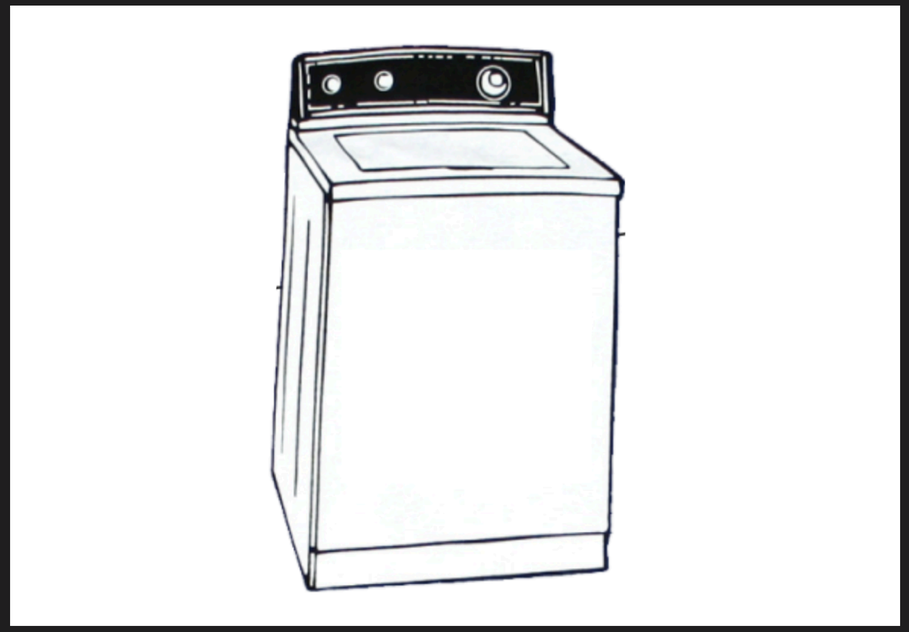
The
SOFTWARE
ESSENTIALIST

# Metaphysical Essential: Abstraction

## Complexity, Theoretical/Physical, The Abstraction Prism & The Stereotypical Architecture

essentialist.dev

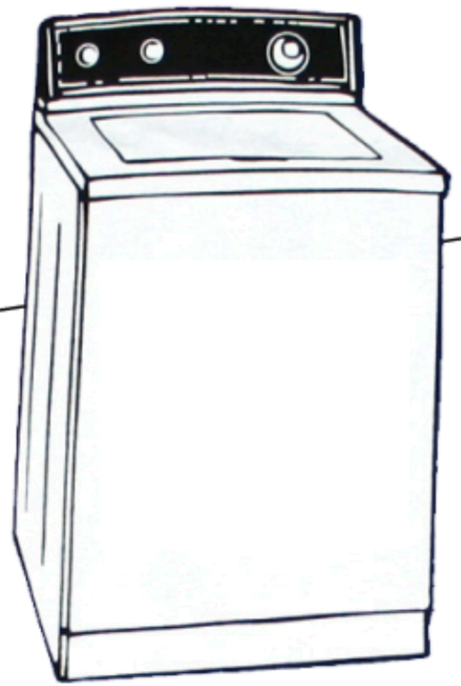
# Have you ever struggled to use one of these?



## Public interface

```
startCycle(options)
```

*Simple and easy for human beings to understand*

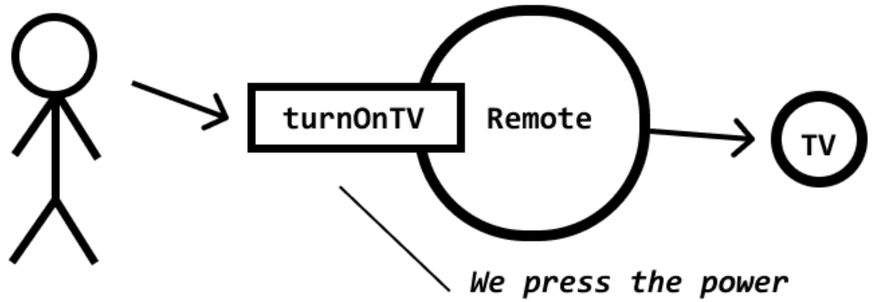


## Implementation

```
startCycle (options) {
  // Parse the options
  // Get access to the physical layer
  // Convert options into commands
  // Lots of low-level code
  // And so on...
}
```

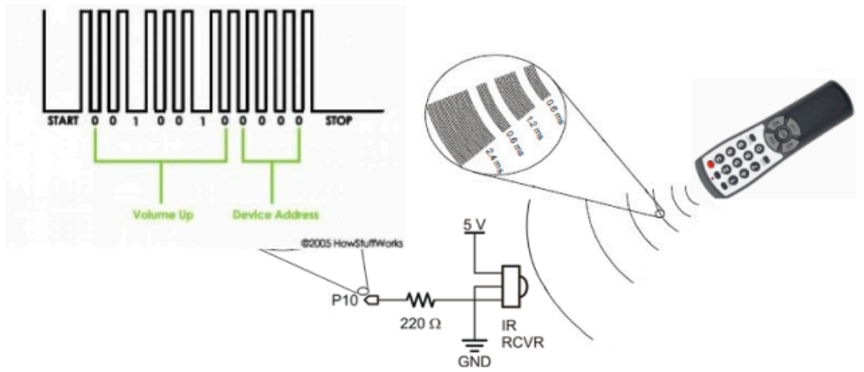
*Implementation complexities abstracted away and understood only by developers and domain experts.*

# Abstraction example: Using a remote control to turn on a TV




## Public interface

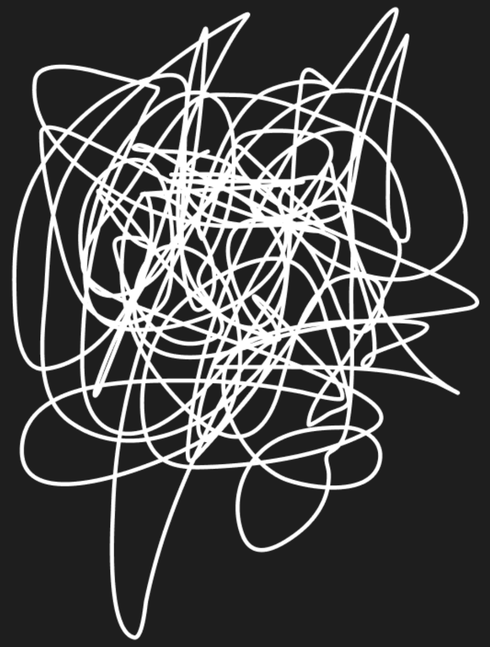
**Public**
**Declarative**
**Human-centered**
**Highest level of abstraction**



