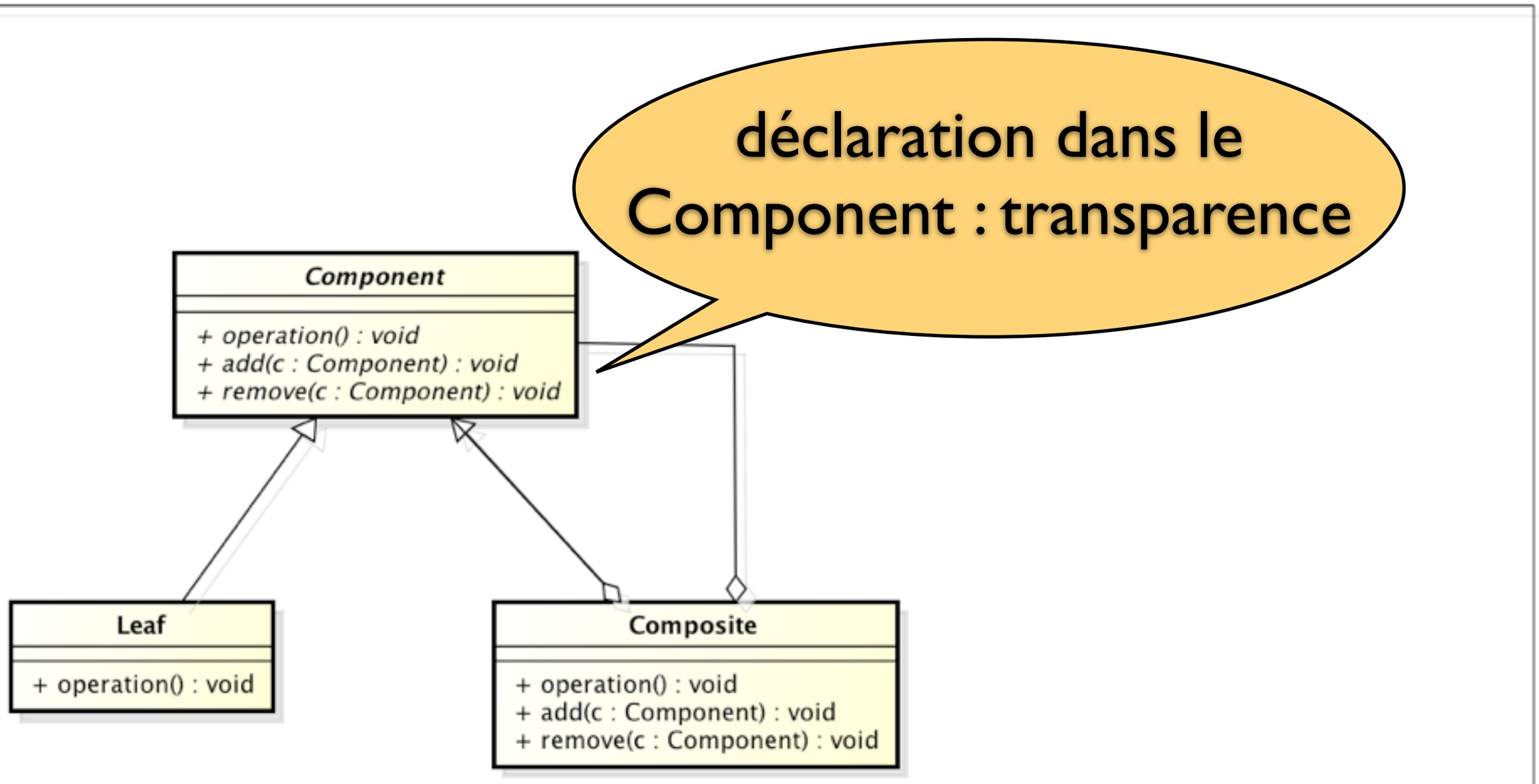
Composite

- définit le comportement des composants ayant des fils
- implante les opérations nécessaires à la gestion des fils

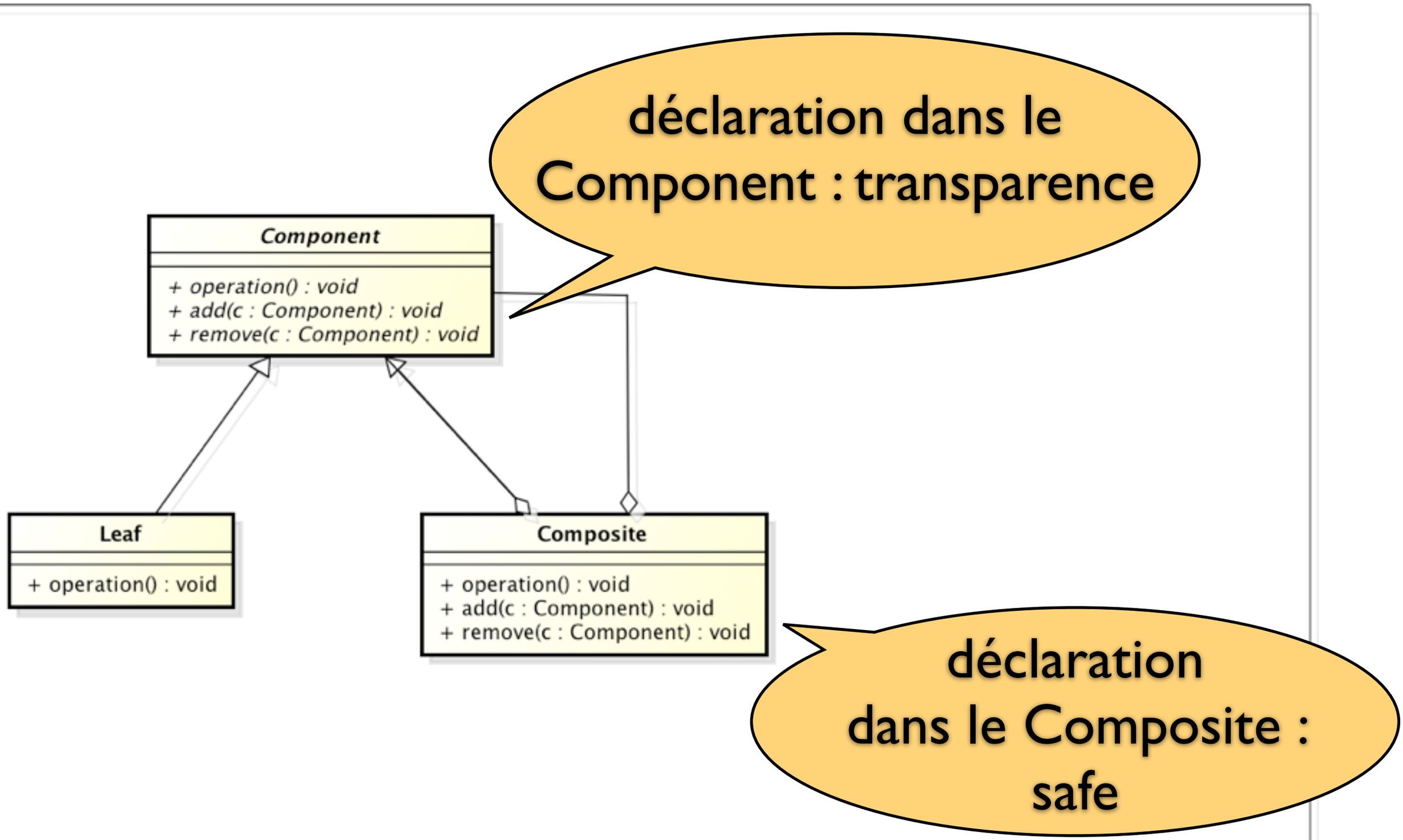
Gestion fils



Gestion fils



Gestion fils

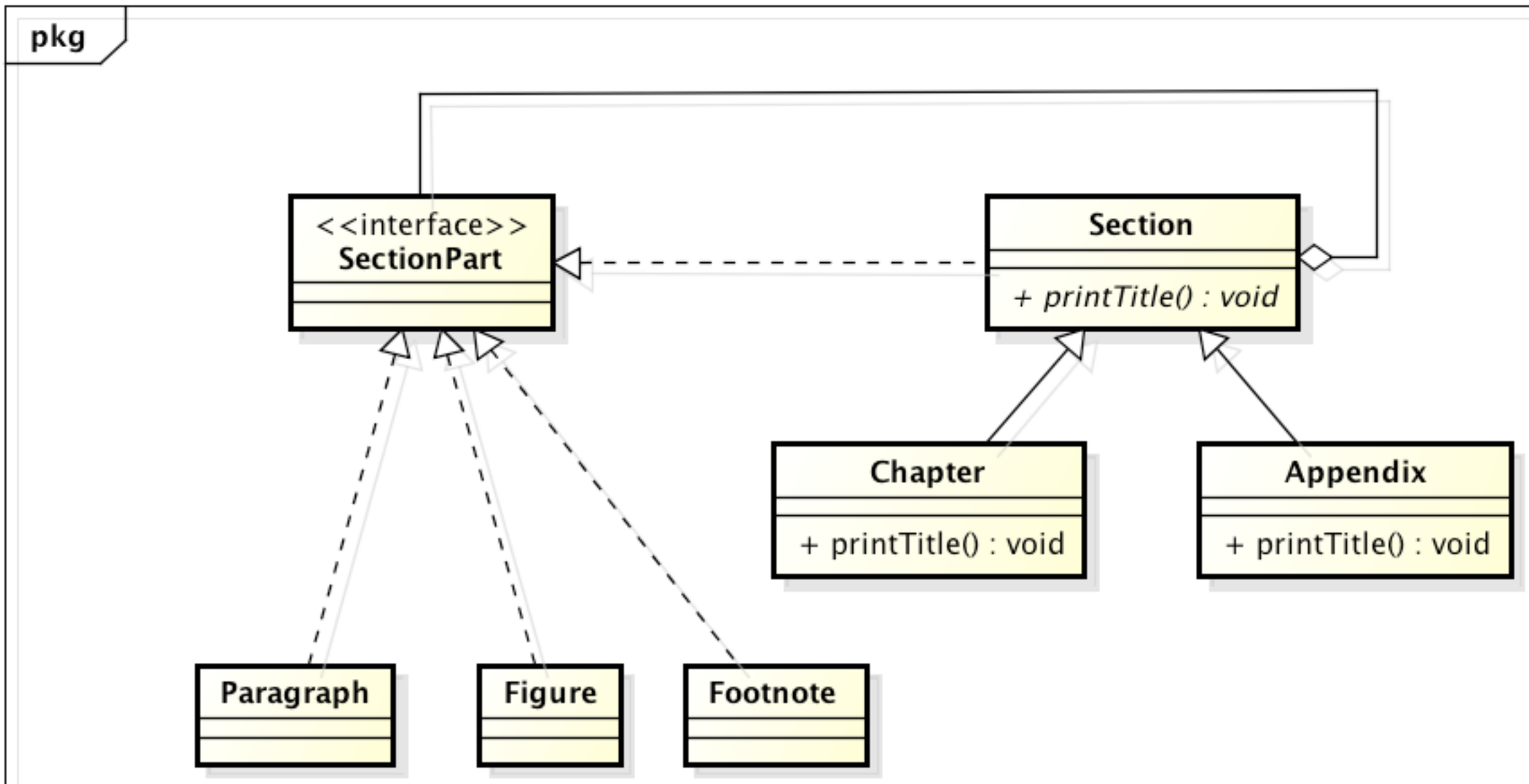


Composite

Indications d'utilisation

- représentation de structures récursives
- traitement uniforme de tous les objets du composite, qu'ils soient terminaux ou non

Plusieurs types de composites



Composite

Implémentation

- garder une référence vers les parents ?
- comment gérer les structures non-arborescentes ?
- l'ordre des enfants est important ?

Composite

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Classification GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Problème

- Il existe un nombre potentiellement variable de handlers pour les requêtes des clients et on doit gérer efficacement les requêtes sans expliciter les relations entre handlers et requêtes.

