**Design Patterns Summarize**

Command Pattern

1. Advantages and disadvantages

Don’t want remote to consist of a set of if statements.

Whenever a new vender class comes out, we have to modify the code and fix the bugs.

Decouple the requester and the performs. (I used to module them and use switch case trie tree. Using Command interface and storing a command object for each button.

If we need to add a new command to our system, we just need to modify only one class (client) in our system. If we use trie tree, it is terrible.)

It is easy to design a command queue.

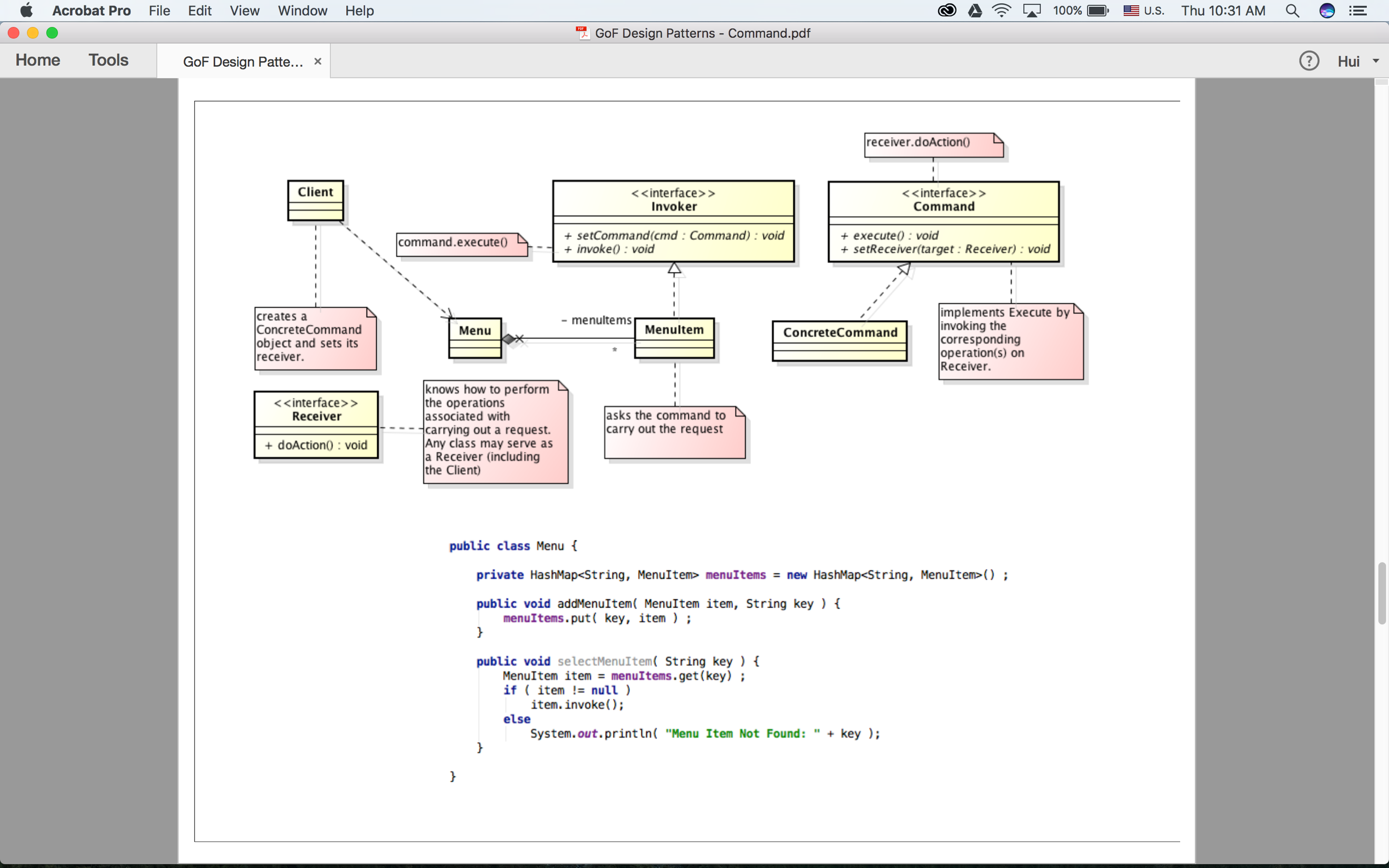
It is convenient to record the commands to logs.

When we received the request from the client, we can decide whether to do the commands.

It is easy to redo.

It is easy to add new commands because the new command class do not effect others.

1. Class Diagram:



3 Demo:

<https://drive.google.com/drive/folders/0B9BRIstFILWqMWdxLUh4dWVaM1E?usp=sharing>

4 Processing:

Menu contains menuItems (Interface)

Each menuItem contains a command (Interface);

Each ConcreteCommand contains a Receiver.

Receiver was used by Client. (Callback)

5 When to use it:

We really need to undo or recover from the former operations.