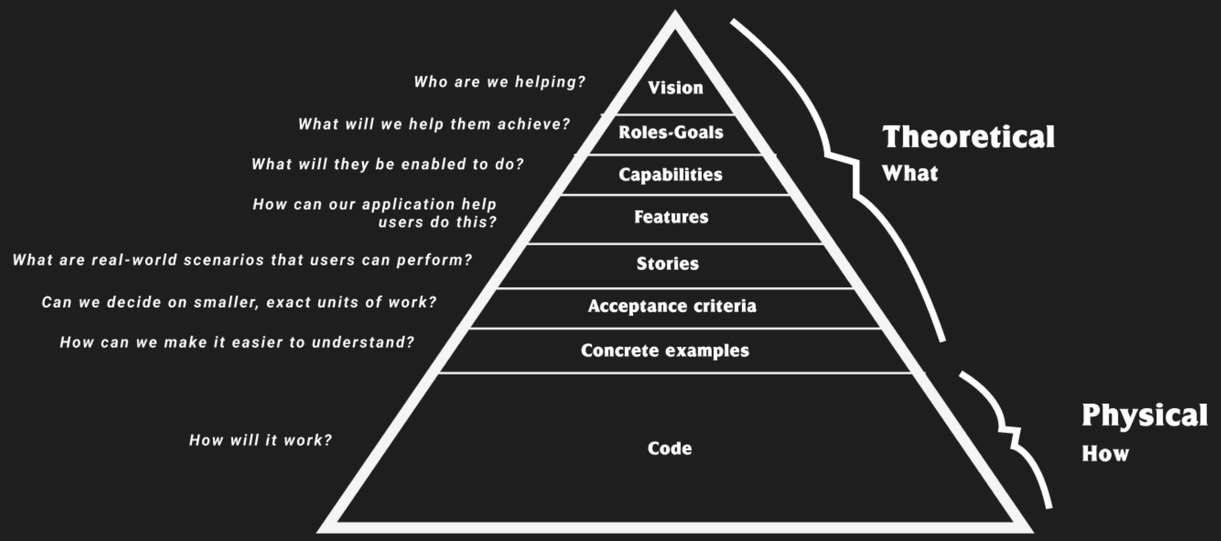

## Implementation

**Private**
**Imperative**
**More technical**
**Lower levels of abstraction**

# Complexity is the default state of the universe



# Abstraction tames complexity




## Metaphysical Essential

# Abstraction

# What we'll cover

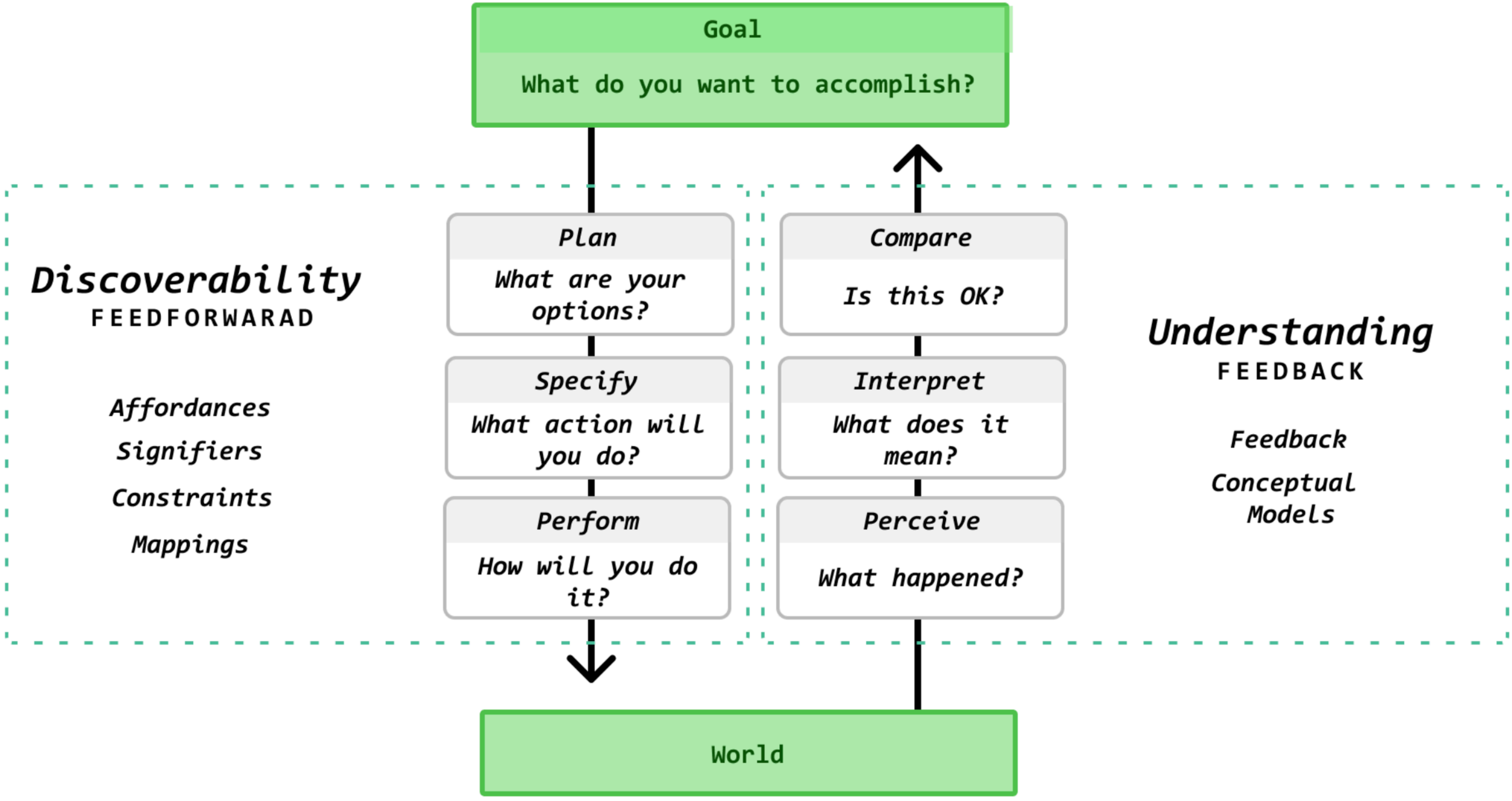
- *What is Abstraction & why is it so important?*
- *How to use Abstraction to tame complexity (ie: by decoupling a goal into Theoretical & Physical)*
- *The layers of Abstraction (ie: the Abstraction Prism) & the Stereotypical Architecture*

# What is Abstraction?

# ABSTRACTION

Essential vs. Accidental
Upfront vs. Emergent
Theoretical vs. Physical
Declarative vs. Imperative
What vs. How
Contract vs. Concretions
Roles vs. Actors
Plans vs. Execution
Not Real vs. Real