Présentation garage

`getDescription()` → Nice F1 car

`getDescription()` → Mustang

`getDescription()` → Volkswagen

`getDescription()` → Gas truck

Présentation garage

`getDescription()` → Nice F1 car

description voiture

`getDescription()` → Mustang

description modèle

`getDescription()` → Volkswagen

description marque

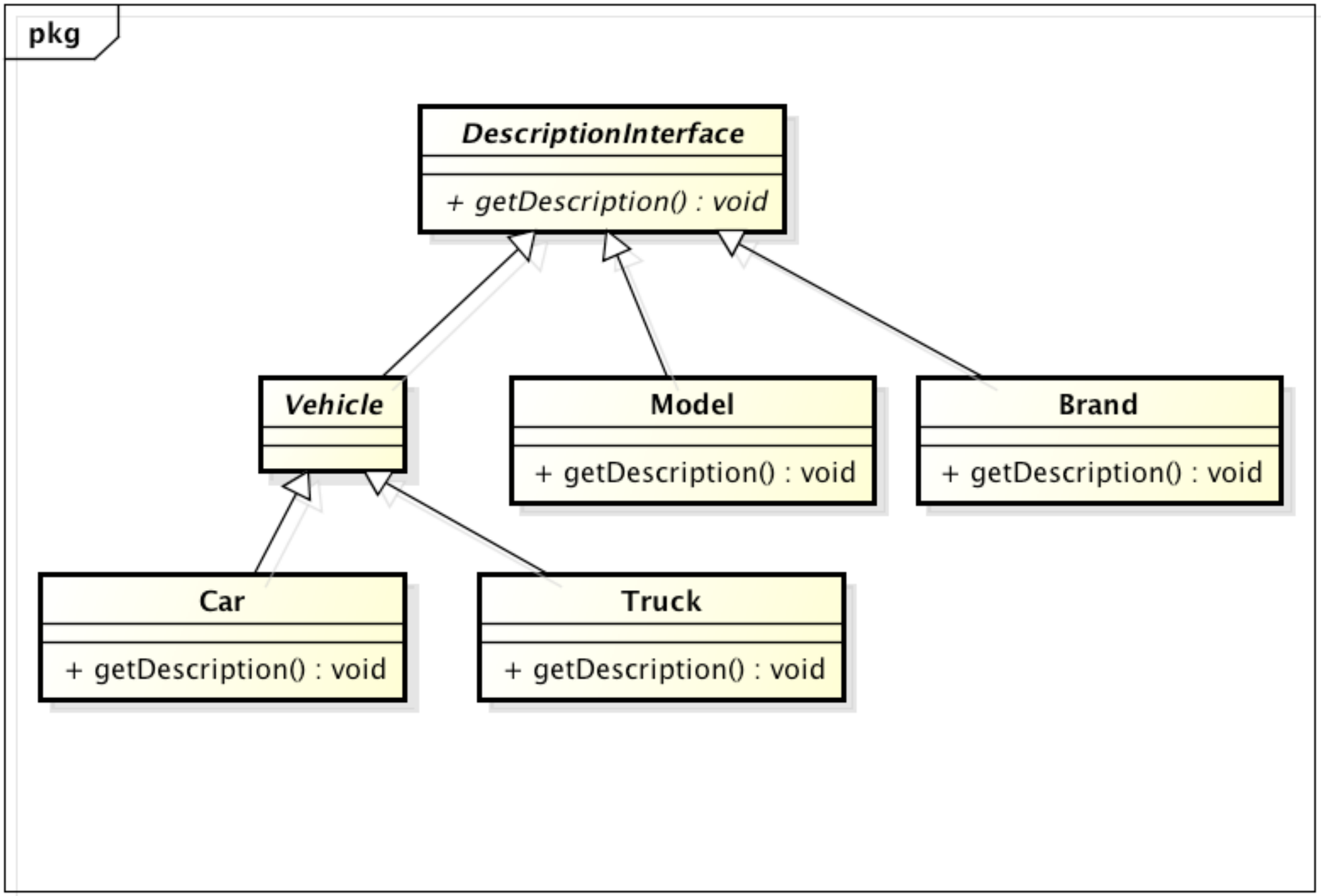
`getDescription()` → Gas truck

description type

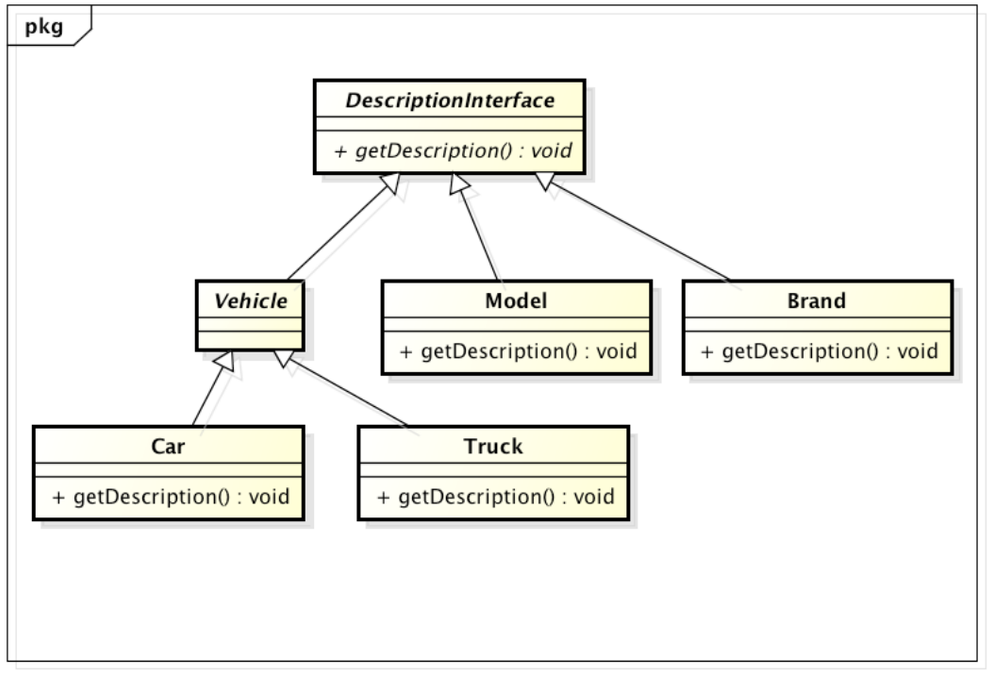
Solution 1

```
class Garage{
  public void presentation(){
    for (Vehicle v : listVehicles){
      if(v instanceof Car){
        if(v.getDescription() != null){
            System.out.println(v.getDescription());
        } else if (v.getModel().getDescription() != null) {
            S.o.p(v.getModel().getDescription());
        } else if (v.getModel().getBrand().getDescription() != null){
            S.o.p(v.getModel().getBrand().getDescription());
        }
      }
      if(v instanceof Truck){
        ...
      }
      ...
  }
}
```

Solution 2


pkg
DescriptionInterface
+ getDescription() : void
Vehicle
Model
+ getDescription() : void
Brand
+ getDescription() : void
Car
+ getDescription() : void
Truck
+ getDescription() : void

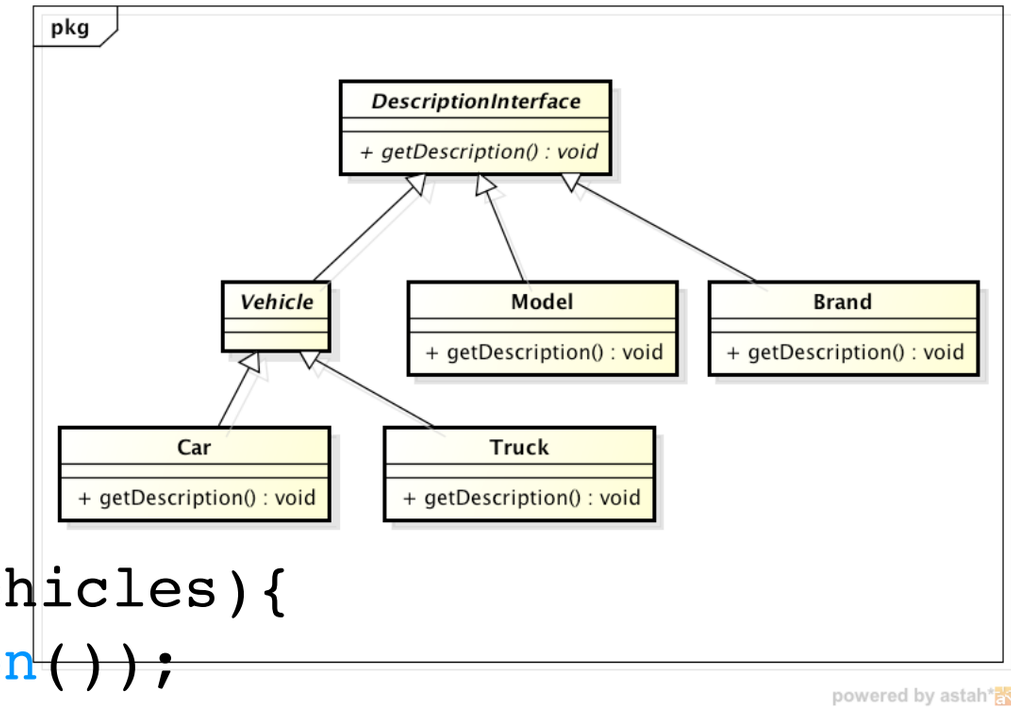
Solution 2



powered by astah*

```
class Car implements ... DescriptionInterface {
  String description;
  Model model;
  public String getDescription(){
        if(description != null){
          return description;
        } else if (model != null &&
                    model.getDescription() != null) {
          return model.getDescription();
        } else
          ...
  }
}
```

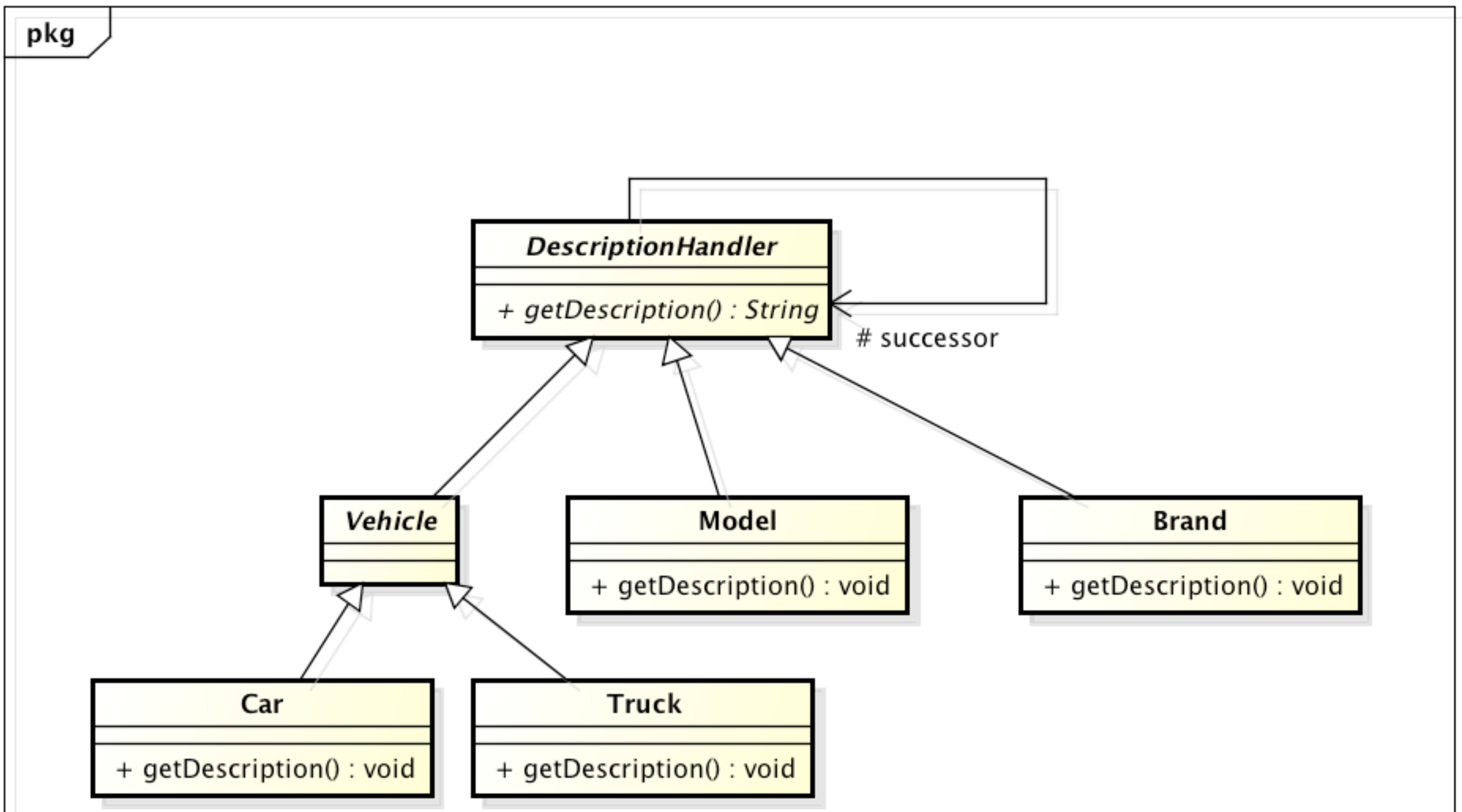
Solution 2




```
class Garage{
  public void presentation(){
    for (DescriptionInterface v : listVehicles){
      System.out.println(v.getDescription());
    }
  }
}
```

```
class Car implements ... DescriptionInterface {
  String description;
  Model model;
  public String getDescription(){
        if(description != null){
          return description;
        } else if (model != null &&
                   model.getDescription() != null) {
          return model.getDescription();
        } else
          ...
  }
}
```

Solution 3


pkg
DescriptionHandler
+ getDescription() : String
# successor
Vehicle
Model
+ getDescription() : void
Brand
+ getDescription() : void
Car
+ getDescription() : void
Truck
+ getDescription() : void

Solution 3

```
class Car extends Vehicle {
  String description;
  Model model;
  public Car(String desc, Model m){
     description = desc;
     model = m;
      successor = model;
  }
  public String getDescription(){
    if(description != null){
        return description;
    }
    return successor.getDescription();
  }
}
class Vehicle implements PresentationHandler {
   ...
}
```

Solution 3

```
class Model implements PresentationHandler
{
  String description;
  Brand brand;
  public Model(String desc, Brand b){
     description = desc;
     brand = b;
     successor = brand;
  }
  public String getDescription(){
    if(description != null){
        return description;
    }
    return successor.getDescription();
  }
```

```
class Car extends Vehicle {
  String description;
  Model model;
  public Car(String desc, Model m){
     description = desc;
     model = m;
     successor = model;
  }
  public String getDescription(){
    if(description != null){
        return description;
    }
    return successor.getDescription();
  }
}
class Vehicle implements PresentationHandler {
   ...
}
```

