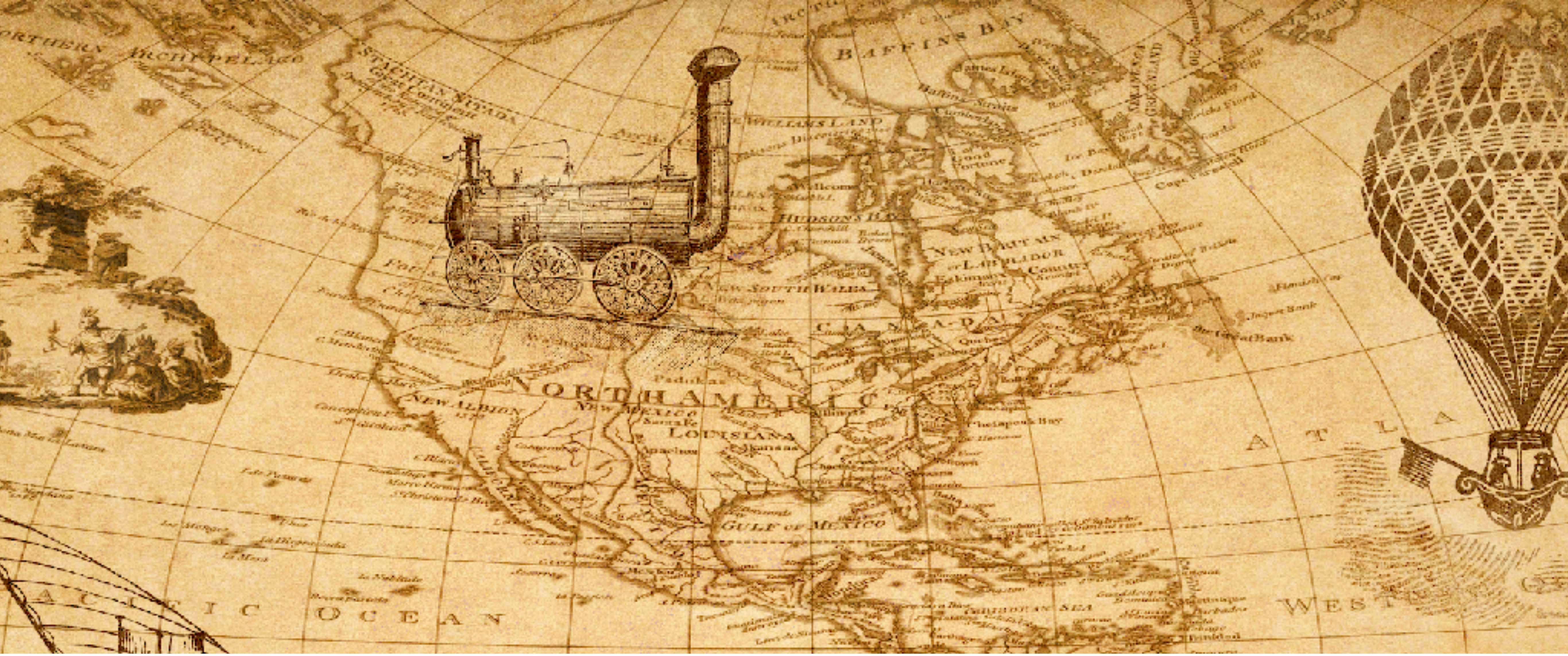
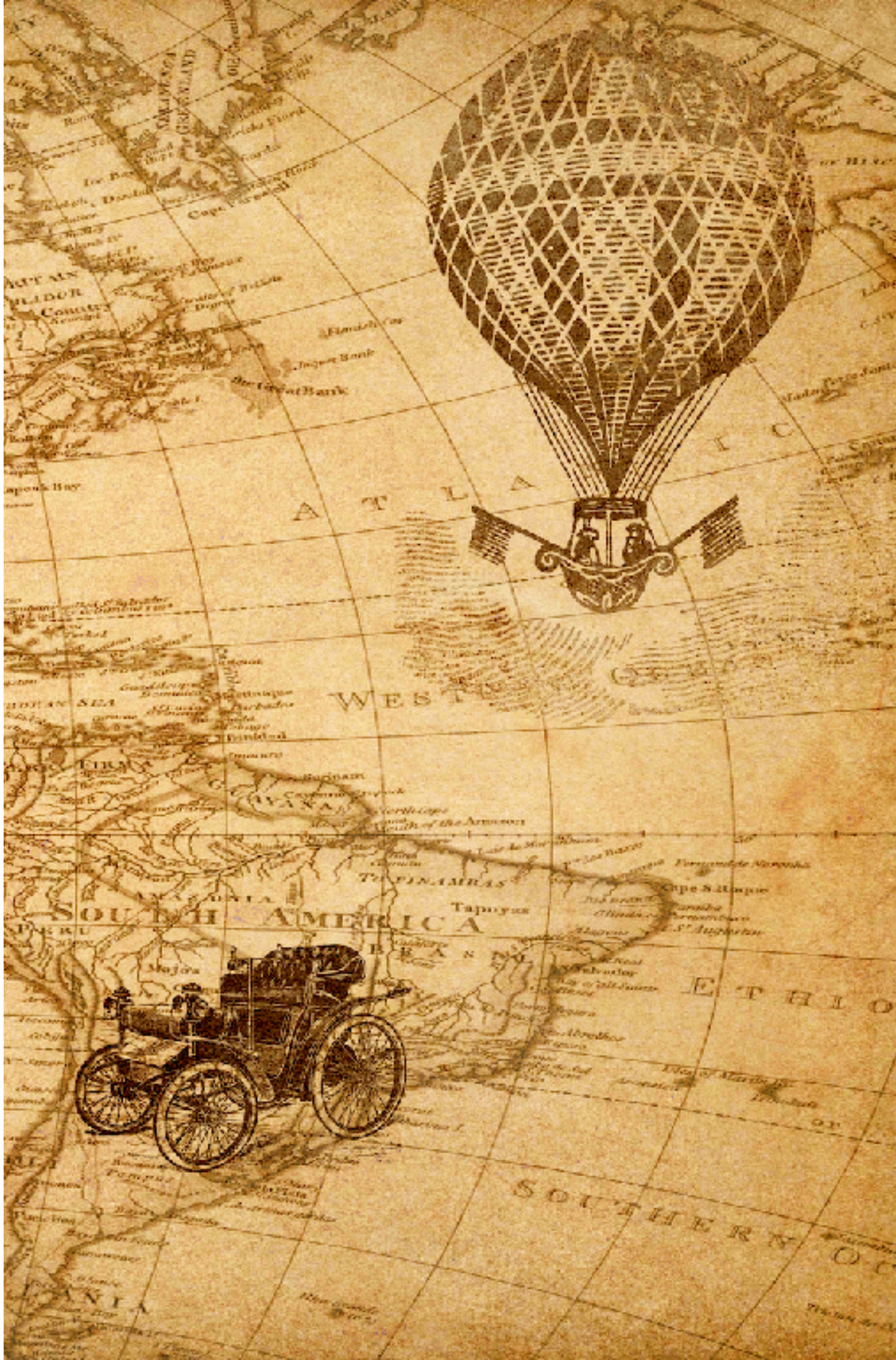