# #1: The Root Principle of Design (discoverability & understanding)

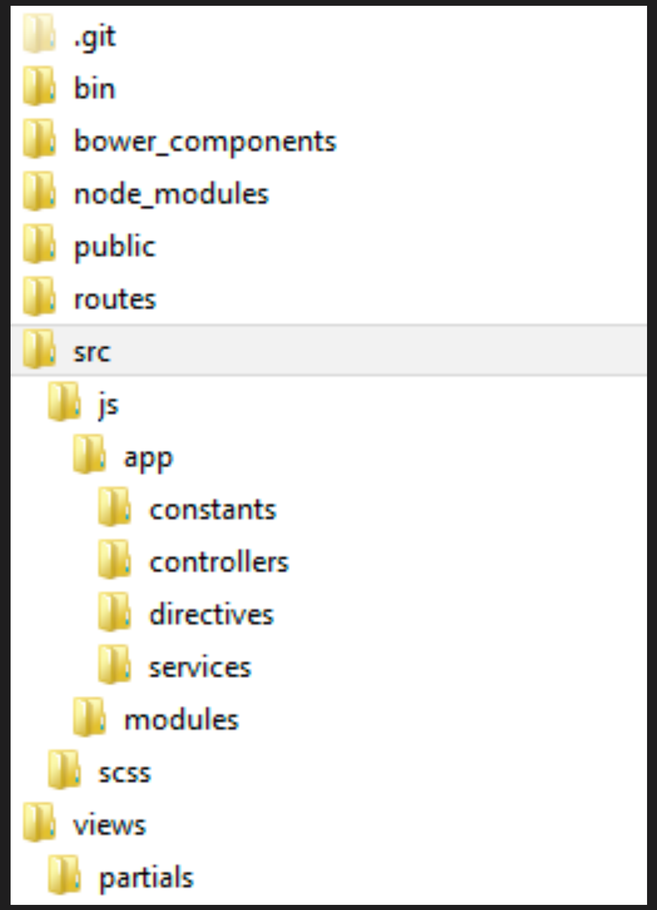

Goal
What do you want to accomplish?
Plan
What are your options?
Specify
What action will you do?
Perform
How will you do it?
Compare
Is this OK?
Interpret
What does it mean?
Perceive
What happened?
World
Discoverability
FEEDFORWARAD
Affordances
Signifiers
Constraints
Mappings
Understanding
FEEDBACK
Feedback
Conceptual Models