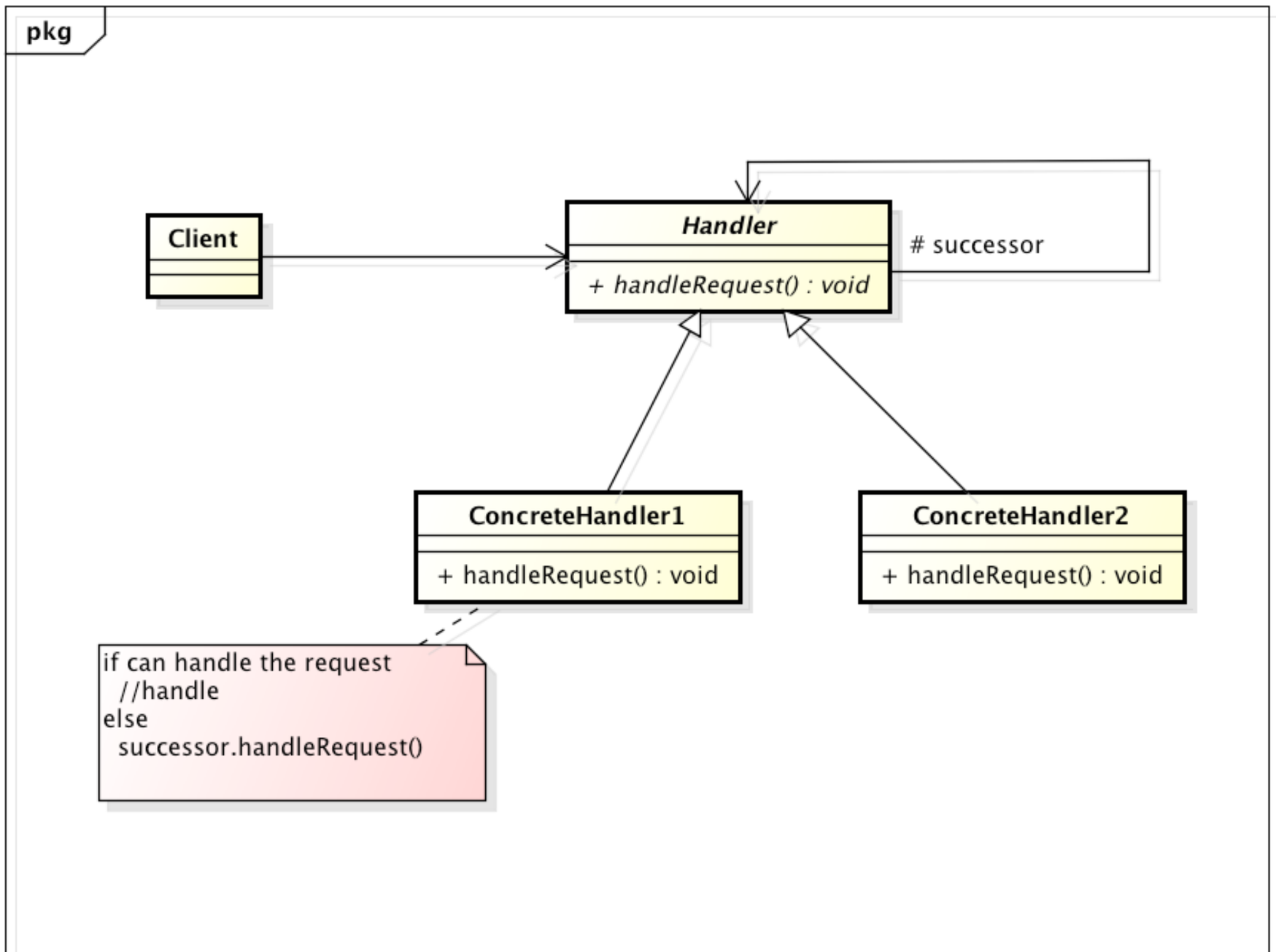
Solution 3

```
class Model implements PresentationHandler
{
  String description;
  Brand brand;
  public Model(String desc, Brand b){
     description = desc;
     brand = b;
     successor = brand;
  }
  public String getDescription(){
    if(description != null){
        return description;
    }
    return successor.getDescription();
  }
```

```
class Car extends Vehicle {
  String description;
  Model model;
  public Car(String desc, Model m){
     description = desc;
     model = m;
     successor = model;
  }
  public String getDescription(){
    if(description != null){
        return description;
    }
    return successor.getDescription();
  }
}
class Vehicle implements PresentationHandler {
   ...
}
```

```
class Truck implements PresentationHandler {
  String nom;
  Model model;
  public Truck(String n, Model m){
     nom = n;
     model = m;
     successor = model;
  }
  public String getDescription(){
    return successor.getDescription();
  }
}
```

Chain of responsability



Chain of responsability

Intention

- éviter le couplage entre l'émetteur d'une requête et son récepteur en offrant la possibilité de traiter la requête à plus d'un objet. Enchaîner les objets receveurs et transmettre la requête dans la chaîne jusqu'à ce qu'elle soit traitée par un objet.
- lancer les requêtes dans un pipeline de traitement qui contient plusieurs handlers possible.

Chain of responsability

Indications d'utilisation

- quand plus d'un objet peut traiter une requête et l'objet qui va la traiter n'est pas connu en avance
- quand les requêtes suivent un modèle « handle or forward » : certaines requêtes sont traitées sur place et d'autres doivent être forwardées à un autre objet pour traitement

Chain of responsability

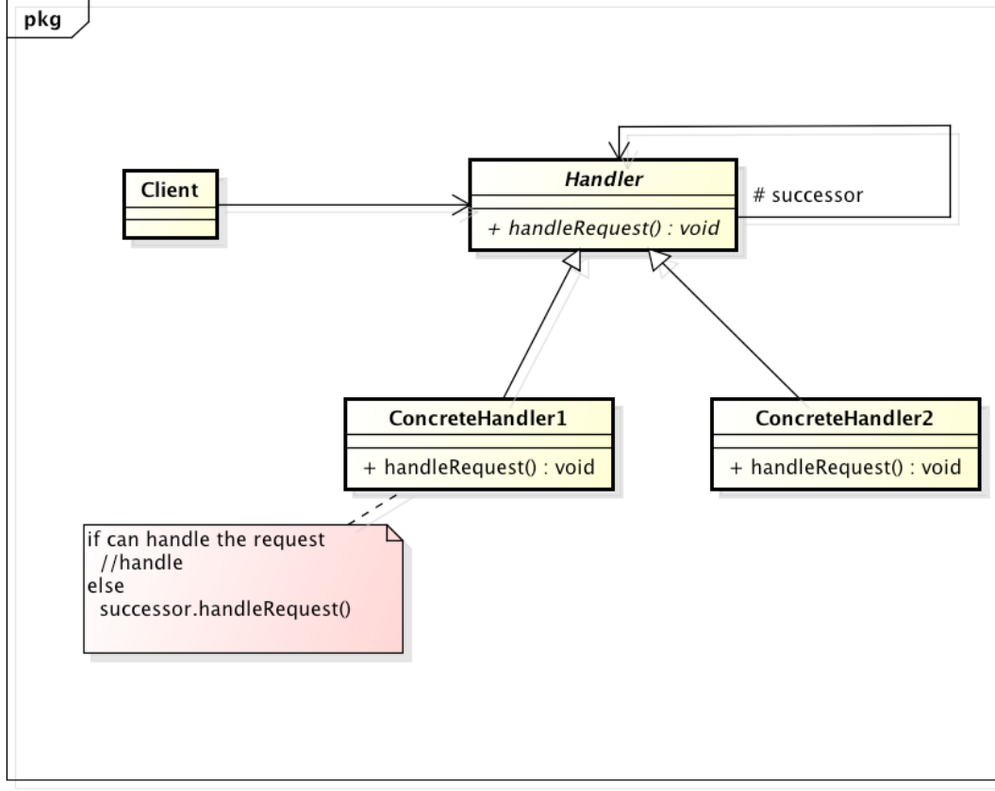
Conséquences

- couplage réduit entre émetteur (de la requête) et récepteur (ils ne se connaissent pas)
- la chaine de traitement peut être modifiée dynamiquement
- *réception non garantie - la requête peut arriver a la fin de la chaine sans être traitée*

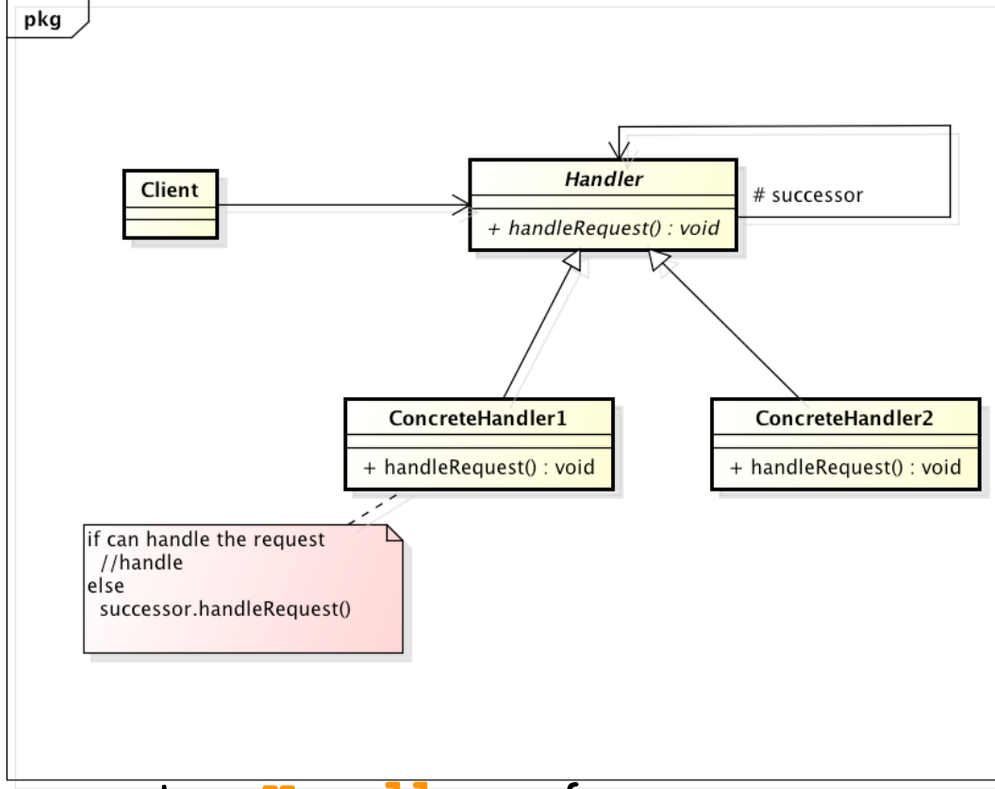
Chain of responsability

Implémentation

- la chaîne de successeurs
  - définir un nouvel attribut (dans le `Handler` ou dans le `ConcreteHandler`)
  - utiliser un attribut existant
- représenter les requêtes
  - invocation à des opérations hard-coded
  - méthodes paramétrées (pas type-safe) + type `Request`


pkg
Client
Handler
+ handleRequest() : void
# successor
ConcreteHandler1
+ handleRequest() : void
ConcreteHandler2
+ handleRequest() : void
if can handle the request
//handle
else
successor.handleRequest()

```java
public interface Handler{
   public void setNext(Handler handler);
   public void handleRequest();
}
```




```java
public class ConcreteHandler implements Handler {
   private Handler next;
   public void setNext(Handler handler) {
      this.next = handler;
   }

   public void handleRequest() {
      if (...) { // can handle
         // handle
      } else {
         this.next.handleRequest();
      }
   }
}
```

```
public interface Handler{
  public void setNext(Handler handler);
  public void handleRequest();
}
        public abstract class AbstractHandler implements Handler {
          private Handler next;
          public void setNext(Handler handler) {
             this.next = handler;
          }
          public void handleRequest() {
            if (this.canHandle()) {
              this.doHandle();
            } else {
              this.next.handleRequest();
            }
          }
          protected abstract boolean canHandle();
          protected abstract void doHandle();
        }

public class ConcreteHandler extends AbstractHandler {
  public boolean canHandle(){ ... }
  public void doHandle() { ... }
}
```

```
public interface Handler{
   public void setNext(Handler handler);
   public void handleRequestOne();
   public void handleRequestTwo();
   ...
}
```

requêtes multiples

```
public interface Handler{
   public void setNext(Handler handler);
   public void handleRequest(Request request);
}
```

requêtes multiples

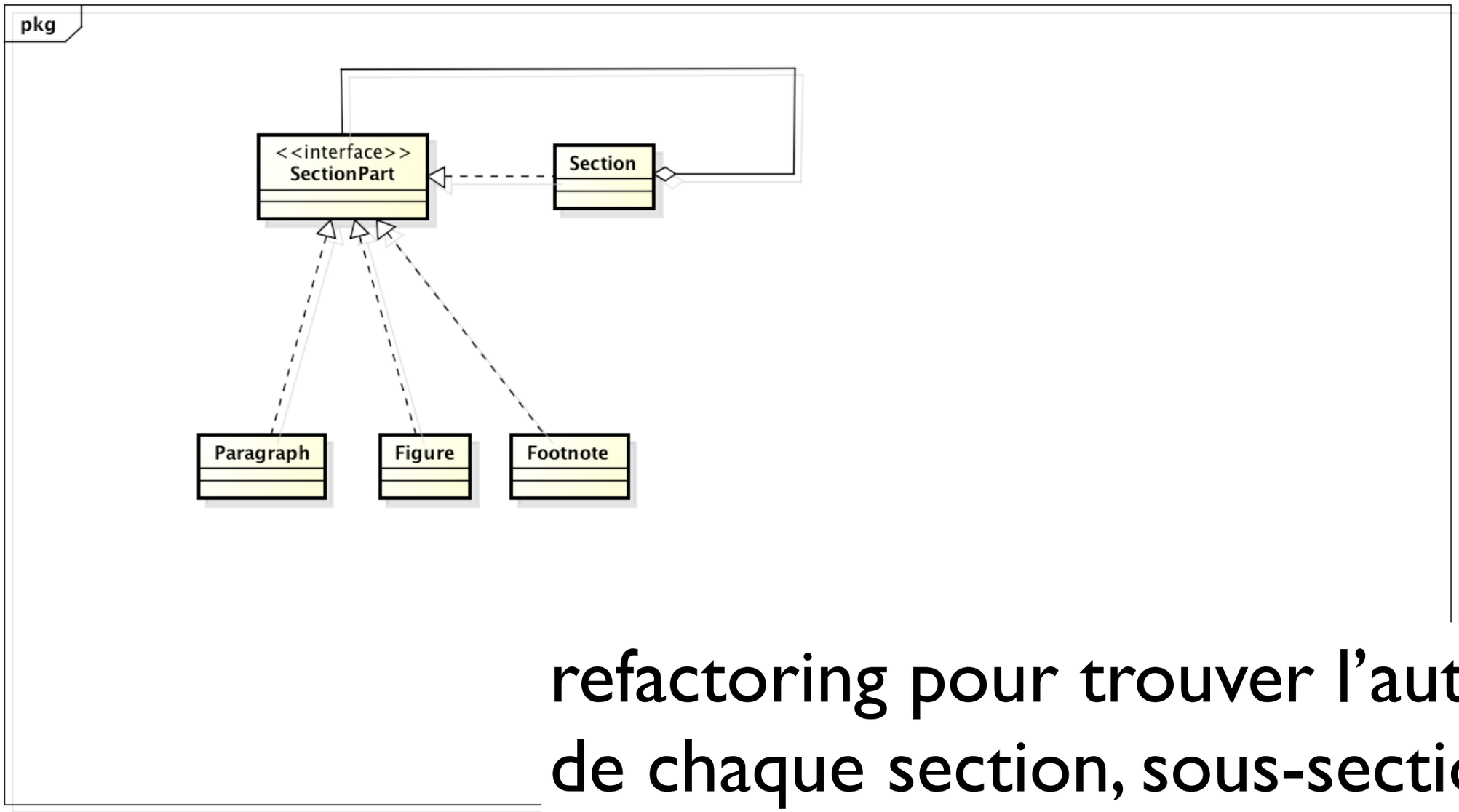
requêtes multiples

```
public interface Handler{
  public void setNext(Handler handler);
  public void handleRequest(Request request);
}
```

