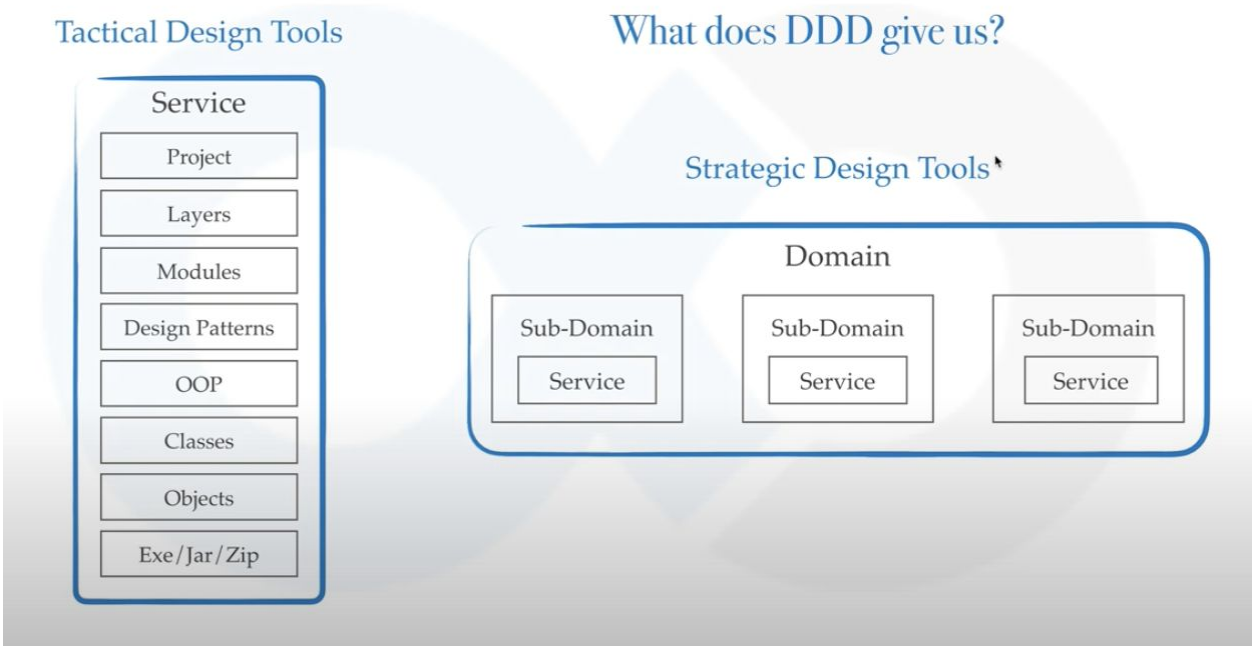
Study along the VDO series of "Mastering the art of designing Microservices Architecture" (10 short VDOs).
After your study, write down a summary of your understanding (submit a pdf file), at least covering the following topics,

Why should we focus on design?
We cannot focus just a part but we focus on overall vision and theme.



What is **Domain Driven Design**?
It is top-down software looking which means we focus on the business objectives (domain) before developing software to satisfy that. In the **domain model**, we treat software stuff as a “service” and “sub-domain” to solve the bigger problems which generally involve with business problems such as e-commerce domain, banking domain, etc. Since DDD is the way to look at things from top down which we have domain on top and DDD gives us two kinds of tools. One most important tool is “**Strategic Design Tools**”, which help to solve problems related to software modelling. Another is “**Tactical Design Tools**”, which help to solve the service problem.

Why should I learn DDD?
To improve an architecture and to make sure that your product/system is needed to the end user.