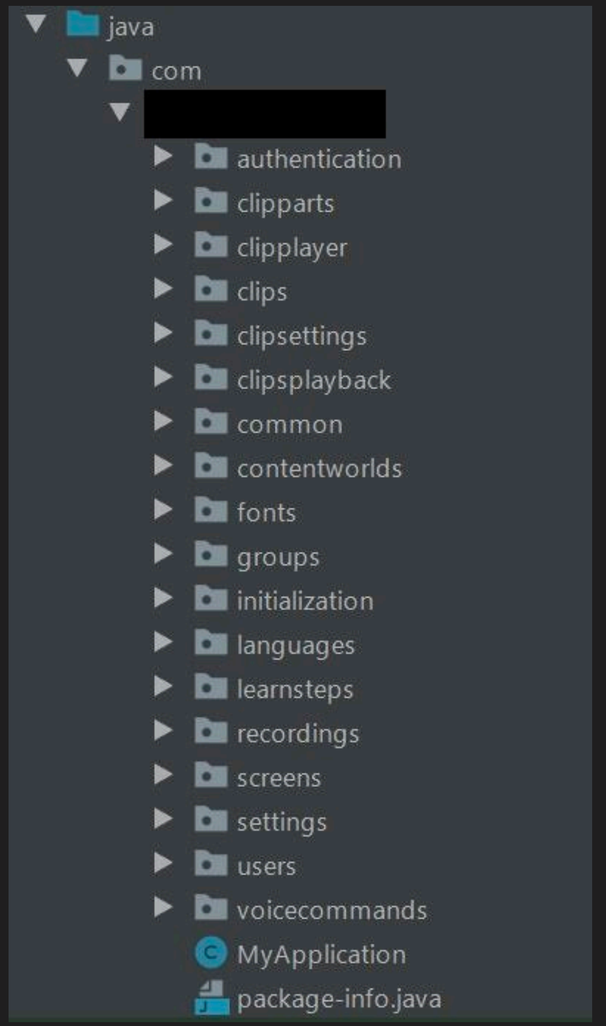
# Reminder: The Key Developer Use Case

**Discover, Understand, Add, Change, Debug, Test, Plan**

**Features**

# Goal = Add Feature



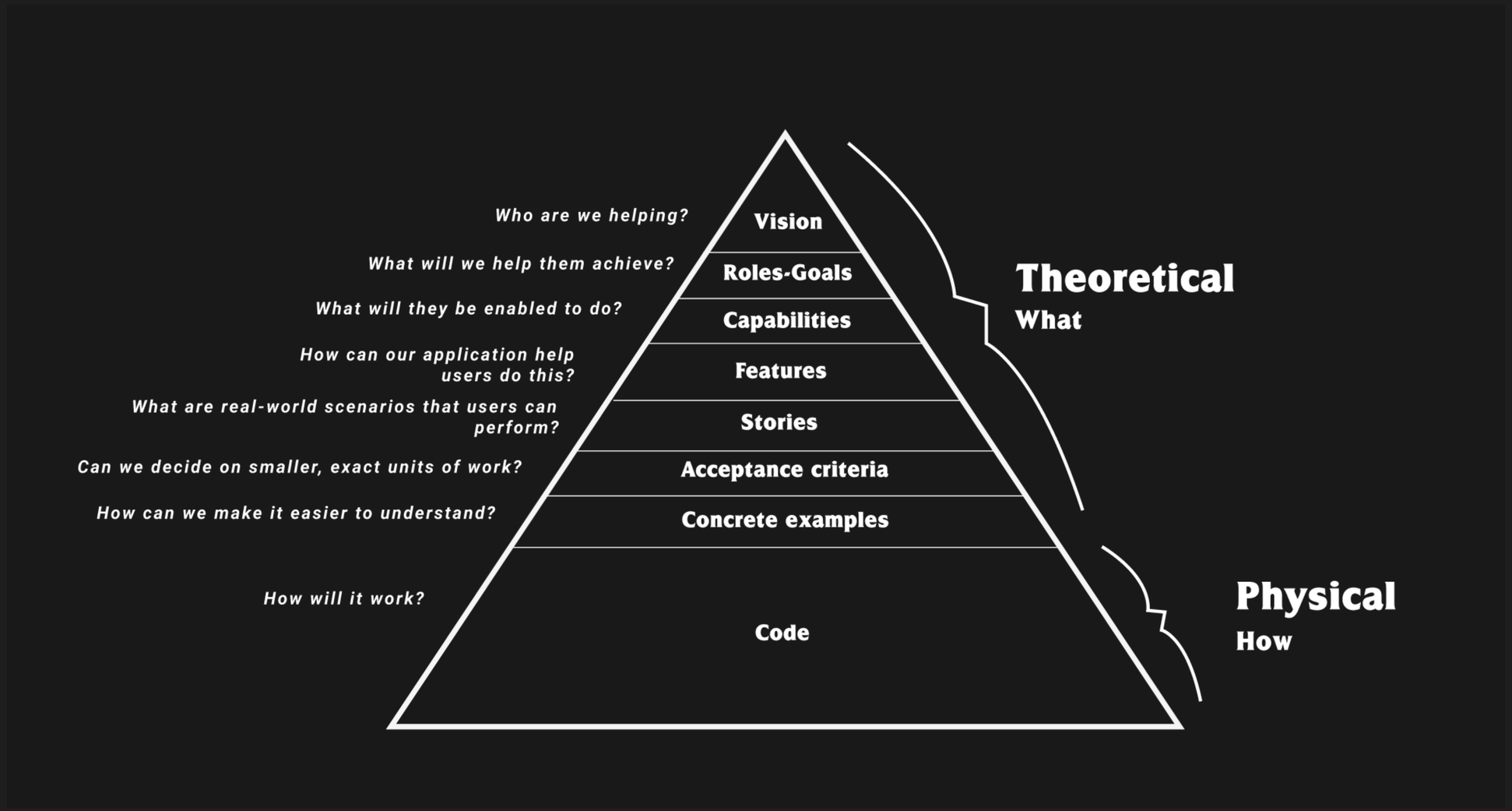
# Goal = Change Feature

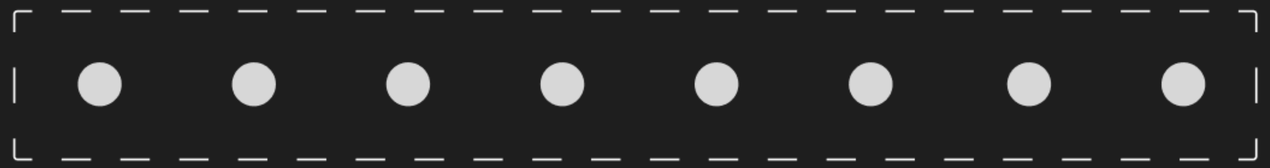
```
module.exports = (app) => {
  app.put('/updateUser', (req, res, next) => {
    passport.authenticate('jwt', { session: false }, (err, user, info) => {
      if (err) {
        console.error(err);
      }
      if (info !== undefined) {
        console.error(info.message);
        res.status(403).send(info.message);
      } else {
        User.findOne({
          where: {
            username: req.body.username,
          },
        }).then((userInfo) => {
          if (userInfo != null) {
            console.log('user found in db');
            userInfo
              .update({
                first_name: req.body.first_name,
                last_name: req.body.last_name,
                email: req.body.email,
              })
              .then(() => {
                console.log('user updated');
                res.status(200).send({ auth: true, message: 'user updated' });
              });
          } else {
            console.error('no user exists in db to update');
            res.status(401).send('no user exists in db to update');
          }
        });
      }
    })(req, res, next);
  });
};
```

```
Feature: Registration
  As a user
  I want to register an account
  So that I can join the community discussions

  Scenario: Successful registration
    Given I am a new user
    When I register with valid account details
    Then I should be granted access to my account
    And I should receive an email with login instructions
```

```
defineFeature(feature, (test) => {
  test('Successful registration', ({ given, when, then, and })
    let createUserInput: CreateUserInput;
    let restfulAPIDriver: RESTfulAPIDriver;
    const compositionRoot: CompositionRoot = new CompositionRoot();
    const server = compositionRoot.getWebServer();
    let response: any;

    beforeAll(async () => {
      await server.start();

      restfulAPIDriver = new RESTfulAPIDriver(server.getHttp());
    });

    given('I am a new user', async () => {
      createUserInput = new UserBuilder()
        .withFirstName('Khalil')
        .withLastName('Stemmler')
        .withUsername('stemmlerjs')
        .withRandomEmail()
        .build();
    });

    when('I register with valid account details', async () => {
      response = await restfulAPIDriver.post('/users/new', createUserInput);
    });

    then('I should be granted access to my account', async () => {
      expect(response.body.success).toBeTruthy();
      expect(response.body.error).toBeFalsy();
      expect(response.body.data.id).toBeDefined();
      expect(response.body.data.email).toEqual(createUserInput.email);
      expect(response.body.data.firstName).toEqual(createUserInput.firstName);
      expect(response.body.data.lastName).toEqual(createUserInput.lastName);
      expect(response.body.data.username).toEqual(createUserInput.username);
    });
```

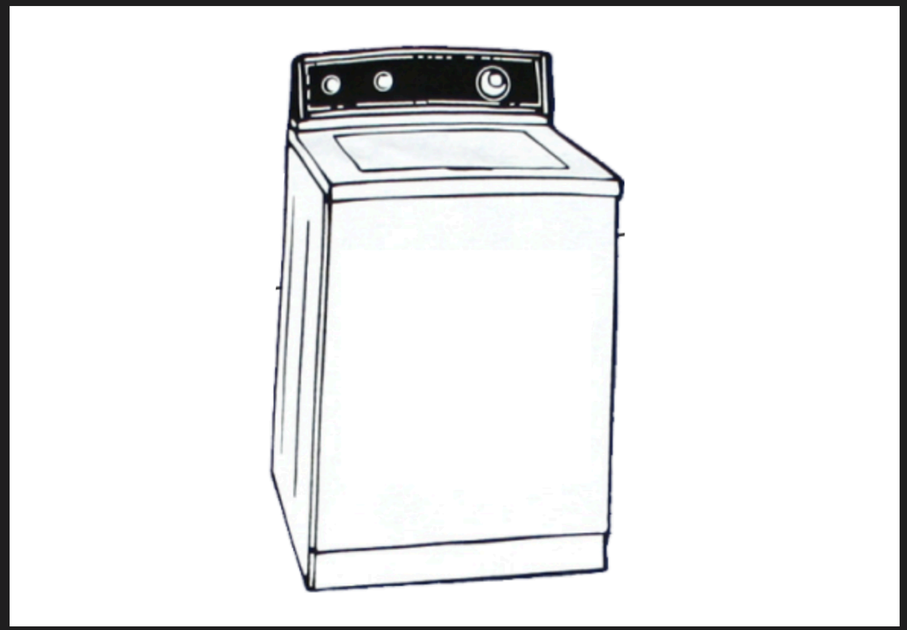
# #2: Separating the Theoretical from the Physical

