DESIGN PRINCIPLES AND VERSION CONTROL

Prepared by : Rupak Koirala and Raj Prasad Shrestha

WEEK 3 REVIEW

HERALD COLLEGE KATHMANDU

- Activity Diagram
- Sequence Diagram
- Communication/Collaboration Diagram
- State Transition Diagram
- UML Models to Code Conversion

AGENDA

- Introduction to Design Principles
- SOLID principles
- Introduction to Design Patterns
- Introduction to Version Control
- Git
- Github, BitBucket
- Git GUI Clients

INTRODUCTION TO DESIGN PRINCIPLES

- **Object Oriented Principles:**

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction (Interface)

- OOP teaches you to construct classes, encapsulate properties and methods inside them, and also develop a hierarchy between them and use them in your code.

WHY DESIGN PRINCIPLES ?

- Writing a book?
- How to take attention of readers?
- Knowing how to construct sentences is not enough to write good essays/articles or books, right?
- You got to know how to divide the subject into smaller topics. You also need to write chapters on those topics, and you need to write prefaces, introductions, explanations, examples, and many other paragraphs in the chapters. **You need to design the overall book and learn some best practices of writing techniques.**

WHY OO DESIGN PRINCIPLES ?

HERALD COLLEGE KATHMANDU

- Similarly in the software world ,only knowing OOP programming it is almost impossible to create a software that is **re-usable, and flexible**.

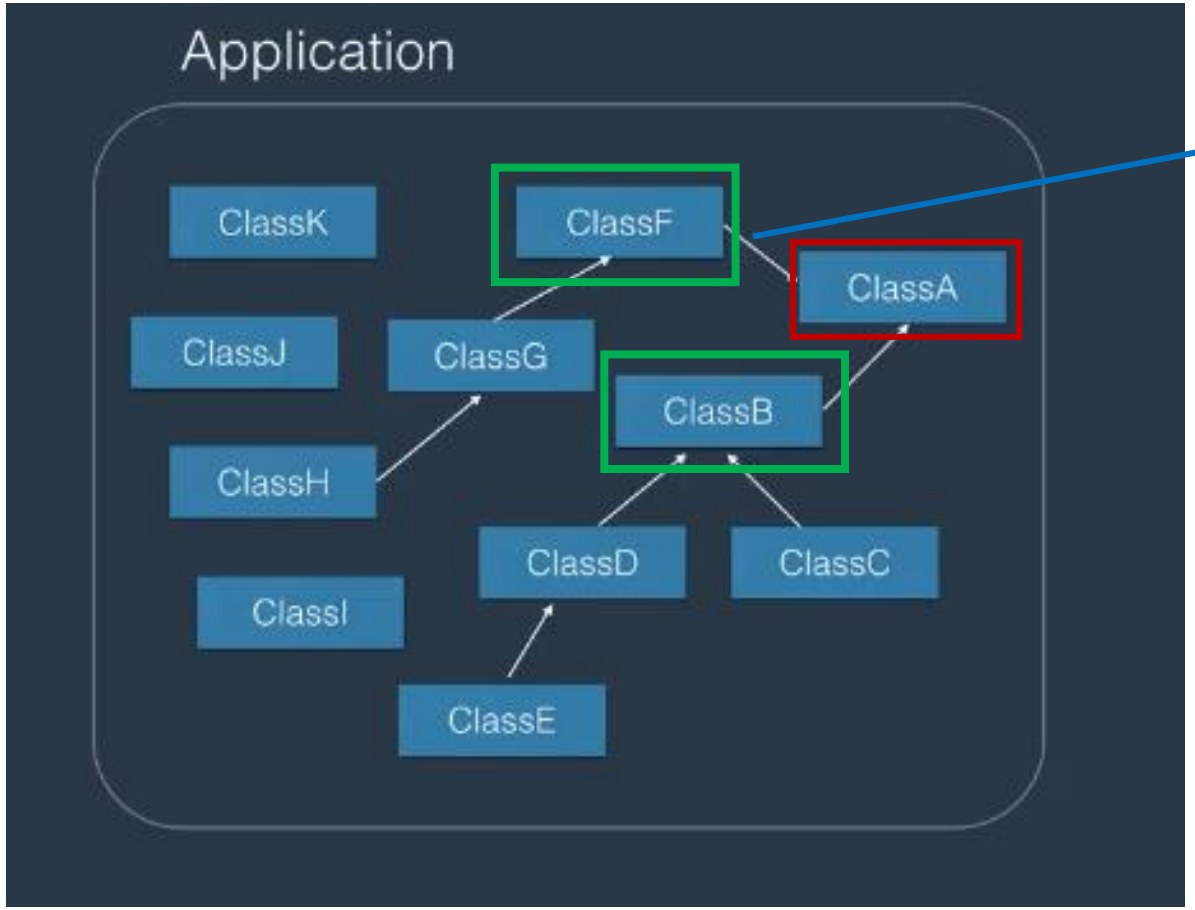
- The universal truth of software is "**your software is bound to change".**
- Because, your **software solves real life business problems and real life business processes evolve and change - always**.
- Your software does what it is supposed to do today and does it good enough. But, is your software smart enough to support "changes"?
- If not, you don't have a smartly designed software
- A smartly designed software **can adjust changes easily; it can be extended, and it is re-usable.**
- And, applying a good "**Object Oriented Design Principles**" is the key to achieve such a smart design.

ADVANTAGES OF USING OOD PRINCIPLES

- Resulting in software code
  - Easier to read
  - Easier to understand
  - Easier to change with minimal effort
  - Easier to extend without changing existing code
  - Reusable

SOME TERMS

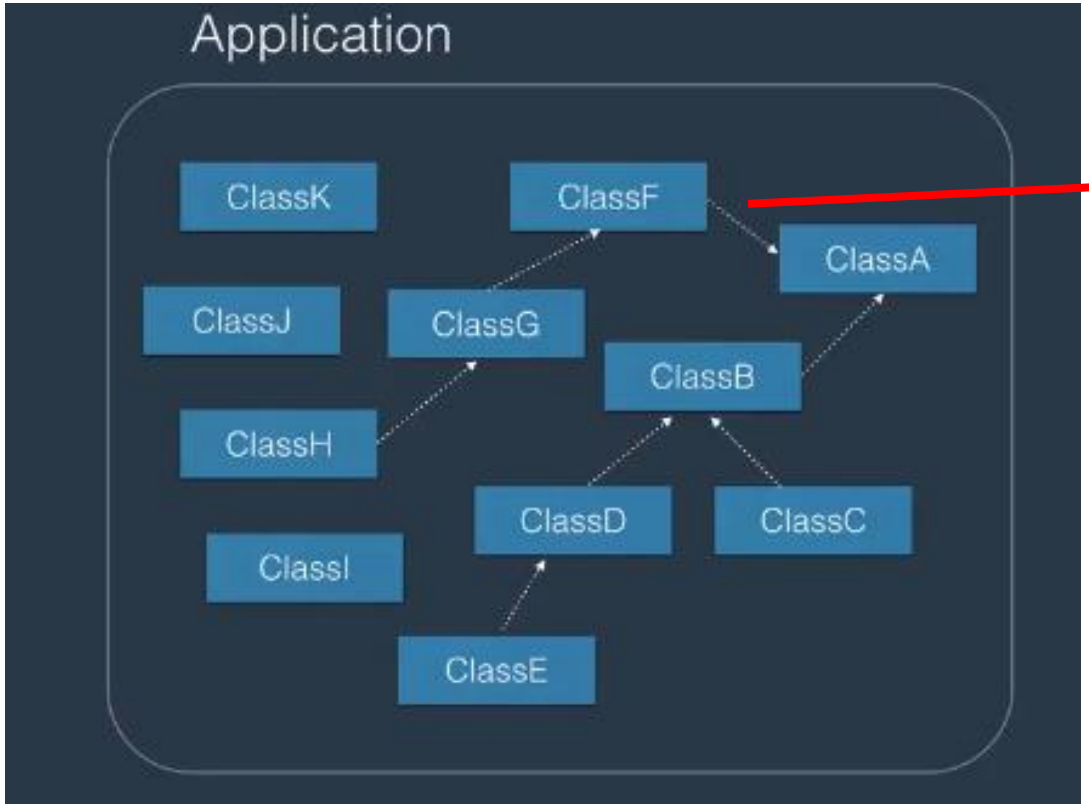
1. **Coupling**:
   - It indicates **the level of dependency between classes**.



- Arrow showing strong dependency between classes
- Assume we need to do some changes in Class A
- As classes F and B are dependent on class A
- So classes F and B also need to be changed or recompiled and redeployed
- **Tightly coupled**

SOLUTION FOR HIGH COUPLING

- **Need of low coupling** so that when we make some changes in class A the changes are isolated (i.e no need to make changes in other classes)
- Using **Abstraction (Interface)**



- Loosely coupled

Advantage:

- Same advantages of OODP priniciples

SOME TERMS

2. **Cohesion**:
   - It is a measure of **how related the responsibilities of a class** are.

```
class DownloadAndStore{
    void downloadFromInternet(){
    }

    void parseData(){
    }

    void storeIntoDatabase(){
    }

    void doEverything(){
        downloadFromInternet();
        parseData();
        storeIntoDatabase();
    }
}
```

Responsibility -1: Downloading data from the internet.

Responsibility -2: Parsing data

Responsibility -3: Save parsed data into database

Problem: This class is having range of responsibilities (unrelated responsibilities) – not a high cohesive class

SOLUTION FOR NOT BEING A HIGH COHESIVE CLASS

- It is better to split the class into multiple classes based upon their responsibility.

```
class InternetDownloader{
void downloadFromInternet(){
    //HTTP
}
   class Parser{
        void parseData() {
            // Regular
            // Jsoup
        }
    }
class DummyDao{
    void storeIntoDatabase() {
        // Jdbc //hibernate
    }
}
```

Responsibility -1: Downloading data from the internet in class Internet Downloader

Responsibility -2: Parsing data in class Parser

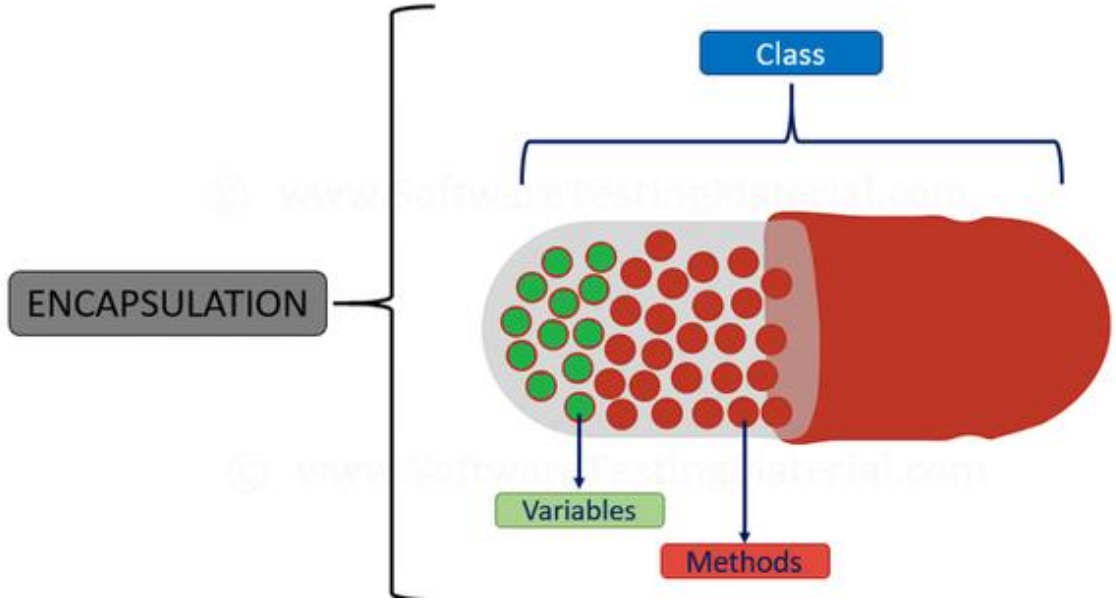
Responsibility -3: Save parsed data into database in DummyDao class

Advantage:
Same advantages of OODP priniciples

SOME TERMS

3. **Encapsulation:**

- Binding object state's data with methods.



OBJECT ORIENTED DESIGN PRINCIPLES

HERALD COLLEGE KATHMANDU

GOALS = LOW COUPLING, HIGH COHESIVE AND STRONG ENCAPSULATION

- Fortunately, others have created stepping stones that lead to these goals.
- Among many design principles out there, there are **five principles** which are abbreviated as the SOLID principles.
- SOLID principles is originally compiled by Robert C. Martin (Uncle Bob) in the 1990s.

Robert C. Martin

SOLID PRINCIPLES

HERALD COLLEGE KATHMANDU

SRP	The Single Responsibility Principle	*A class should have one, and only one, reason to change.*
OCP	The Open Closed Principle	*You should be able to extend a classes behavior, without modifying it.*
LSP	The Liskov Substitution Principle	*Derived classes must be substitutable for their base classes.*
ISP	The Interface Segregation Principle	*Make fine grained interfaces that are client specific.*
DIP	The Dependency Inversion Principle	*Depend on abstractions, not on concretions.*

S- SINGLE RESPONSIBILITY PRINCIPLE

"A class should have one and only one responsibility".

- One can clearly see that this product has not just one single responsibility.
- There are several responsibilities assembled in a single unit. We have 2 knifes, one can opener, one bottle opener, an awl and a corkscrew.
- Violation of Single Responsibility priniciple

S- SINGLE RESPONSIBILITY PRINCIPLE(SRP)

For example:

Let's take the class A which does the following operations:

1. Open a database connection
2. Fetch data from database
3. Write the data in an external file

Class A violates SRP !

S- SINGLE RESPONSIBILITY PRINCIPLE(SRP)

- **Problems**:
  - Class A handles lot of operations.

    Suppose any of the following change happens in future like :

    1. New database
    2. Adopt ORM to manage queries in the database
    3. Change in output structure
- So in all the cases the above class would be changed.
- Which might affect the implementation of the other two operations as well.
- **Solution**:
  - So ideally according to SRP there should **be three classes each having the single responsibility.**

O- OPEN/CLOSED PRINCIPLE

"Classes should be open for extension but closed for modification".

- **Open for extension and closed for modification:**
  -> open for extensions means that the behavior of the class can be extended without modifying the class.
- It can be achieved by **Abstraction.**

O- OPEN/CLOSED PRINCIPLE EXAMPLE

- Shipped phone comes with the basics like camera operation, actual calls, text messages, etc.
- But via the app store, you can *extend* the phone's capabilities to allow you to manage your to do list etc.
- Your phone's core system(Kernel OS) code are closed for the modification.