What is Strategic design?
It thinks in terms of **“Contexts”**

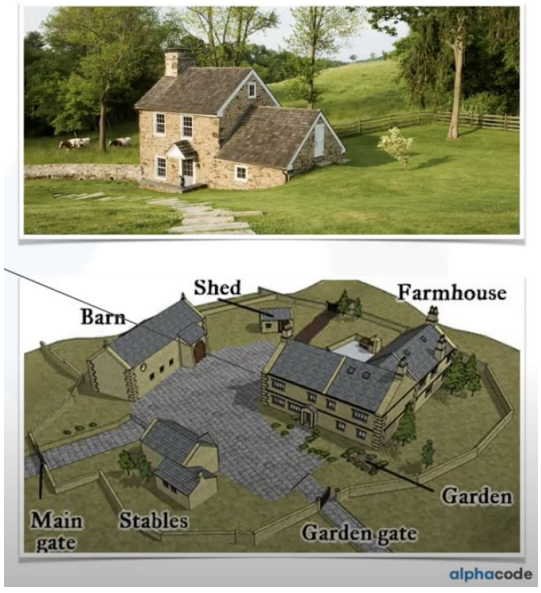
What is context?
The meaning of a word or statement depends on context. Different contexts give different meanings and decisions.

Strategic Design Tools
We focus on the important three concepts in the figure, which are **Bounded Context**, **Context Map**, and **Ubiquitous Language**. To make it simple, supposed that you are a civil engineer, need to build a house, and following these steps:

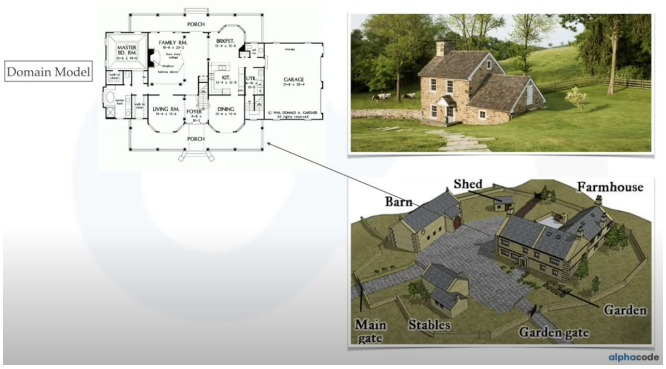
1. Definitely, you have to ask the owner which kind of house they want. We assumed that they would choose a farmhouse.
2. You will talk to the **“Domain expert” (maybe the owner itself)**

“What is a domain expert?” - It is a person who has a lot of knowledge about that domain. In this case, the domain expert has to know about the farmhouse while you only know how to build complicated infrastructure but do not know what required features of the farmhouse.

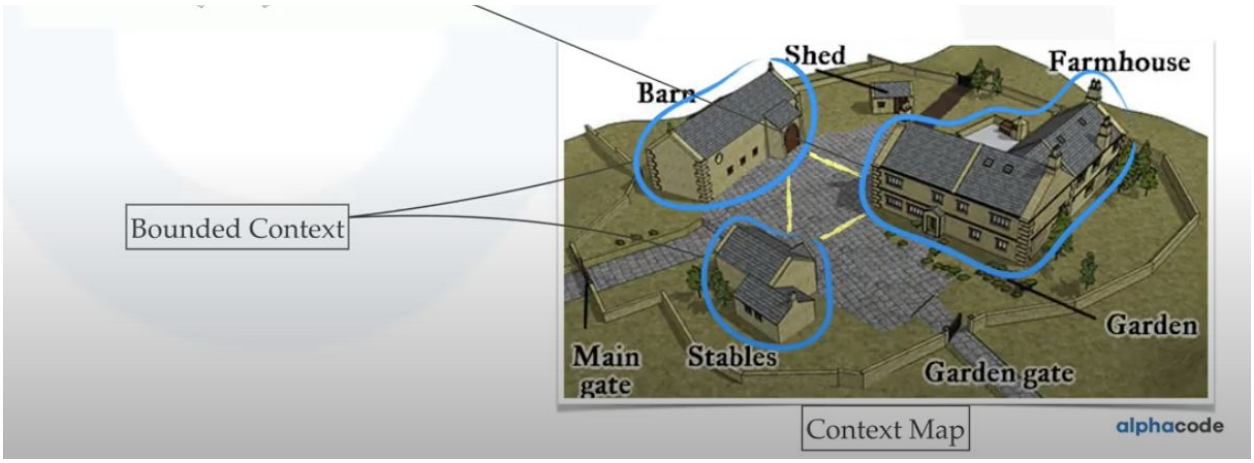
3. Then, you find out the core values . For example, the owner wants green lawns for relaxing and he wants it to be very secure or suitable for parties.
4. You see what others have done or previous work. For example, visit those farmhouses that you or the owner think is nice. And you will see what are the common patterns among those or what they have done right.
5. After you gather enough information, you have to create a mental picture of that farmhouse.