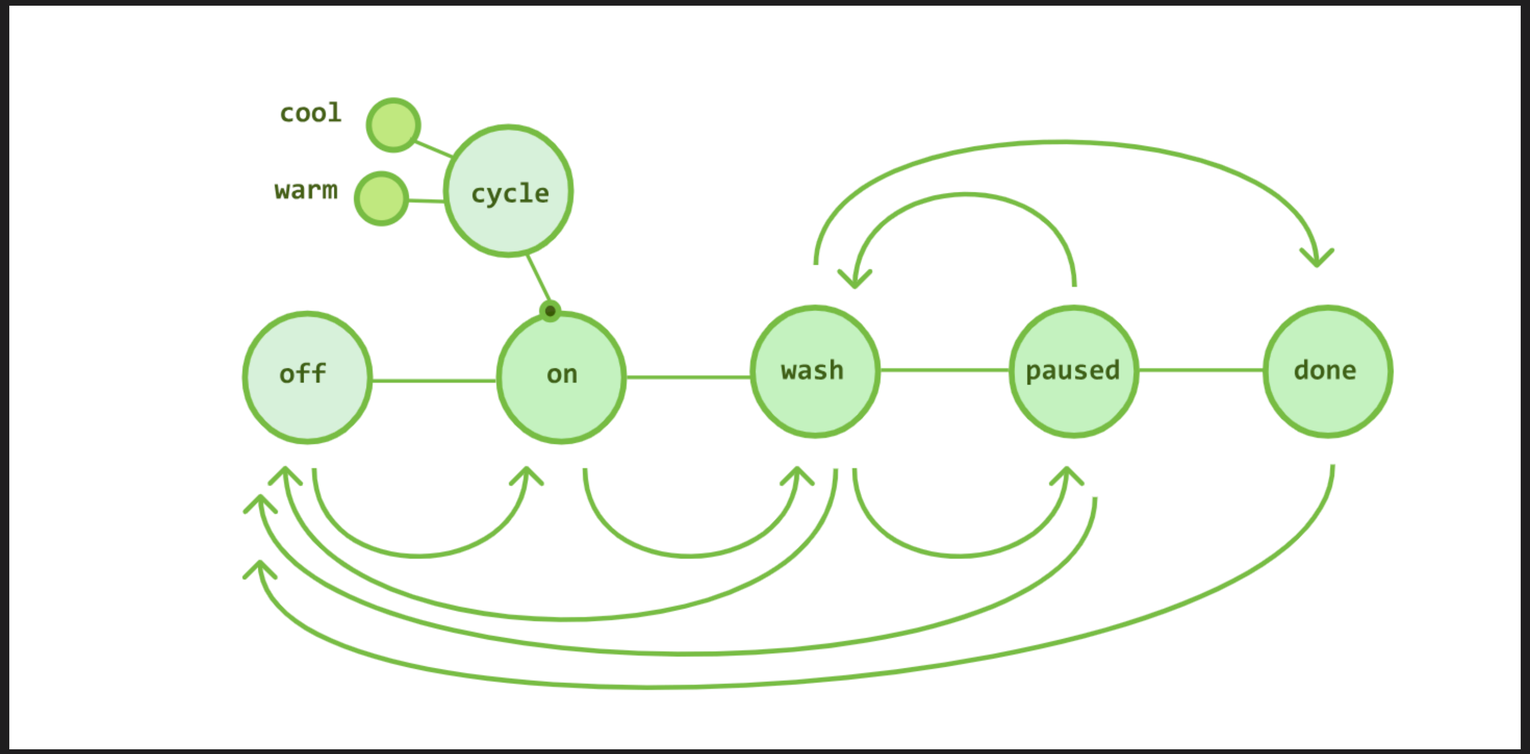

Who are we helping?
Vision
What will we help them achieve?
Roles-Goals
What will they be enabled to do?
Capabilities
How can our application help users do this?
Features
What are real-world scenarios that users can perform?
Stories
Can we decide on smaller, exact units of work?
Acceptance criteria
How can we make it easier to understand?
Concrete examples
How will it work?
Code
Theoretical
What
Physical
How



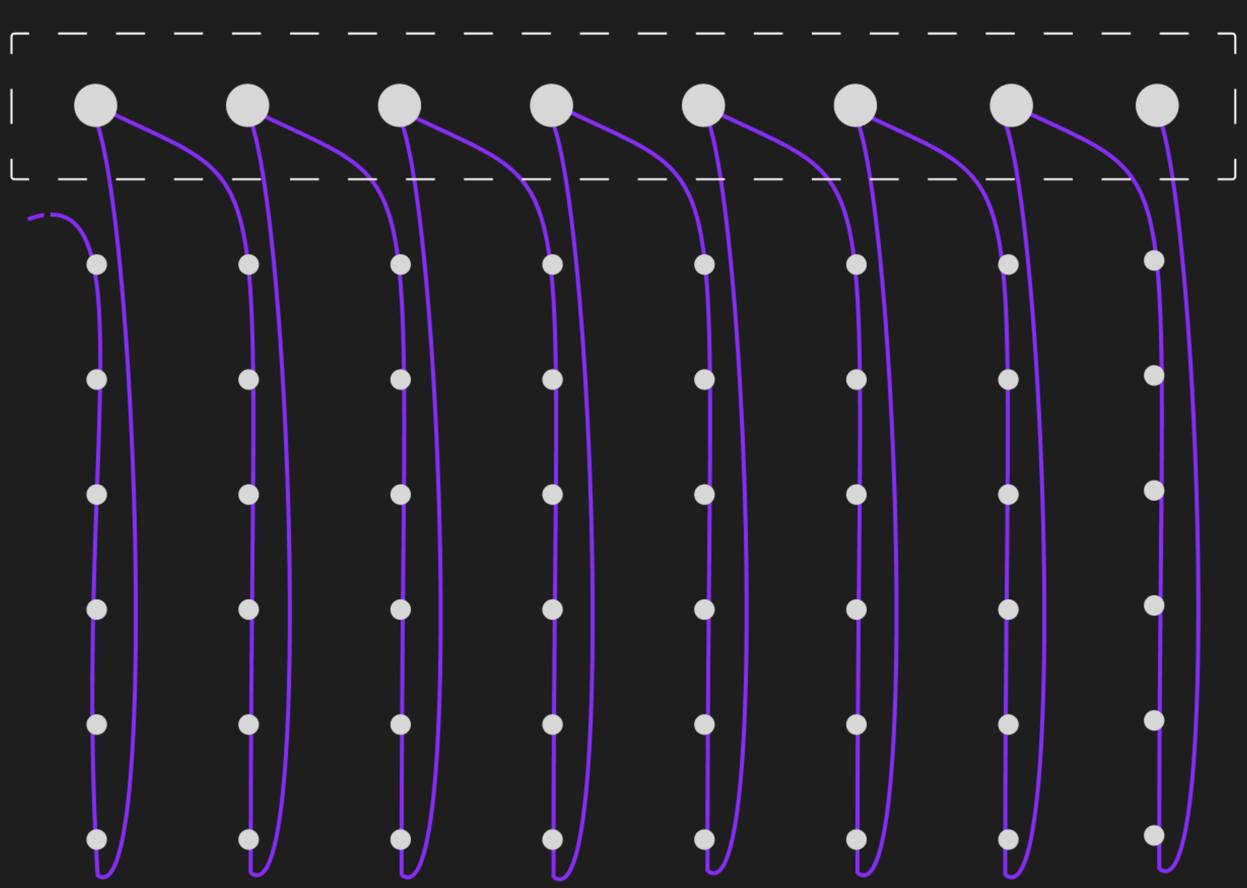
# Theoretical

# For example, let’s discuss the washing machine again




cool
warm
cycle
off
on
wash
paused
done

```
// Options for the wash cycle
type WashOptions = {
  dryLevel: 'low' | 'medium' | 'high'
  temperature: 'cold',
  duration: 'hour',
  ecoEnabled: false
}

// The abstraction
class WashingMachine {

  // Private instance variables
  ...

  public startCycle (options: WashOptions): void {
      // Parse the options
    // Get access to the physical layer
    // Convert options into commands
    // Lots of low-level code
    // And so on...
    ...
  }

  // More methods
  ...

}
```