EXAMPLE

```
public double Area(object[] shapes)

    double area = 0;

    foreach (var shape in shapes)
        if (shape is Rectangle) {
                Rectangle rectangle = (Rectangle) shape;
                area += rectangle.Width*rectangle.Height;
        } else {
                Circle circle = (Circle)shape;
                area += circle.Radius * circle.Radius * Math.PI;
        }
    return area;
```

Violation of OCP!!

- Adding more shapes (Square etc)requires you to modify Area method code.

SOLUTION- ABSTRACTION (ALLOWING EACH OF THE SHAPE TO DEFINE THE AREA METHOD)

HERALD COLLEGE KATHMANDU

```
public abstract class Shape
{
    public abstract double Area();
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}
```

Open for extension

```
public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}
```

Closed for modification

```
public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }
    return area;
}
```

L – LISKOV SUBSTITUTION PRINCIPLE

HERALD COLLEGE KATHMANDU

"Derived classes must be substitutable for their base classes".

EXAMPLE

The "Liskov's Substitution Principle" is just a way of ensuring that **inheritance** is used correctly

- Ostrich is a Bird (definitely it is!) and hence it inherits the Bird class. Now, can it fly?

Ostrich

This violates LSP!!

KingFisher

SOLUTION

- So, even if in real world this seems natural, in the class design, Ostrich should not inherit the Bird class, **and there should be a separate class for birds that can't really fly and Ostrich should inherit that**.

NonFlyingBird

Ostrich

FlyingBird

KingFisher

I – INTERFACE SEGREGATION PRINCIPLE

"Clients should not be forced to depend upon interfaces that they do not use".

Shubho: Sure, here is your explanation:

Suppose you want to purchase a television and you have two to choose from. One has many switches and buttons, and most of those seem confusing and doesn't seem necessary to you. Another has a few switches and buttons, which seems familiar and logical to you. Given that both televisions offer roughly the same functionality, which one would you choose?

Farhana: Obviously the second one with the fewer switches and buttons.

Shubho: Yes, but why?

Farhana: Because I don't need the switches and buttons that seem confusing and unnecessary to me.

Shubho: Correct. Similarly, suppose you have some classes and you expose the functionality of the classes using interfaces so that the outside world can know the available functionality of the classes and the client code can be done against interfaces. Now, if the interfaces are too big and have too many exposed methods, it would seem confusing to the outside world. Also, interfaces with too many methods are less re-usable, and such "fat interfaces" with additional useless methods lead to increased coupling between classes.

This also leads to another problem. If a class wants to implement the interface, it has to implement all of the methods, some of which may not be needed by that class at all. So, doing this also introduces unnecessary complexity, and reduces maintainability or robustness in the system.

The Interface Segregation principle ensures that Interfaces are developed so that each of them have their own responsibility and thus they are specific, easily understandable, and re-usable.

I – INTERFACE SEGREGATION PRINCIPLE

- Interfaces should contain only the necessary methods and not anything else.

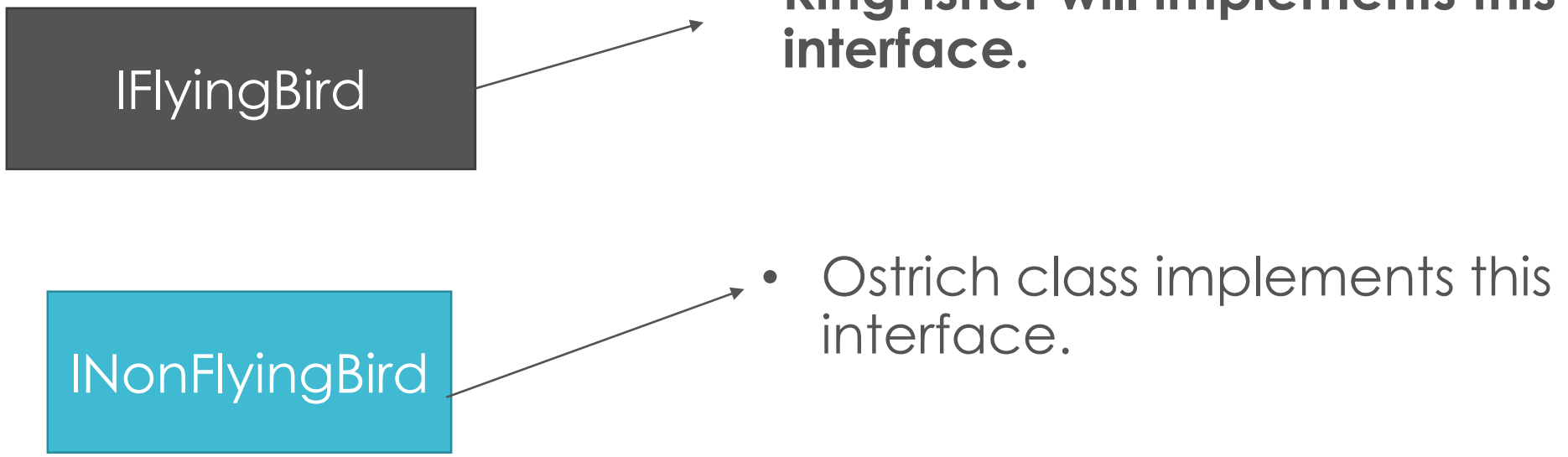
- This interface has many bird behaviors along **with fly () method.**

- Say Ostrich class implements the IBirdInterface, it has to implement the **fly() method unnecessarily(Ostrich doesn't fly)**

This violates ISP!!!

SOLUTION:

- Split the interface into two different interfaces, INonFlyingBird and IFlyingBird.

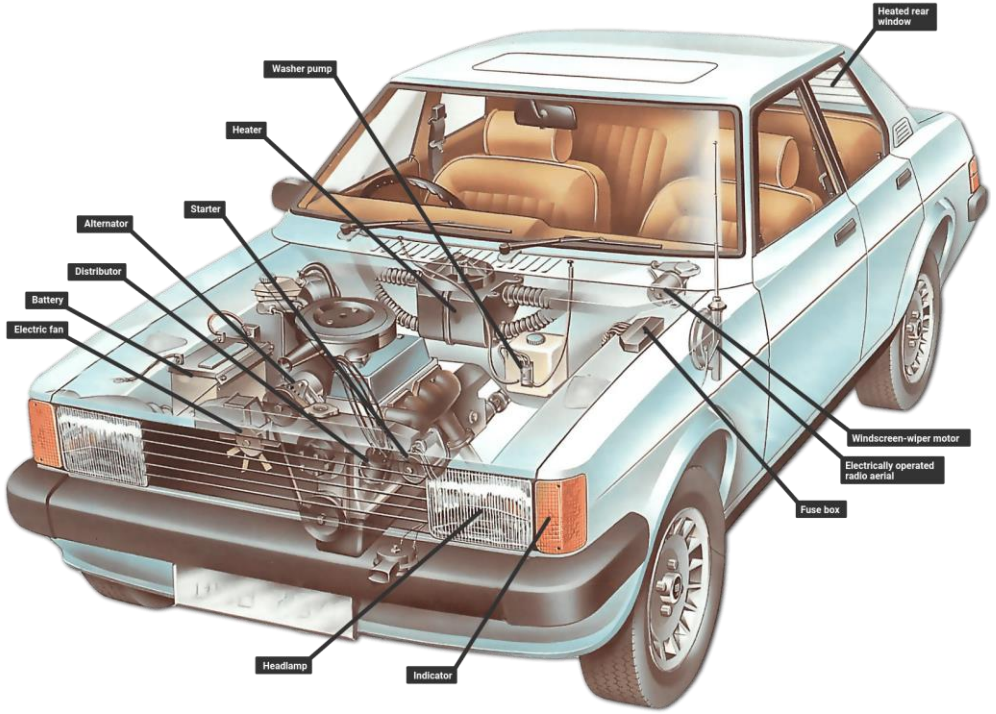


- **KingFisher will implements this interface.**
- Ostrich class implements this interface.

D – **DEPENDENCY INVERSION PRINCIPLE**

HERALD COLLEGE KATHMANDU

"**High level modules should not depend upon low level modules. Rather, both should depend upon abstractions.**"

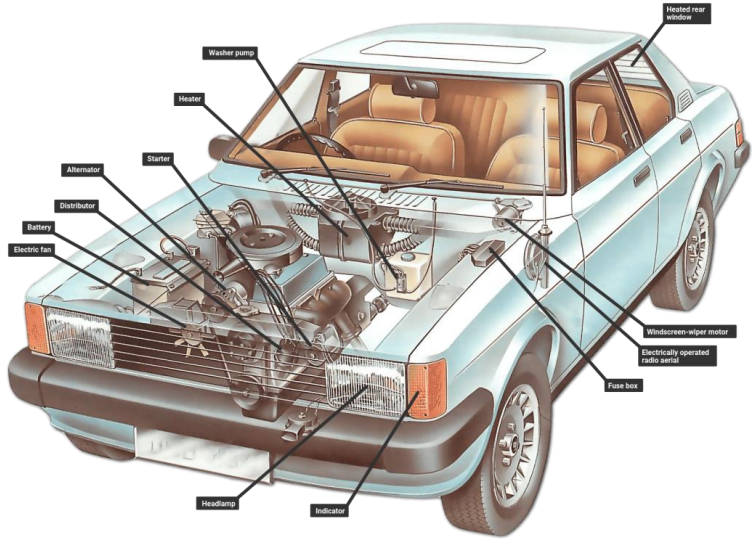


- Car is composed of lots of objects like the engine, the wheels, the air conditioner, and other things
- None of these things are rigidly built within a single unit; rather, each of these are "**pluggable**" so that when the **engine or the wheel has problem, you can repair it** (without repairing the other things) and you can even replace it.

EXAMPLE

- "While replacement, you just have to ensure that the engine/wheel conforms to the car's design (say, the car would accept any 1500 CC engine and will run on any 18 inch wheel).
- Also, the car might allow you to put a 2000 CC engine in place of the 1500 CC, given the fact that the manufacturer (say, Toyota) is the same.

Depend upon abstractions(interfaces) rather than concrete classes.
Use Interface
Main aim => loosely coupled



EXAMPLE:

Low level class

```
class Developer {
        public void work(){
                //....
        }
}
```

```
class Architect{
        public void work(){
                //....
        }
}
```

High level Class

```
class Manager{
        Developer dev;
        Architect archi;
        public void addEmployee(Developer dev, Architect archi){
                //...
        }
}
```

This violates DIP!!
If you want to add "Tester" you have to modify Manager class also .

SOLUTION:

Low level class

```
interface Employee {
        public void work(){
                //....
        }
}
```

```
class Architect implements Employee{
        public void work(){
                //....
        }
}
```

```
class Developer
implements Employee{
        public void work(){
                //....
        }
}
```

High level Class

```
class Manager{
        Employee emp;
        public void addEmployee(Employee emp){
                //…
        }
}
```

If you want to add "Tester" you don't need to modify Manager class now .

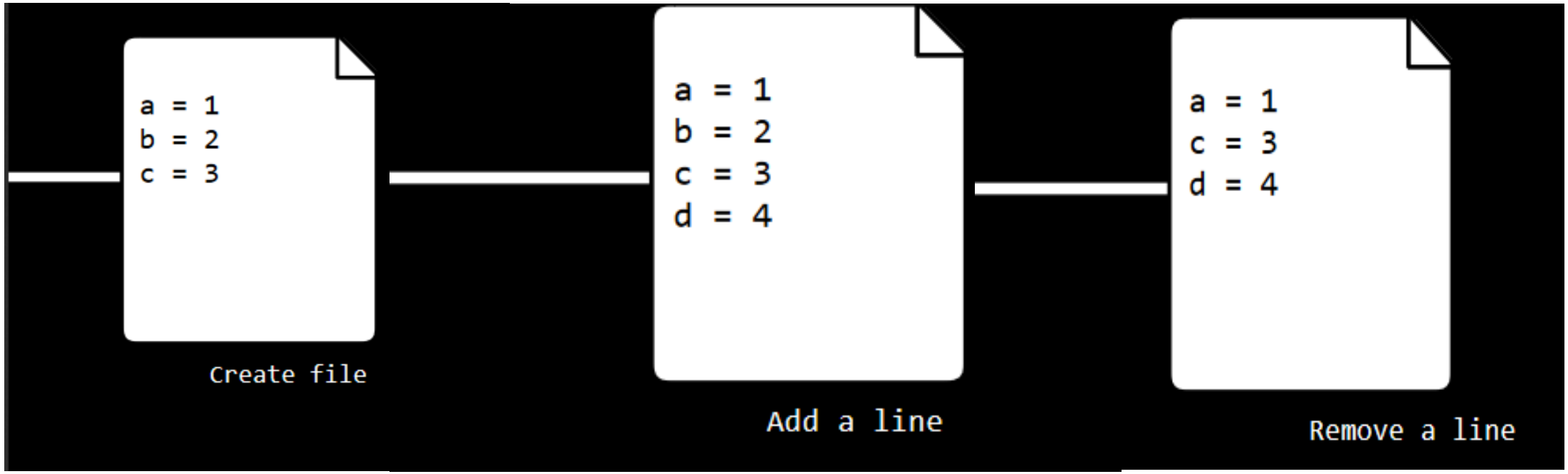
INTRODUCTION TO VERSION CONTROL

How to make sure that:

