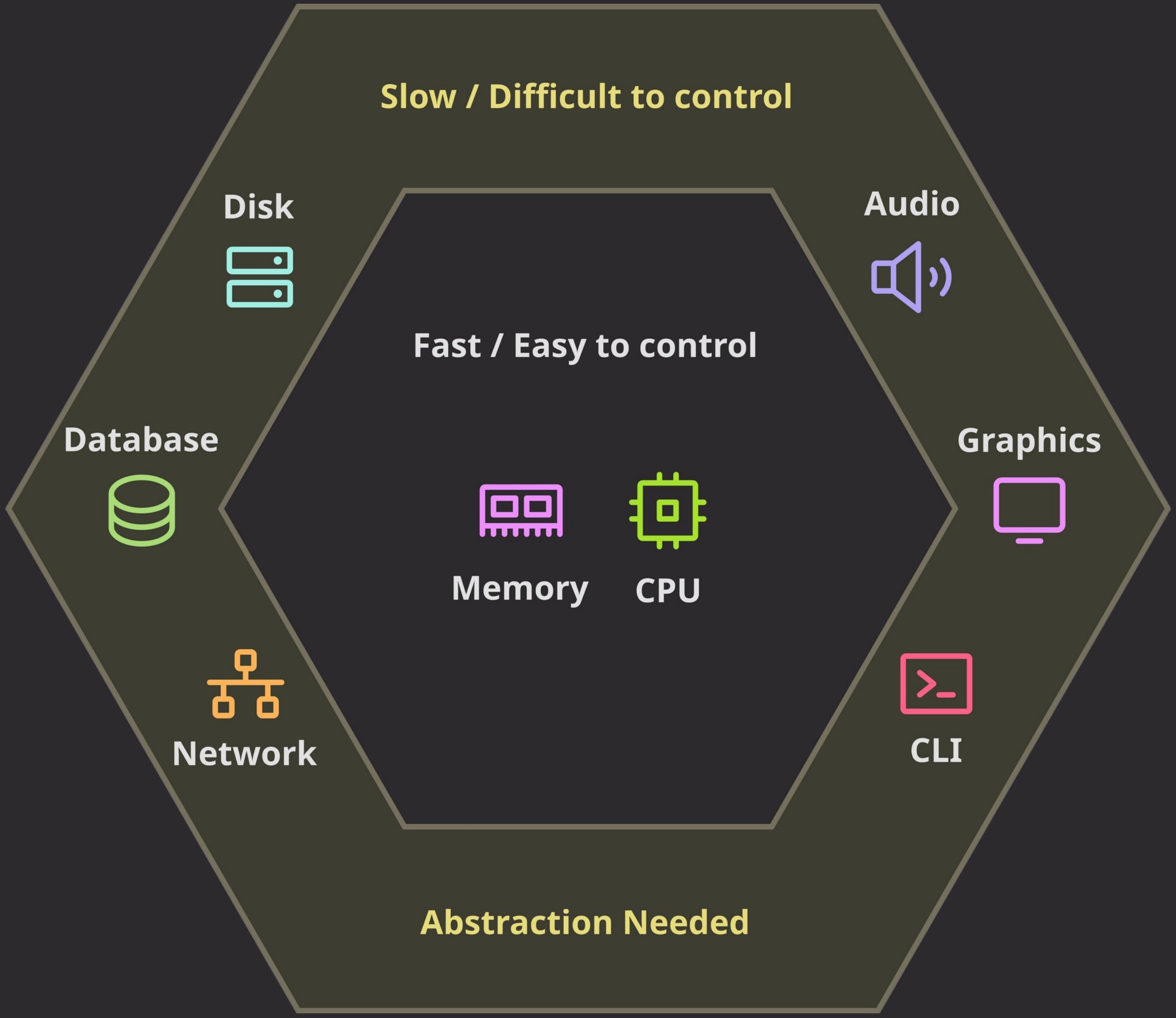
Maintainable Code | Traits

Traits

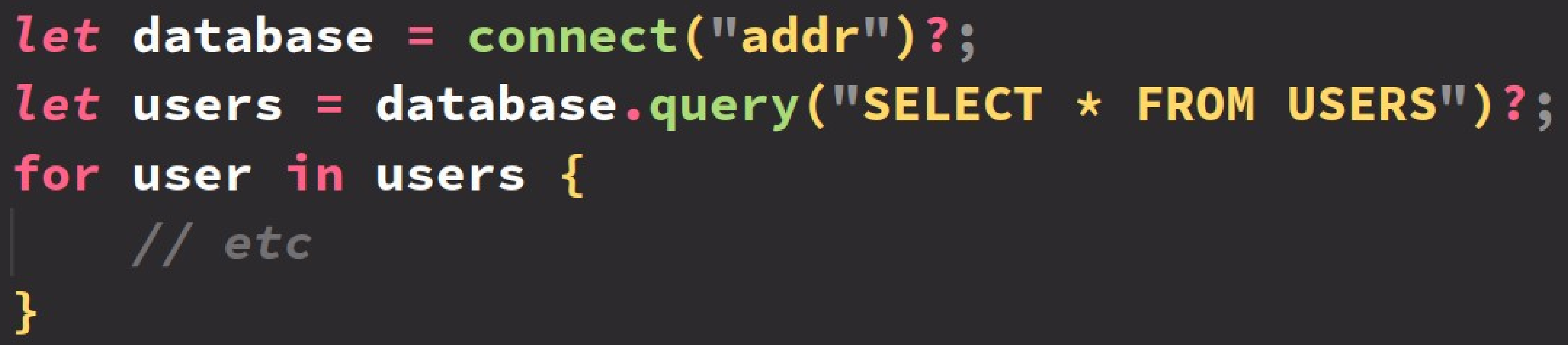
- Traits describe expected inputs and outputs
  - Use when interacting with external systems
    - Can substitute with a test version
    - Can change to a different system later
  - Use trait objects for common behaviors
    - Multiple image encoders, multiple web scrapers, multiple signal processors, etc


Slow / Difficult to control
Disk
Audio
Fast / Easy to control
Database
Graphics
Memory
CPU
Network
CLI
Abstraction Needed

Without traits

```rust
let database = connect("addr")?;
let users = database.query("SELECT * FROM USERS")?;
for user in users {
    // etc
}
```

Without traits



```
let database = connect("addr")?;
let users = database.query("SELECT * FROM USERS")?;
for user in users {
    // etc
}
```

- How do we:
  - test?
  - change databases?
  - add extra logging?
  - do something else: replay from events?

With traits

```rust
trait Data {
    fn get_users(&self) -> Result<Vec<User>, DataError>;
}
let data = SqlDatabase::connect("addr")?;
let users = data.get_users()?;
for user in users {
    // etc
}
```

Multiple implementations possible

```rust
let data = SqlDatabase::connect("addr")?;
let data = TestDatabase::default();
let data = KVStore::in_memory();
let data = FileSystemStore::new("/mnt");
let data = Cluster::connect("10.11.12.13");