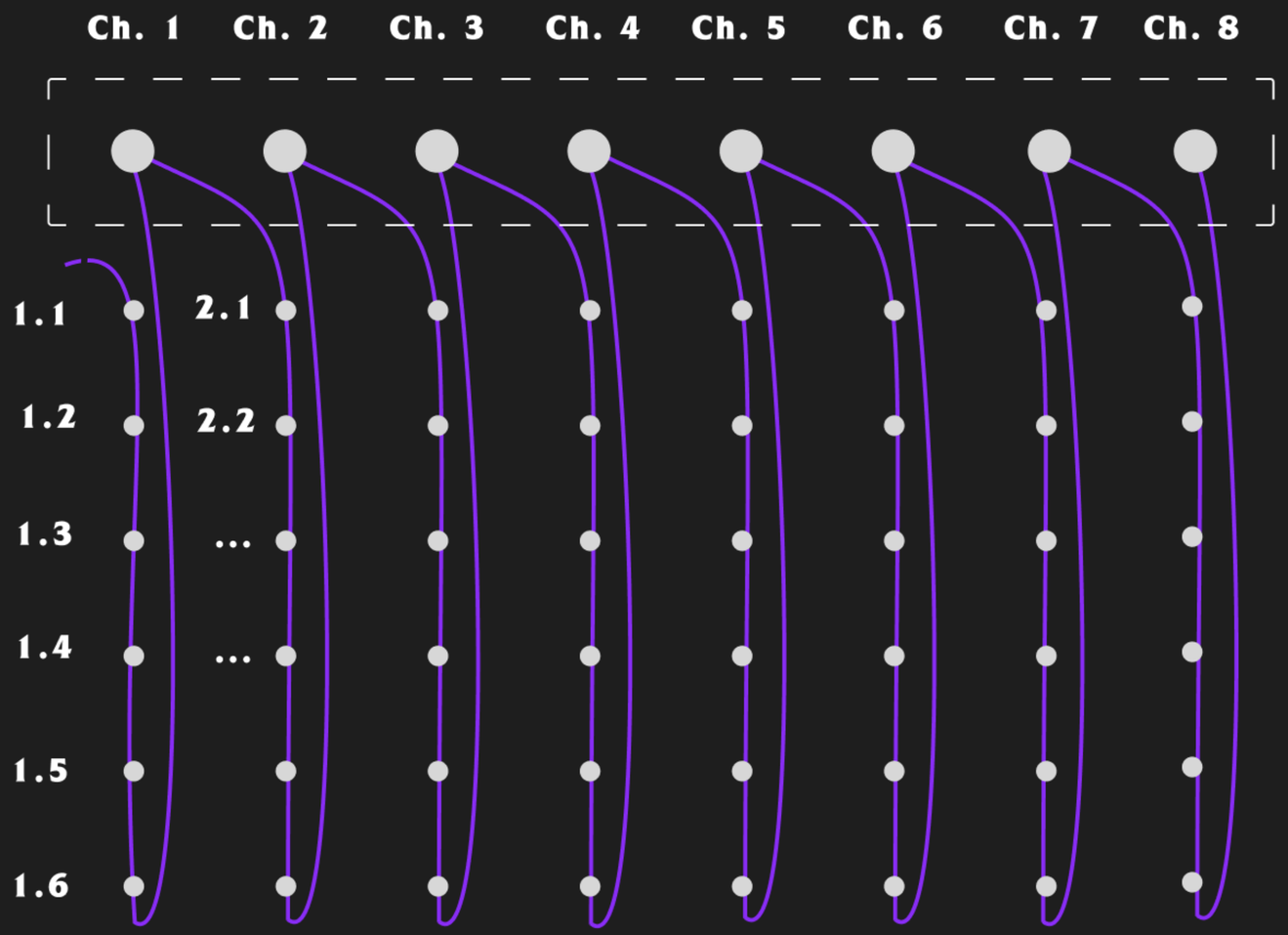
Events

**Theoretical**

Steps

**Physical**

# Consider the chapters of a book




Events

**Theoretical**

Steps

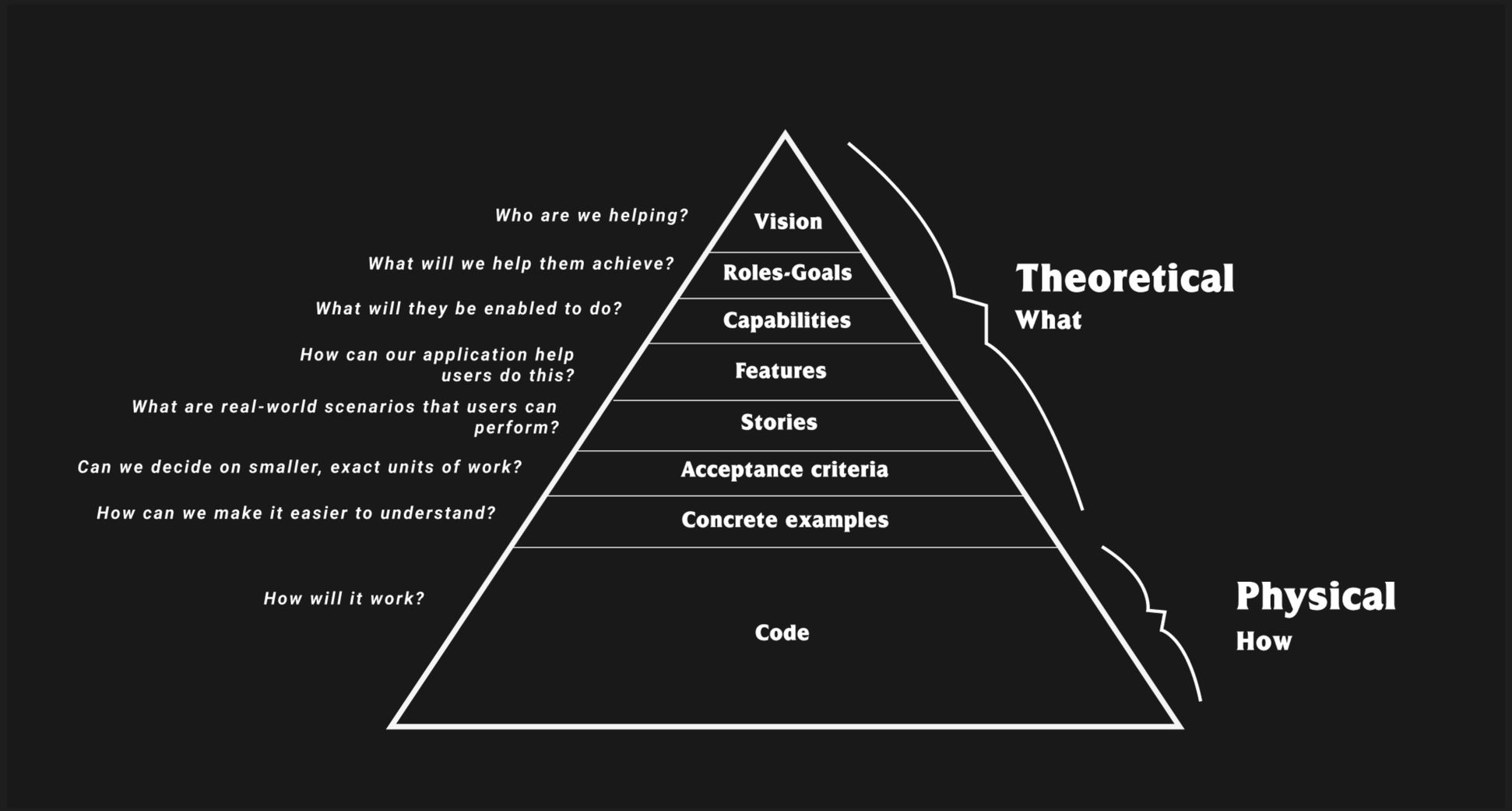
**Physical**

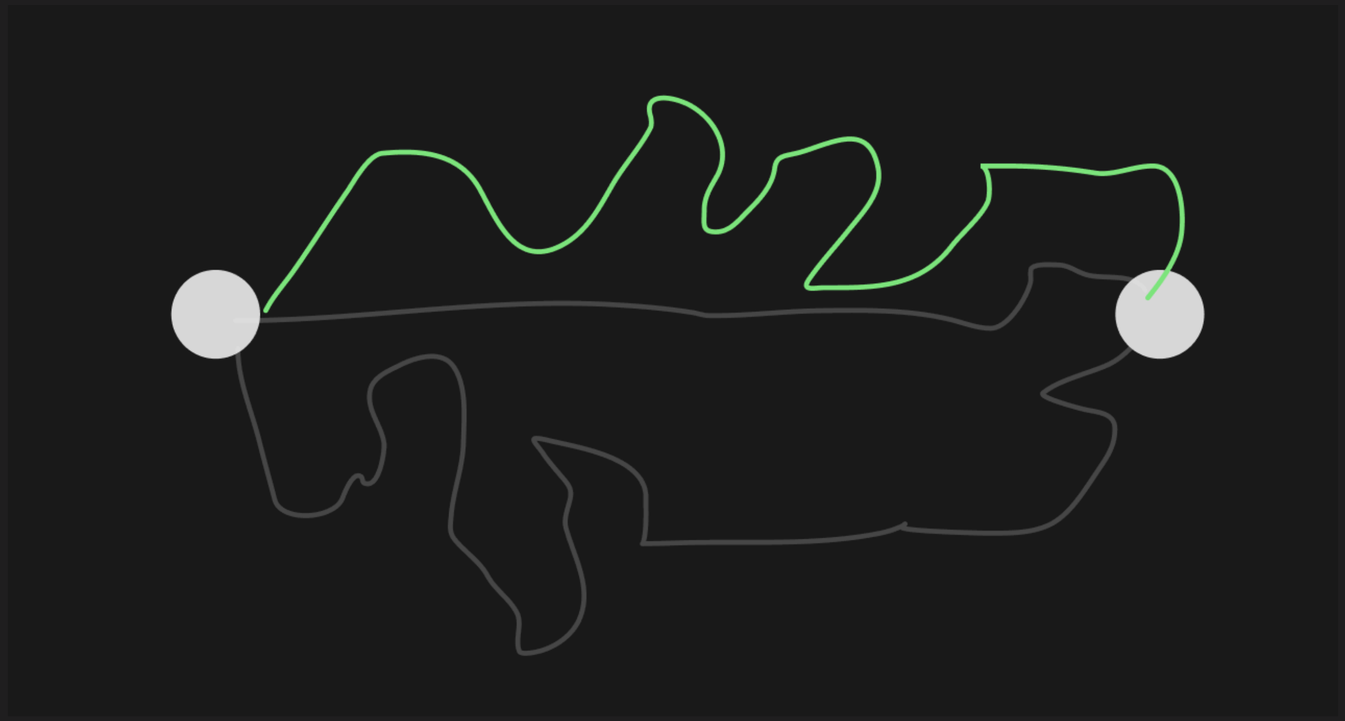
# Guidelines for implementing abstraction

## Step #1: Clarify the end goal

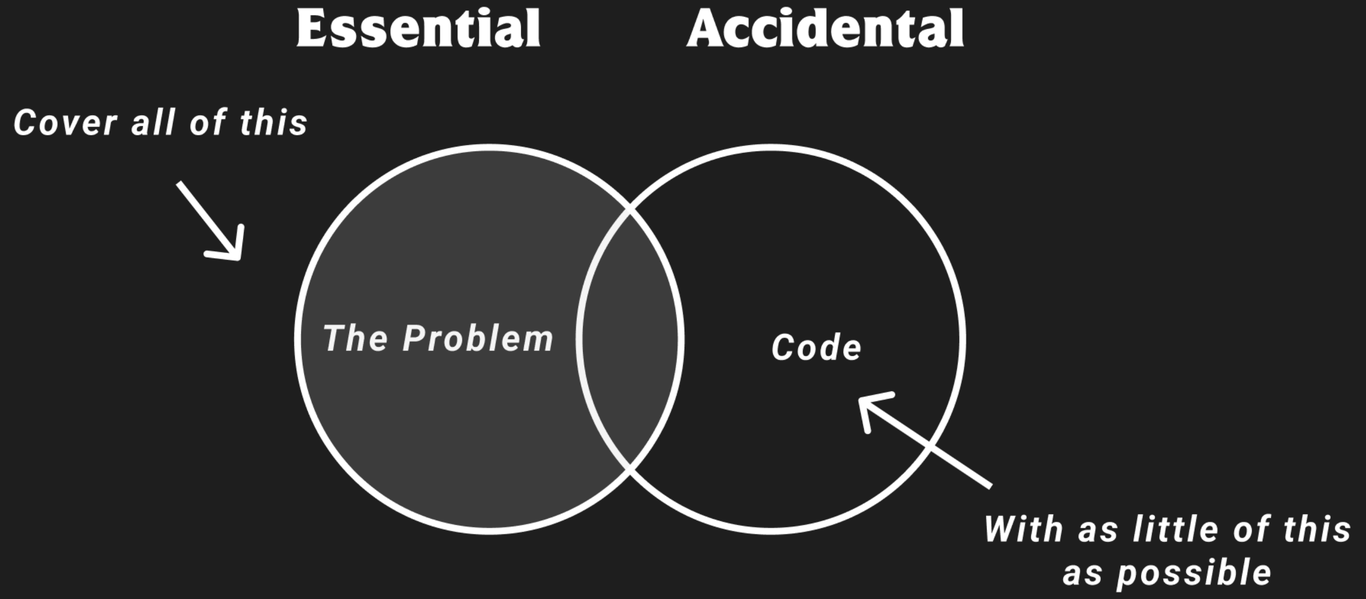
## Step #2: Think backwards

## Step #3: Implement the minimum amount of code necessary (the physical steps)


Who are we helping?
Vision
What will we help them achieve?
Roles-Goals
What will they be enabled to do?
Capabilities
How can our application help users do this?
Features
What are real-world scenarios that users can perform?
Stories
Can we decide on smaller, exact units of work?
Acceptance criteria
How can we make it easier to understand?
Concrete examples
How will it work?
Code
Theoretical
What
Physical
How