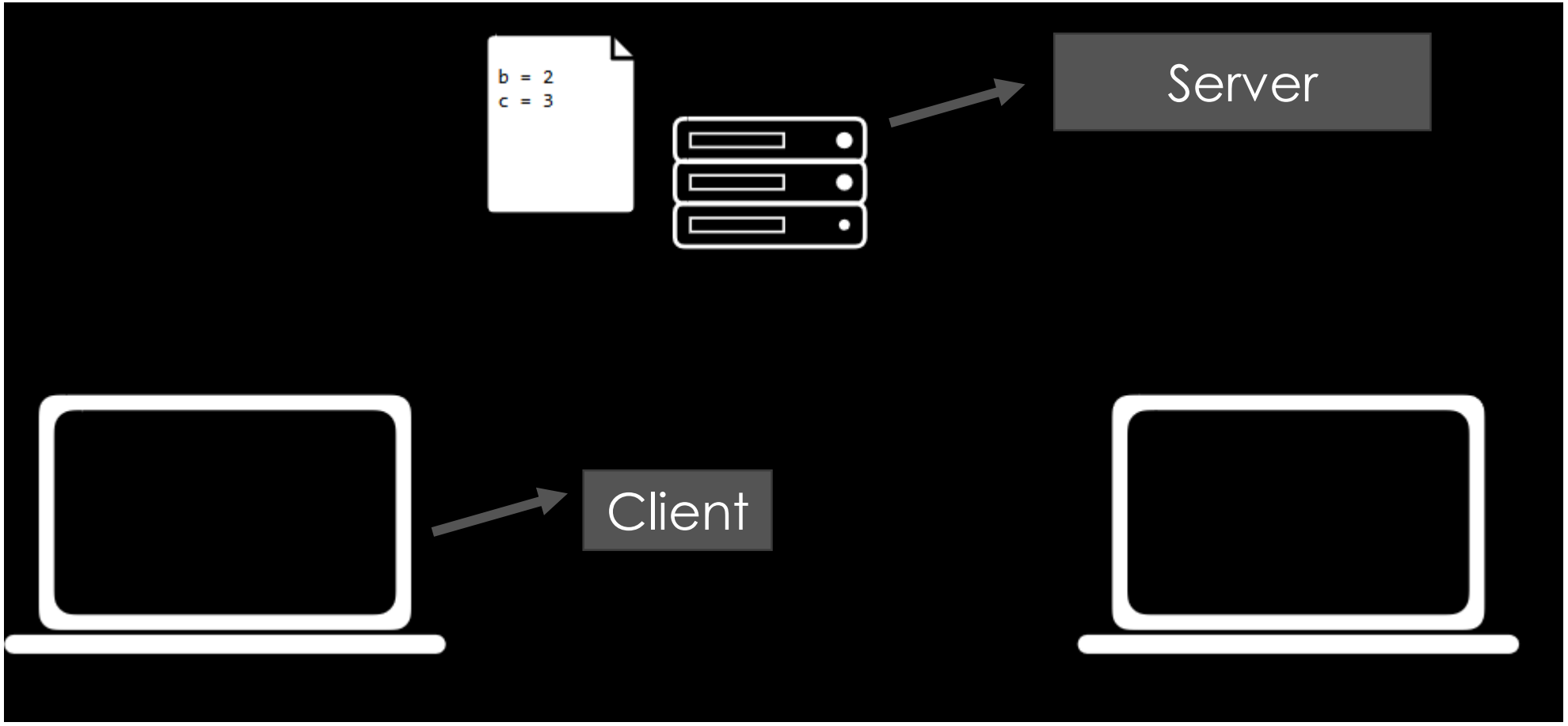
1. Keep track of different versions of code
2. Share/Collaborate the code with people
3. Test your code without losing the original code

- Git is one of the popular version control software/tool.