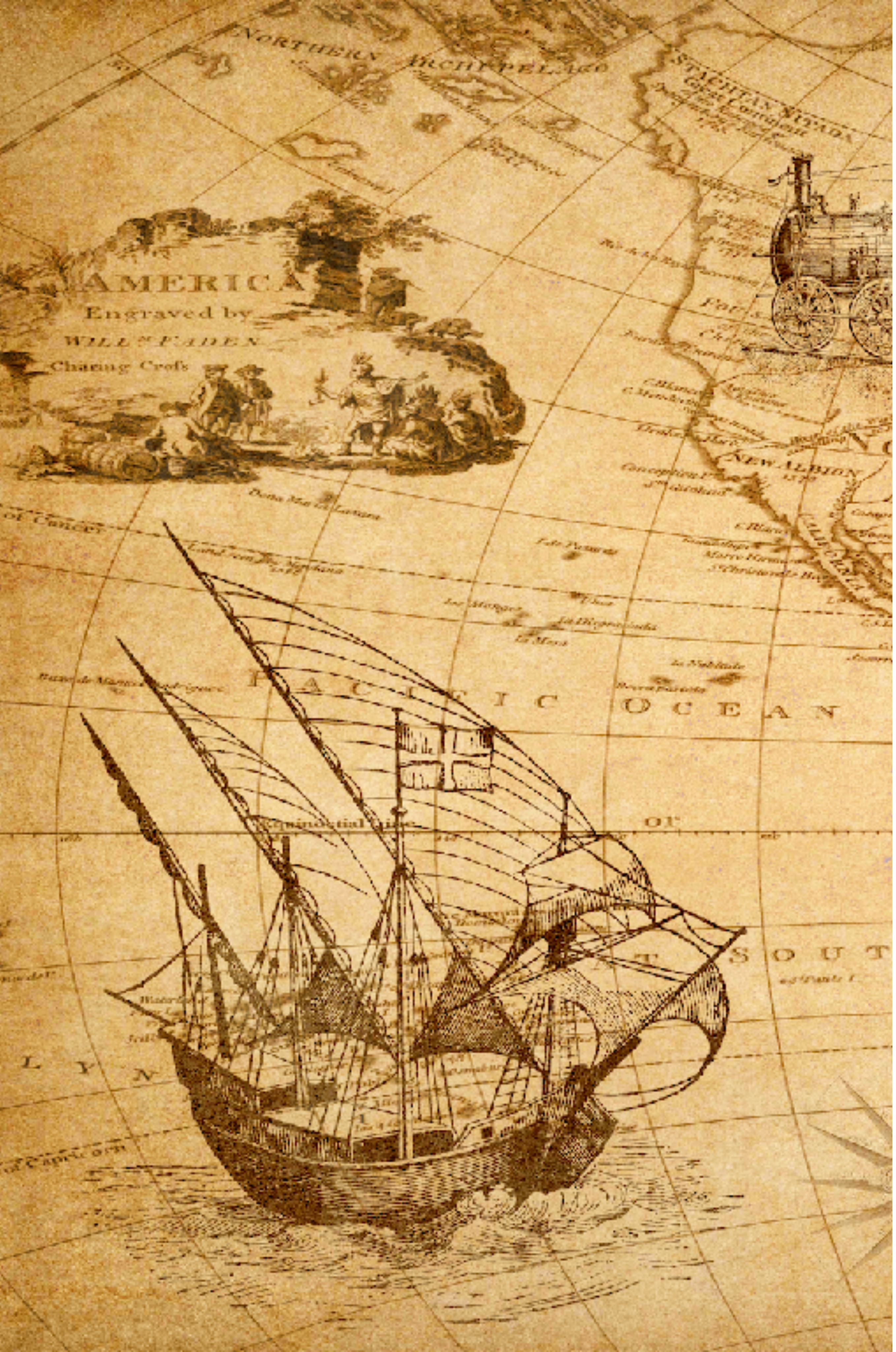