enchainement

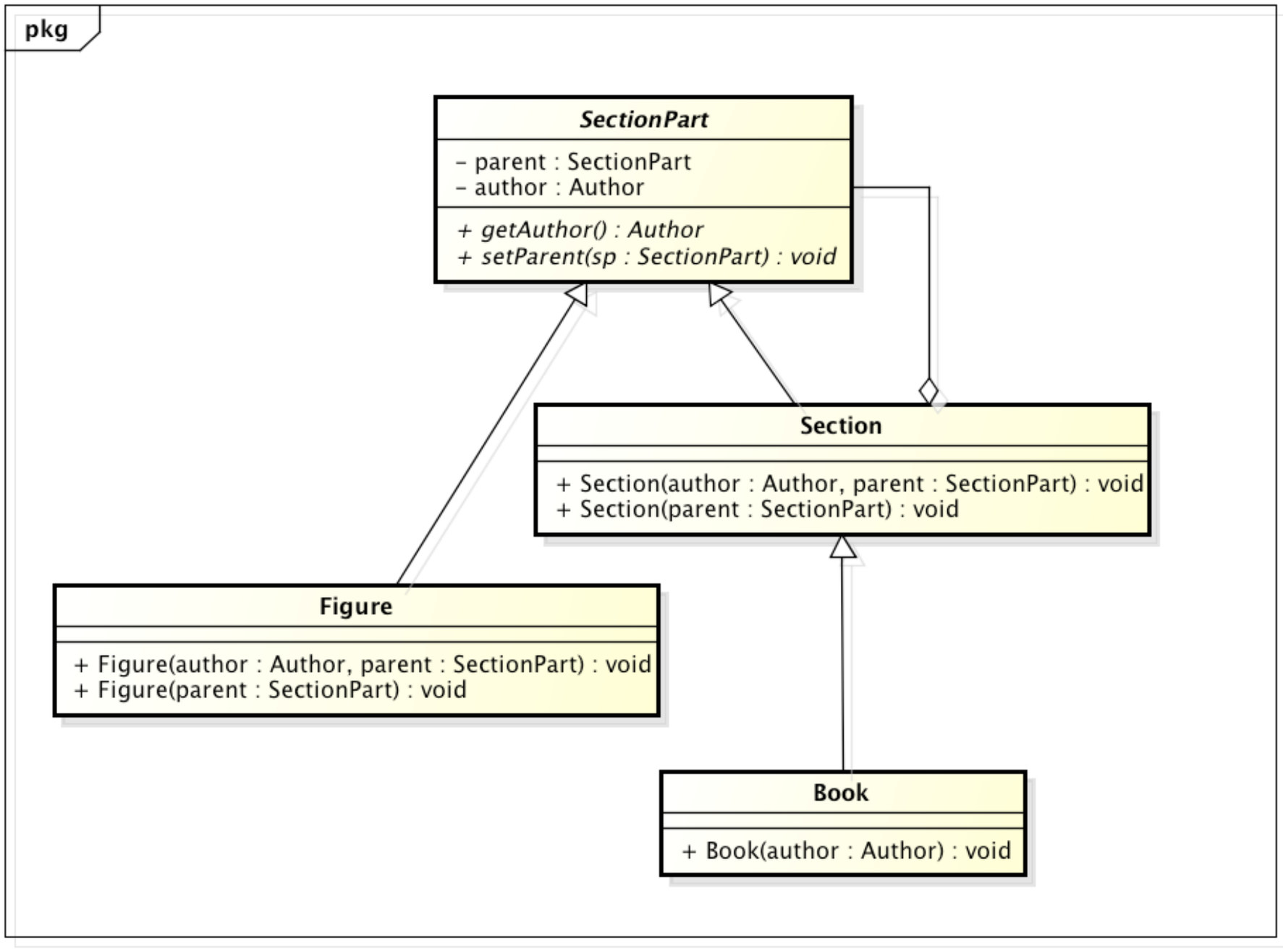
```
public class ConcreteHandler implements Handler {
  private Handler next;
  public void setNext(Handler handler) {
    if ( this.next == null ) {
      this.next = handler;
    } else {
      this.next.setNext(handler);
    }
  }

  public void handleRequest(Request request) {
    if (request...) {
      ...
    } else if (...){
      ...
    }
  }
}
```

Exercice



refactoring pour trouver l'auteur de chaque section, sous-section, paragraphe, footnote, figure, ...

Auteurs



powered by astah*

Chain of responsability

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Classification GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Problème

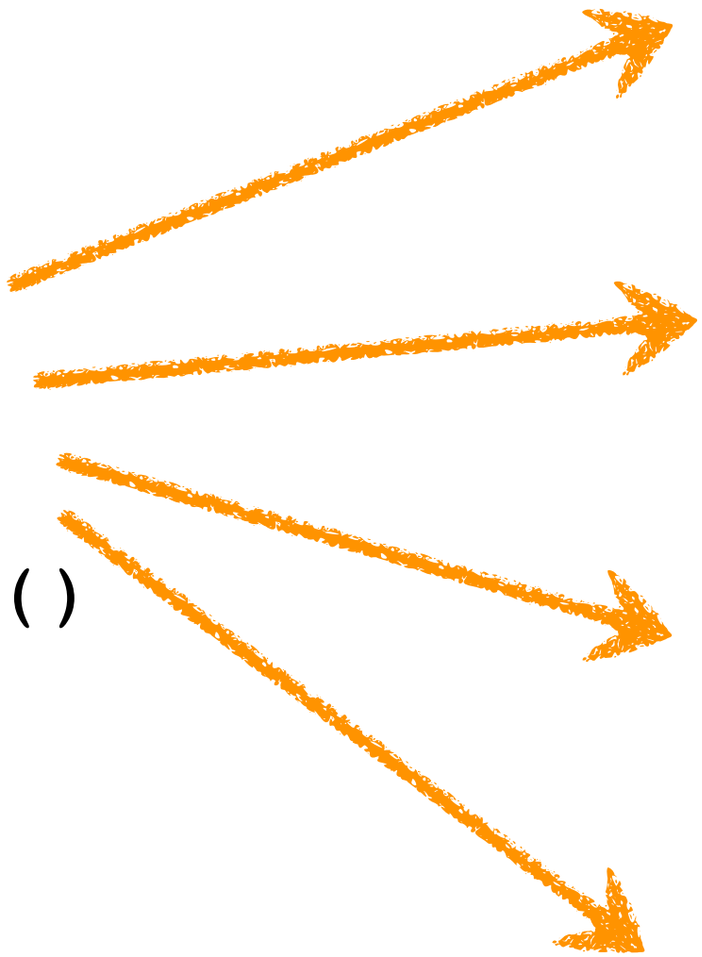
- Comment intégrer plusieurs possibilités alternatives de résolution dans une application ?
- Comment découpler la définition des possibilités alternatives de leur utilisation (dans l'application principale) ?

Exemple

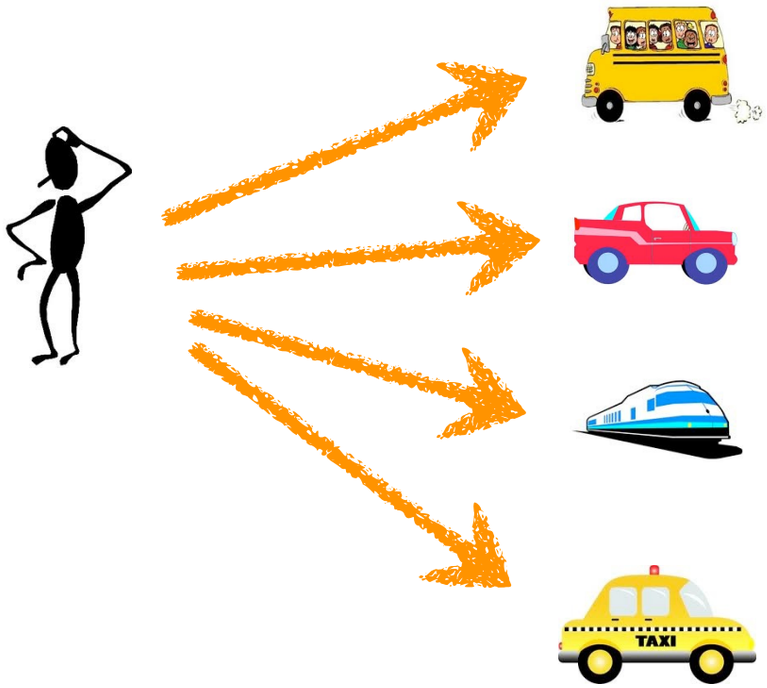
goToAirport()

Exemple

goToAirport()

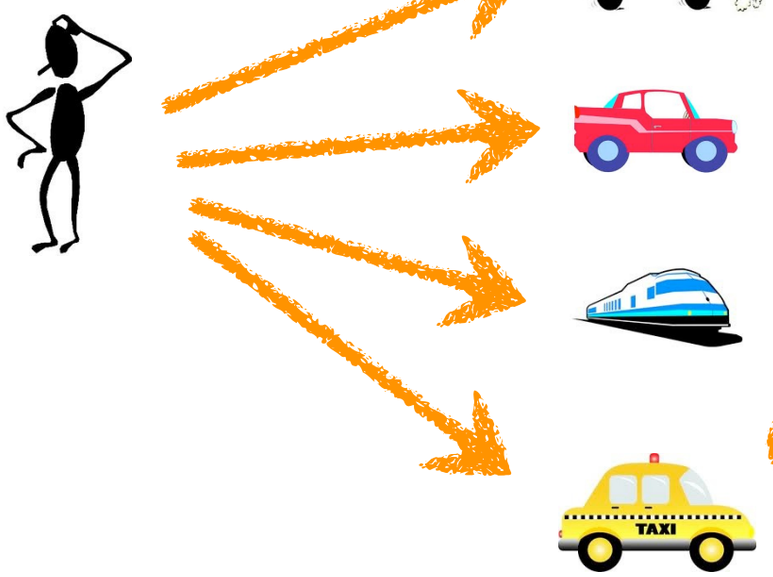


Solution 1




```
public class Person {
    private int age;
    private String name;
    private Car car;
    public void goToAirport() {
        if(age < 18){
            Bus bus = Bus.getAirportBus();
            bus.openDoor();
            bus.getOn(this);
            bus.closeDoor();
        } else if(car != null) {
            car.start();
            car.drive(this);
        } else if(...) {
            ...
        } else {
            System.out.println(name+“walks”);
        } ...
```

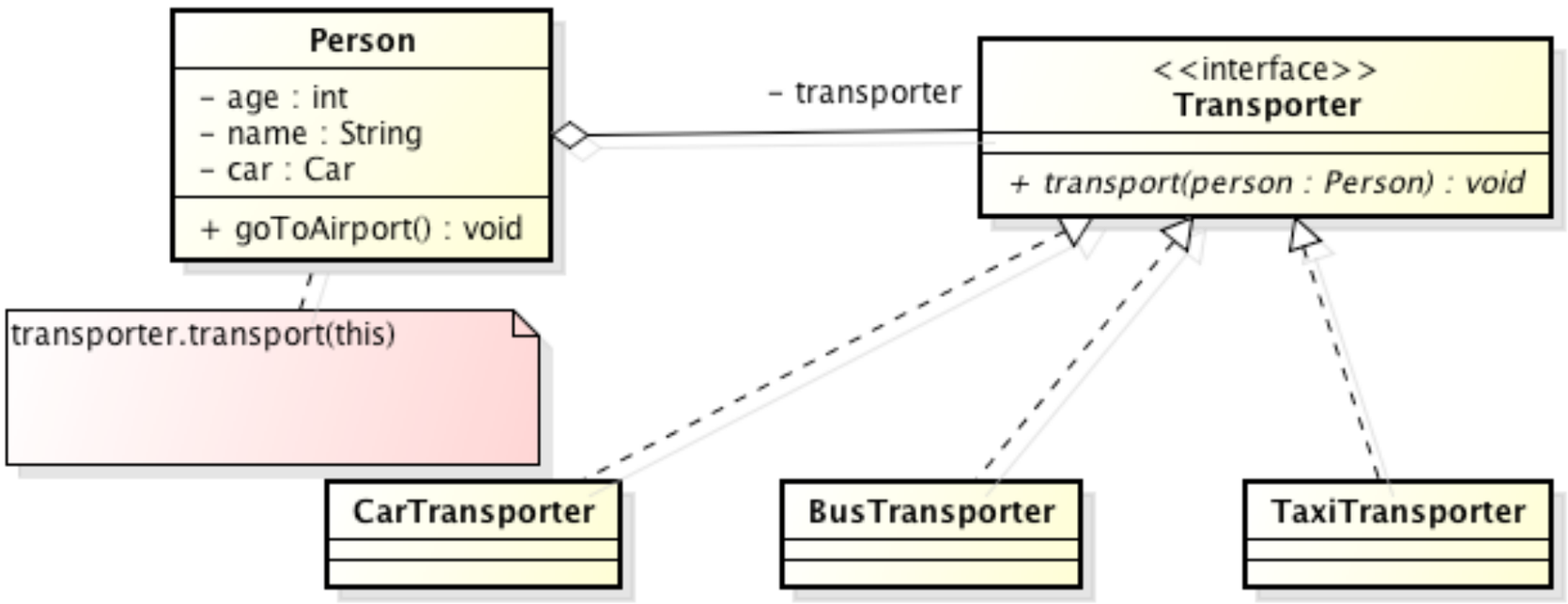
Solution 1




```
public class Person {
    private int age;
    private String name;
    private Car car;
    public void goToAirport() {
        if(age < 18){
            Bus bus = Bus.getAirportBus();
            bus.openDoor();
            bus.getOn(this);
            bus.closeDoor();
        } else if(car != null) {
            car.start();
            car.drive(this);
        } else if(...) {
            ...
        } else {
            System.out.println(name+“walks”);
        } ...
```