GIT - KEEPING TRACK OF CHANGES TO CODE

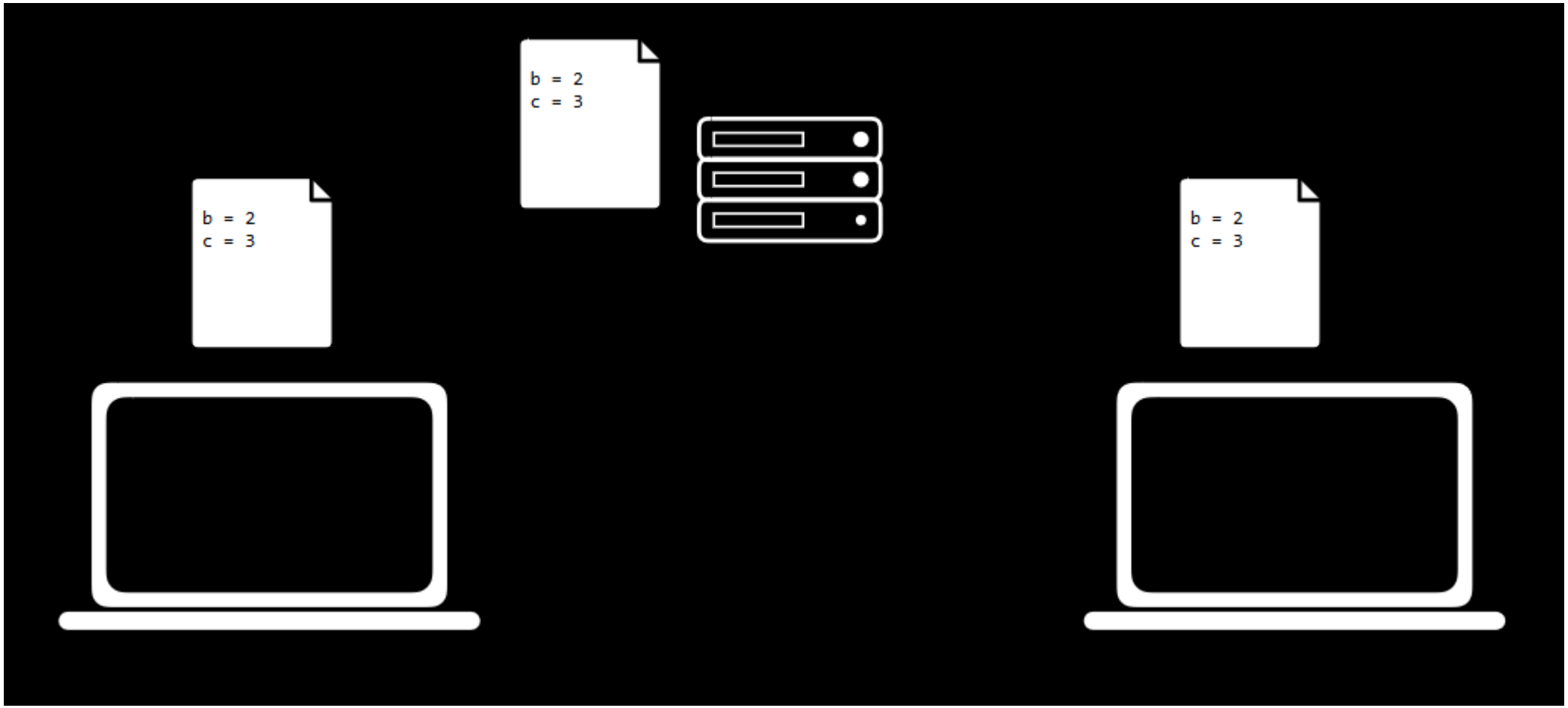


GIT- SYNCHRONIZES CODE BETWEEN DIFFERENT PEOPLE

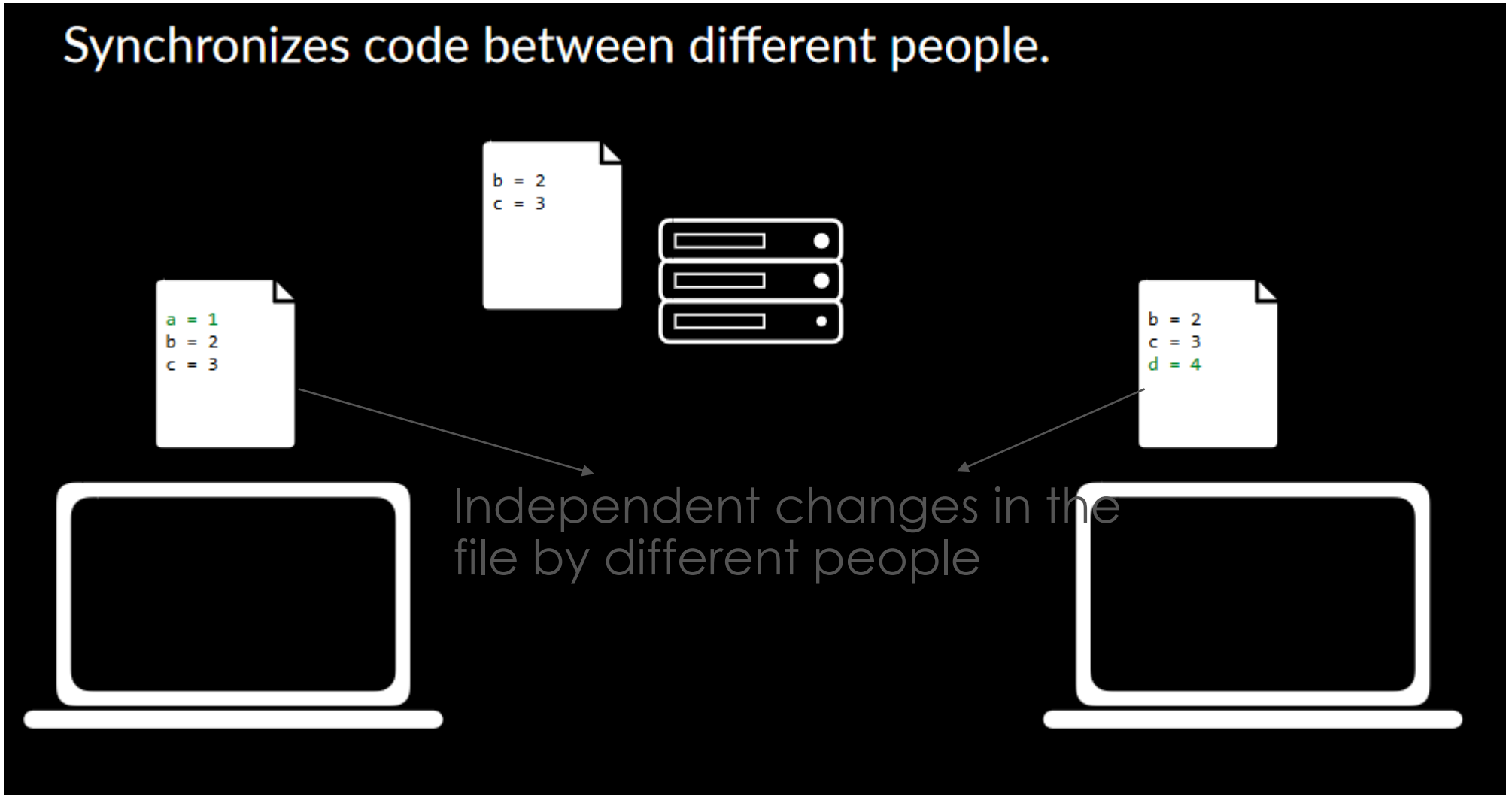


GIT- SYNCHRONIZES CODE BETWEEN DIFFERENT PEOPLE

HERALD COLLEGE KATHMANDU

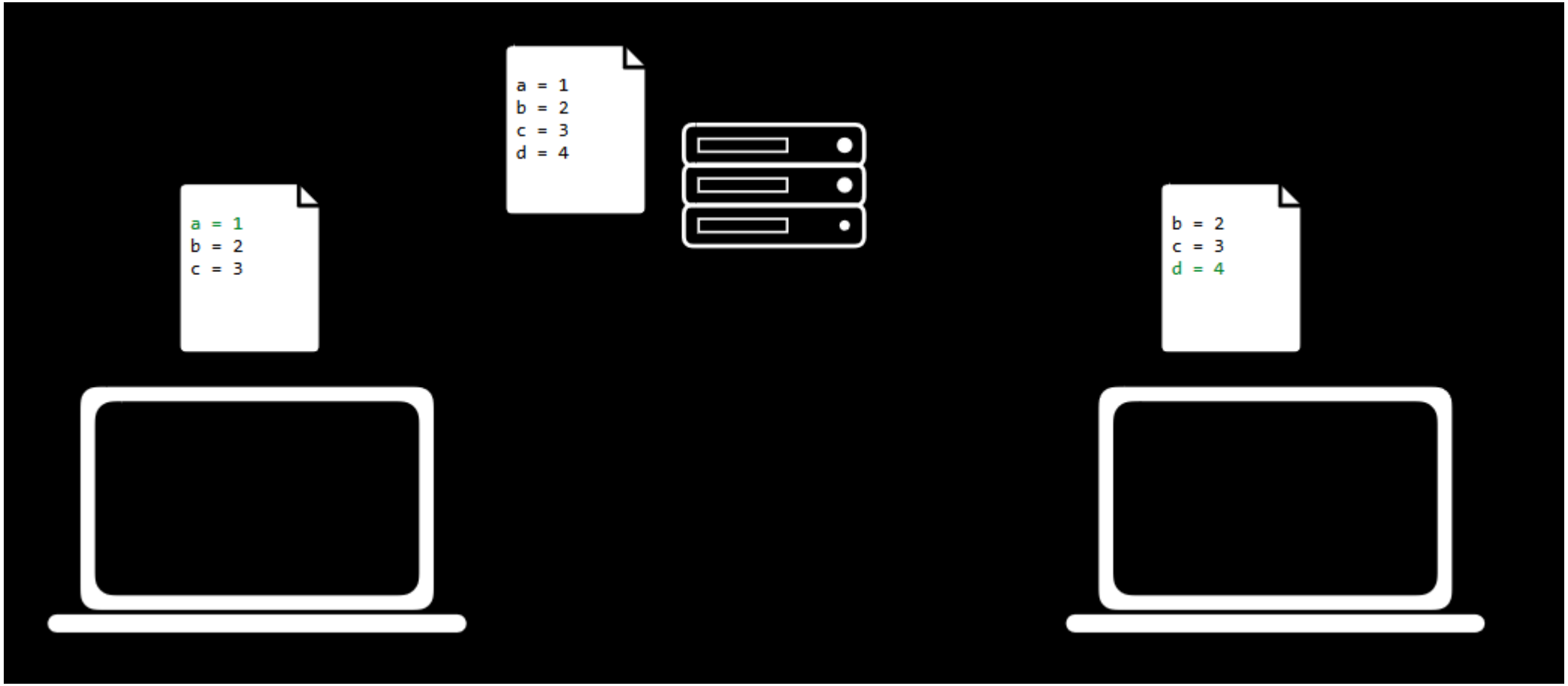


GIT- SYNCHRONIZES CODE BETWEEN DIFFERENT PEOPLE

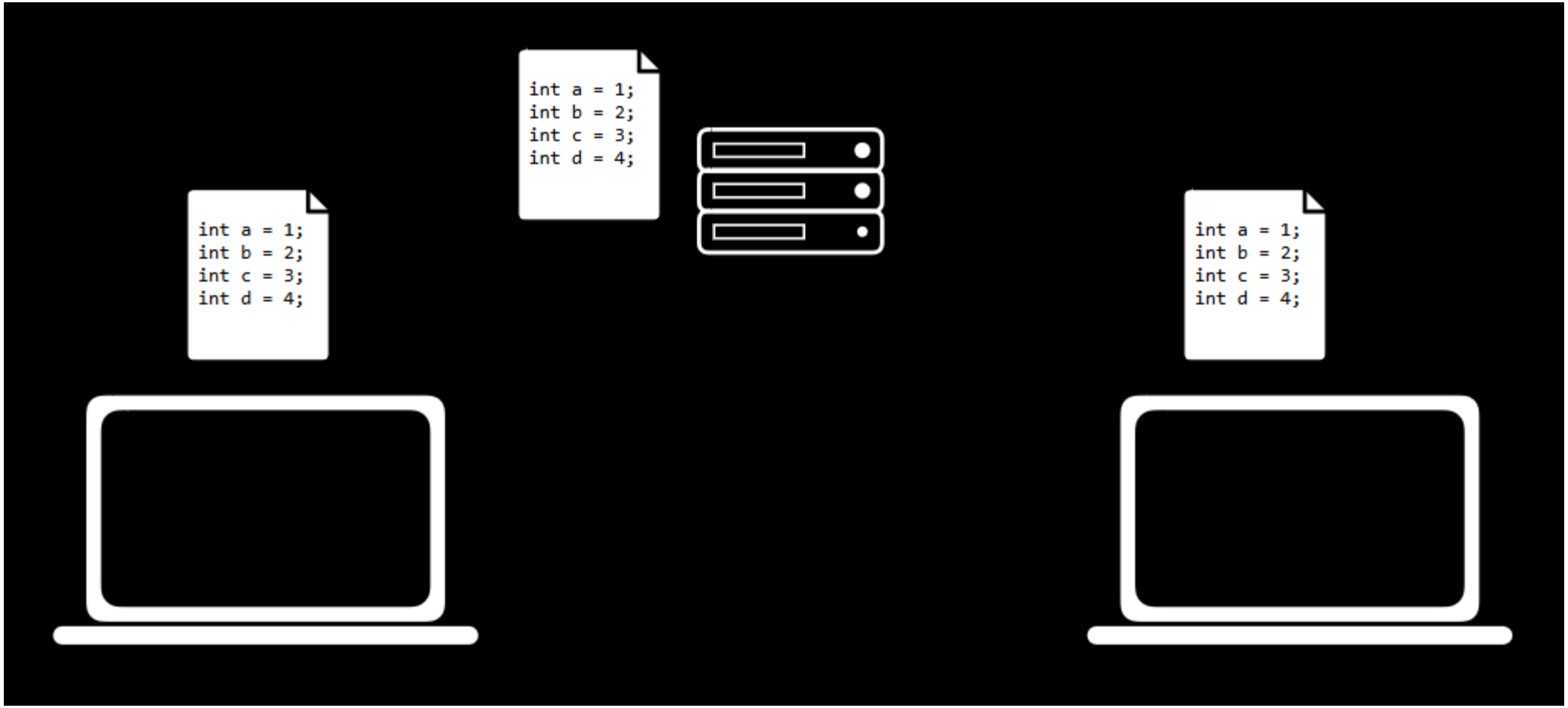


GIT- SYNCHRONIZES CODE BETWEEN DIFFERENT PEOPLE

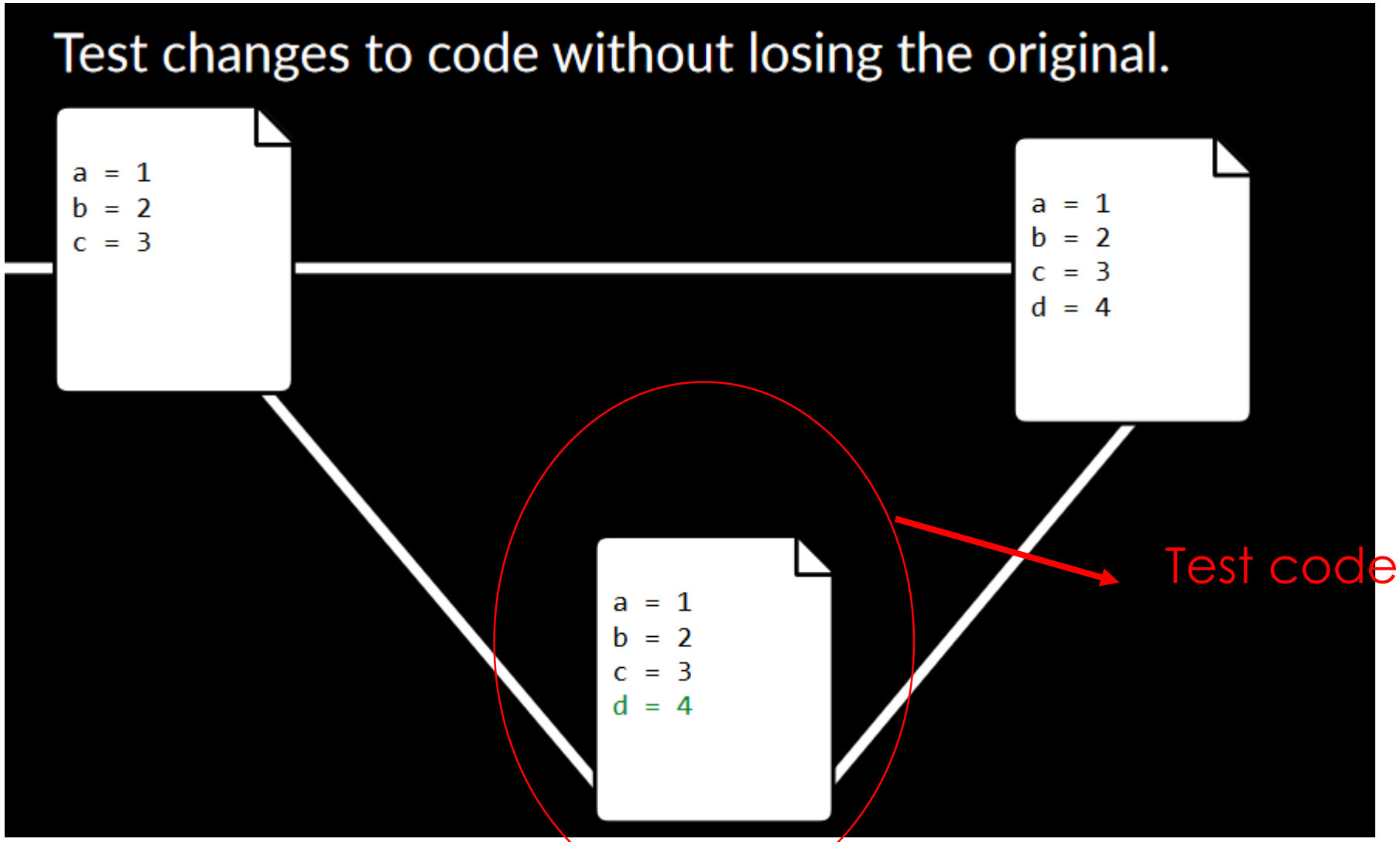
HERALD COLLEGE KATHMANDU



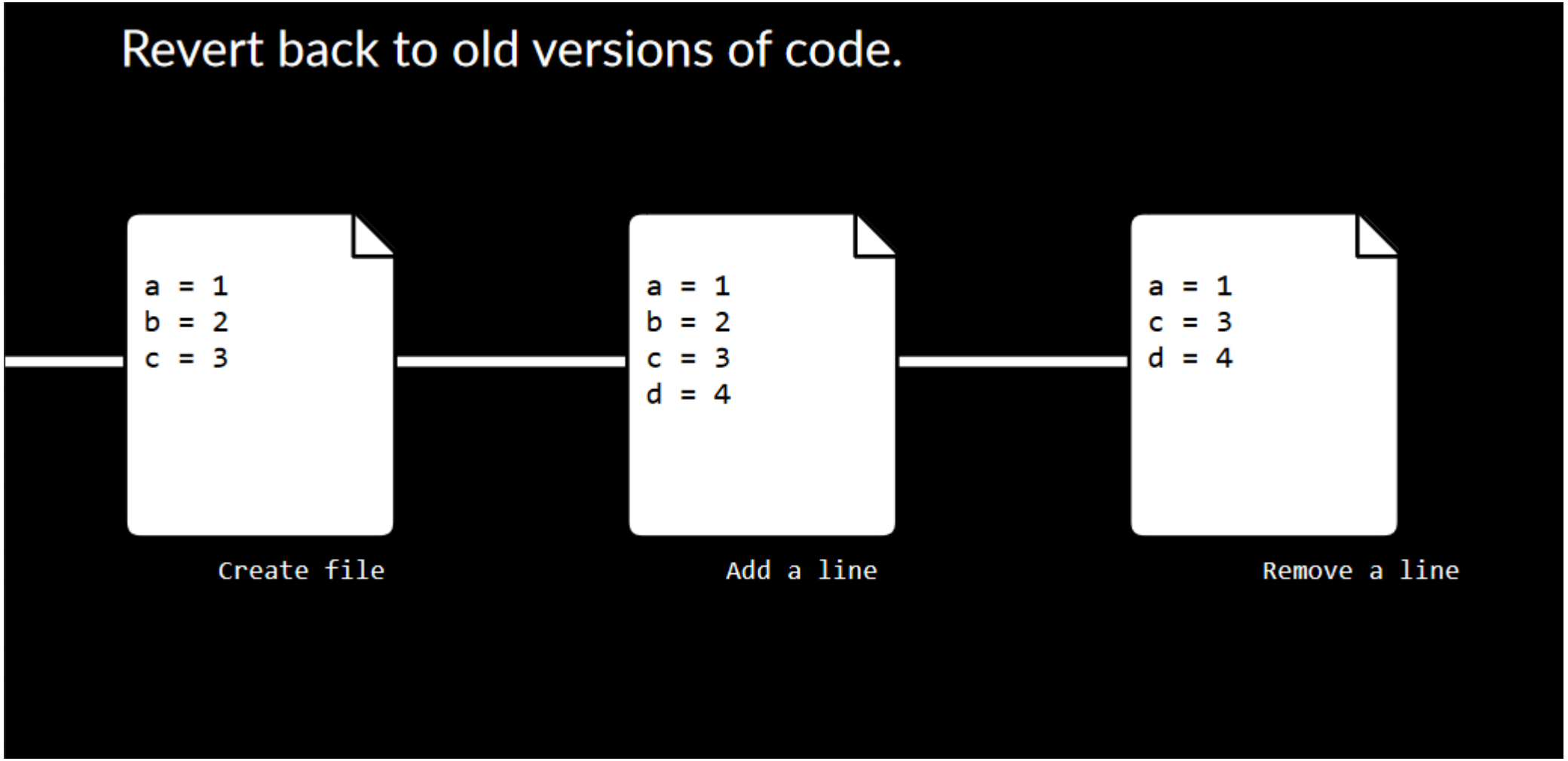
GIT- SYNCHRONIZES CODE BETWEEN DIFFERENT PEOPLE



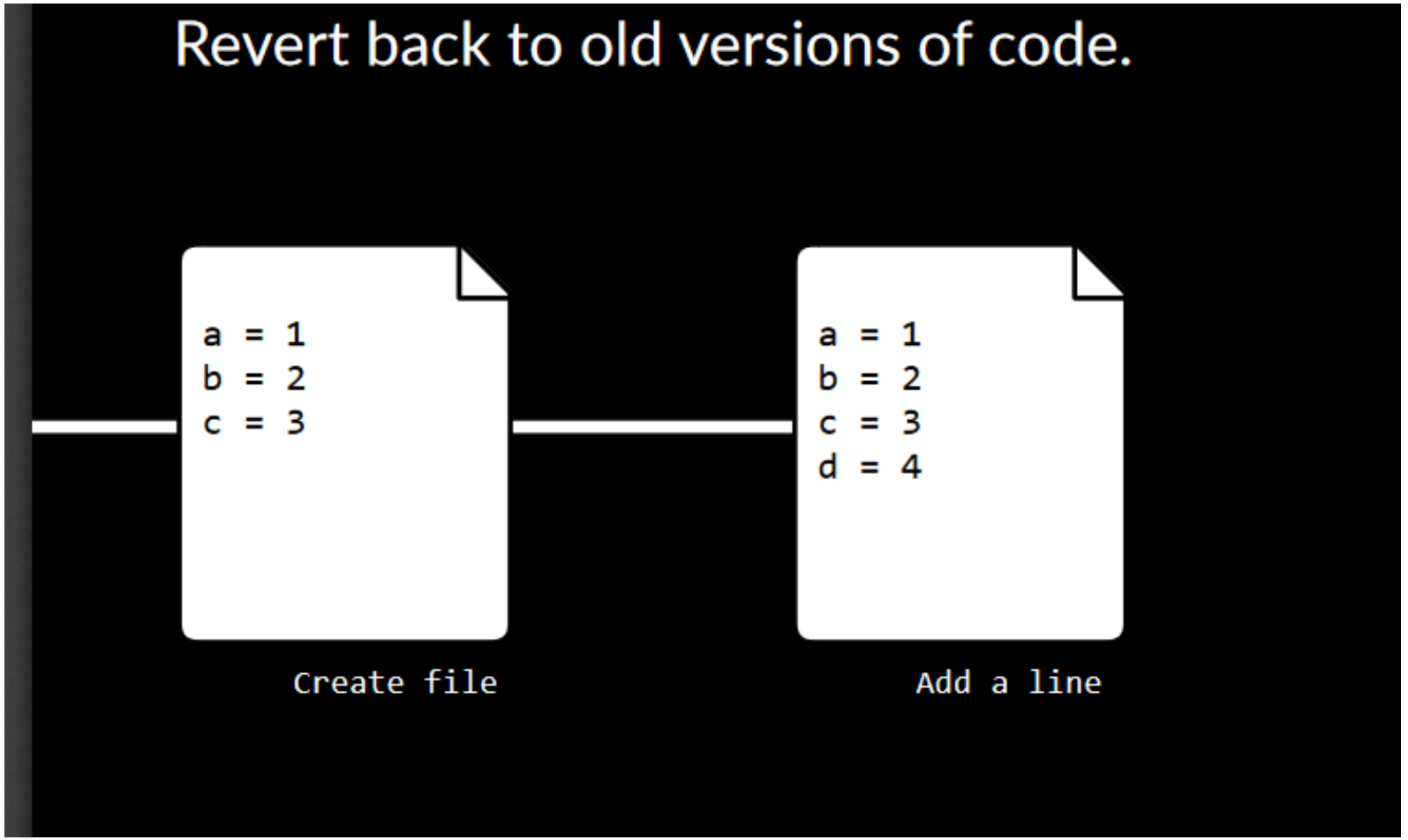
GIT- TEST CHANGES TO CODE WITHOUT LOSING THE ORIGINAL