ABSTRACT DATA REPRESENTATION

SELF CARE

- Look for the needs of others
- Respond with grace
- Say thank you
- Take interest in others





DATA ABSTRACTION

- Previously we discussed the concept of data abstraction
- Separate the underlying representation from the usage of the data type
- We've worked through this with multiple data types
- Rational numbers, sets, and so on

EXAMPLE: COMPLEX NUMBERS

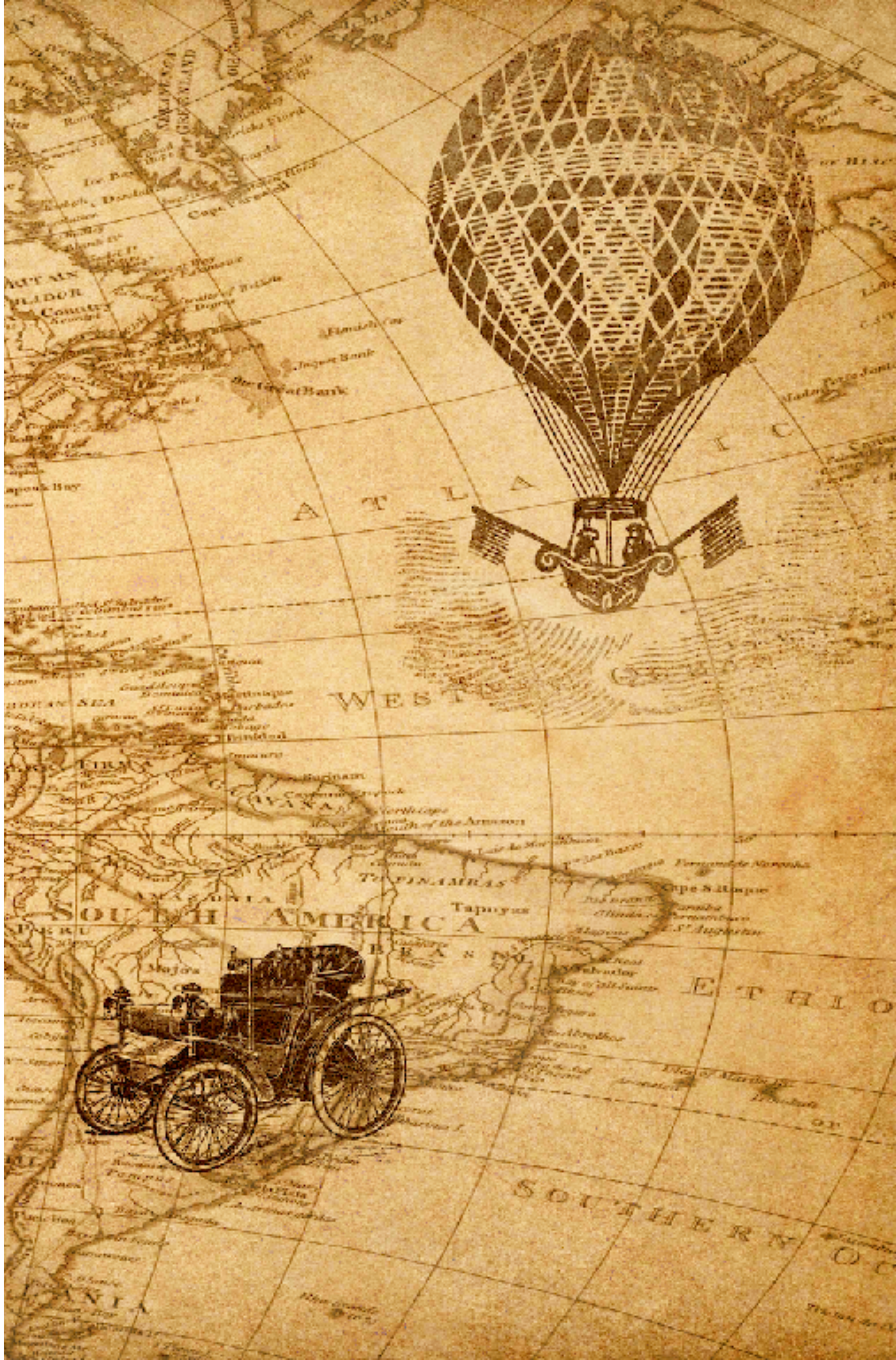
- We've previously discussed a possible implementation of rational numbers
- Constructed by building a pair from a numerator and denominator

- Could we represent a complex number in another way?
- How about a whole number part and a decimal part?
- Use a fixed mantissa and digits of precision, perhaps
- How would addition work?

CONVERTING NUMBER FORMATS

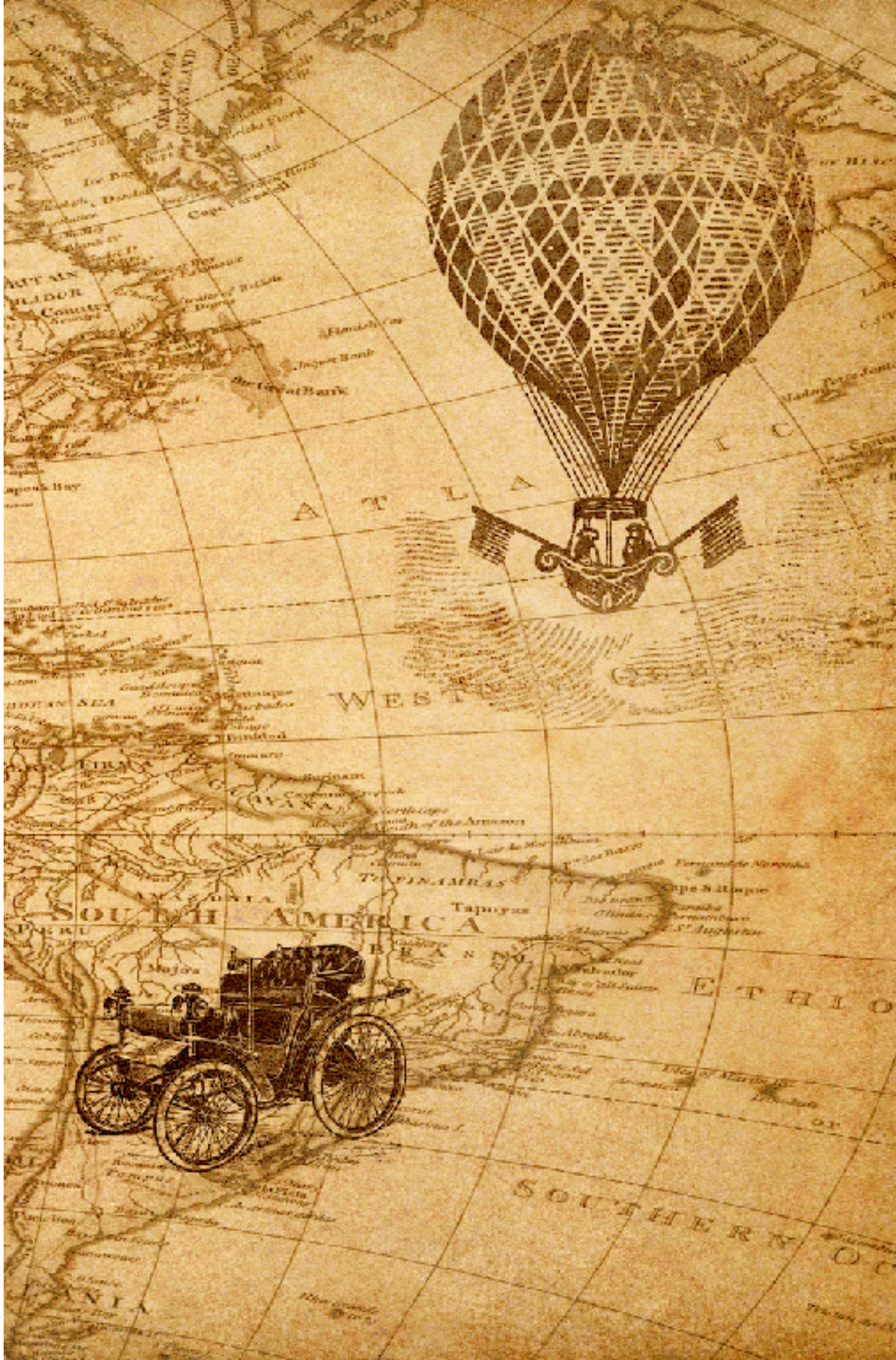
- When we have multiple data formats, it's important to provide conversions between similar formats

