

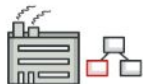
A decorative graphic on the left side of the slide. It consists of a blue parallelogram and a light green parallelogram, both tilted at an angle. The blue shape is in the foreground, and the green shape is partially behind it. They are set against a dark blue background with faint, lighter blue diagonal stripes.

Design

Patterns

Creational patterns

These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.



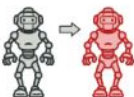
Factory Method



Abstract Factory



Builder



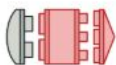
Prototype



Singleton

Structural patterns

These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.



Adapter



Bridge



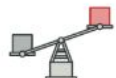
Composite



Decorator



Facade



Flyweight



Proxy

Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.



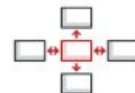
Chain of Responsibility



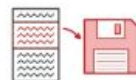
Command



Iterator



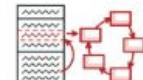
Mediator



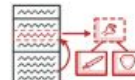
Memento



Observer



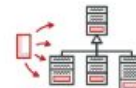
State



Strategy



Template Method



Visitor

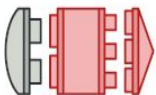


Structural Design Patterns

Structural Design Patterns الگو های ساختاری هستند و ما تمرکز مون بر روی این است که چگونه کلاس ها و اشیاء را باهم ترکیب کنیم که ساختار های بهینه و کارآمد بسازیم که بتوان به راحتی آن ها را تغییر داد باشد

Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

Structural Design Patterns



Adapter

Allows objects with incompatible interfaces to collaborate.



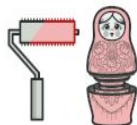
Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.



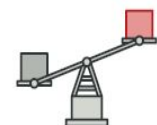
Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



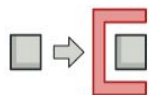
Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.



Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.



Proxy


Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Facade
6. Flyweight
7. Proxy

A decorative graphic on the left side of the slide consisting of two overlapping parallelograms. The front one is blue and the back one is a light green. They are positioned diagonally, with the blue one partially covering the green one.

Composite

Structural Design Patterns



فرض کنید یک رستوران دارید که در منوی آن چندین آیتم وجود دارد (مثل برگر , ساندویچ , پیتزا , سیب زمینی , نوشابه کوکا , دوغ ...) و آیتم هایی همچون پک خانوادگی نیز وجود دارد که شامل چندین آیتم منوی شما هست

حال اگر ما قیمت ثابتی برای پک خانوادگی خود در نظر بگیریم با تغییر قیمت آیتم ها ما به ناچار باید قیمت پک خانوادگی را تغییر دهیم

Admin



قیمت نوشابه کوکا را
به 20 هزار تومان
تغییر بده

چون قیمت نوشابه از
10 به 20 تغییر کرد
سبد کوکا را 60 کن
و قیمت پک خانوادگی
رو 829 تبدیل کن