GIT- REVERT BACK TO OLD VERSIONS OF CODE

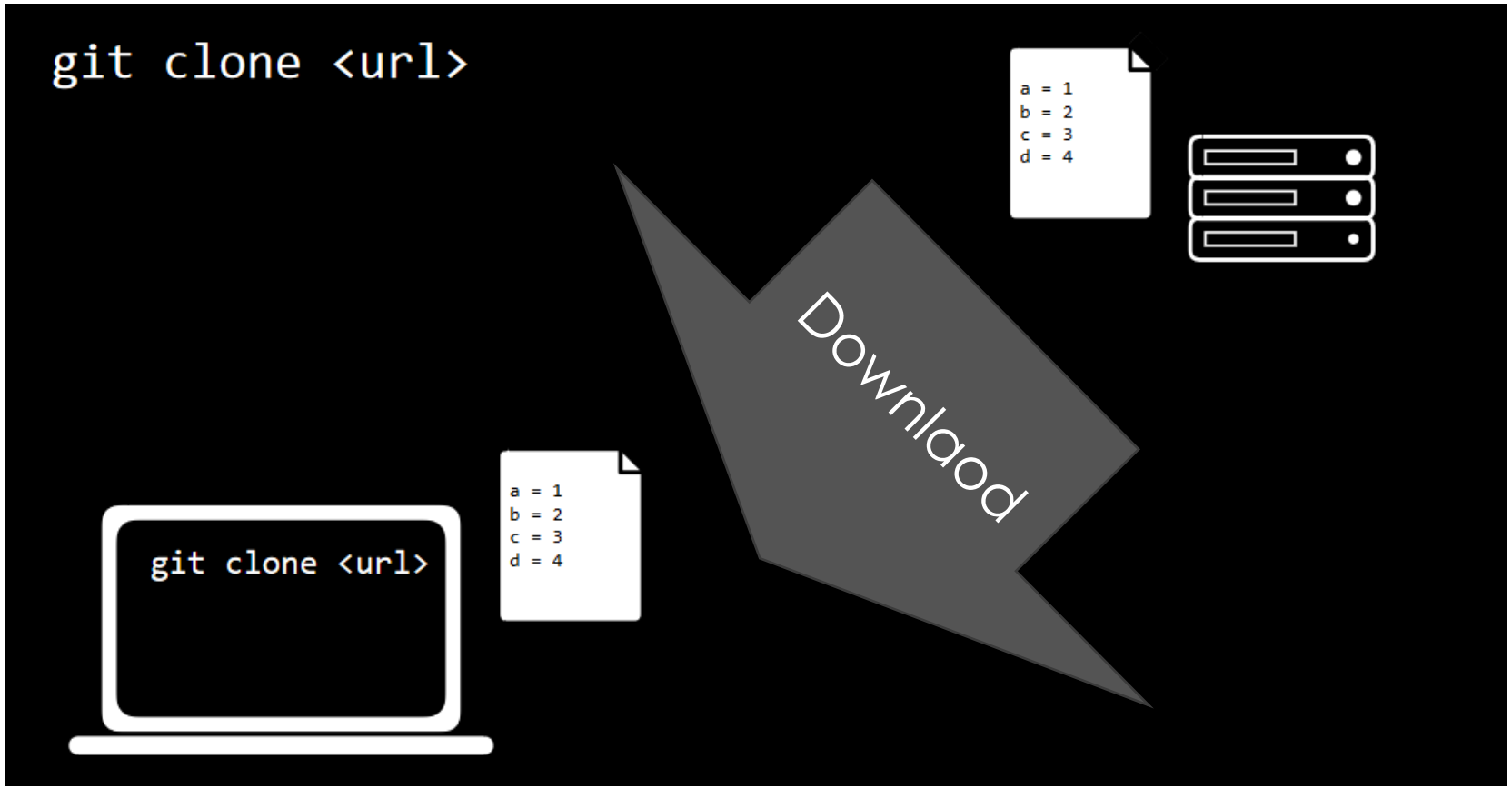


GIT- REVERT BACK TO OLD VERSIONS OF CODE

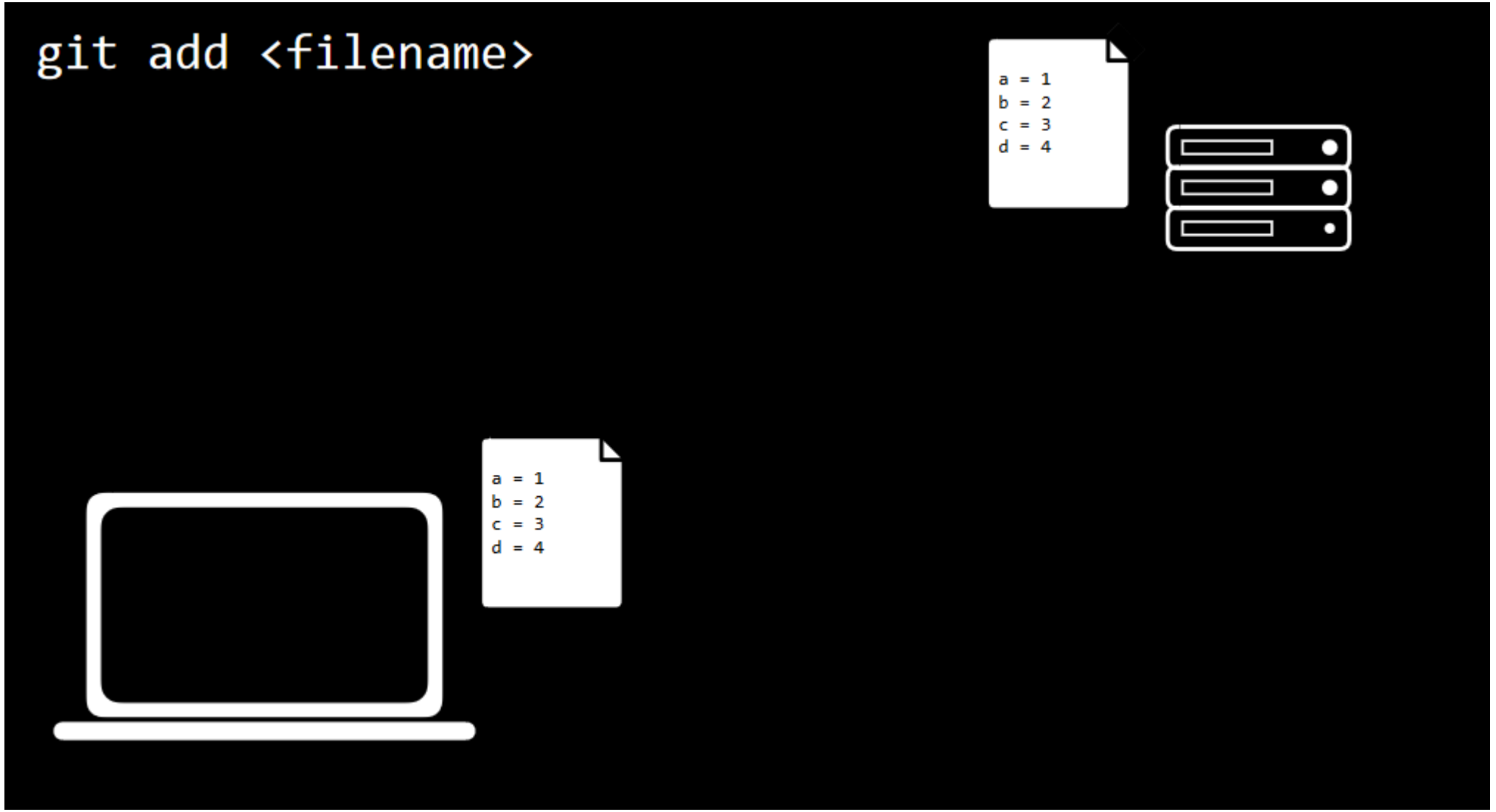


GITHUB

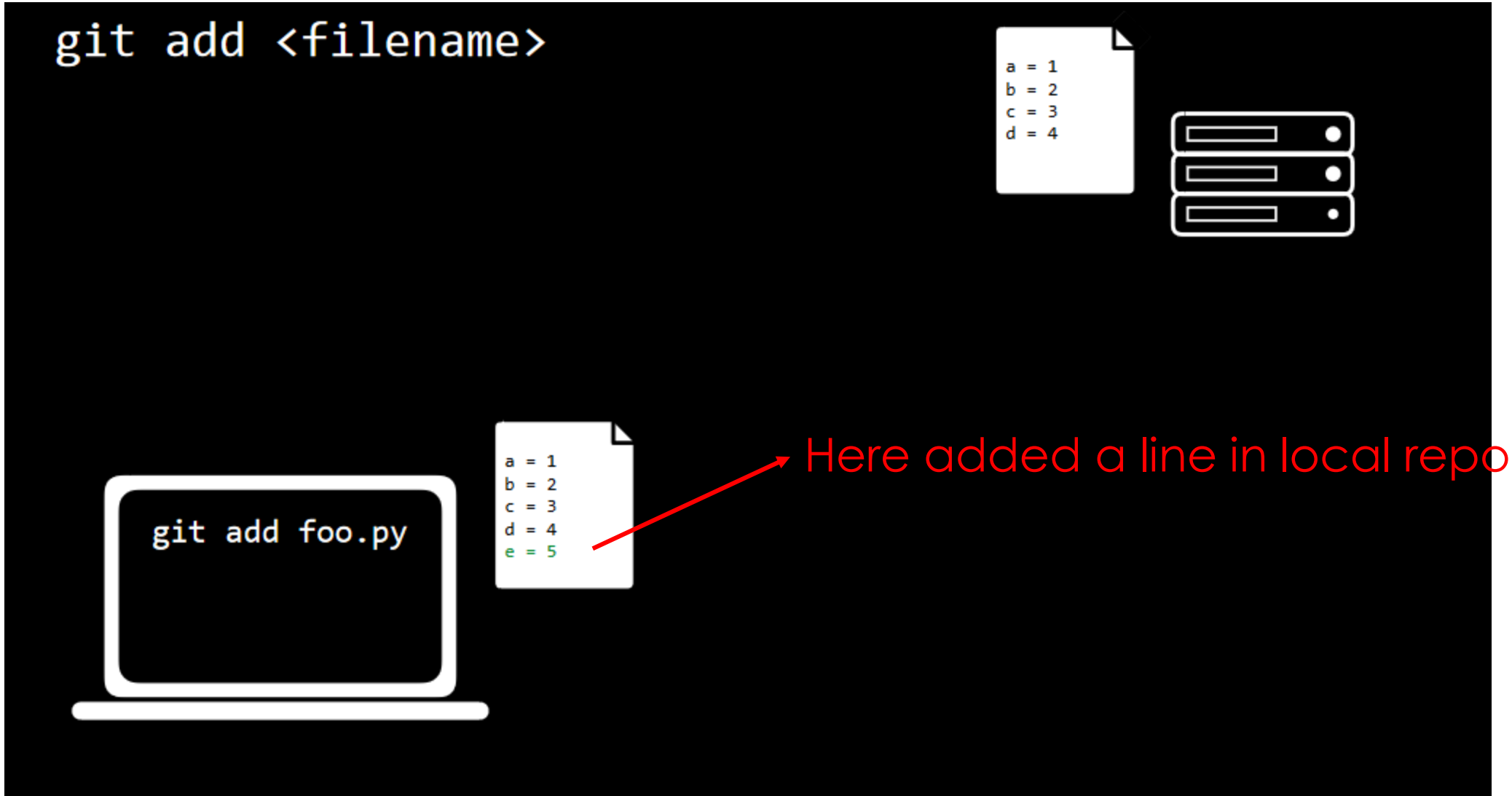
- It is a website to store remote git repositories.