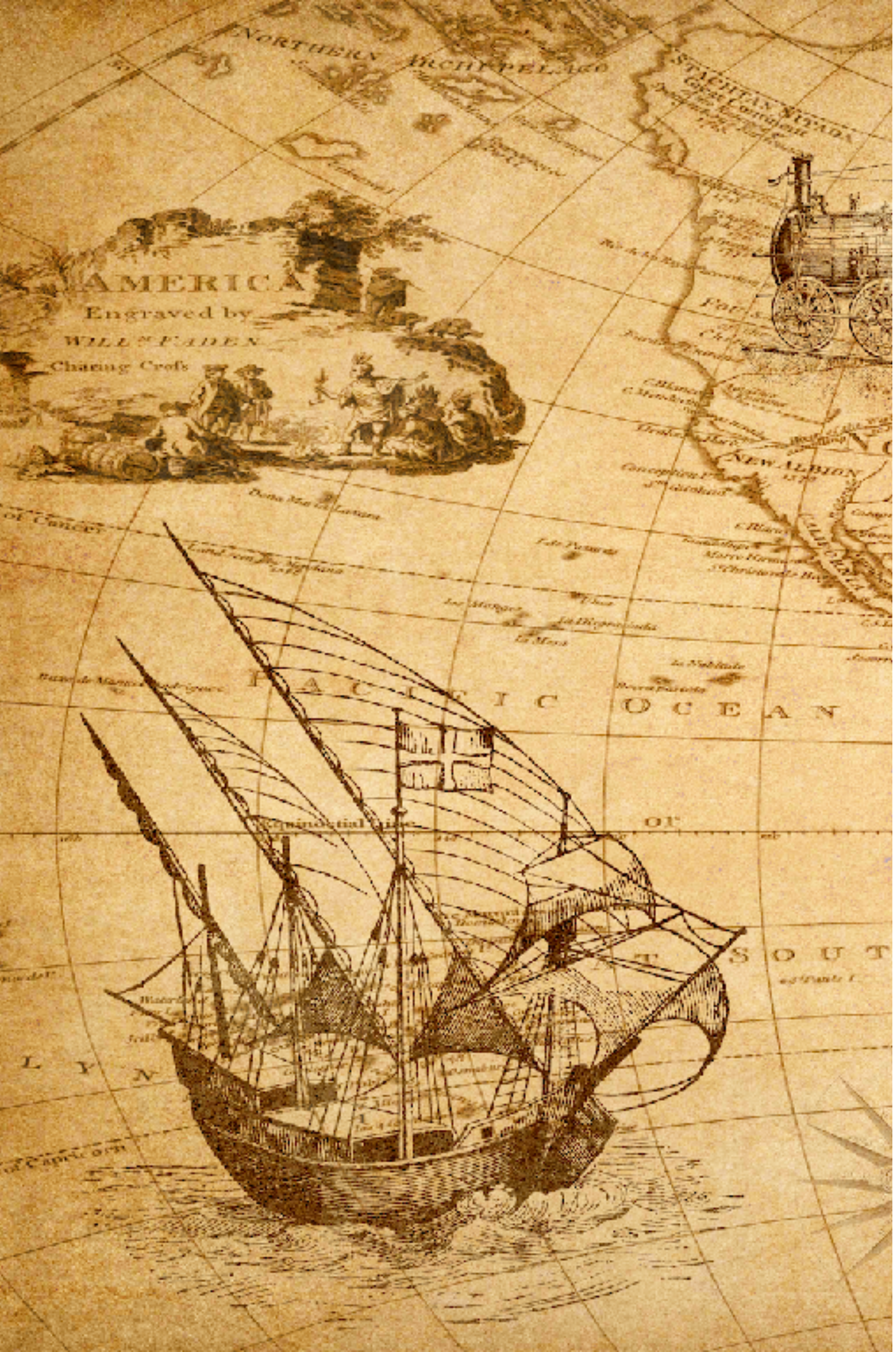
- We can easily see this in strongly typed languages such as Rust
    - `let a = 1;`
    - `let b = a as f32;`



LOW LEVEL REPRESENTATIONS

- These low level representations in Rust don't provide the abstraction we need to further our discussion
- For example, on modern hardware, floating point numbers and integers are held in different registers entirely
- For our discussion, we can pretend they're in the same register type
- (What does this mean? Take the compiler class in the fall.)