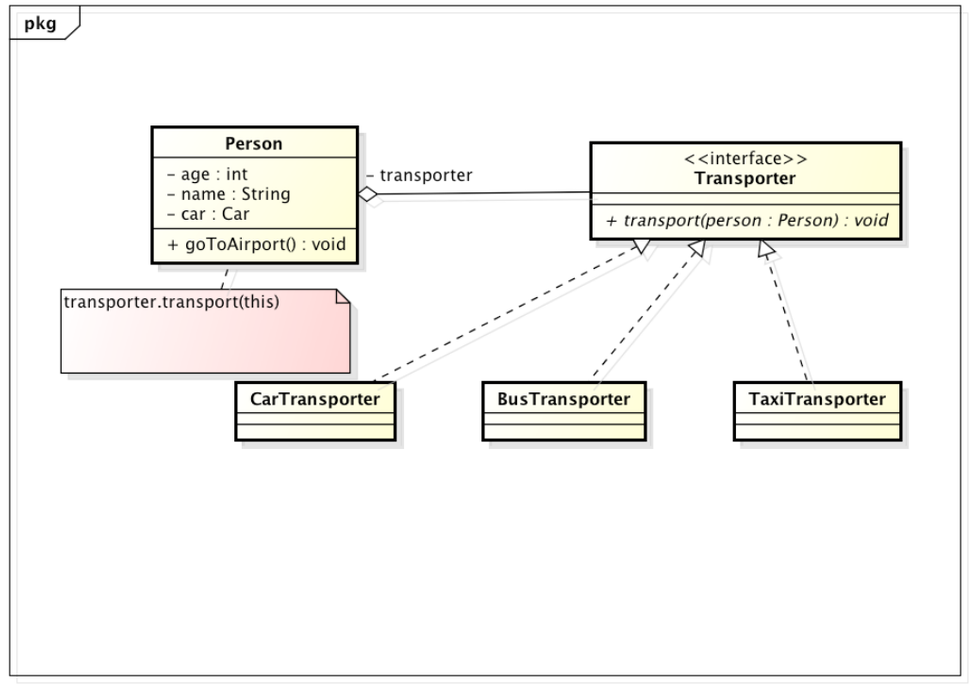
alternatives

Solution 2


Person
- age : int
- name : String
- car : Car
+ goToAirport() : void
- transporter
<<interface>>
Transporter
+ transport(person : Person) : void
transporter.transport(this)
CarTransporter
BusTransporter
TaxiTransporter

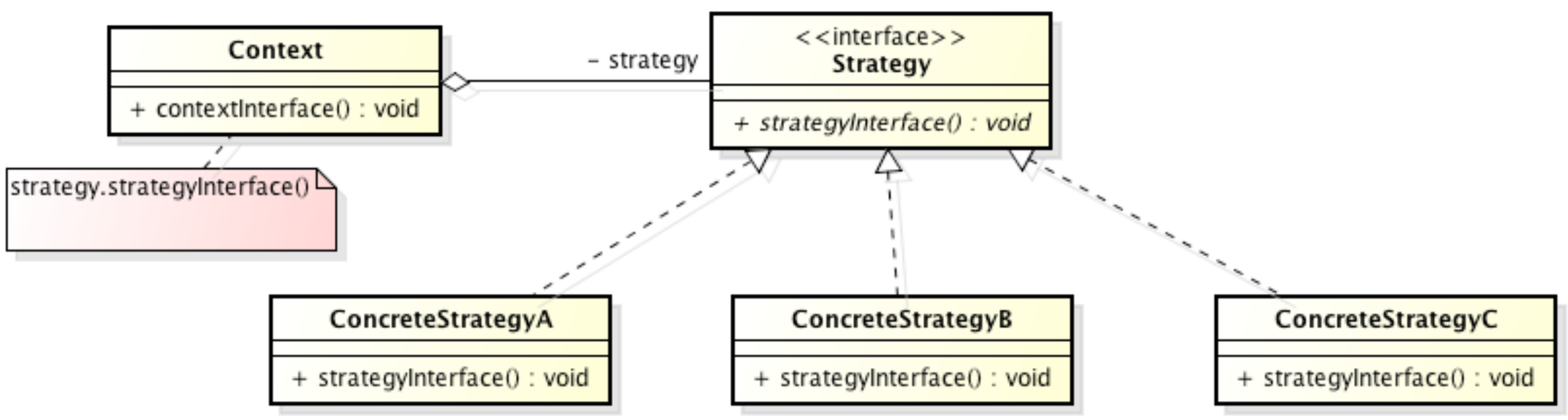
Solution 2

```
public class Person {
    private int age;
    private String name;
    private Car car;
    private Transporter trans;

    public void goToAirport() {
      if(trans != null)
        trans.transport(this);
      else
        System.out.println(name+“walks”);
    }
    public void setTransporter(...) {
      ...
    }
...
```



Strategy

Structure



Strategy

Intention

- Définir une famille d'algorithmes et les rendre interchangeables
- Les stratégies évoluent indépendamment des clients qui les utilisent
- Capturer l'abstraction en une interface, cacher les détails d'implantation dans les classes dérivées

Strategy

Indications d'utilisation

- plusieurs classes ne différent que par leur comportement
- une classe définit de nombreux comportements dans des structures conditionnelles
- nécessité de disposer de diverses variantes d'un algorithme
- un algorithme utilise des données que les clients n'ont pas à connaître

Null Object Pattern

```
public class Person {
    private Transporter trans;

    public void goToAirport() {
      if(trans != null)
        trans.transport(this);
      else
        System.out.println(name+“walks”);
    }
    public void setTransporter(...) {
      ...
    }
...
```

Null Object Pattern

traitement différent en fonction d'une valeur null

```
public class Person {
    private Transporter trans;

    public void goToAirport() {
      if(trans != null)
        trans.transport(this);
      else
        System.out.println(name+“walks”);
    }
    public void setTransporter(...) {
      ...
    }
...
```

Null Object Pattern

```
public class Person {
    private Transporter trans = new NullTransporter();
    public void goToAirport() {
        trans.transport(this);
    }
...
}

public NullTransporter implements Transporter {
    public void transport(Person p){
        System.out.println(p.name+“walks”);
    }
}
```

Résumé Strategy

- identifier un algorithme (i.e. un comportement) que le client souhaite utiliser via un « flex point »
- spécifier la signature de l'algorithme dans une interface
- cacher les implantations alternatives dans des classes dérivée
- les clients de l'algorithme se connectent à l'interface

Strategy

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Classification GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Problème

Problème

- Comment ajouter du comportement ou des informations d'état à des objets individuels au run-time ?

Problème

- Comment ajouter du comportement ou des informations d'état à des objets individuels au **run-time ?**
  - l'héritage n'est pas une option car il est statique et donc le comportement ajouté s'applique à toute une classe d'objets

Commander un café

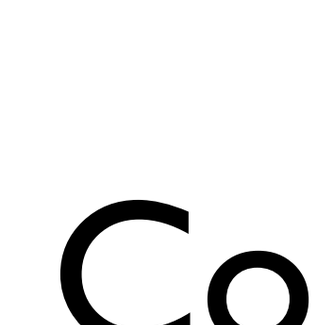
espresso
1€50

Commander un café

cream
0€50

espresso
1€50

Commander un café



cream
0€50

espresso
1€50

vanilla
0€50

Commander un café

cost()

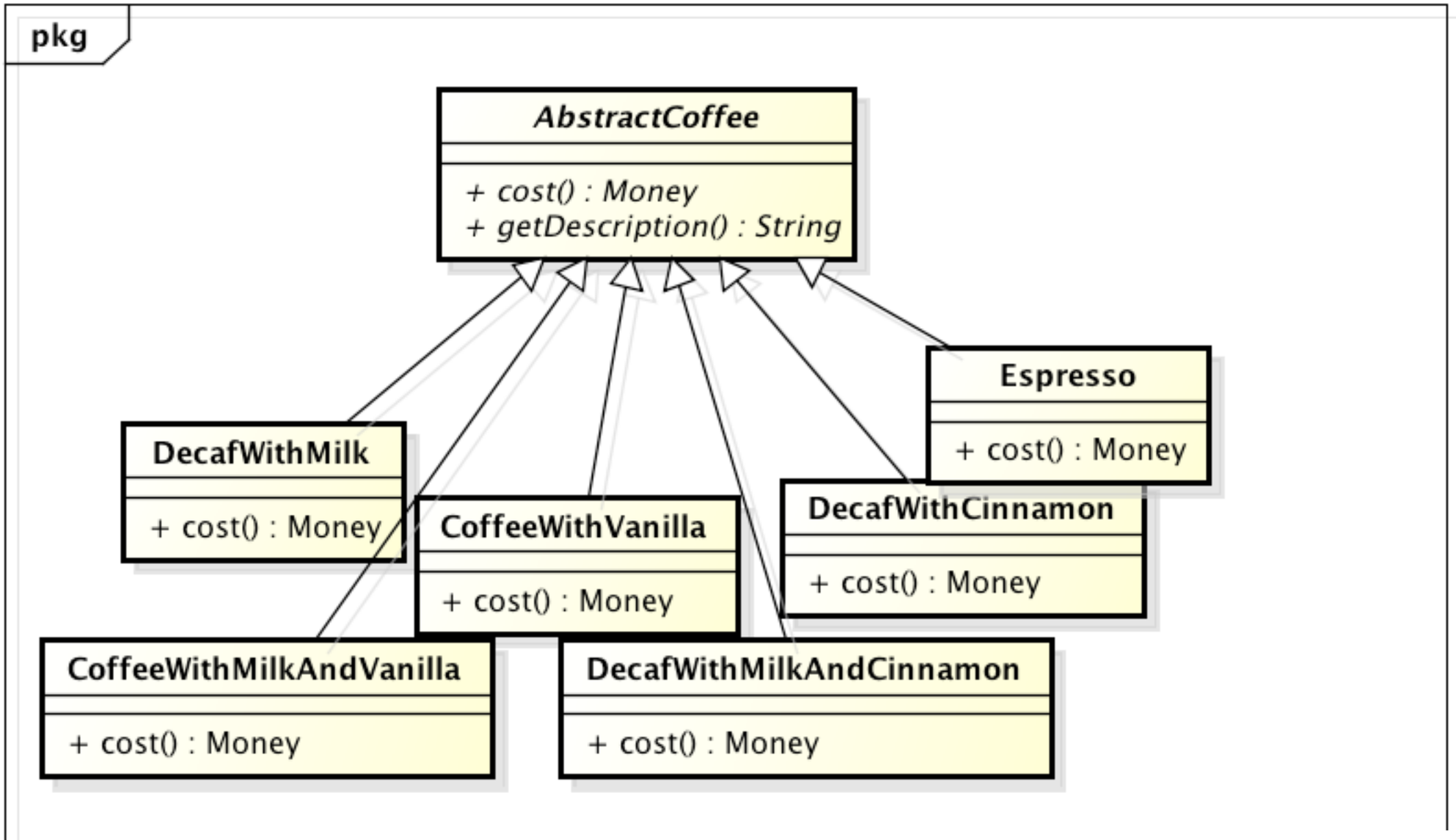
2€50

cream
0€50

espresso
1€50

vanilla
0€50

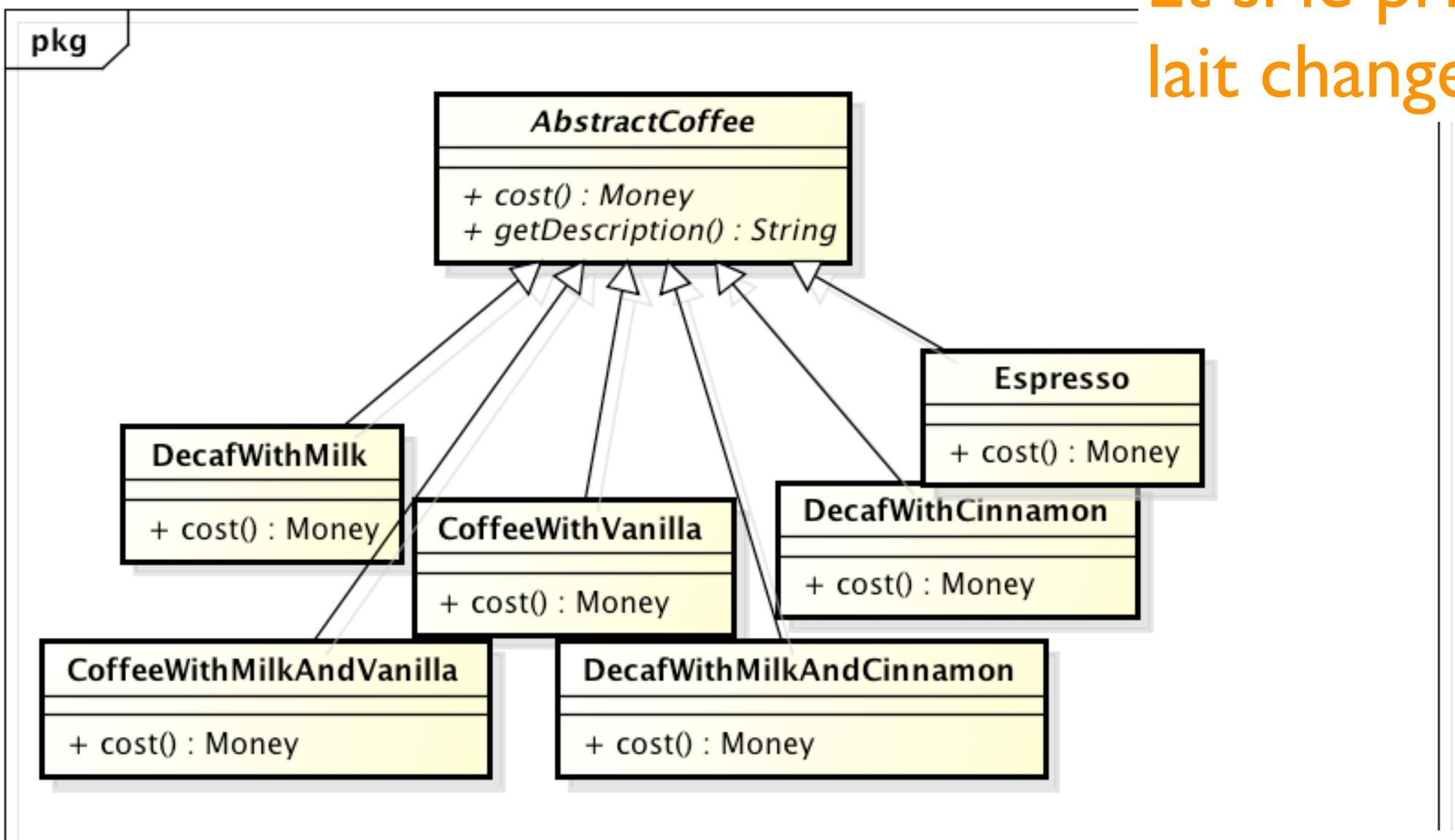
Solution 1



La méthode `cost()` tient compte de tous les suppléments.