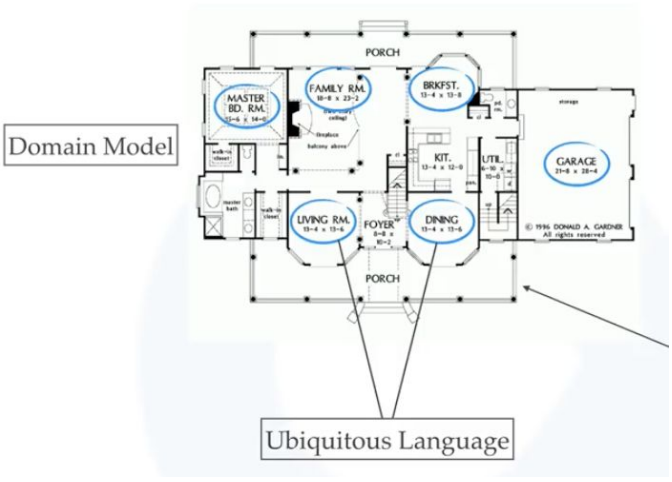
6. You will endetail those mental pictures and get the domain model. At this point our domain is farmhouse with the requirements: barn, shed, garden, gate, etc. which are known as subdomain.



As in the picture, things with blue paint are known as **bounded contexts.** These bounded contexts have relationship among each other and if you draw yellow line of how they are related to each other that is known as **context map**



Also, as you see things that are painted blue like a family room or breakfast room or whatever. All of these terms combined together, we called **ubiquitous language**.



However, we will explain those three concepts again, one by one. Start with

Ubiquitous language

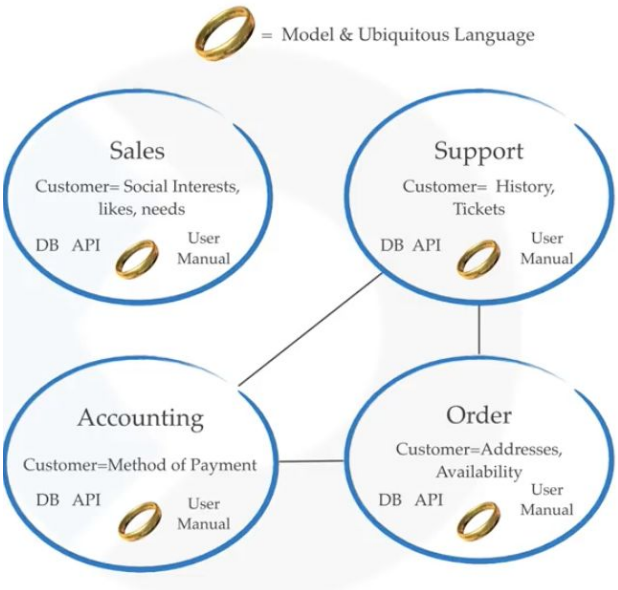
It is the official language that the team decided to use to communicate along with the same understanding between them. For example, we use the keywords we discuss as the module and class name to correspond. This language has to be not too technical, it must be general.

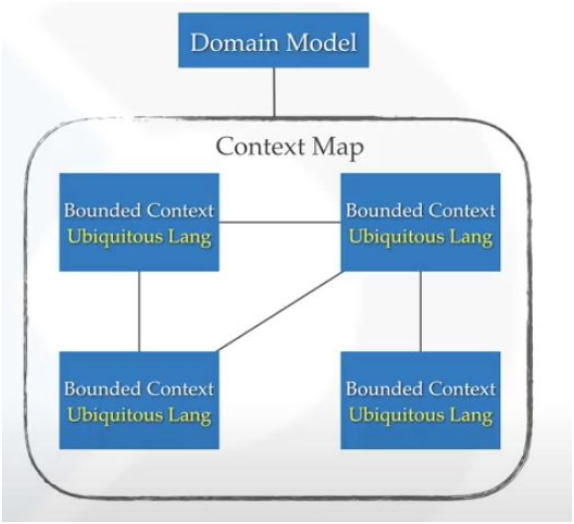
Bounded context

In the e-commerce business, there are many people with different roles. Certainly, they consider “customer” in their minds with their own model and ubiquitous language. Moreover, they have their own databases, APIs, and user manuals.