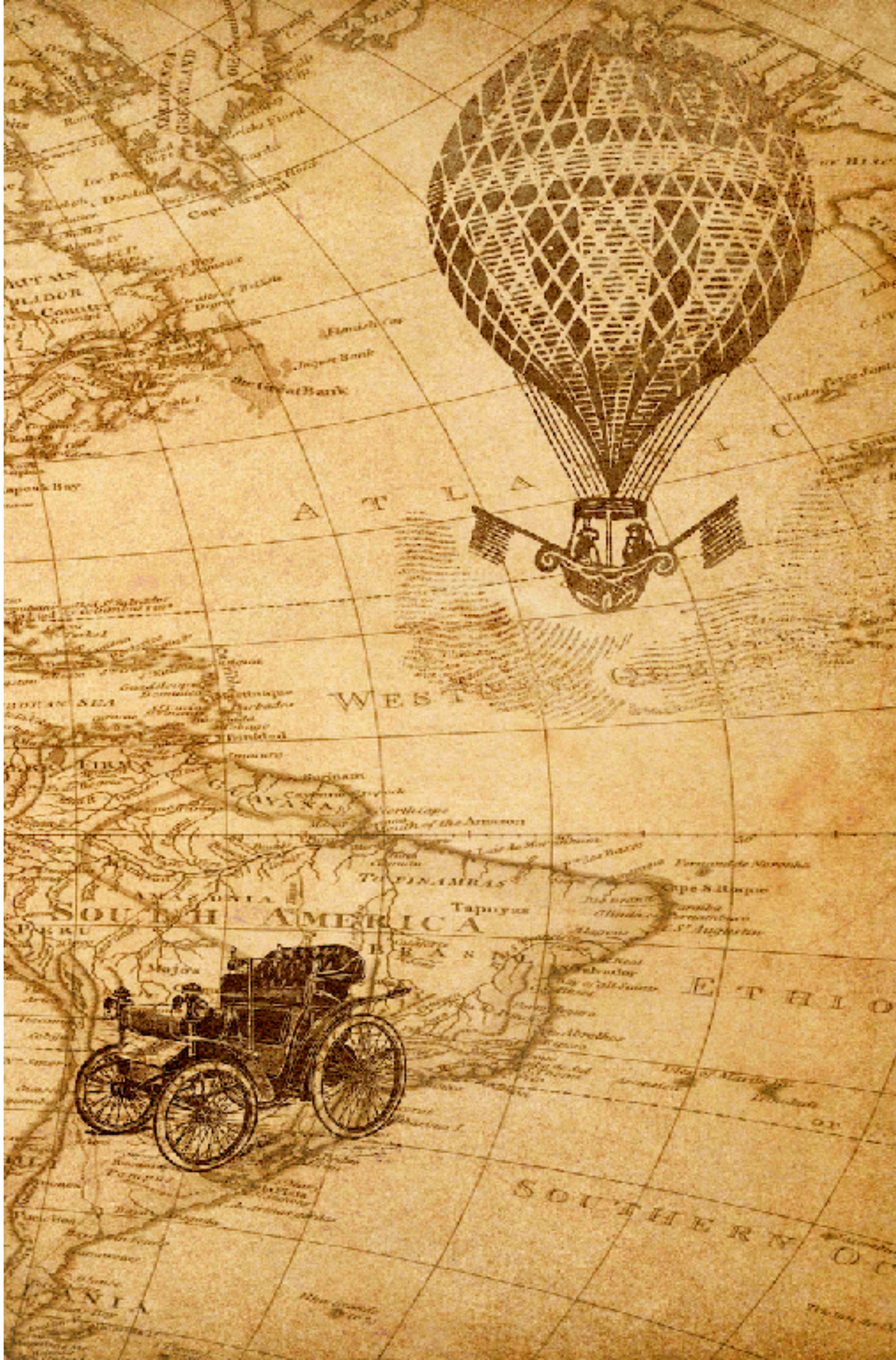


PRINCIPLE OF LEAST COMMITMENT

- This is the law of protocol-oriented (or contract-based) programming at its core
- We leave the decision about how to represent data undecided until the last possible minute
- We should even feel free to change our minds multiple times during development of a system

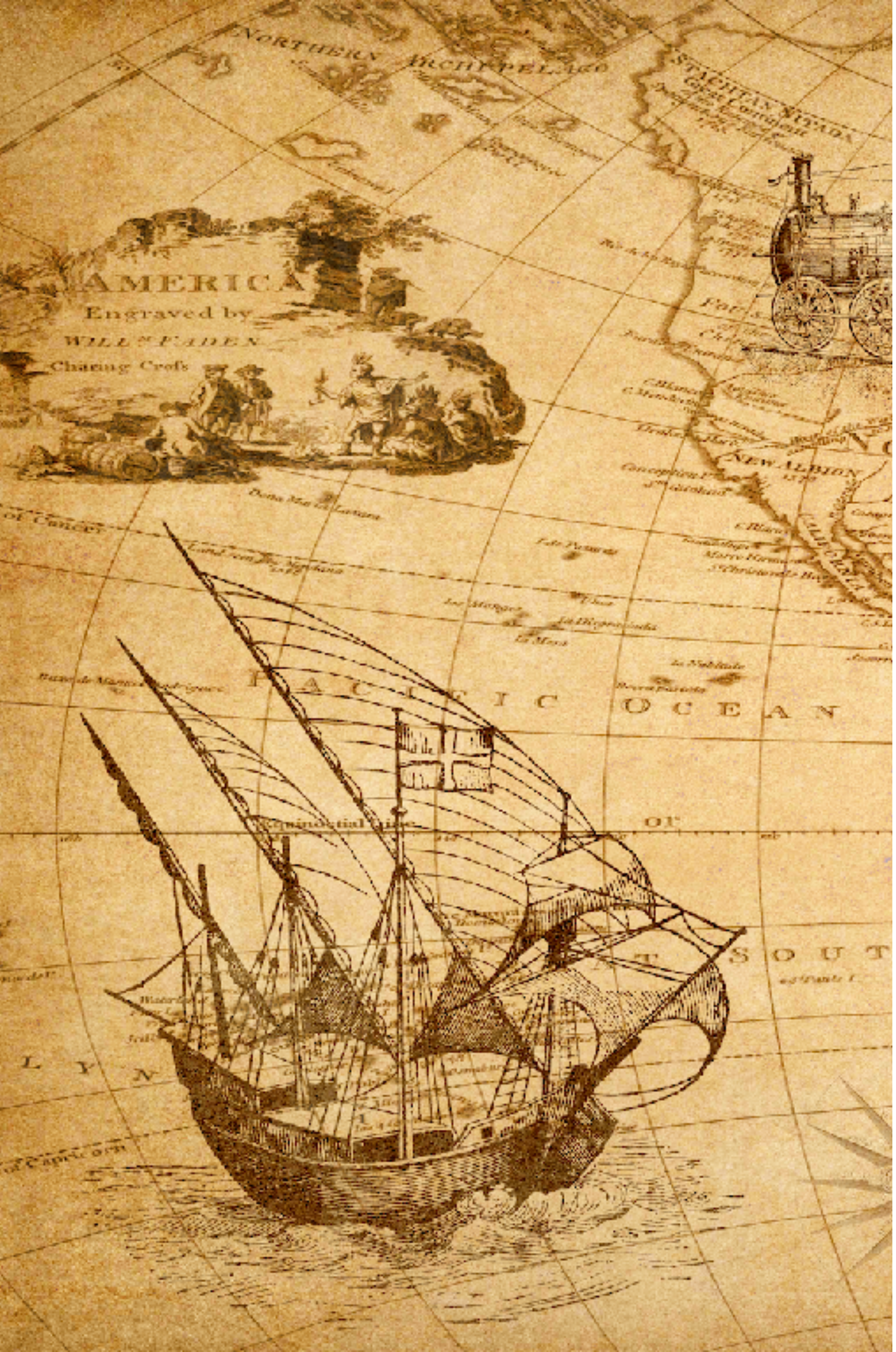
HOW TO DIFFERENTIATE?

- If we have multiple representations of data implemented via pairs — such as complex numbers provided as both rational numbers and fixed point / mantissa based numbers, then how do we know which implementation to use?



COMPLEX NUMBERS

```
(define (make-rat numerator denominator)
        (cons numerator denominator))

(define (make-fpm whole decimal)
        (cons whole decimal))

(make-rat 2 3)
(2 3)
(make-fpm 2 3)
(2 3)
```



DATA TAGGING

- We can solve the problem of being able to determine the type of data by using a tagging technique
- We put a small amount of data in front of our actual data, to indicate what its type is
- This is a fundamental technique in some object oriented languages, such as smalltalk and objective-c