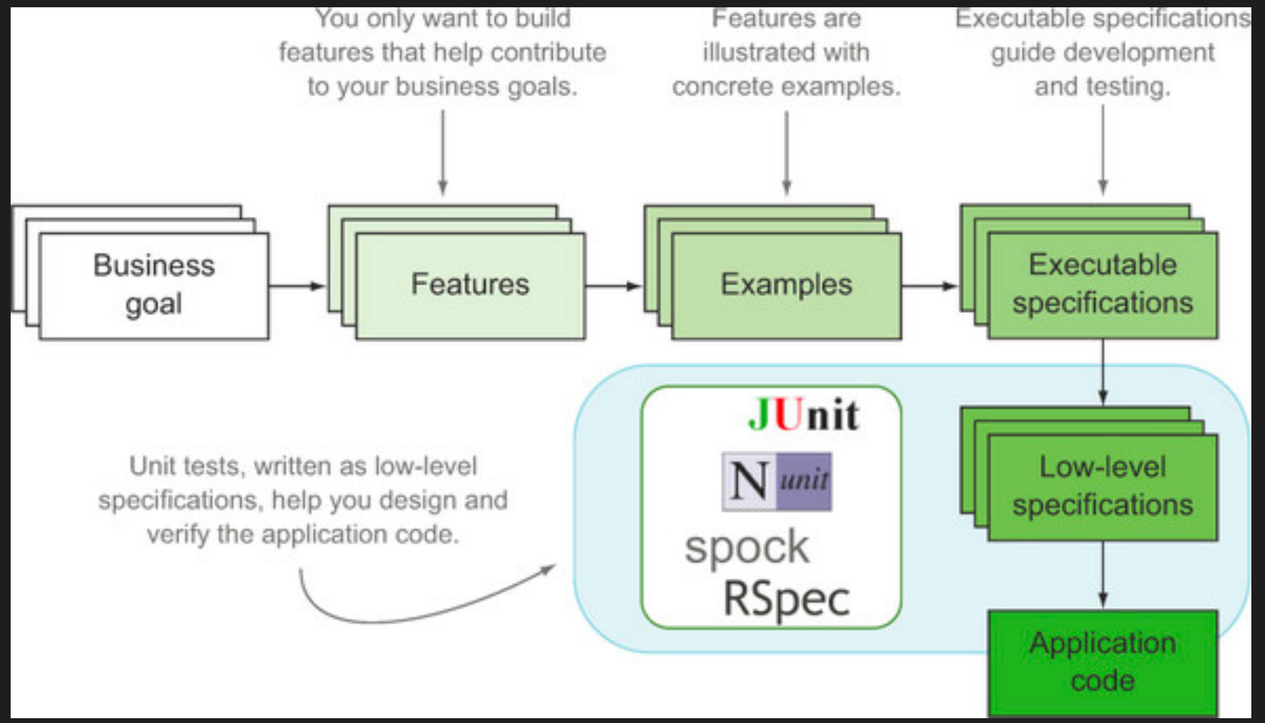
# There are two types of Complexity



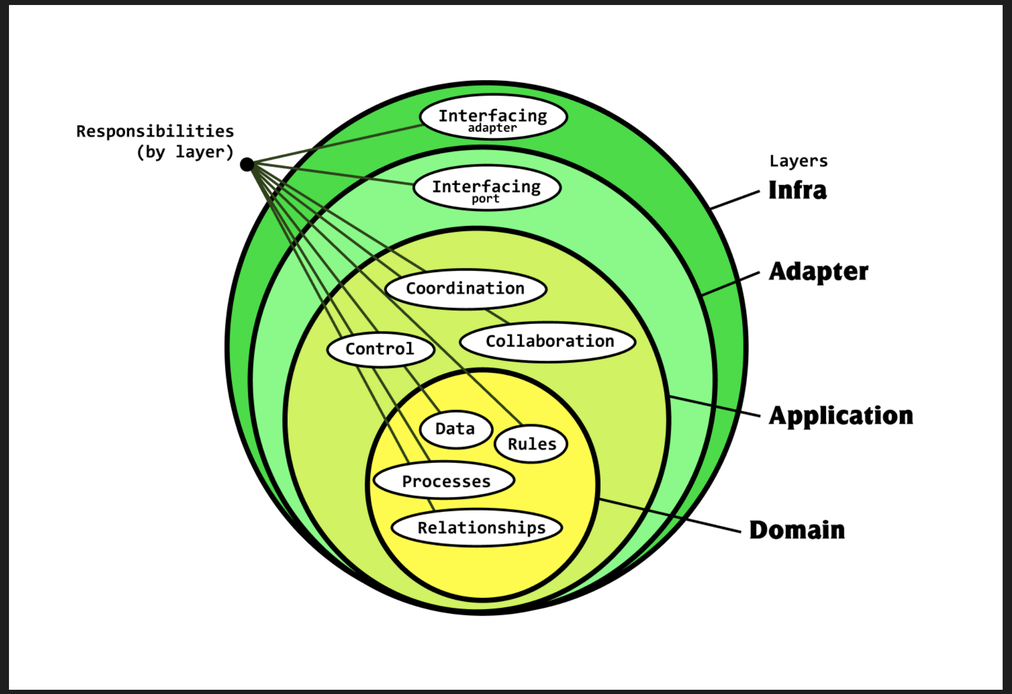
# Common challenges

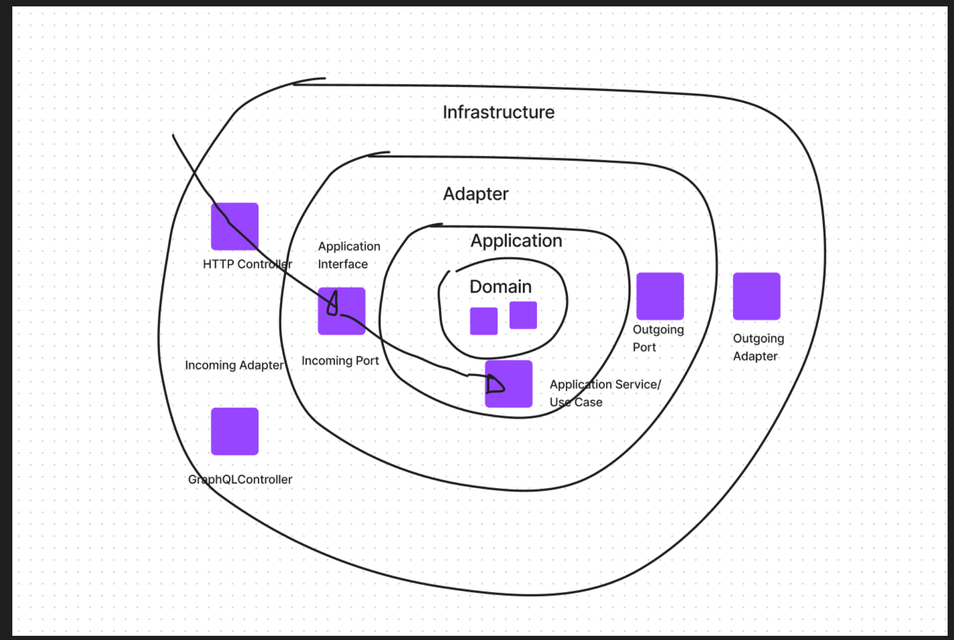
- *Perfectionism*
  - *Getting stuck in 1.1.2, 1.1.3*
- *Starting from the Physical (Code-First)*

# What about the Physical?



# The Physical Layers are Stereotypical



# What we learned

**Remember: You are an Abstraction Designer. You visualize and realize the solutions to real-life problems with code. You Give Back Time and Enrich Time using Abstraction.**