Context map

If we draw the relationship, it will be a context map showing how these contexts interact each other.

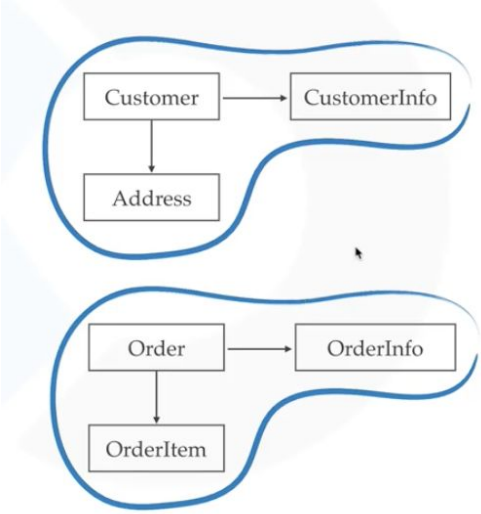




Tactical Design Tools

Value Objects
One of the best things about good design and reduces complexity and forces ubiquitous language. They do not care about uniqueness like you would like to have 100$ notes, not the particular one. Also coming with auto-validating, for example, 100 cents will be rounding to 1 buck or negative value for money.

Entities
It can be uniquely identified using an ID and consists of value objects.



Aggregates
If the object graph is too big then we need to aggregate them as a collection of entities and values under a single transaction boundary. For example, if the Address or one of Order items is changed, you have to change Customer and Order as well. So, all these have strict transactional boundaries and they have to be consistent all the time. An aggregate always has a root entity, for this figure, Customer and Order are root entities. If the root has been deleted, all entities will be gone.

Factories and Repositories
They are basically used for aggregate. For example, suppose you want to cook pancakes. Now you have all needed ingredients but you do not have a factory to create them. Those ingredients are nothing but entities and values. So, the factory helps you create new aggregates and you also have a repository helping you get persisted aggregates.
- services

Implementing Domain Driven Design

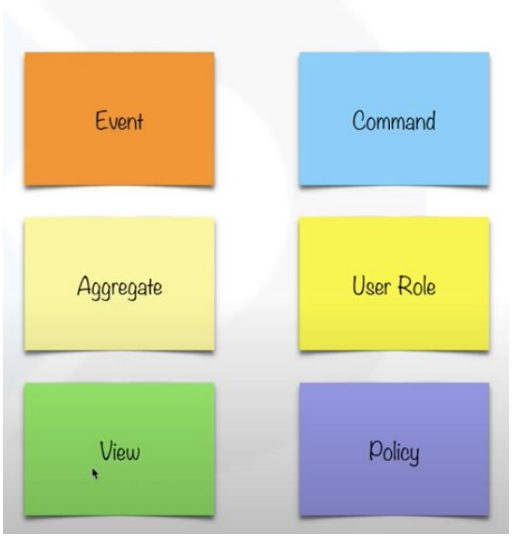
How would you discuss your understanding with domain experts?

The answer is **Event Storming** which is an exercise for creating domain Models for strategic design. To simplify, it is a brainstorming workshop among domain experts and technology people to understand the events in a system and aimed to achieve a common understanding of the domain in which software must operate.

How does event storming work?

The following will be an event storming session but not that completely one.