TAGGING DATA

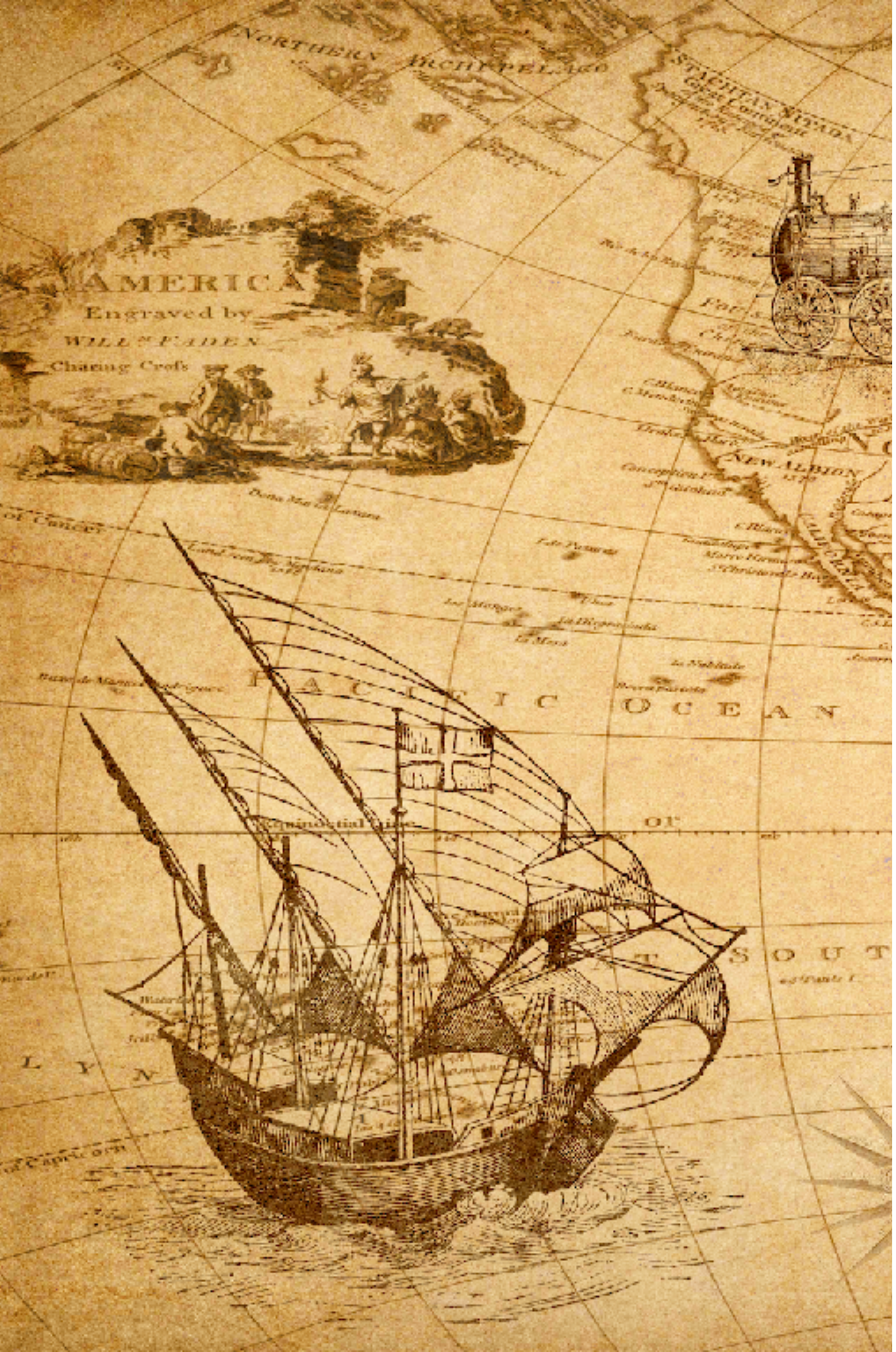
```
(define (attach-tag type-tag contents)
  (cons type-tag contents))

(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum: TYPE-TAG" datum)))

(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum: CONTENTS" datum)))
```

USING TAGGING

```
(define (rational? z)
        (eq? (type-tag z) 'rational))
(define (fixedpoint? z)
        (eq? (type-tag z) 'fixedpoint))
```

UPDATING MAKE-RAT

```
(define (make-rat num denom)
        (attach-tag 'rational (cons num denom)))
```



SYSTEMIC CHANGE

- A systemic change like this does have some serious implications
- Never use direct accessors
  - for example, "add-rat" should never make a direct call to *car / cdr*, but instead should use a *get-numerator* method instead
- This is much easier to do if planned from the outset
- Replace generic selectors with tagged selectors

DISPATCH ON TYPE

DIFFERENT SYSTEMS FOR RECTANGLES

TLBR	*Origin / Size*	*Center/Extent*
area-tlbr	*area-originsize*	*area-centerextent*
perim-tlbr	*perim-originsize*	*perim-centerextent*

A system like this is said to be brittle, because each type has a uniquely named function

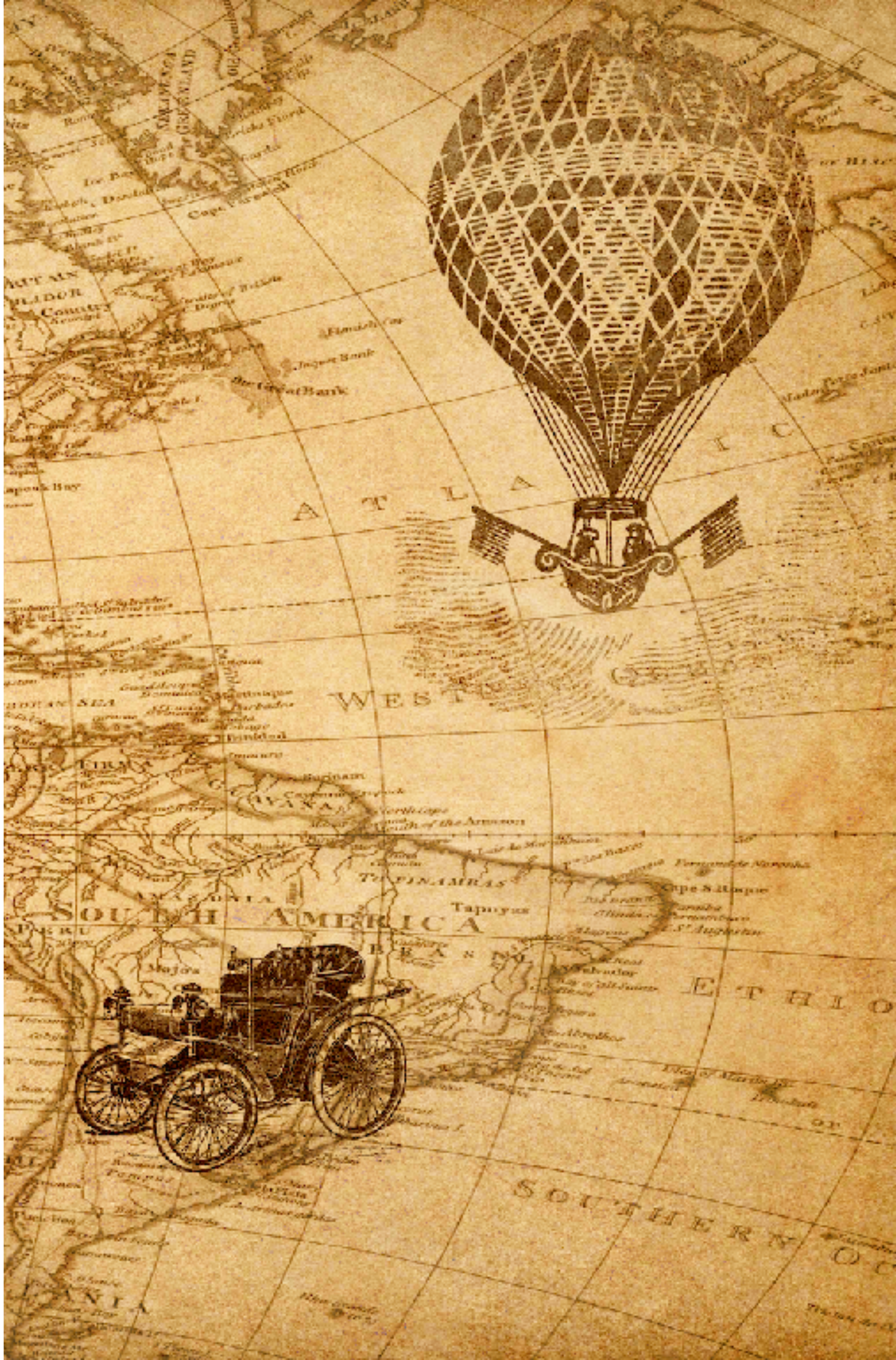
What if we accidentally put together some conflicting names?