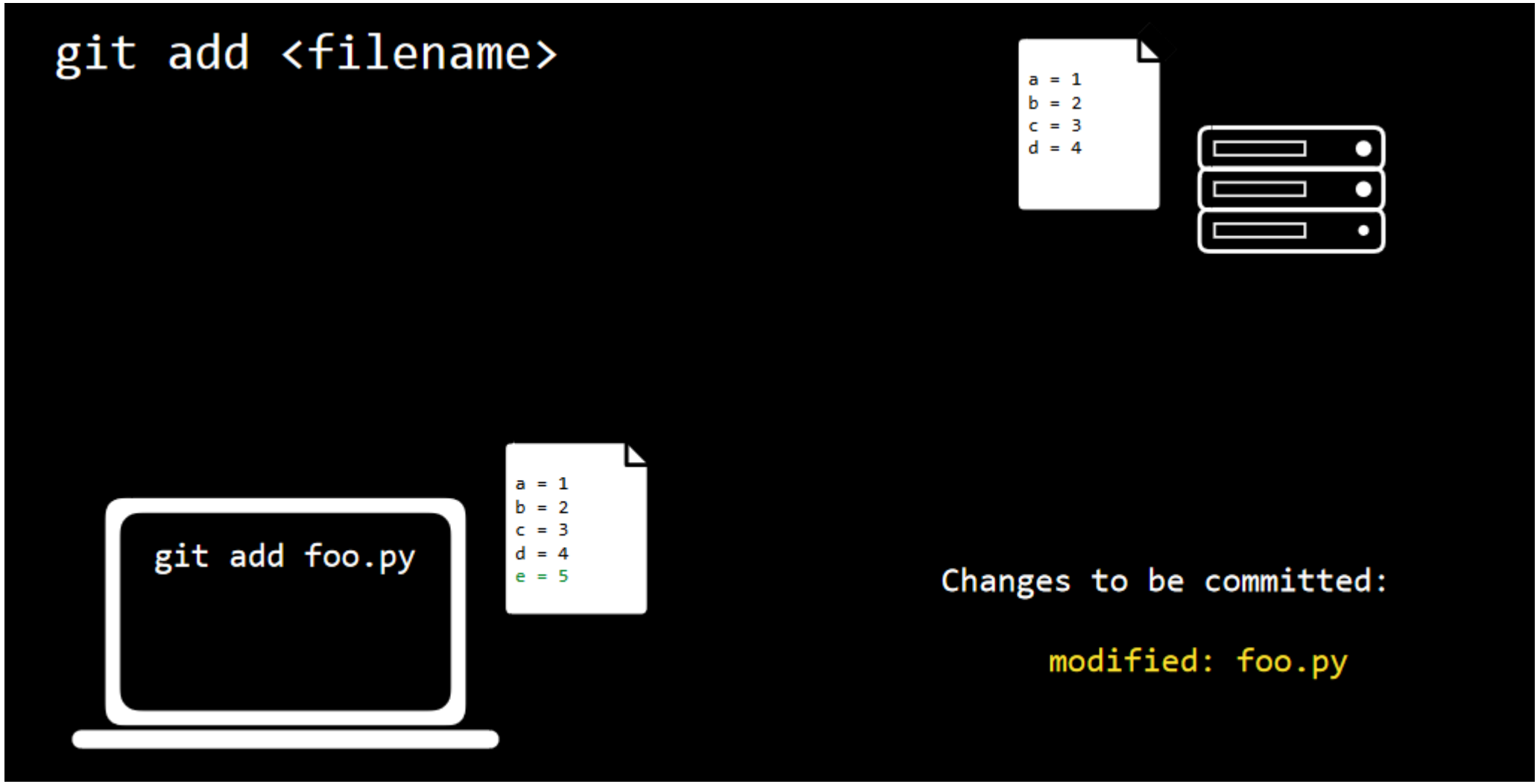
GIT ADD



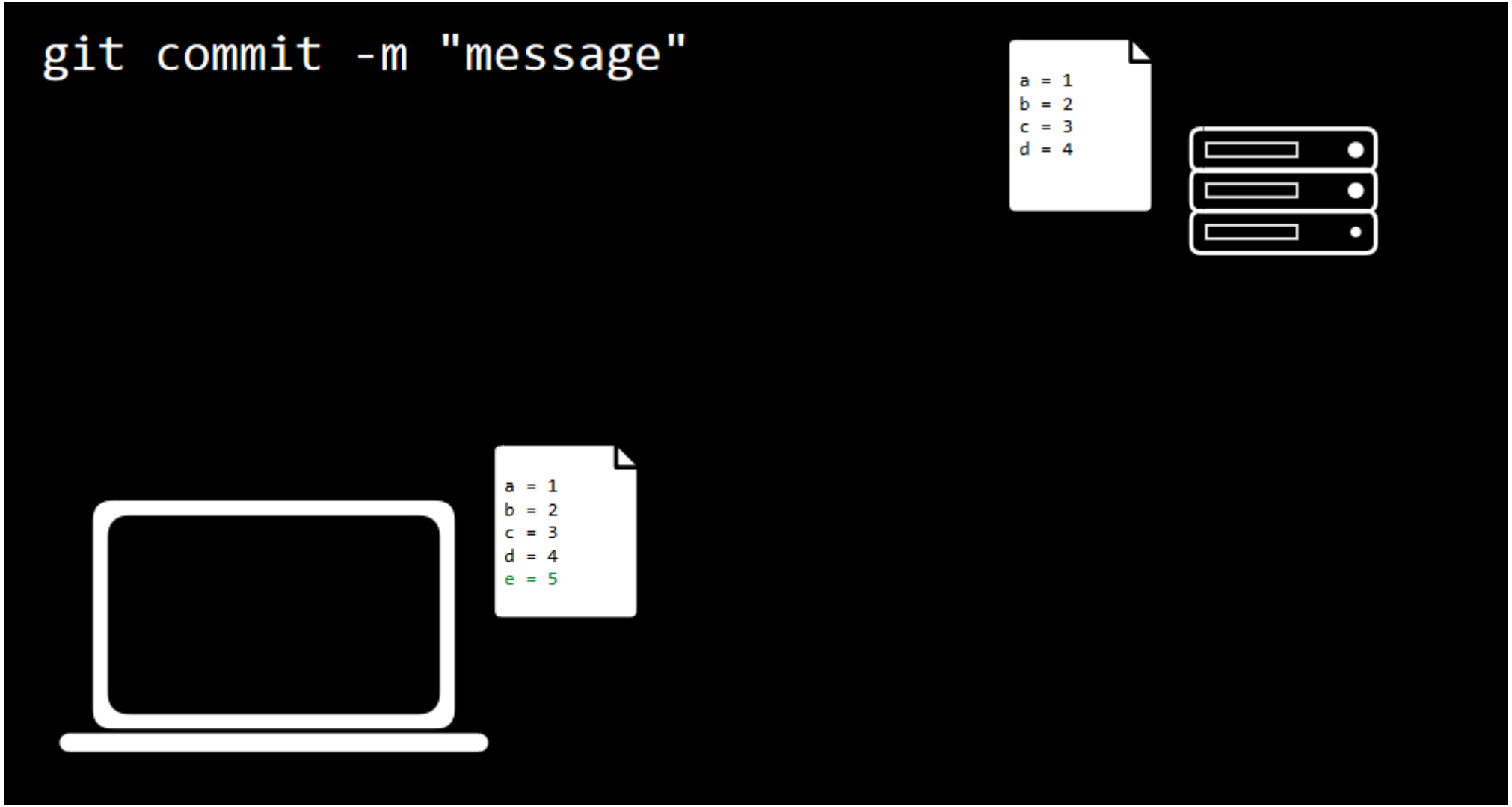
GIT ADD


git add <filename>
a = 1
b = 2
c = 3
d = 4
git add foo.py
a = 1
b = 2
c = 3
d = 4
e = 5
Here added a line in local repo

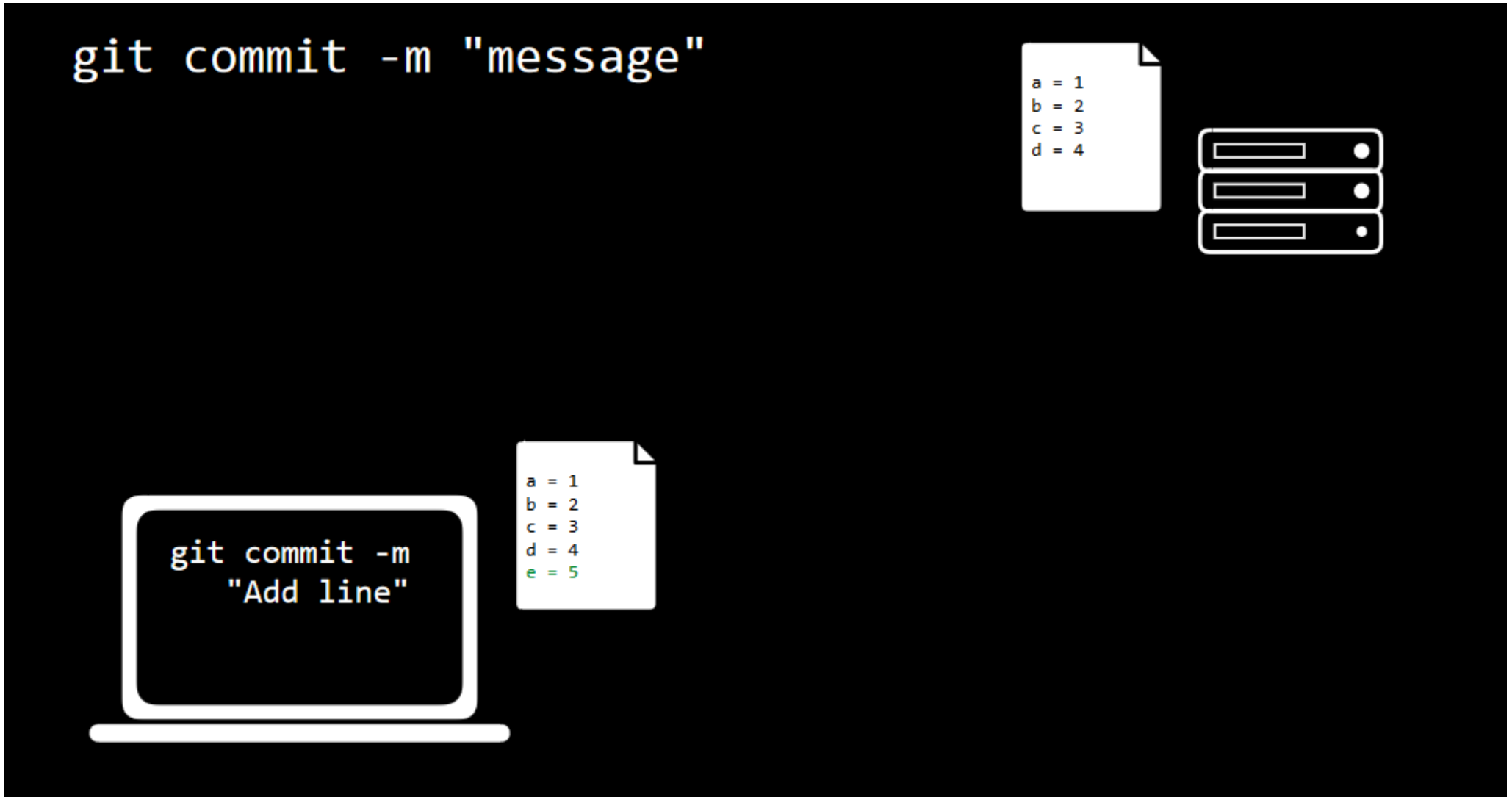
GIT ADD


git add <filename>
a = 1
b = 2
c = 3
d = 4
git add foo.py
a = 1
b = 2
c = 3
d = 4
e = 5
Changes to be committed:
modified: foo.py

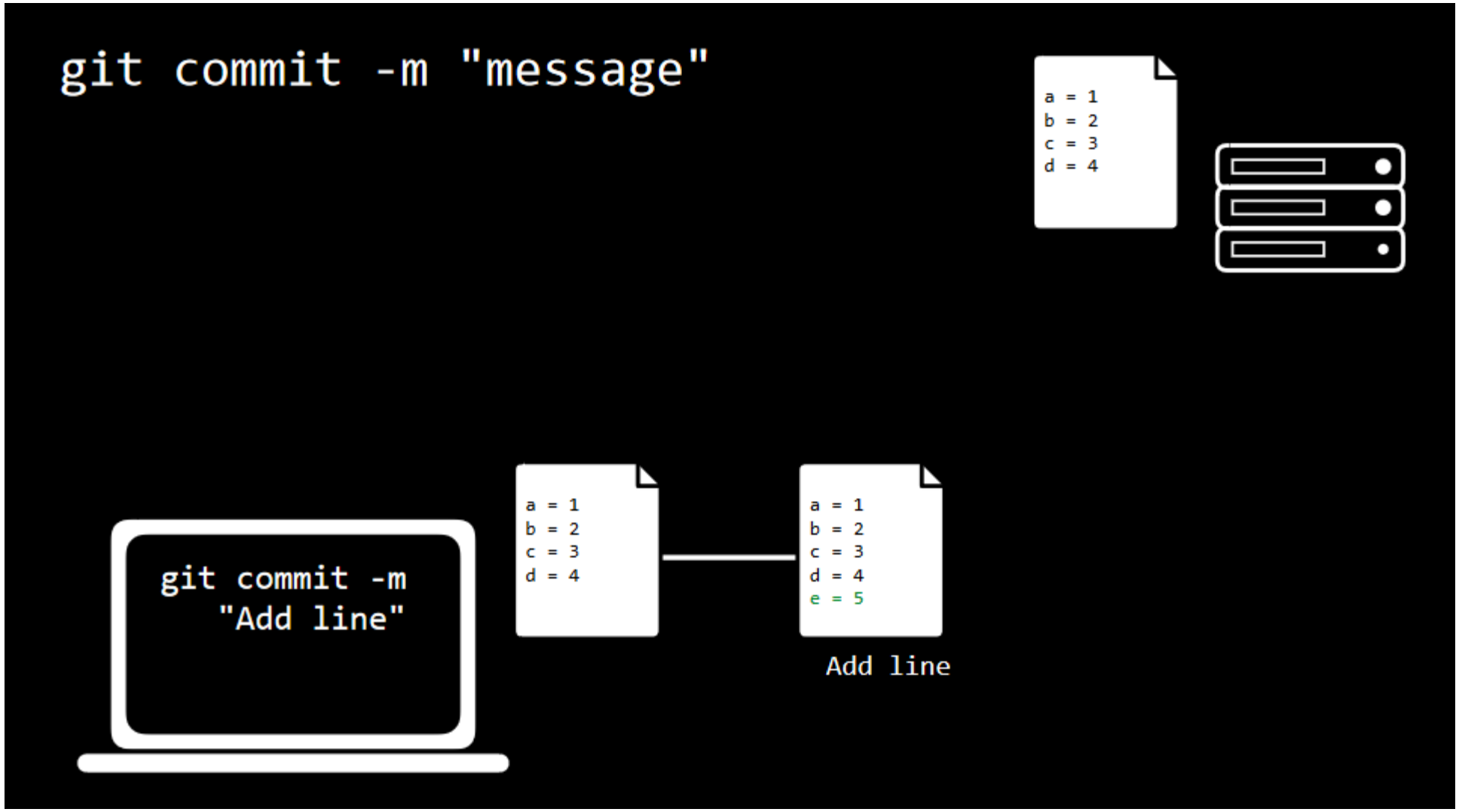
GIT COMMIT



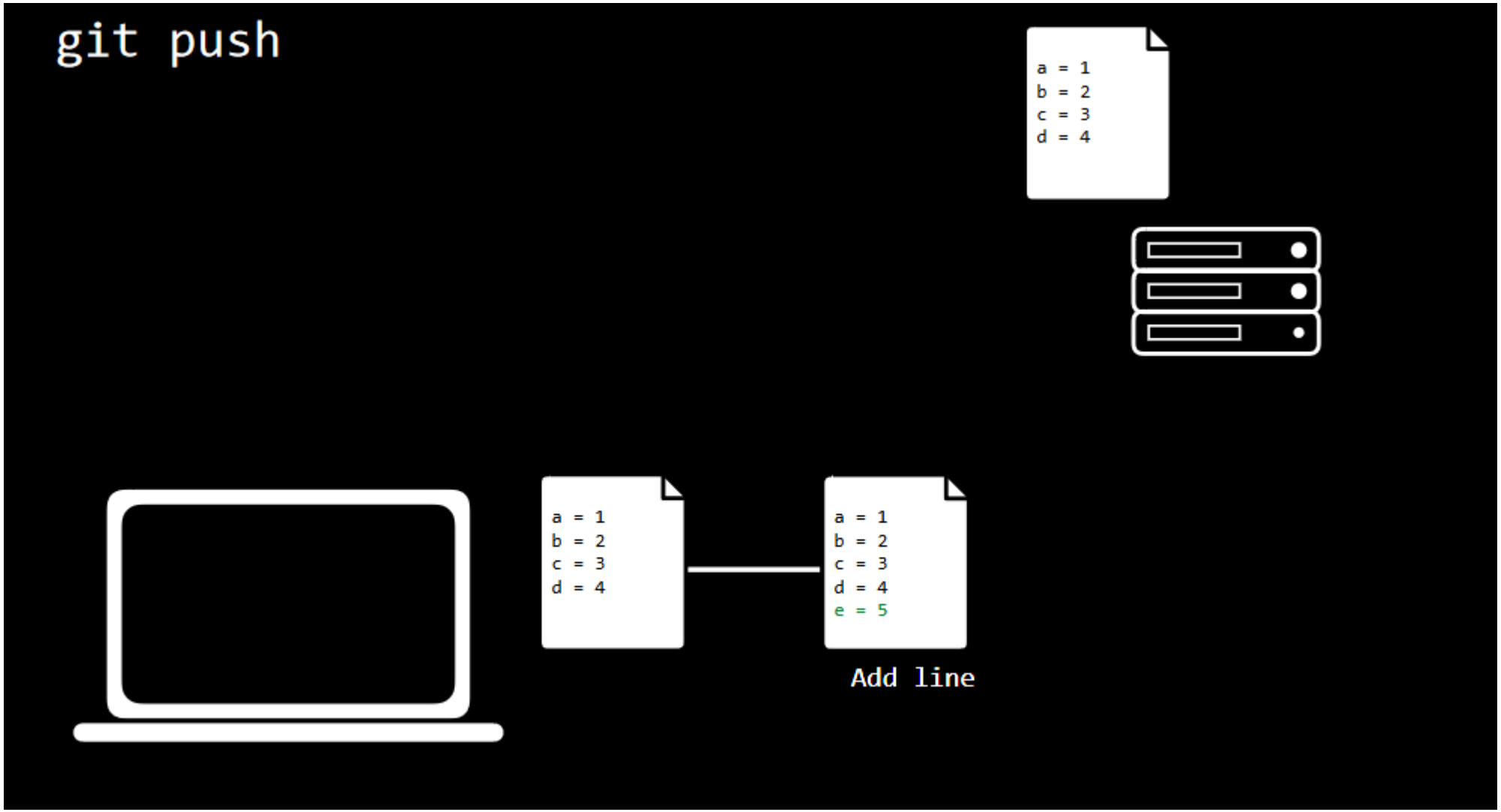
GIT COMMIT


git commit -m "message"
a = 1
b = 2
c = 3
d = 4
git commit -m
"Add line"
a = 1
b = 2
c = 3
d = 4
e = 5