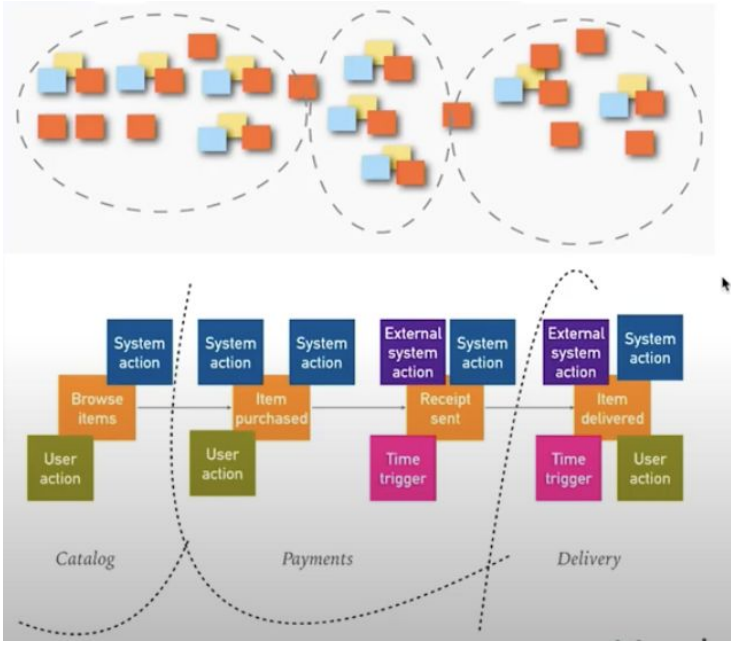
1. Bring the right people in such as domain experts and technician
2. Take sticky notes and colour code them to events, commands, policies, errors, roles, and aggregates. And all of these terms will compose your ubiquitous language.



3. Take a whiteboard and start writing interesting events in sequence. Below examples are stock orders and items in e-commerce. Notice that all the events are in the past tense which means that they have already happened. And some events are created by other events or some commands.

4. So, you have to start adding commands, aggregates, and policies.

5. Start finding your bounded context. For example, there are “Catalog Domain”, “Payments domain”, and “Delivery domain”.



Implementing DDD in code

Value Objects aka Domain Primitives: to make sure that attributes have which types to make them easy to maintain. Note that int, float, or string are called primitive for Java domain.

```
public class User {
  String name;
  String email;
  String mobile;
  ...
}


public class Product {
  int rating;
  int quantity;
  String title;
  String description;
  BigDecimal price;
  ...
}
```

So, you have to create ‘your’ primitive. For example, PhoneNumber, so you can check whether it has a maximum digit. Especially with ‘Money’ instead of BigDecimal.

```
public class User {
  Name name;
  Email email;
  PhoneNumber mobile;
  ...
}


public class Product {
  Rating rating;
  Quantity quantity;
  Title title;
  Description description;
  Money price;
  ...
}
```

You would say that it would be numerous classes to code. Anyway, this below is an example of some neat ways of code you can create with using less coding. Begin with the typical one.

```
public class ShippingController extends Controller{
  private CartService cartService;

  public void process(ShippingForm form) throws Exception, SQLException{
    CartBean cartBean = cartService.getCart(form.getCartId());
    BigDecimal cost = null;

    if (cartBean.getTotal() <= 100){
      cost = new BigDecimal(4.99);
      if (form.getOption() == 1) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(2.99));
          }
          if (i.getCat() == 'O') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
        }
      }
      if (form.getOption() == 2) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(4.99));
          }
          if (i.getCat() == 'O') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
```

```
public class ShippingController extends Controller{
  private CartService cartService;

  public void process(ShippingForm form) throws Exception, SQLException{
    CartBean cartBean = cartService.getCart(form.getCartId());
    BigDecimal cost = null;

    if (cartBean.getTotal() <= 100){
      cost = new BigDecimal(4.99);
```

Apparently, the code is not that obvious. Let the red marks make it more clearly. As you see, there are too many magic numbers. Magic number is a string literal that does not make any sense being repeatedly used like 4.99, we do not know what it is. Next, there is duplicate code in the two yellow boxes as shown. Not that obvious but as you see the same pattern of code.