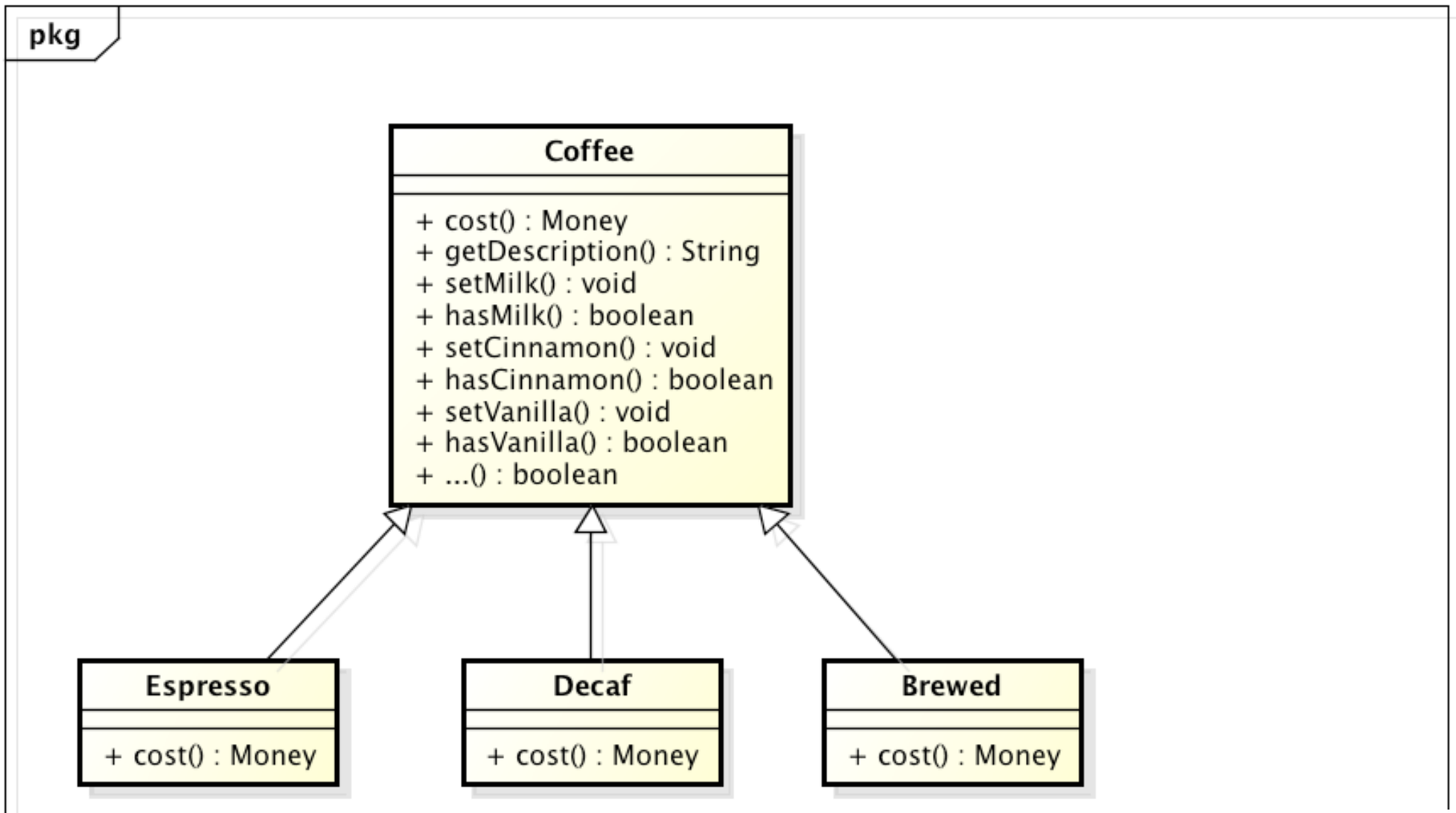
Solution 1

Et si le prix du lait change?



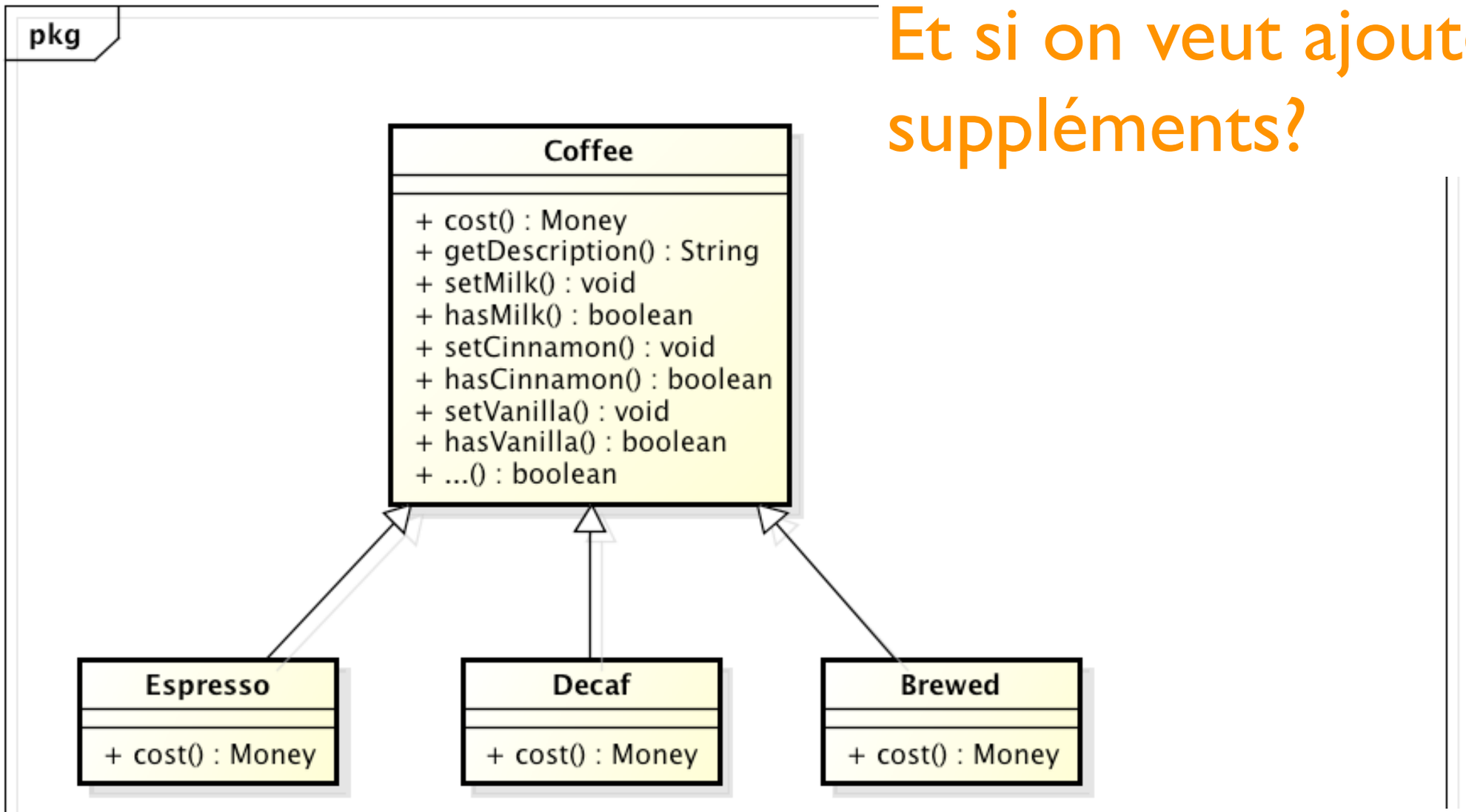
La méthode `cost()` tient compte de tous les suppléments.

Solution 2



La méthode `cost()` retourne le prix du café plus le `cost()` calculé par `SpecialCoffee` en fonction des suppléments.

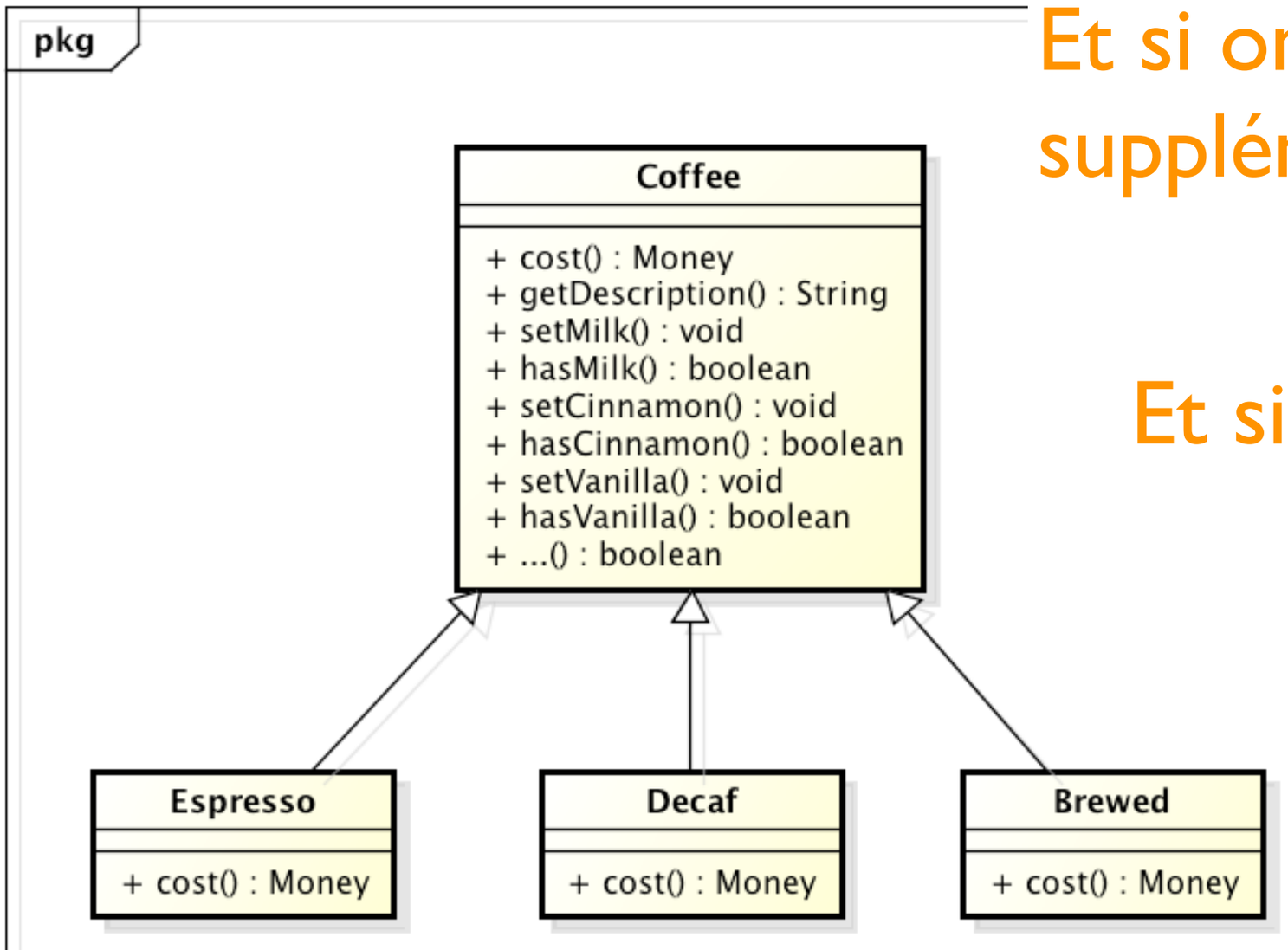
Solution 2



Et si on veut ajouter des suppléments?

La méthode `cost()` retourne le prix du café plus le `cost()` calculé par `SpecialCoffee` en fonction des suppléments.

Solution 2

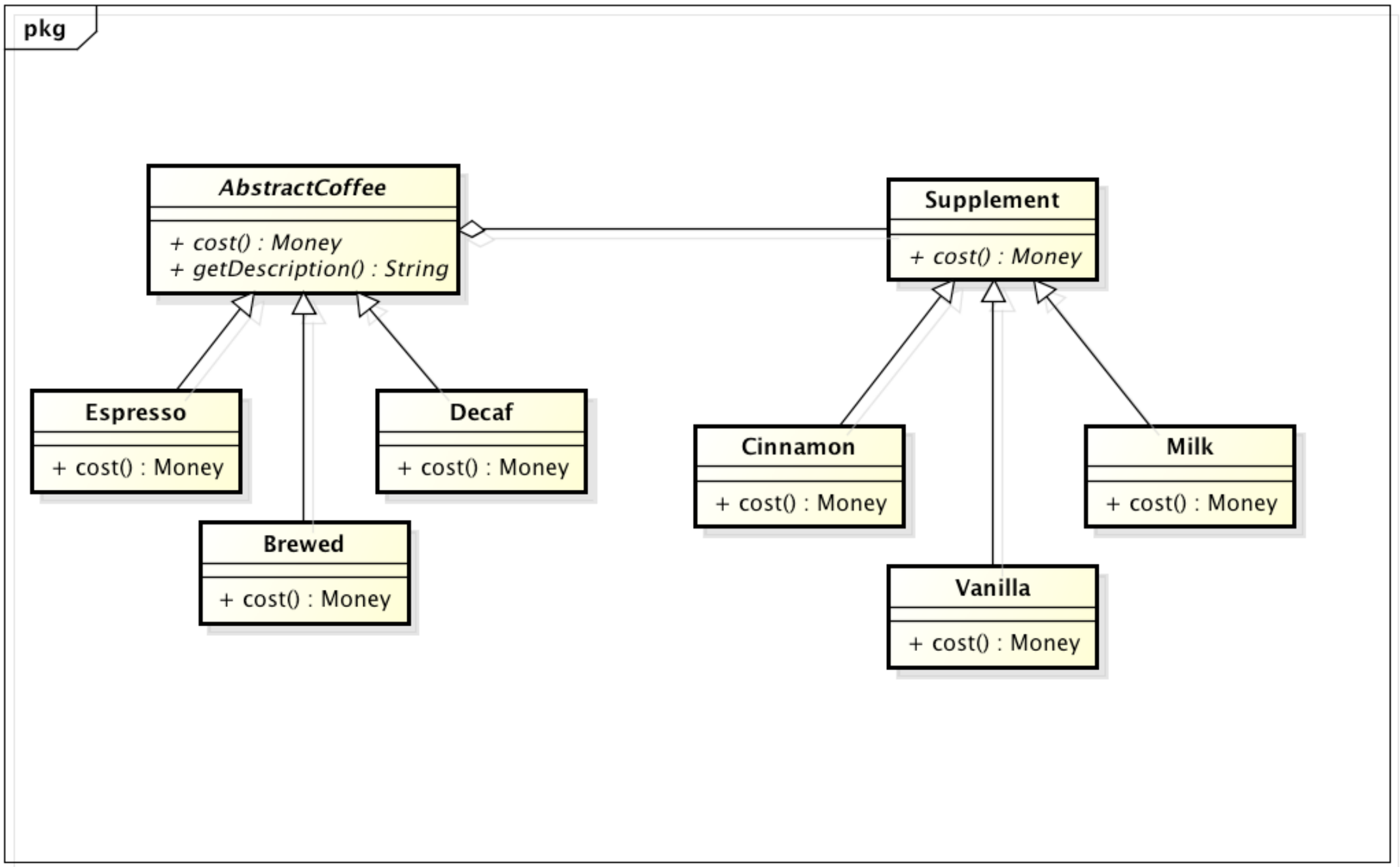


Et si on veut ajouter des suppléments?