Decorate Pattern

1. Advantages and disadvantages

Open-closed principle: Open for extension, closed for modification.

(It means that the system will be steady if we add new code such as new classes to our system instead of modifying the existing code in the system when new requirement comes out.

If the part need to be changed frequently, we need to use abstractions.)

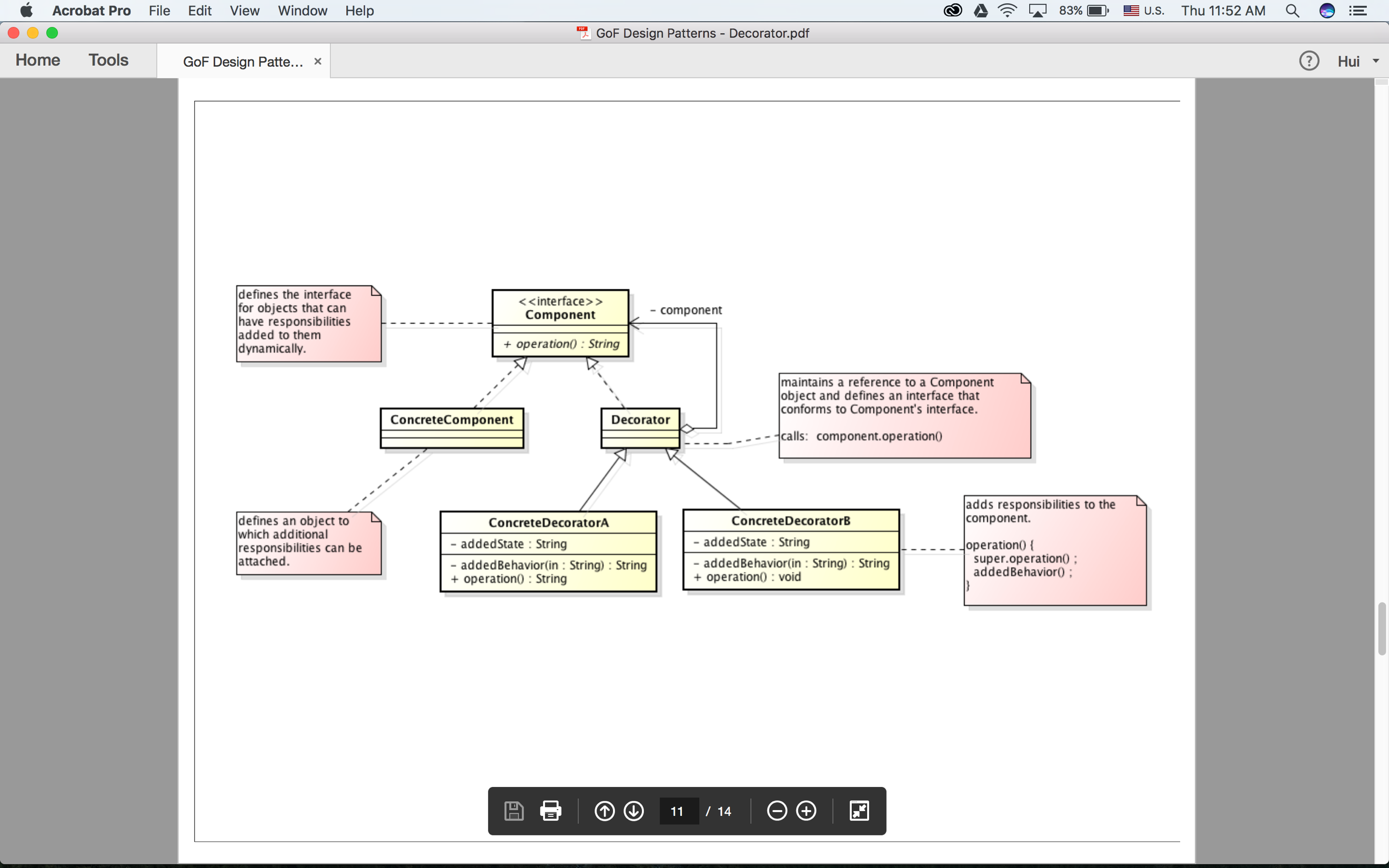
Adding more functions dynamically.

(Burger is the basic class which we must use every time, but chees or toppings are some features we may use in the specific situation. We already make chees to be a ConcreteDecorater)

Divide the most important function from decorator.

(If we want to get more food, we just need to modify the client to attach something to the main function instead of modifying the Burger class.)

1. Class Diagram:



1. Demo:

<https://drive.google.com/drive/folders/0B9BRIstFILWqYm5NZTJwU0VqOG8?usp=sharing>

1. Processing:

ConcreteComponent contains basic functions of the systems.

Decorator contains Component (interface).

ConcreteDecorator extends Decorator, uses super operation and adds its own state and overwrite its own operation.