Customer



قیمت پک خانوادگی چنده

منوی رستوران

قیمت رو بگو

پک خانوادگی

799

تغییر بده ✓

تغییر بده ✓

سبد کوکا
30

سبب زمینی

برگر

تغییر بده ✓

بطری کوکا
1
10

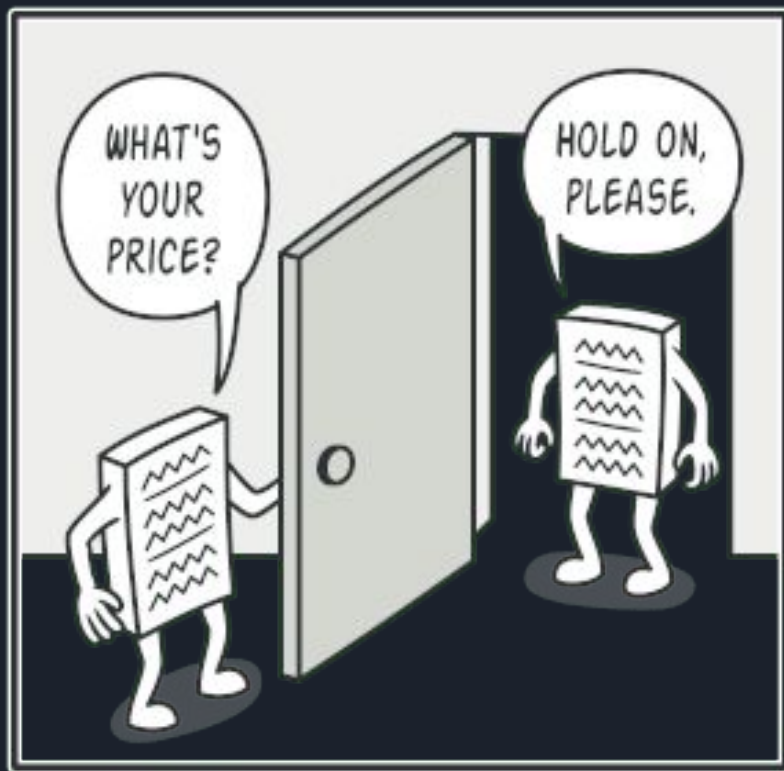
بطری کوکا
2
10

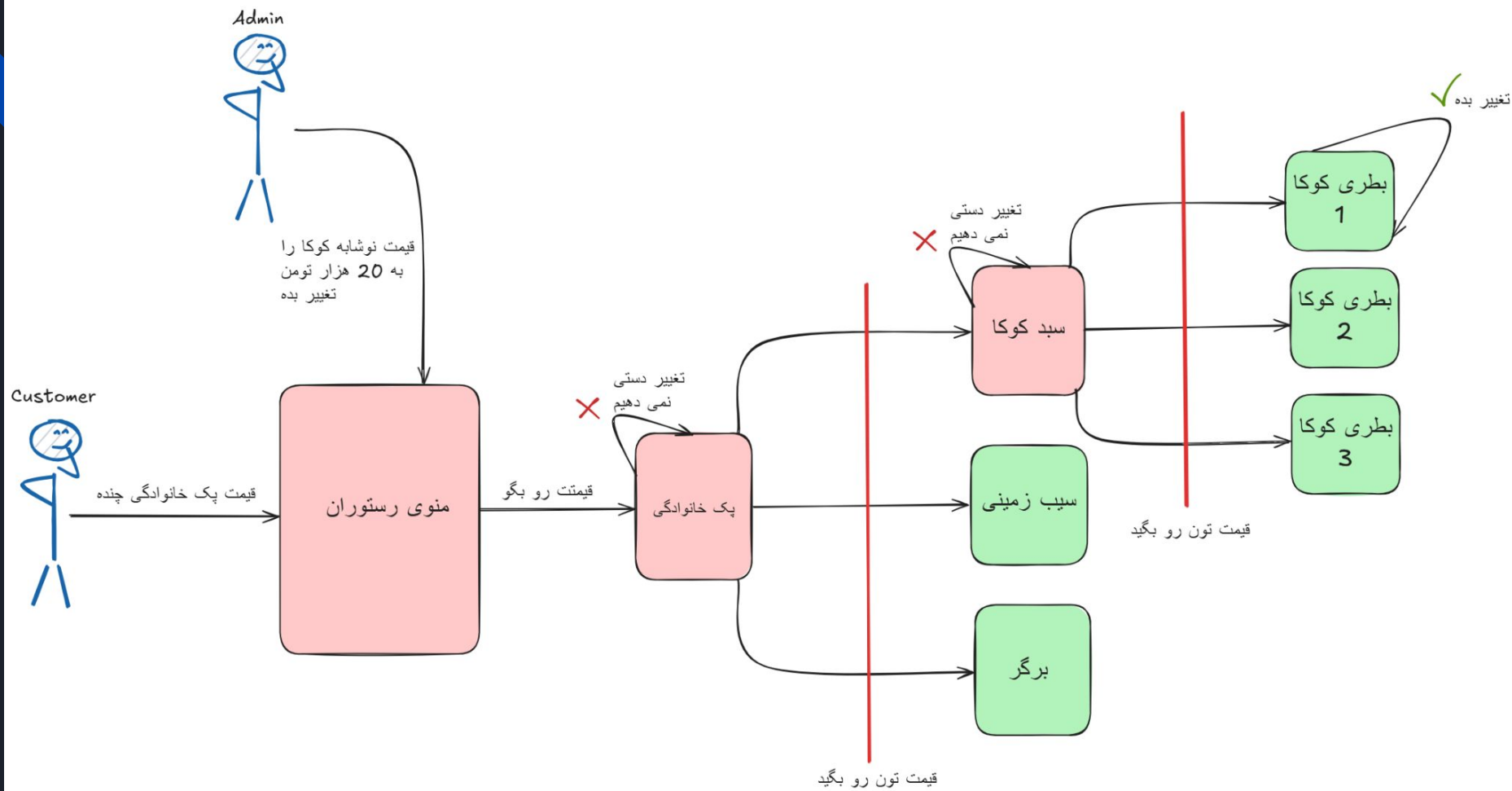
بطری کوکا
3
10



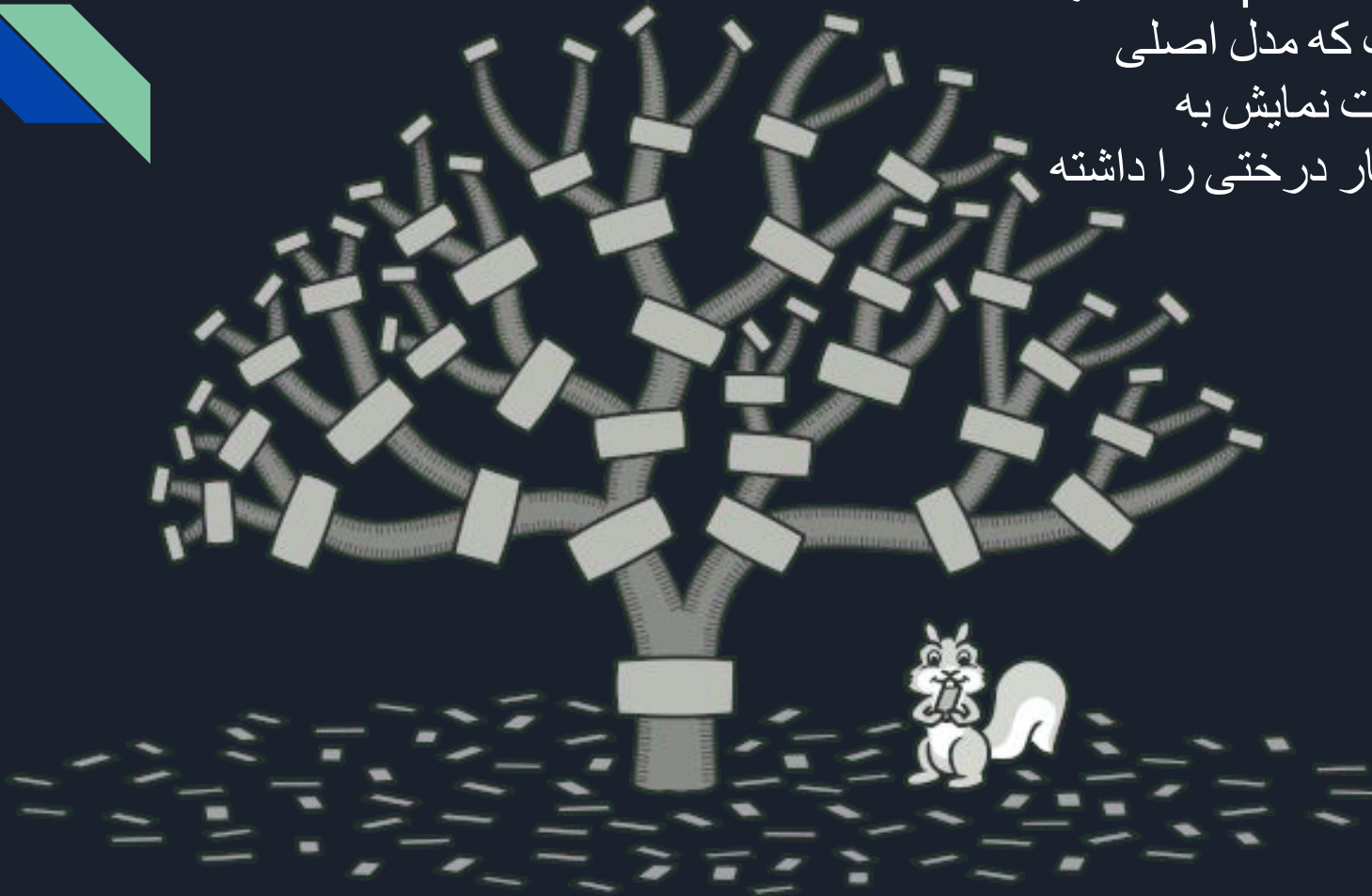
Composite


حالا اگر از **Composite** بخوایم استفاده کنیم به جای این که برای پک خانوادگی و سبد کوکا قیمتی ثابت در نظر بگیریم می توان به پک خانوادگی اعلام کرد که شامل سبد کوکا , برگر , سیب زمینی است و در صورتی که کسی این پک را انتخاب کرد هر آیتم به صورت انفرادی قیمت خود را اعلام میکند و دیگر نیازی نیست با تغییر قیمت هر کدام از آیتم ها قیمت پک خانوادگی را نیز تغییر دهیم





استفاده از الگوی Composite تنها
زمانی منطقی است که مدل اصلی
برنامه‌ی شما قابلیت نمایش به
صورت یک ساختار درختی را داشته
باشد.