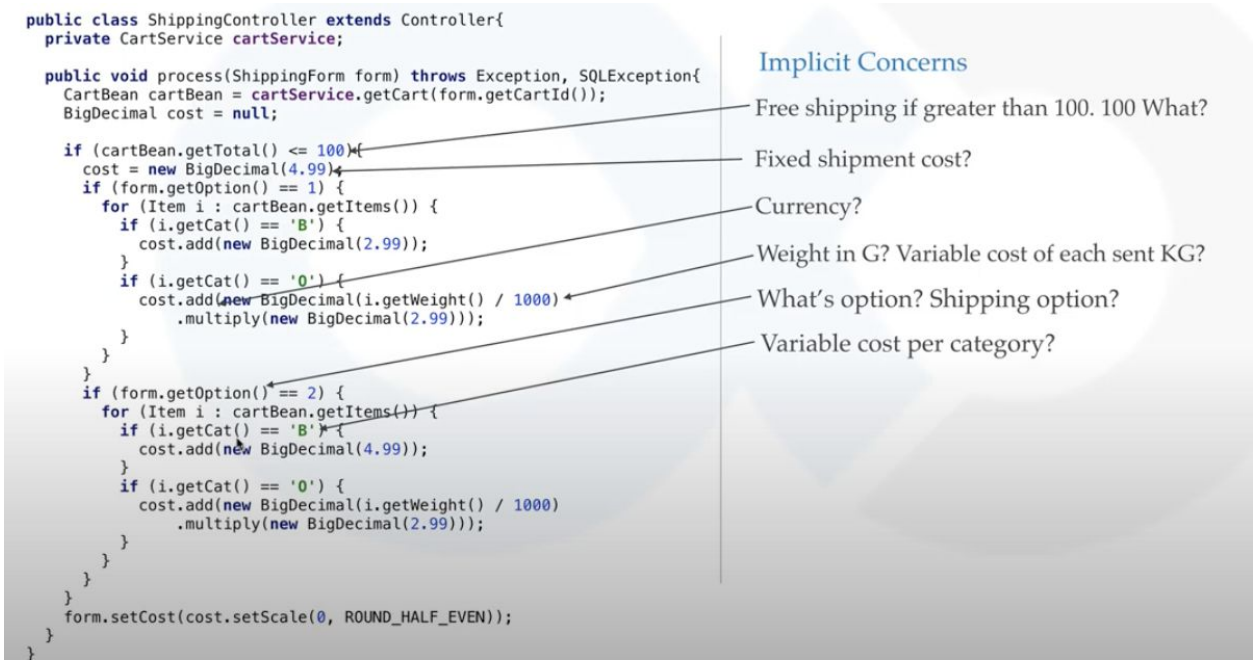
```
public class ShippingController extends Controller{
  private CartService cartService;

  public void process(ShippingForm form) throws Exception, SQLException{
    CartBean cartBean = cartService.getCart(form.getCartId());
    BigDecimal cost = null;

    if (cartBean.getTotal() <= 100){
      cost = new BigDecimal(4.99);
      if (form.getOption() == 1) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(2.99));
          }
          if (i.getCat() == 'O') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
        }
      }
      if (form.getOption() == 2) {
        for (Item i : cartBean.getItems()) {
          if (i.getCat() == 'B') {
            cost.add(new BigDecimal(4.99));
          }
          if (i.getCat() == 'O') {
            cost.add(new BigDecimal(i.getWeight() / 1000)
                .multiply(new BigDecimal(2.99)));
          }
        }
      }
    }
    form.setCost(cost.setScale(0, ROUND_HALF_EVEN));
  }
}
```

For the green marks, they are primitive obsessions as you can see how many times a ‘BigDecimal’ has repeated just in this small size of code. And the blue mark is the mixed concern where the checking SQLException is supposed to be caught somewhere else. The last one, purple marks are fuzzy terminology where using the same ‘i’, it is okay, but it is not acceptable.

Let’s see what our implicit concerns are. According to DDD, you should try to make everything in your code explicit. But implicit means you are assuming the person can read and understand the code such as what is 4.99. Therefore, it must not depend on a person to infer it out of the context.



Thus, we shall begin rewriting this as a Domain Driven Code. First, we use an intention revealing interface which calculates the shipping cost by taking the cart in and option that the person has selected. And the following conditions are almost like an ENGLISH which are understandable. You will notice why we have to create two classes (RegularShipping and PrimeShipping) instead of methods. Because everything related to which type of shipments will be in that type of class. So, if you have some constants that are used for calculation in the class, it will be in the class (like 4.99). Moreover, there might be more than one method that requires those constants. And we create a new variable (shippingCost) because this Money class is immutable.