{When we use these ConcreteDecorators, we need to new ConcreteDecoratorA(new ConcreteDecoratorB(new ConcreteDecoratorC(new ConcreteComponent()))), or use decorate method as follows:

b.decorate(a);

c.decorate(b);

d.decorate( c );

e.decorate(d);}

1. When to use it:

When we need to separate the core functions to the conditional used functions and the situations often change such as Burger ordering menu.

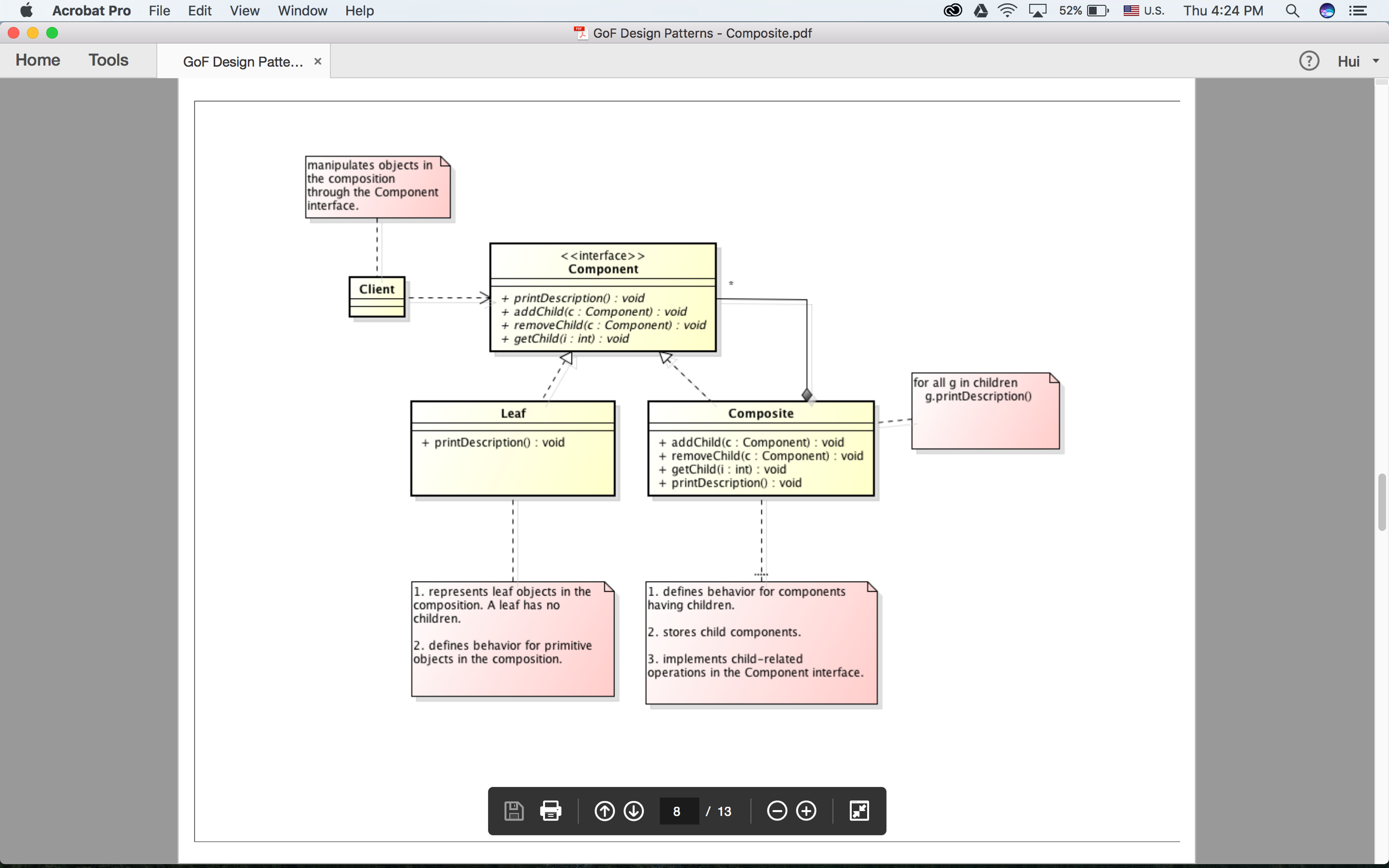
Composite Pattern

1. Advantages and disadvantages

Users can deal with a leafNode as well as Components.

(It means that users can get any components if they want, because they use the same interfaces. Composite can add child which helps users to build their own components.)

1. Class Diagram:



1. Demo:

<https://drive.google.com/drive/folders/0B9BRIstFILWqM1Y4Q1NPX1lUOEE?usp=sharing>

1. Processing:

Clients just need to add the tree nodes they wanted.

Both leaf and Composite implement the same interface, which means that users can use them transparently.

Client need to build the trees with composite which can add or remove a child from the tree.

Even if the structure of the tree is changed, client can also build it conviently.

1. When to use it:

Composite order.addchild(“String”) and build the tree.

User can use composite objects as well as leaf.