الگوی Composite به شما اجازه می‌دهد اشیاء را به صورت درختی
سازماندهی کنید

Component : یک کلاس یا اینترفیس انتزاعی که عملیات مشترک بین
اشیای Leaf و Composite را تعریف می‌کند شامل عملیات هایی که همه
عناصر باید داشته باشند

Leaf : اشیاء انتهایی که دیگر زیرمجموعه ندارند وظیفه اصلی Leaf انجام
عملیات نهایی است

Composite : اشیائی که می‌توانند شامل چند Component یا چندین
Leaf باشند مدیریت فرزندان را انجام می‌دهد



Relations with Other Principle

ترکیب Composite با Single Responsibility Principle :

Component فقط وظیفه دارد یک قرارداد مشترک برای **Leaf** و **Composite** فراهم کند

Leaf فقط وظیفه دارد منطق مخصوص خودش (Operation واقعی اش) را پیاده‌سازی کند

Composite فقط مسئول مدیریت فرزندان و فراخوانی عملیات آنها است

ترکیب Composite با Open/Closed Principle :

ما می‌توانیم بدون تغییر در **Component** یا در **Composite** ، انواع جدیدی از **Leaf** یا **Composite** ایجاد کنیم

ترکیب Composite با Liskov Substitution Principle :

Leaf و Composite هر دو زیرنوع Component هستند کلاينت می تواند بدون دانستن اینکه شیء دریافتی Leaf است یا Composite، فقط از طریق متد های Component با آن کار کند

ترکیب Composite با Interface Segregation Principle :

وقتی Add, Remove, GetChild را در Component قرار می دهیم، Leaf آن ها را به ارث می برد در حالی که Leaf از این متدها استفاده ای ندارد این کمی نقض ISP است، چون Leaf مجبور است متد هایی داشته باشد که منطقی نیست است یا استثنا می دهد برای رعایت بهتر ISP می توانیم اینترفیس های جداگانه تعریف کنیم، مثلاً IComponent برای عملیات اصلی و یک IComposite برای عملیات مدیریت فرزندان Leaf فقط IComponent را پیاده کند و Composite هر دو را پیاده سازی کند

ترکیب Composite با Dependency Inversion Principle :

کلاينت و حتی کلاس Composite نباید به نوع های Concrete وابسته باشند، بلکه باید با Component (اینترفیس/کلاس انتزاعی) کار کنند وقتی Composite لیست فرزندان را نگه می دارد، آن را به صورت `List<Component>` نگه می دارد نه `List<Leaf>` یا `List<Composite>`



Relations

with

Other

Patterns

- You can use **Builder** when creating complex **Composite** trees because you can program its construction steps to work recursively.
- **Chain of Responsibility** is often used in conjunction with **Composite**. In this case, when a leaf component gets a request, it may pass it through the chain of all of the parent components down to the root of the object tree.
- You can use **Iterators** to traverse **Composite** trees.
- You can use **Visitor** to execute an operation over an entire **Composite** tree.
- You can implement shared leaf nodes of the **Composite** tree as **Flyweights** to save some RAM.
- **Composite** and **Decorator** have similar structure diagrams since both rely on recursive composition to organize an open-ended number of objects.

A *Decorator* is like a *Composite* but only has one child component. There's another significant difference: *Decorator* adds additional responsibilities to the wrapped object, while *Composite* just "sums up" its children's results.

However, the patterns can also cooperate: you can use *Decorator* to extend the behavior of a specific object in the *Composite* tree.

- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.



Relations with Other Patterns

: Builder و Composite

Builder می‌تواند برای ساخت تدریجی یک ساختار Composite استفاده شود به جای اینکه مستقیماً Composite و Leaf را new و Add کنی، یک Builder داری که به صورت مرحله ای آن‌ها را بسازد و درخت را سرهم کند

: Chain of Responsibility و Composite

گاهی درخت Composite را با Chain of Responsibility ترکیب می‌کنند به این صورت که اگر یک Leaf یا Composite نتوانست یک درخواست را پاسخ دهد، آن را به والدش (یا فرزندانش) پاس بدهد

: Iterator و Composite

چون Composite ساختار درختی دارد، پیمایش (Traversal) نیاز می‌شود می‌توانی برای پیمایش یکنواخت کل این ساختار از Iterator استفاده کنی



Composite و Flyweight :

Composite وقتی تعداد زیادی Leaf داشته باشیم ممکن است حافظه زیادی مصرف کند می توانی Leaf ها را به صورت Flyweight طراحی کنی (یعنی داده اشتراکی بین چند Leaf)

Decorator و Composite :

هر دو از یک قرارداد انتزاعی (Component) استفاده می کنند در **Decorator**، به جای ساختار درختی، یک زنجیره تک شئی است که هر Decorator دور یک Component را می گیرد در **Composite**، چندین Component زیر یک Composite جمع می شوند

A decorative graphic on the left side of the slide. It consists of a blue parallelogram and a light green parallelogram, both tilted at an angle. The blue shape is in the foreground, and the green shape is partially behind it. They are set against a dark blue background with diagonal stripes.

This is

the end