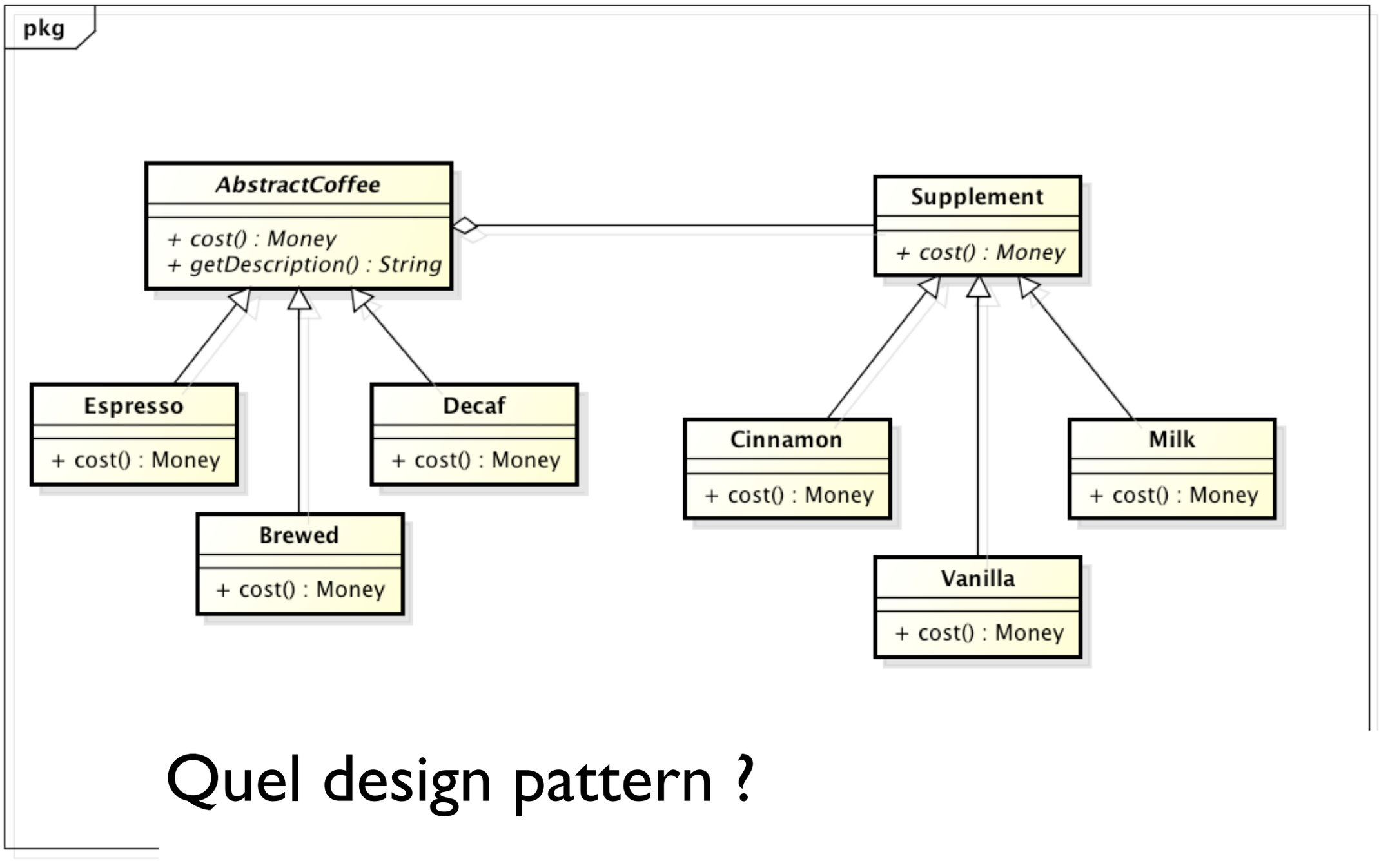
Et si on veut un extra milk ?

La méthode `cost()` retourne le prix du café plus le `cost()` calculé par `SpecialCoffee` en fonction des suppléments.

Solution 3


pkg
AbstractCoffee
+ cost() : Money
+ getDescription() : String
Espresso
+ cost() : Money
Decaf
+ cost() : Money
Brewed
+ cost() : Money
Supplement
+ cost() : Money
Cinnamon
+ cost() : Money
Milk
+ cost() : Money
Vanilla
+ cost() : Money
powered by astah*

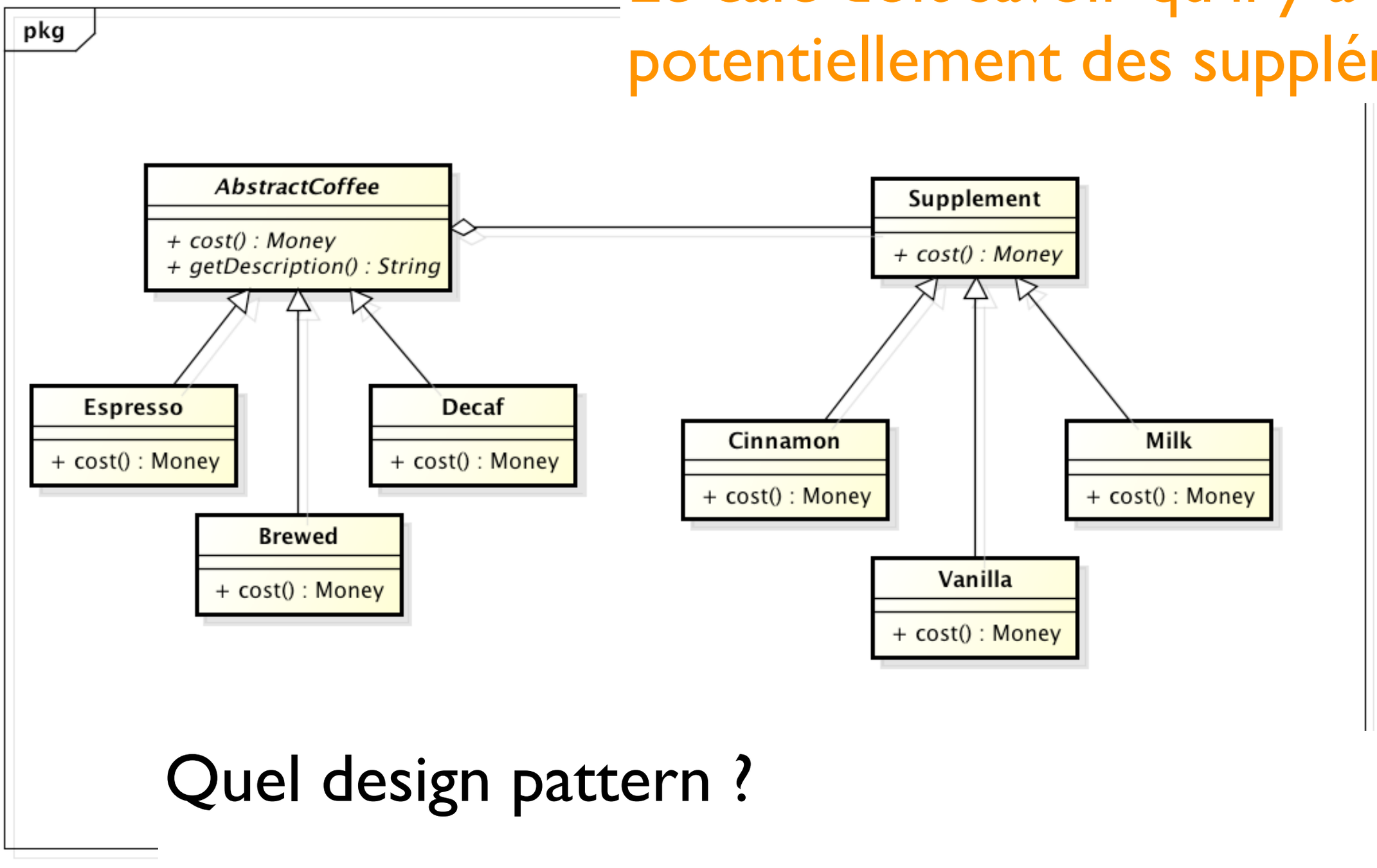
Solution 3



Quel design pattern ?

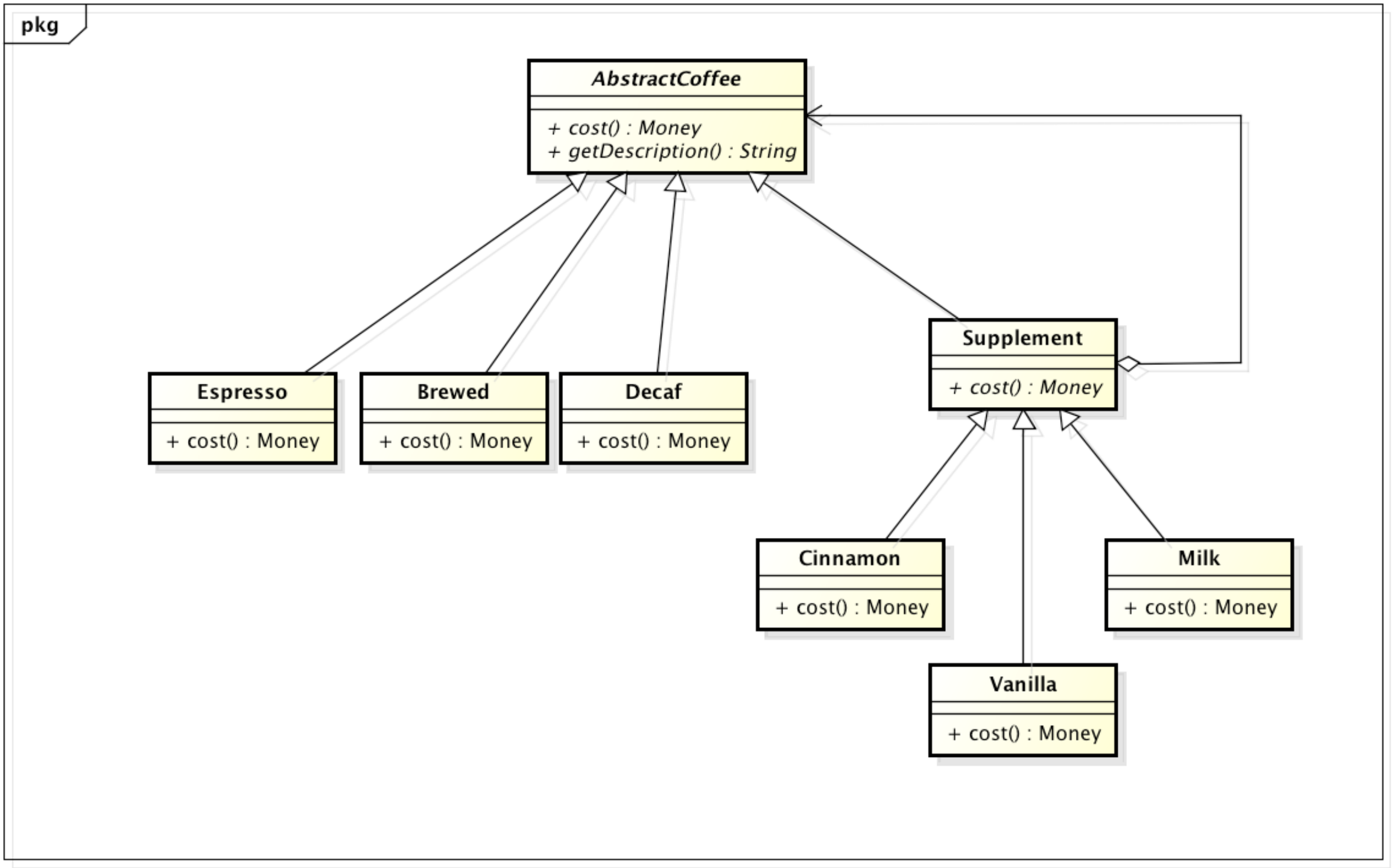
Solution 3

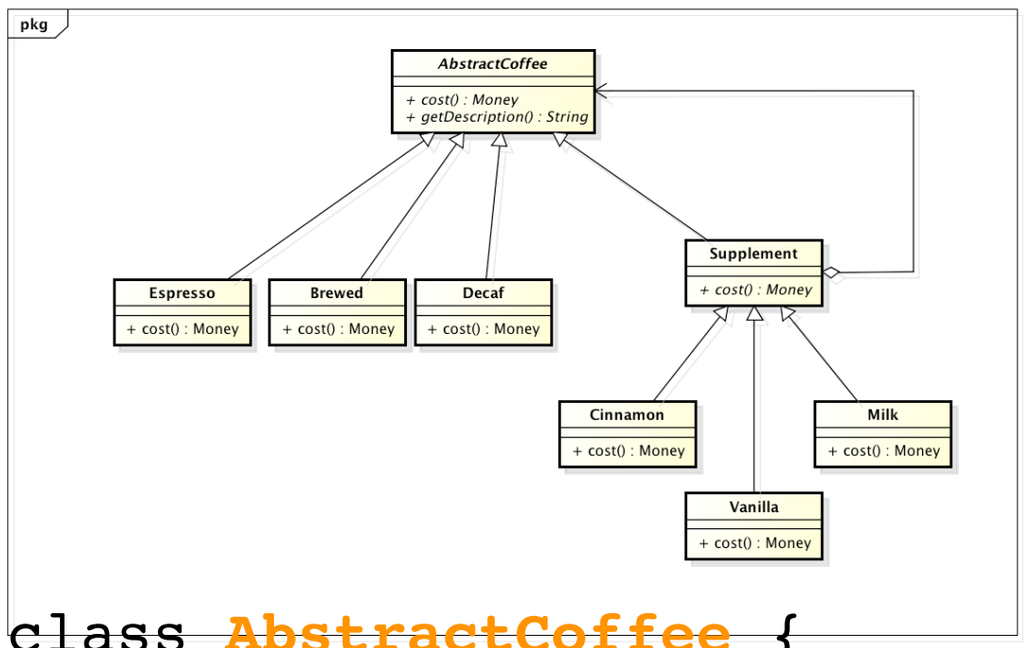
Le café doit savoir qu'il y a potentiellement des suppléments ?