What if we can't remember all the names?

What if we shouldn't care what type we're working on?

CRAFTING PACKAGES

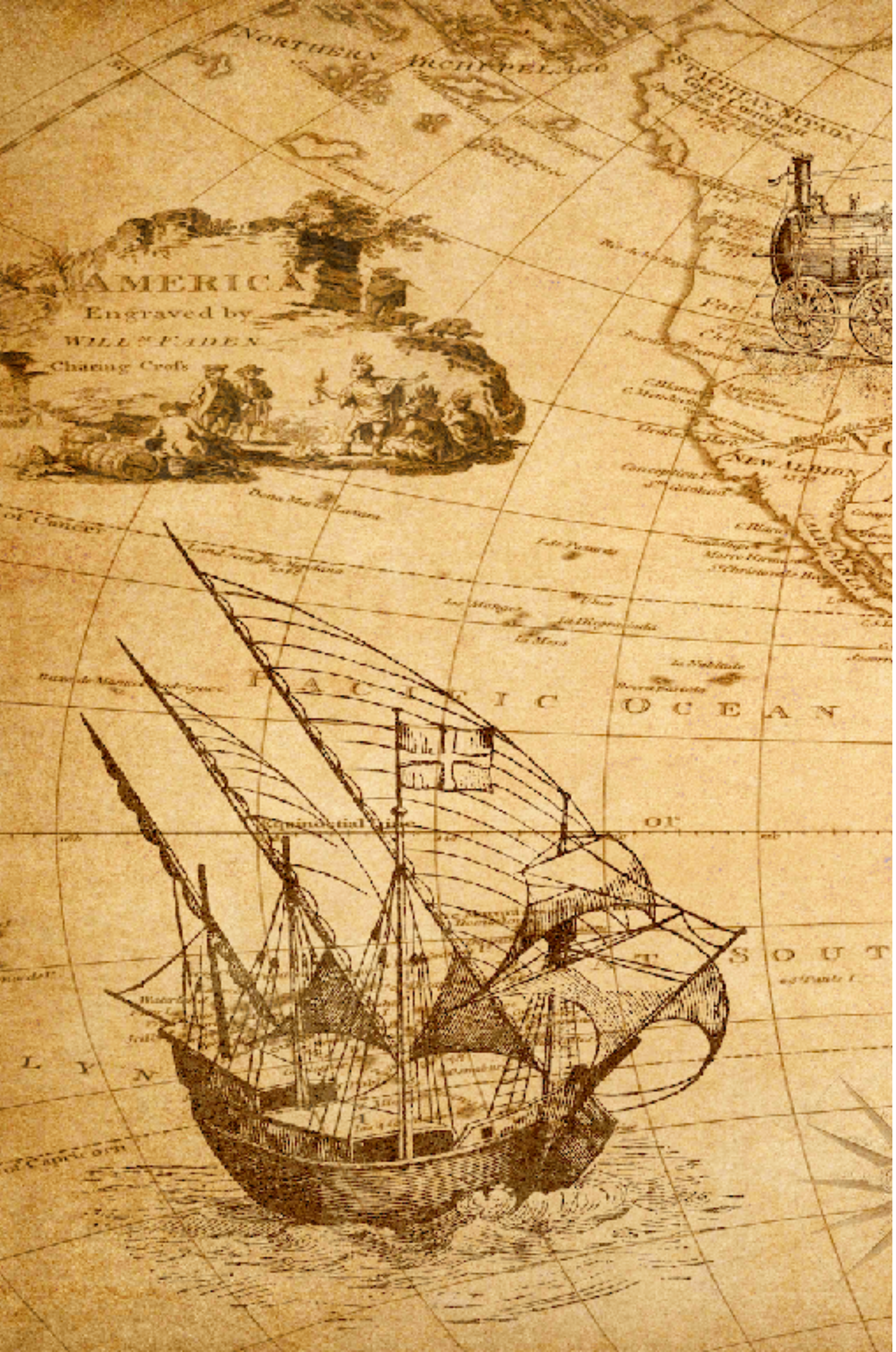
- One way to handle multiple interfaces for a single set of functions is to create a *package* to handle the different implementations
- This can be done with defined functions or lambdas
- Define a function name and the implementation, each package-dependent
- Suppose we have a method: `(put <op> <type> <def>)`