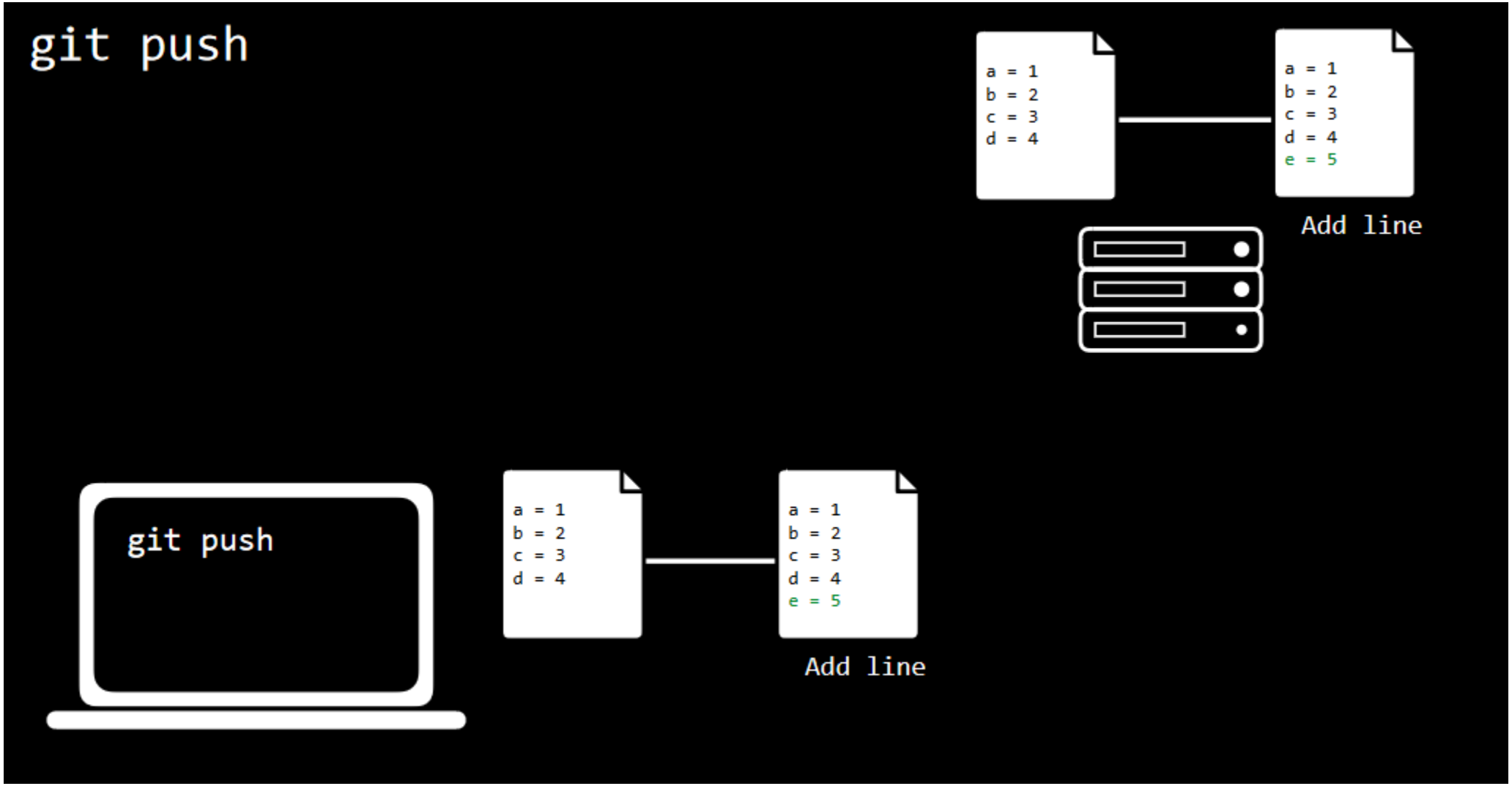
GIT COMMIT


git commit -m "message"
a = 1
b = 2
c = 3
d = 4
git commit -m
"Add line"
a = 1
b = 2
c = 3
d = 4
a = 1
b = 2
c = 3
d = 4
e = 5
Add line

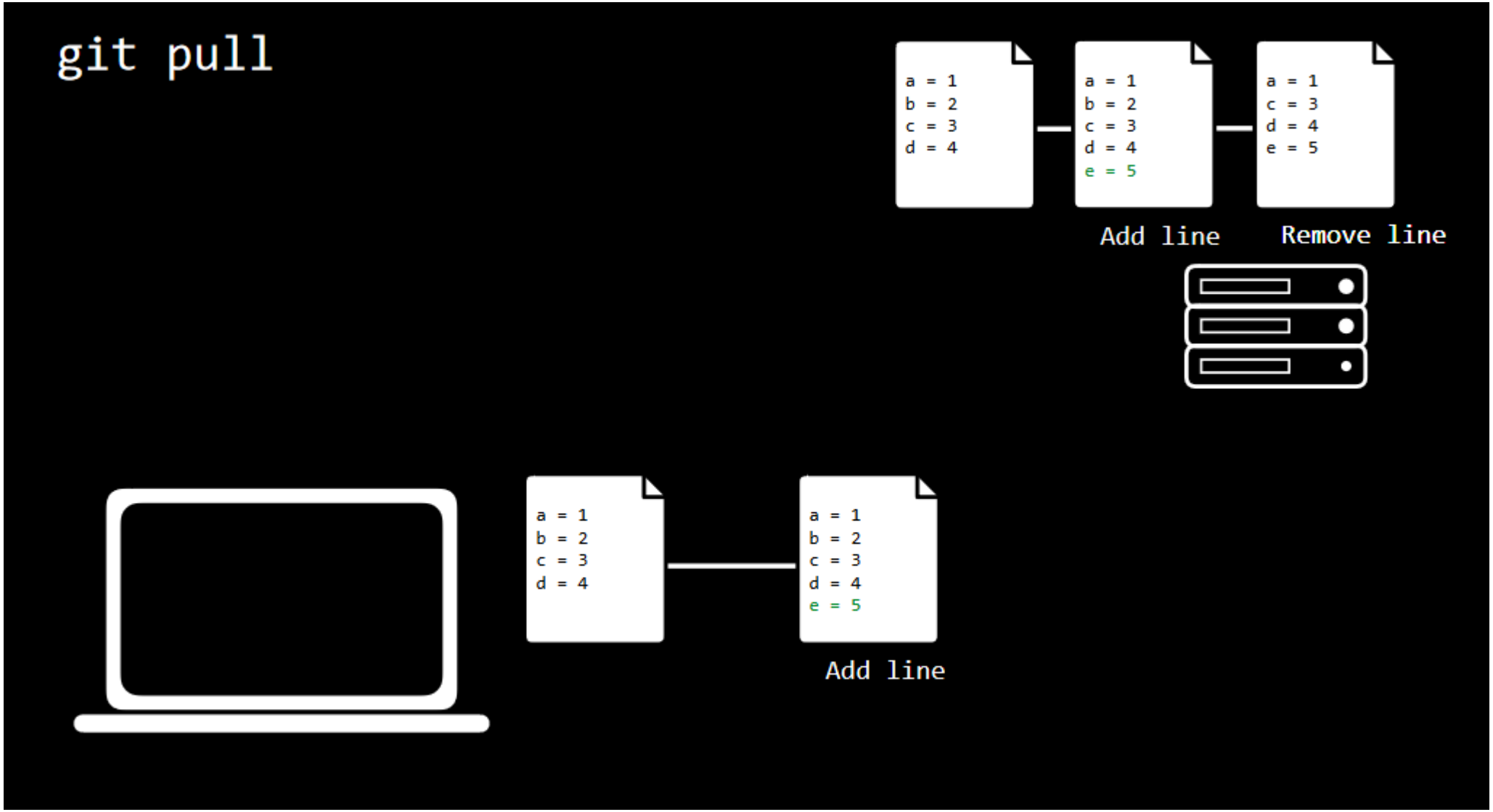
GIT PUSH = UPLOAD



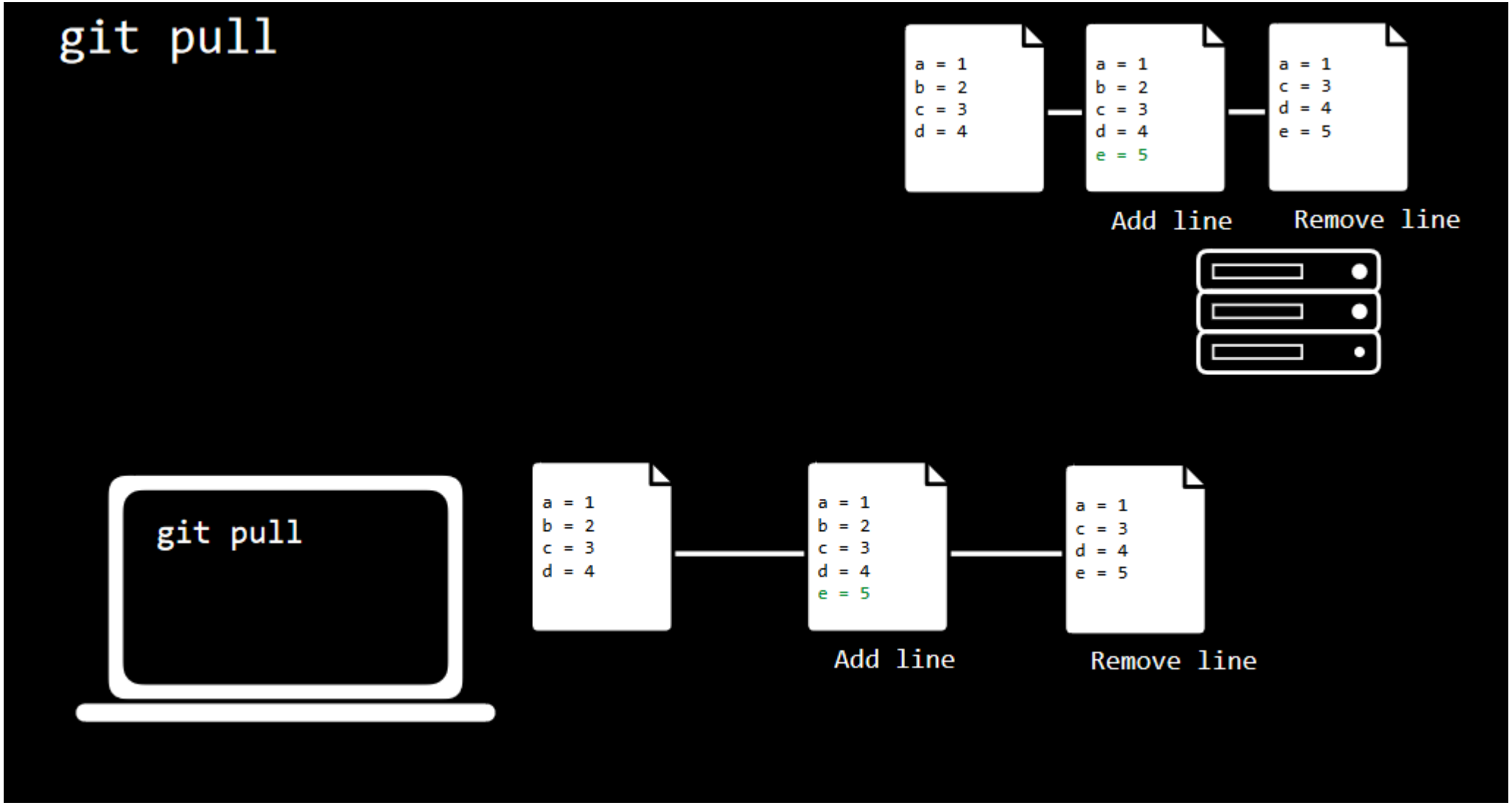
GIT PUSH


git push
a = 1
b = 2
c = 3
d = 4
a = 1
b = 2
c = 3
d = 4
e = 5
Add line
git push
a = 1
b = 2
c = 3
d = 4
a = 1
b = 2
c = 3
d = 4
e = 5
Add line