Quel design pattern ?

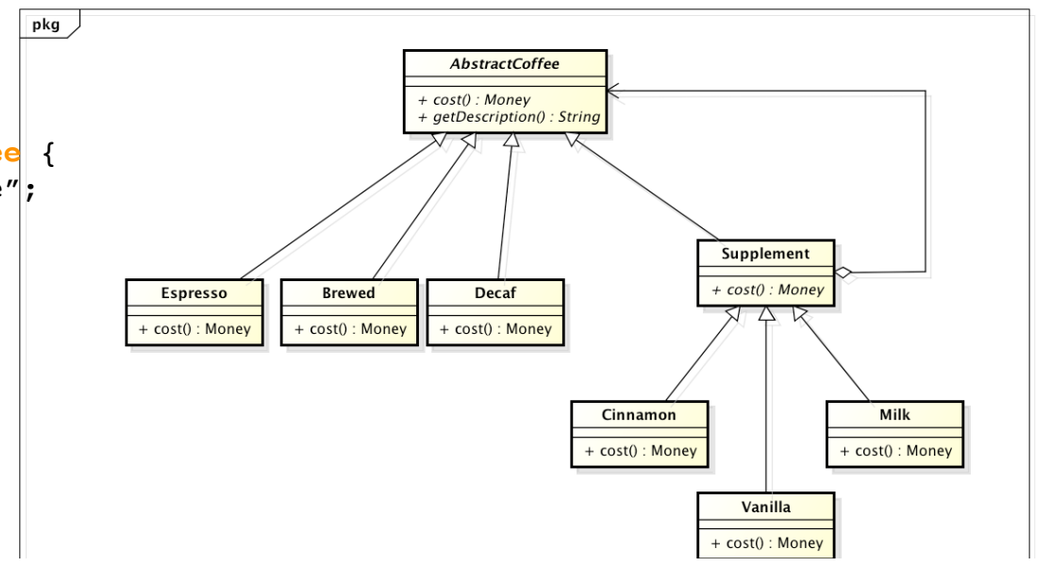
Solution 4


pkg
AbstractCoffee
+ cost() : Money
+ getDescription() : String
Espresso
+ cost() : Money
Brewed
+ cost() : Money
Decaf
+ cost() : Money
Supplement
+ cost() : Money
Cinnamon
+ cost() : Money
Vanilla
+ cost() : Money
Milk
+ cost() : Money




```
public abstract class AbstractCoffee {
  String description = “Some coffee”;
  public String getDescription() {
     return description;
  }
  public abstract Money cost() ;
}


public class Espresso extends AbstractCoffee {
  public Espresso(){
    description = “Espresso”;
  }
  public Money cost(){
    return 1.50;
  }
}
```

```java
public abstract class AbstractCoffee {
  String description = "Some coffee";
  public String getDescription() {
     return description;
  }
  public abstract Money cost() ;
}
```

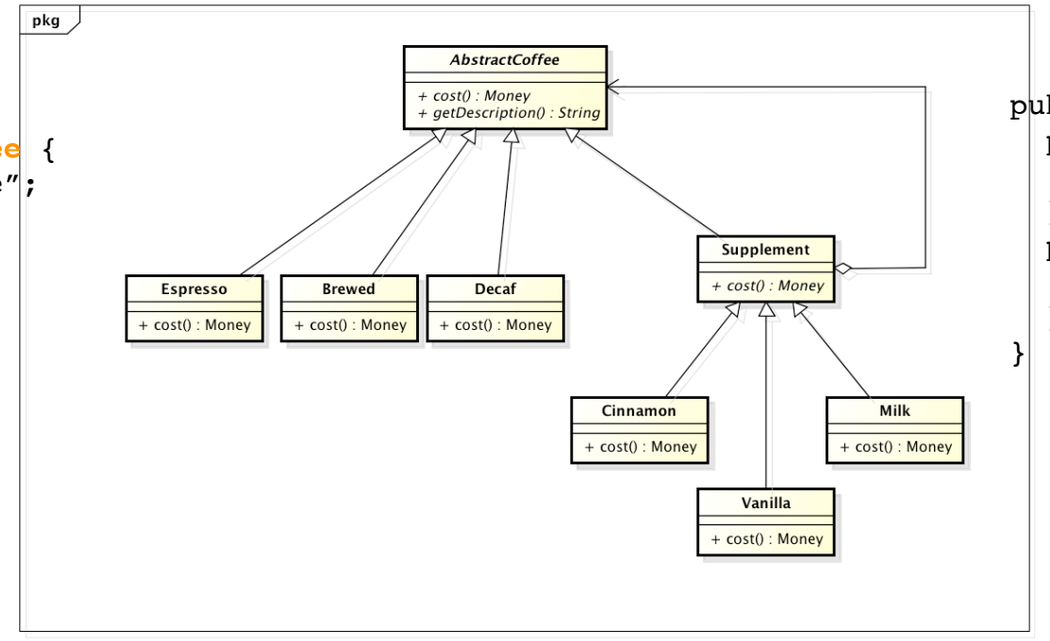



```java
public class Espresso extends AbstractCoffee {
  public Espresso(){
    description = "Espresso";
  }
  public Money cost(){
    return 1.50;
  }
}
```

```java
public abstract class Supplement extends AbstractCoffee {
  public abstract String getDescription();
}
```

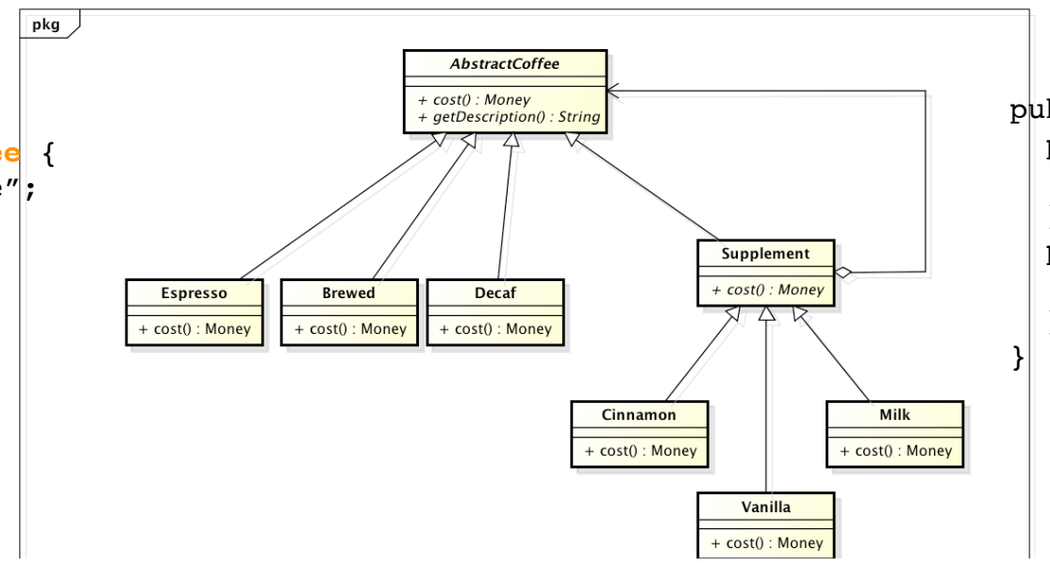
```java
public class Milk extends Supplement {
  AbstractCoffee coffee;
  public Milk(AbstractCoffee coffee){
    this.coffee = coffee;
  }
  public String getDescription() {
     return coffee.getDescription()+", with Milk";
  }
  public Money cost(){
    return 0.50 + coffee.cost();
  }
}
```

```
public abstract class AbstractCoffee {
  String description = “Some coffee”;
  public String getDescription() {
     return description;
  }
  public abstract Money cost() ;
}
```




```
public class Espresso  extends AbstractCoffee {
  public Espresso(){
    description = “Espresso”;
  }
  public Money cost(){
    return 1.50;
  }
}
```

```
public abstract class AbstractCoffee {
  String description = “Some coffee”;
  public String getDescription() {
     return description;
  }
  public abstract Money cost() ;
}
```




```
public class Espresso  extends AbstractCoffee {
  public Espresso(){
    description = “Espresso”;
  }
  public Money cost(){
    return 1.50;
  }
}
```

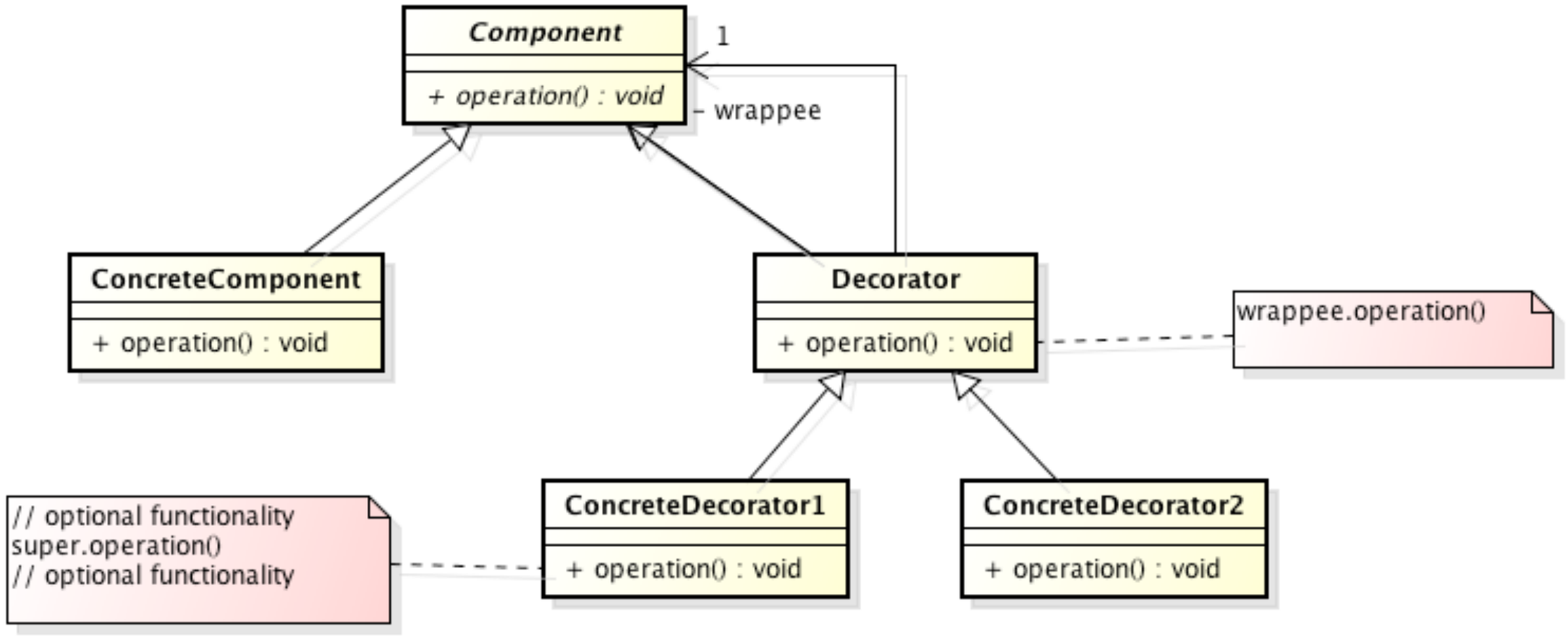
```
public abstract class Supplement extends AbstractCoffee {
  public abstract String getDescription();
}
```

```
public class Milk extends Supplement {
  AbstractCoffee coffee;
  public Milk(AbstractCoffee coffee){
    this.coffee = coffee;
  }
  public String getDescription() {
     return coffee.getDescription()+”, with Milk”;
  }
  public Money cost(){
    return 0.50 + coffee.cost();
  }
}
```

```
public class CoffeeShop {
    public static void main(String[] args){() {
        AbstractCoffee espresso = new Espresso();
        AbstractCoffee milkEspresso = new Milk(espresso);

        S.o.p(milkEspresso.getDescription() +
                    “ costs “ + milkEspresso.cost());

    }
}
```

```
> Espresso, with milk costs 2.00
```

Decorator


Component
+ operation() : void
1
- wrappee
ConcreteComponent
+ operation() : void
Decorator
+ operation() : void
wrappee.operation()
// optional functionality
super.operation()
// optional functionality
ConcreteDecorator1
+ operation() : void
ConcreteDecorator2
+ operation() : void

Decorator

Intention

- attacher dynamiquement des capacités additionnelles à un objet; fournir une alternative flexible à l'héritage pour étendre les fonctionnalités
- "embellissement" d'un objet principal au travers des emballages récursives spécifiés par un client

Decorator

Indications d'utilisation

- ajouter dynamiquement des capacités de manière transparente à un objet sans affecter les autres objets (de la classe)
- définir des capacités qui peuvent être retirées
- l'extension par héritage produirait un nombre trop important d'extensions indépendantes
- l'héritage est interdit/impossible

Decorator

Conséquences

- personnalisation d'objets plus flexible que l'héritage statique → ajouter et enlever des responsabilités au run-time
- possibilité d'ajouter 2 fois le même decorator
- ajouter des responsabilités d'une manière incrémentale
- *multiplication des classes (les décorations)*

Decorator

Implémentation

- utilisation d'interfaces pour la conformité (du décorateur avec l'objet décoré)
- pas forcément besoin d'un décorateur abstrait
- maintenir une classe de base légère

Decorator

Implémentation

- utilisation d'interfaces pour la conformité (du décorateur avec l'objet décoré)
- **pas forcément besoin d'un décorateur abstrait**
- **maintenir une classe de base légère**

Decorator est fait pour le changement d'aspect
Strategy est fait pour le changement radical d'approche

Decorator

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor