**Builder Method:**

You use the builder design pattern when you want to have many classes help in the creation of an object. By having different classes build the object you can then easily create many different types of objects without being forced to rewrite code.

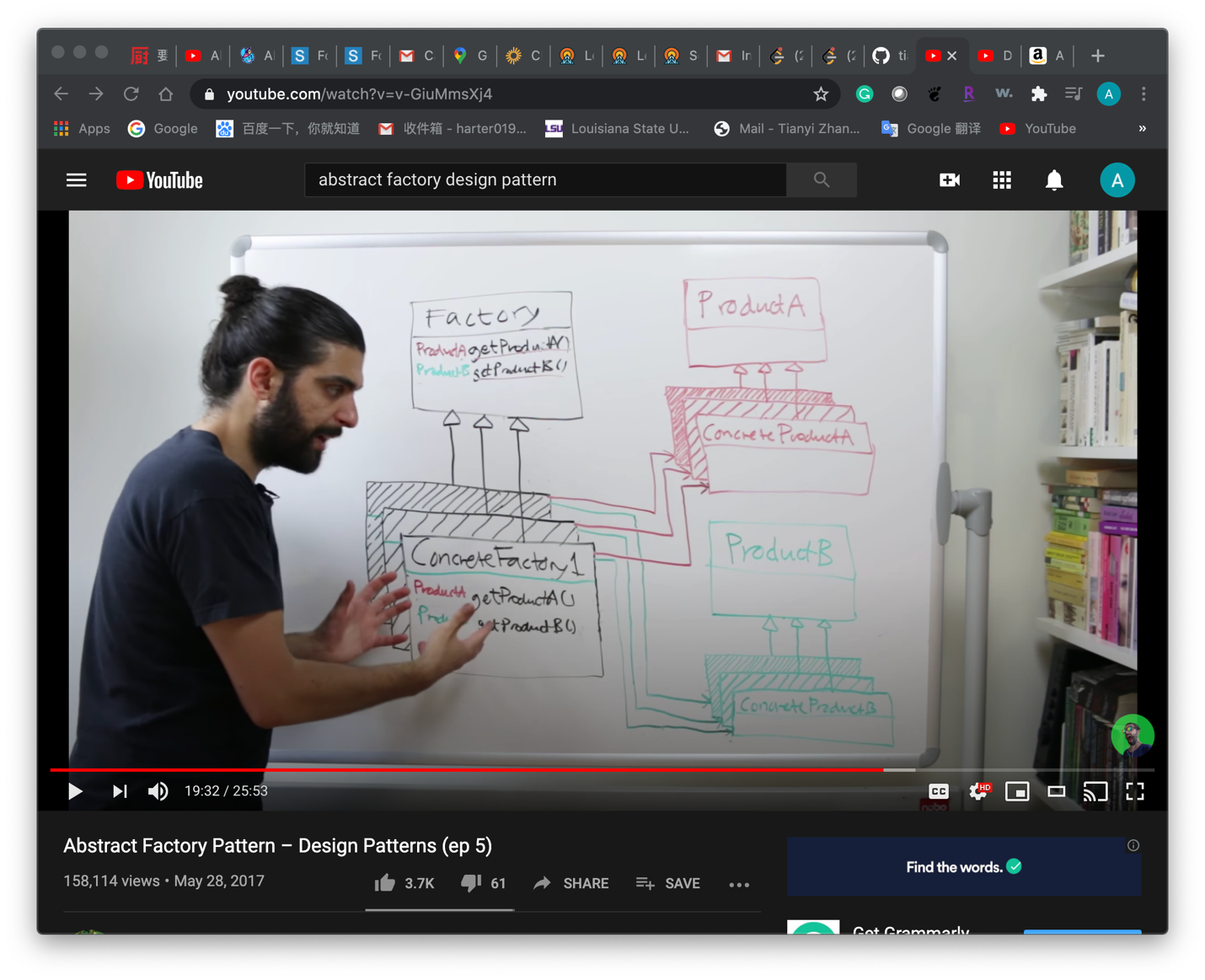
The Builder pattern provides a different way to make complex objects like you'd make using the Abstract Factory design pattern.

**Factory Method:**

You use the Factory design pattern when you want to define the class of an object at runtime. It also allows you to encapsulate object creation so that you can keep all object creation code in one place.

The Factory pattern allows you to create objects without specifying the exact class of object will be created.

**Abstract Factory Pattern:**

****

**Prototype method**

Intent: Lets you copy existing objects without making your code dependent on their classes.

The prototype provides flexibility to create complex objects cheaply. The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations.

The existing object then acts as a prototype, and the newly copied object may change the same properties only if required. This approach saves resources and time, especially when object creation is a heavy process.

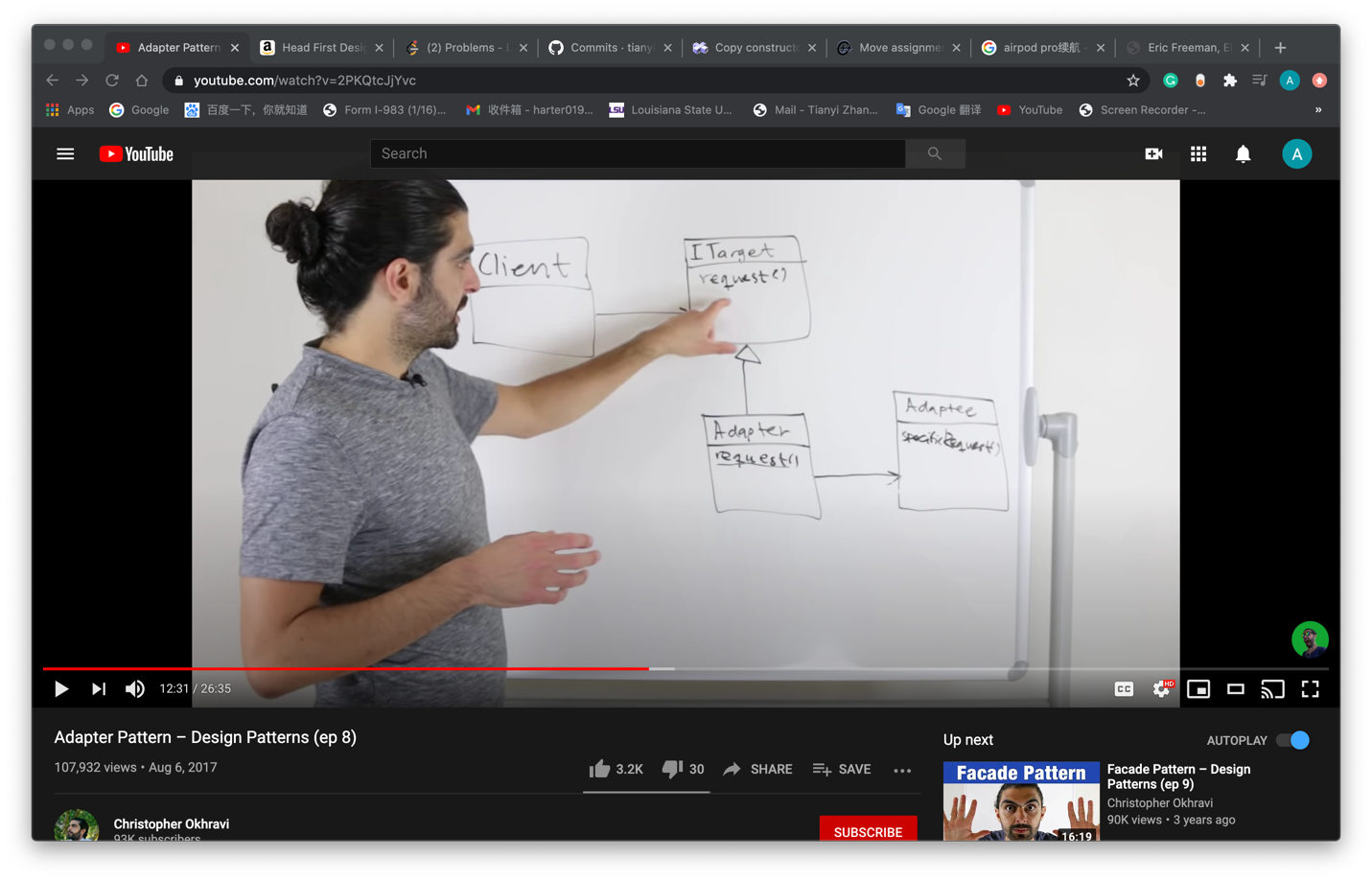
Question: Prototype design pattern to be used when creation is costly, but we do create in the clone.

You must be wondering that in Prototype Factory we show above, we are creating instances in the copy constructor. Isn't that expensive. Yes, it is. But just think about HTTP request, its header consist version, encoding type, content type, server-type, etc. Initially, you need a find out these parameters using respective function calls. But once you got these, these are not going to change until connection closed. So there is no point in doing function calls to extract these params over & over. What cost us here is not parameters but their functions to extract value.

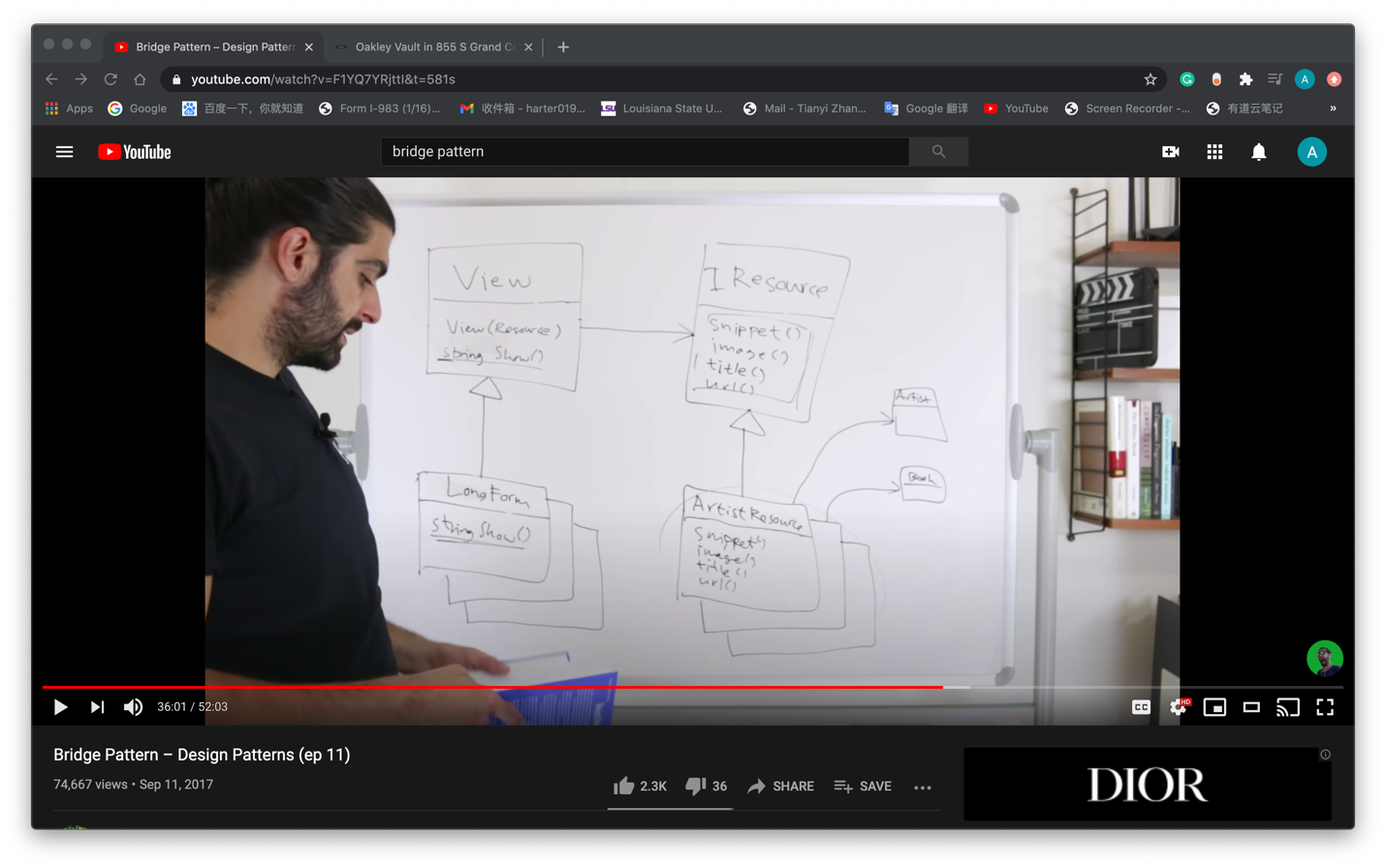
**Concept-Model Idiom**

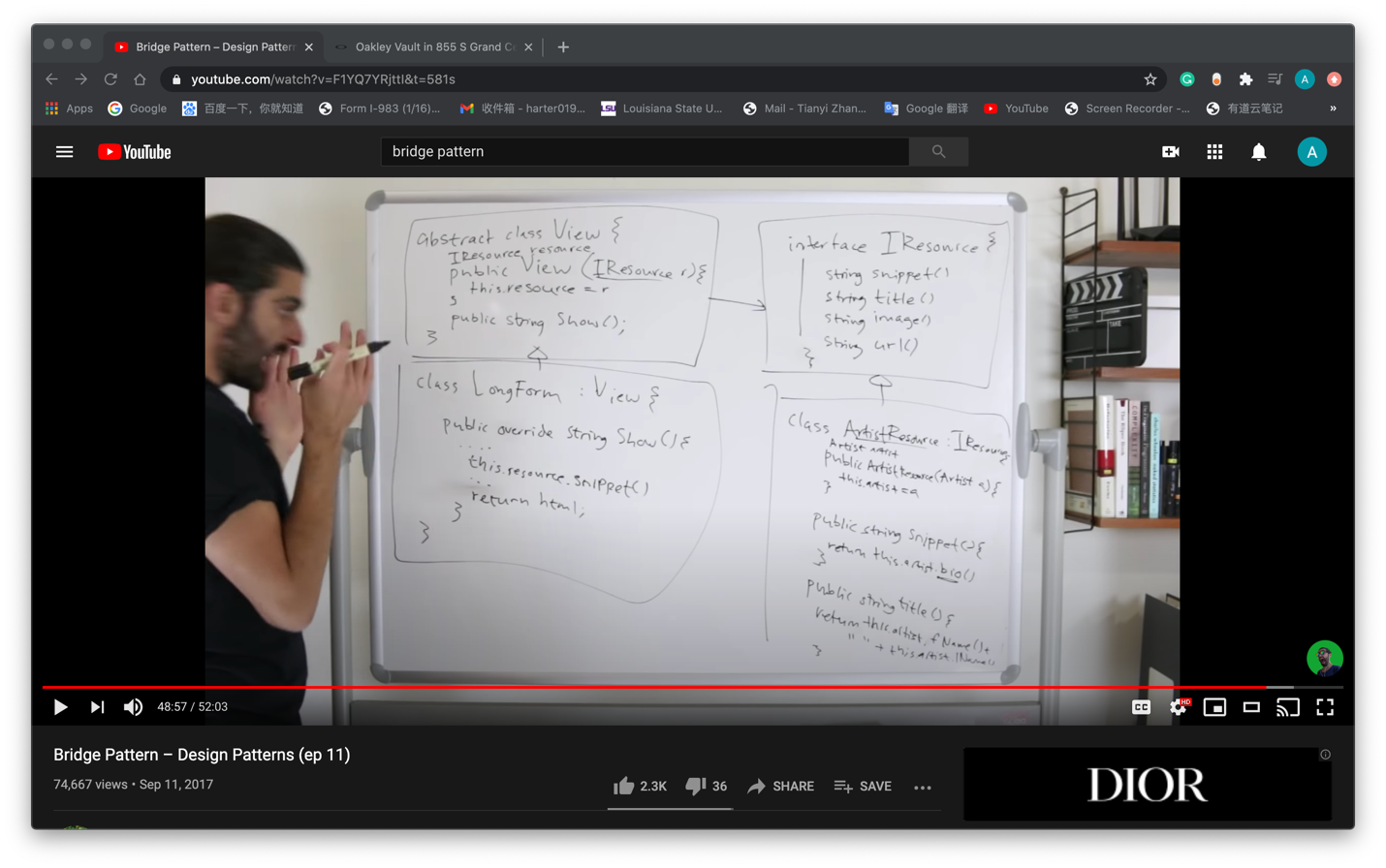
we treat our own code the same as library code. This makes task truely generic. No matter where the object code from, it just work. That class doesn’t need to know about the interface, or heap allocation, or even polymorphism. It just need to satify the task concept.