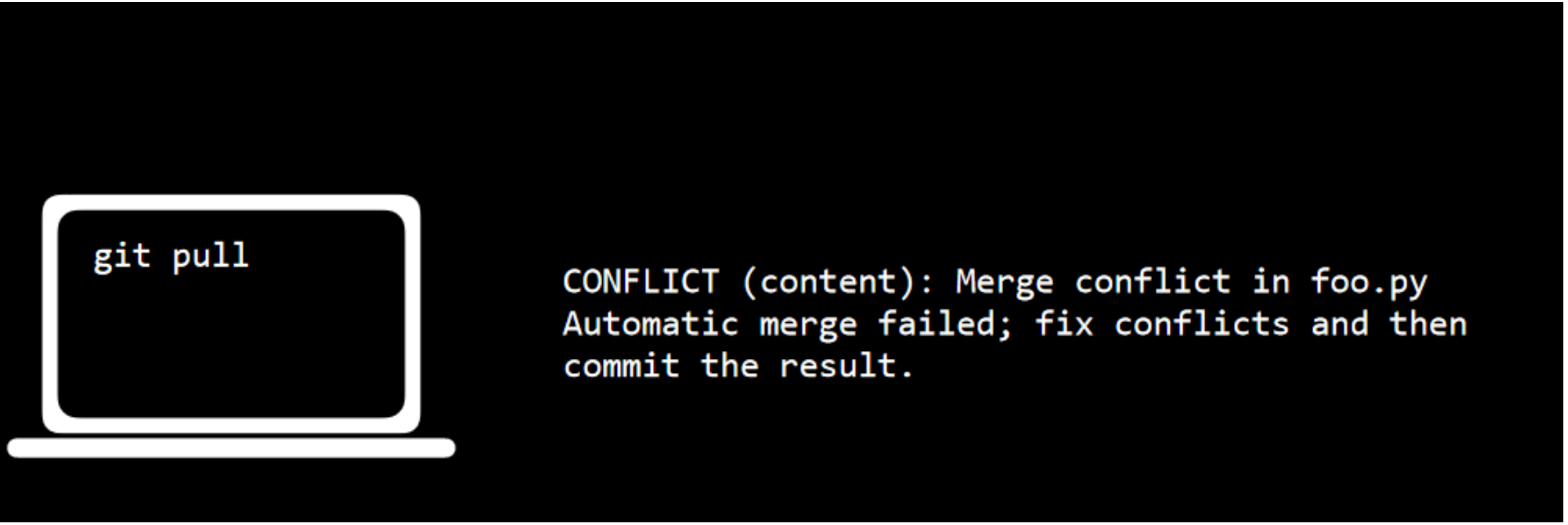
GIT PULL


git pull
a = 1
b = 2
c = 3
d = 4
a = 1
b = 2
c = 3
d = 4
e = 5
a = 1
c = 3
d = 4
e = 5
Add line
Remove line
a = 1
b = 2
c = 3
d = 4
a = 1
b = 2
c = 3
d = 4
e = 5
Add line

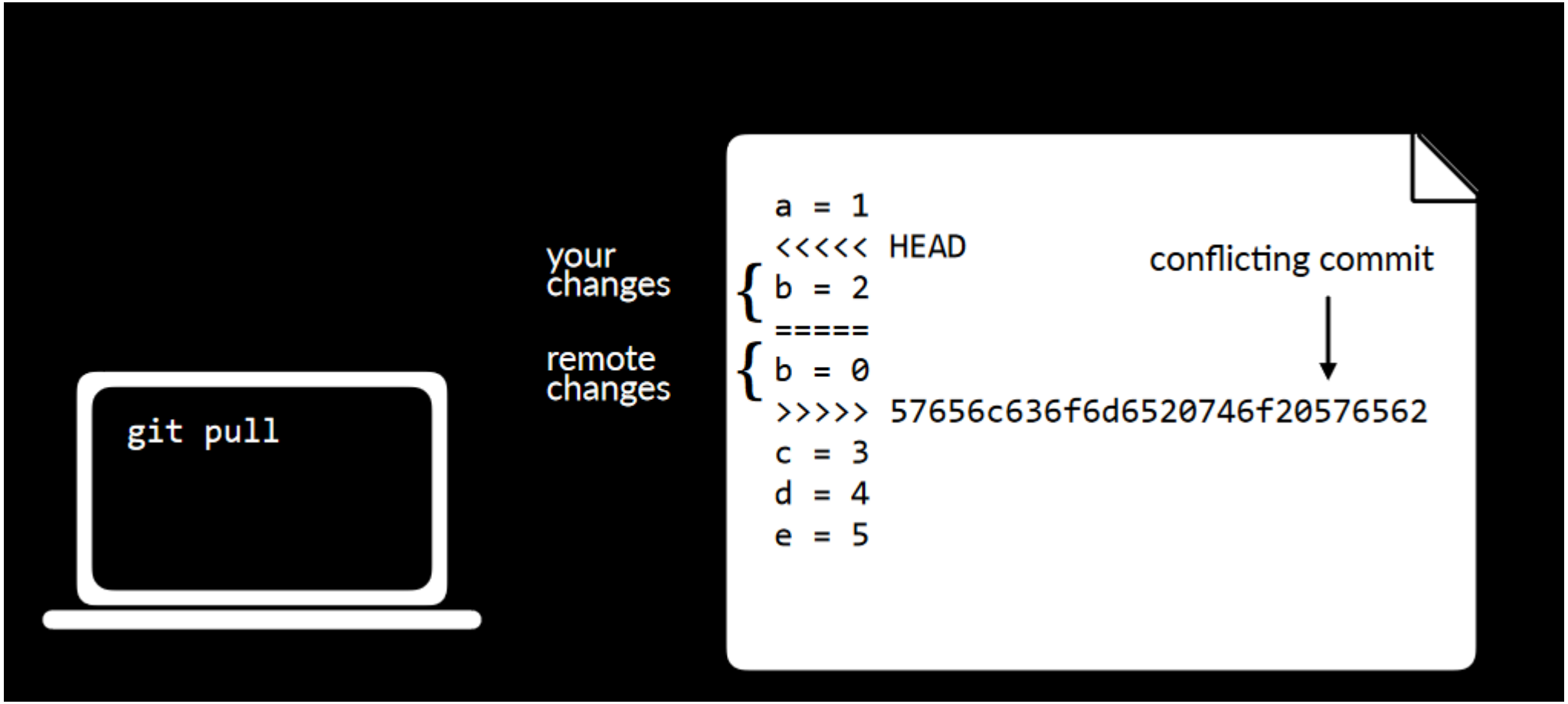
GIT PULL


git pull
a = 1
b = 2
c = 3
d = 4
a = 1
b = 2
c = 3
d = 4
e = 5
Add line
a = 1
c = 3
d = 4
e = 5
Remove line
git pull
a = 1
b = 2
c = 3
d = 4
a = 1
b = 2
c = 3
d = 4
e = 5
Add line
a = 1
c = 3
d = 4
e = 5
Remove line

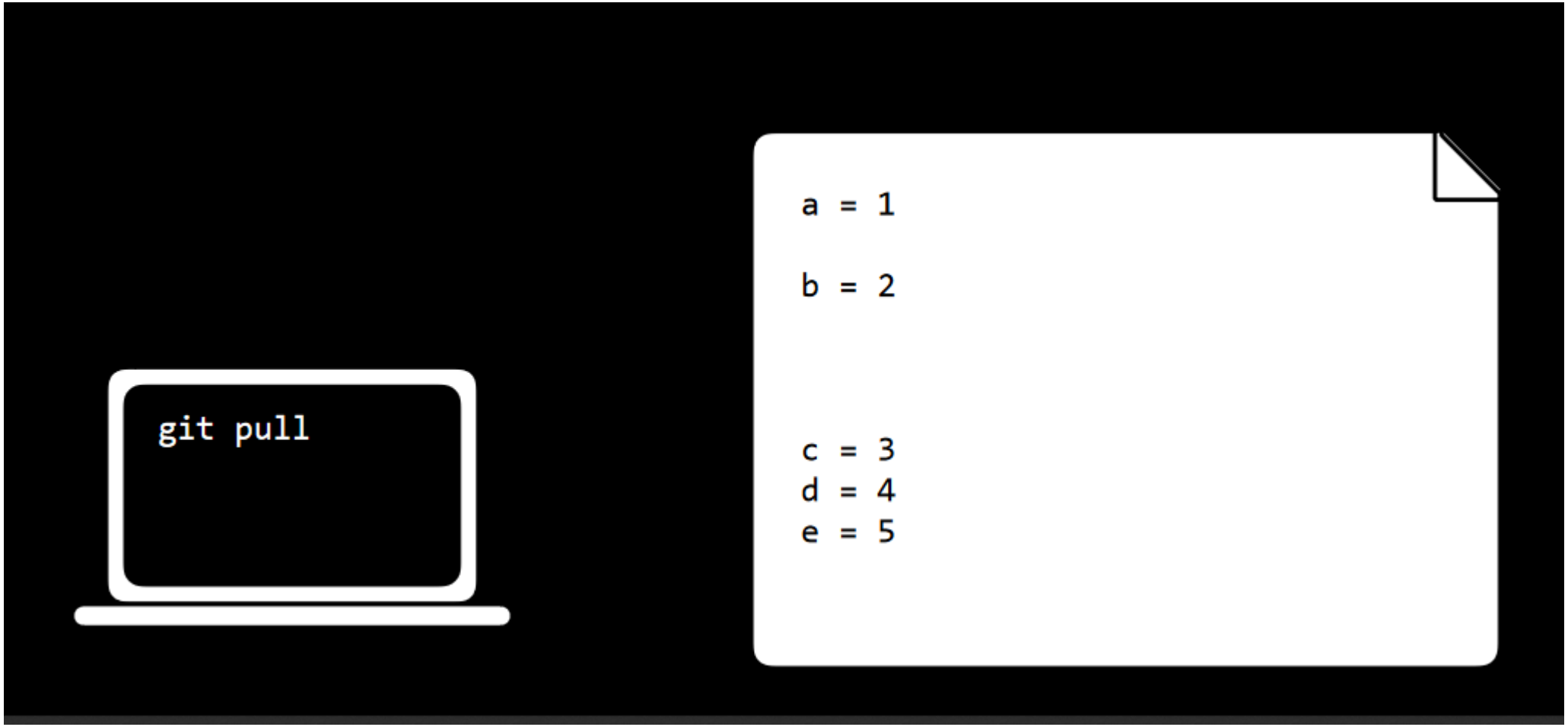
MERGE CONFLICTS



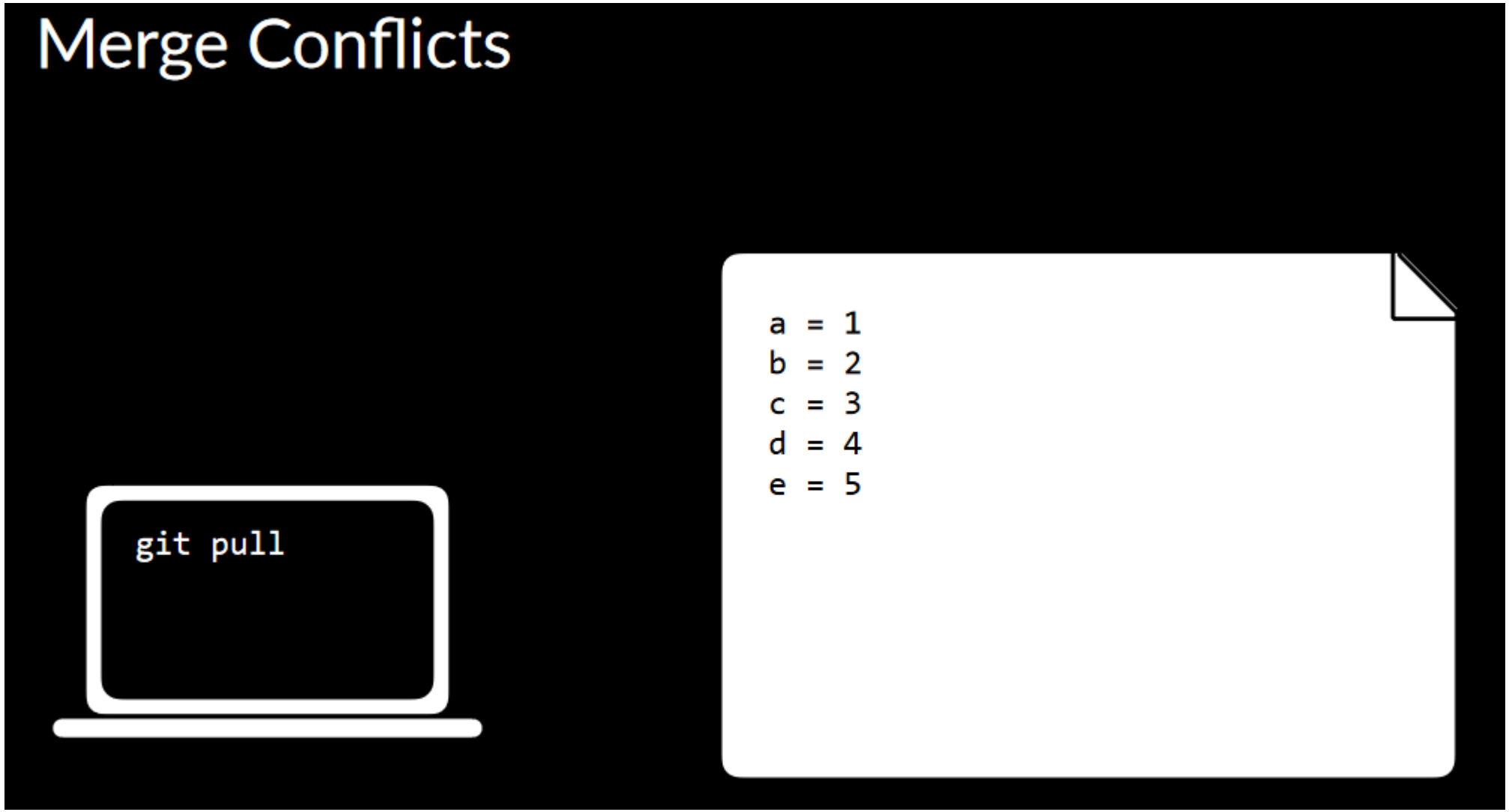
MERGE CONFLICTS



MERGE CONFLICTS - SOLUTION



MERGE CONFLICTS - SOLUTION



SUMMARY

- Introduction to Design Principles
- SOLID principles
- Introduction to Design Patterns
- Introduction to Version Control
- Git
- Github,
- Git Basic Commands

REFERENCES

1. https://www.codeproject.com/Articles/93369/How-I-explained-OOD-to-my-wife
2. https://lostechies.com/wp-content/uploads/2011/03/pablos_solid_ebook.pdf
3. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

4 . https://git-scm.com/book/en/v2