let users = data.get_users()?;
for user in users {
    // etc
}
```

Full usage

```rust
fn print_users<D>(data: &D) -> Result<(), DataError>
where
    D: Data,
{
    let users = data.get_users()?;
    for user in users {
        println!("{user:?}");
    }
    Ok(())
}
let data = SqlDatabase;
print_users(&data);
```

Easier testing

```rust
let data = TestDatabase::default()
        .seed_users_from_ages(&[31, 22, 37]);

let users = data.get_users().unwrap();
let avg = average_user_age(&users);

assert_eq!(avg, Some(30.0));
```

Trait signatures

```rust
trait Notifier {
    fn send(&self, msg: String) -> Result<(), NotifierError>;
}
```

Trait signatures

```rust
trait Notifier {
    fn send(&self, msg: String) -> Result<(), NotifierError>;
}
```

- **String** is too general
  - Additional metadata?
  - Message length restriction

Trait signatures

```rust
struct Notification(String);

trait Notifier {
    fn send(&self, msg: Notification) -> Result<(), NotifierError>;
}
```

- Use custom types or new types for inputs and outputs
- ***Always*** create a custom error type for methods that can fail

Recap

- Use traits when interfacing with an external system
- In trait declarations:
  - Use new types or trait-specific types for inputs and outputs
    - Avoid using third-party types from other crates
  - Always create custom errors for traits