**Adapter Pattern**

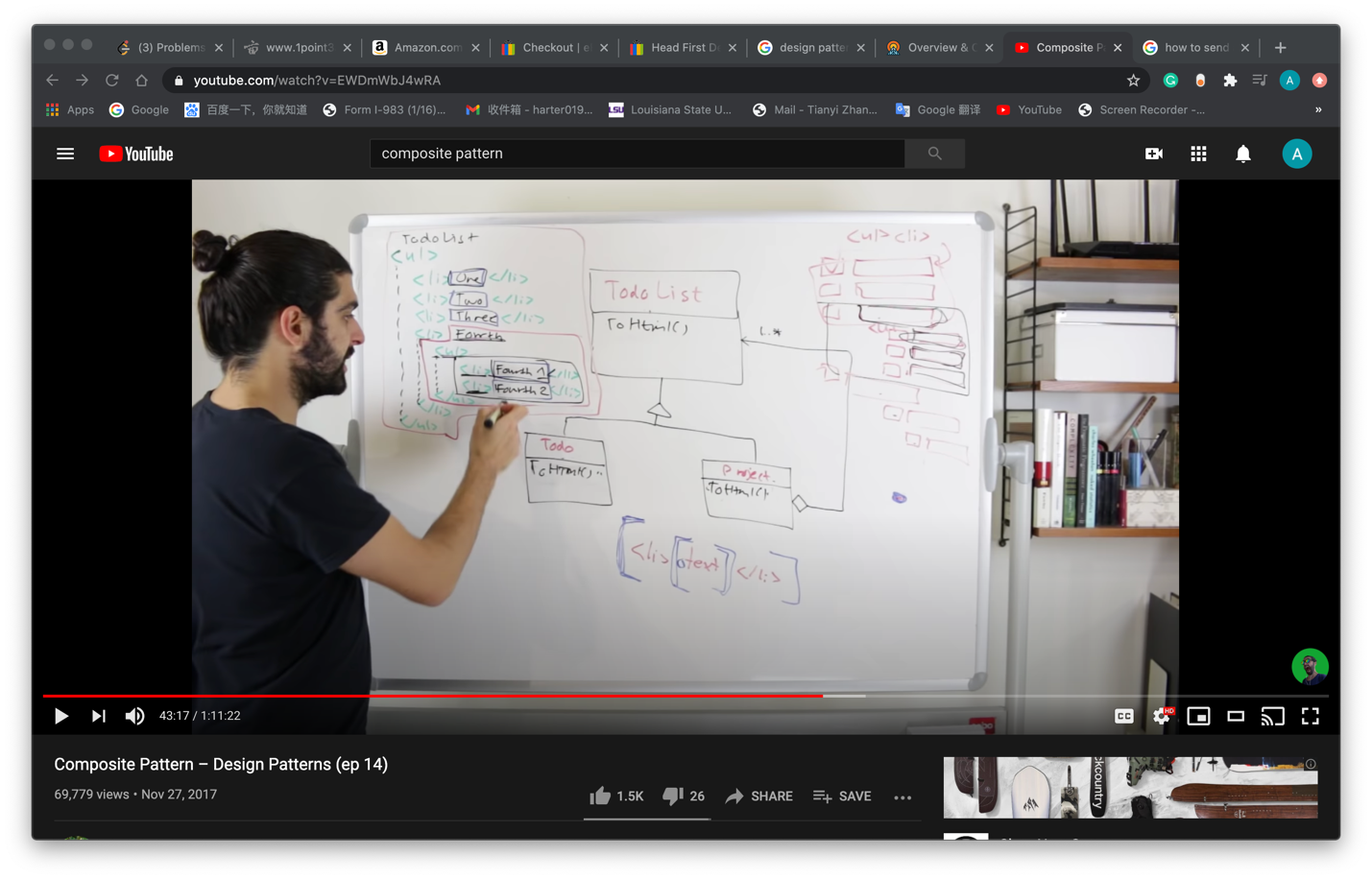


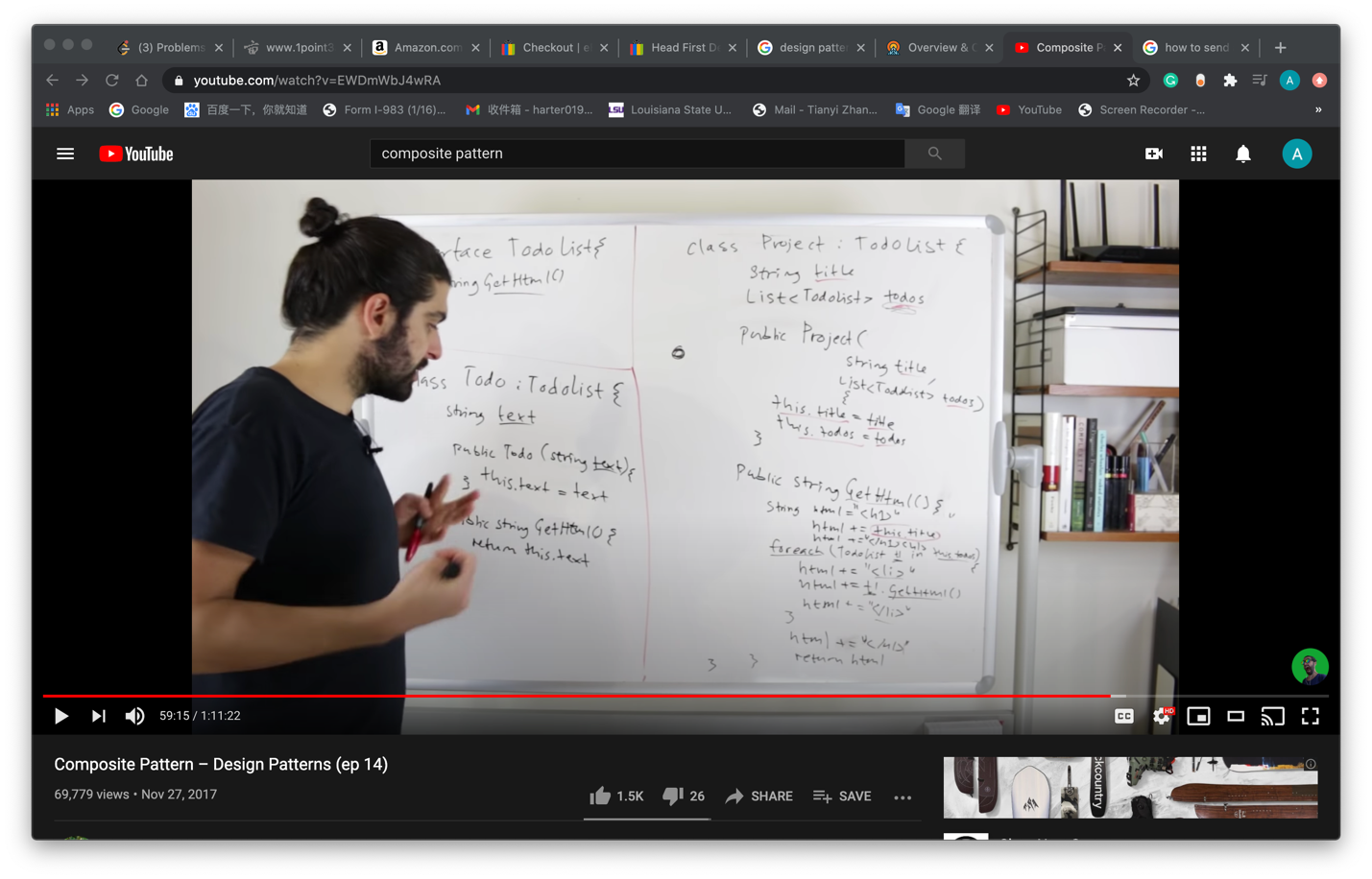
**Bridge Pattern**

****

****

**Composite Pattern**

****

****