CREATING AN IMPLEMENTATION

```
(put 'area '(tlbr) area-tlbr)
```

Here we define a method for 'area' for the 'tlbr' type



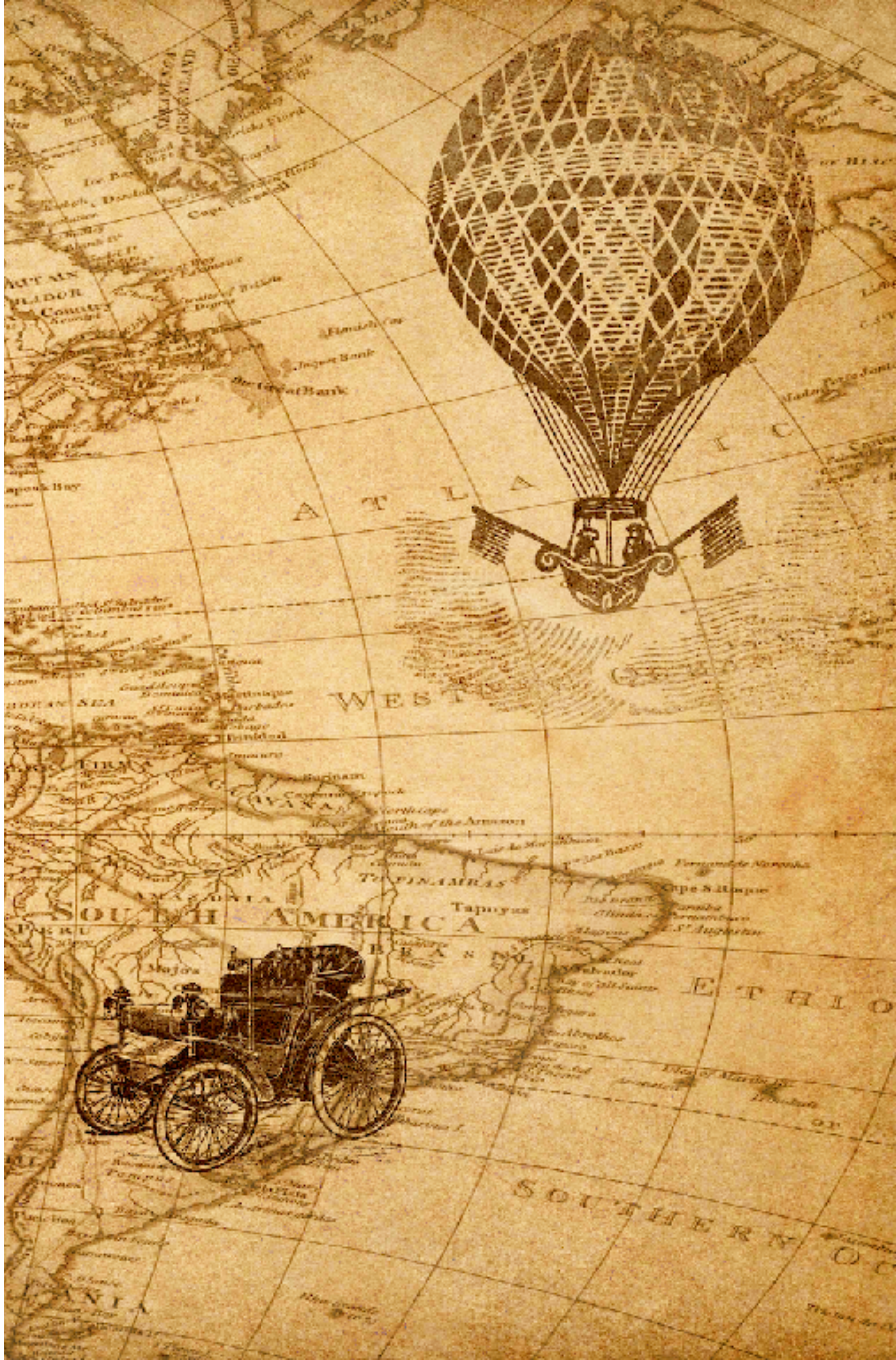
CONCRETE IMPLEMENTATION

- We’ll likely do something like this when crafting our interpreter in the final segment of the semester
- We won’t go further at this time

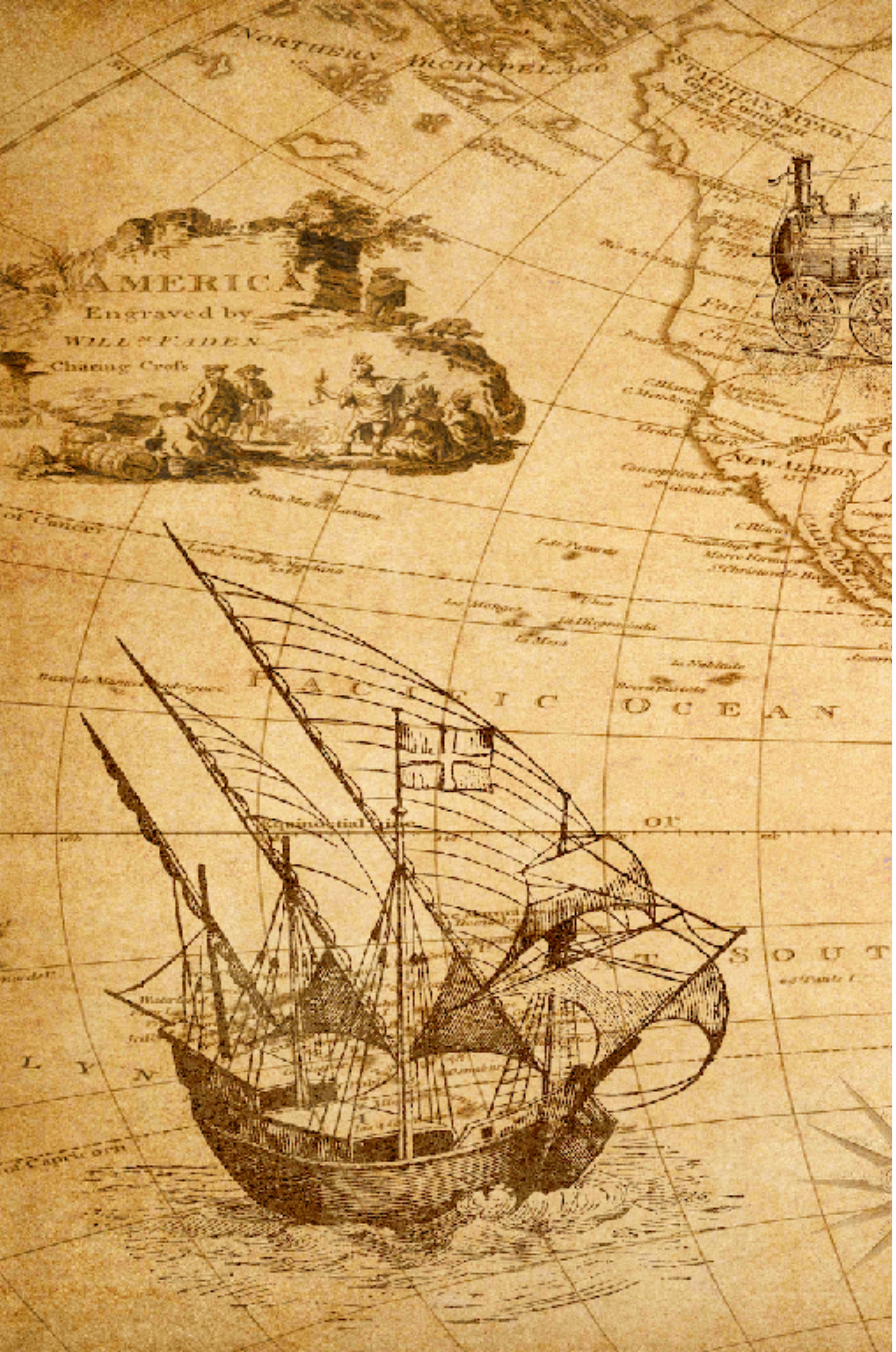
MESSAGE PASSING

- Instead of maintaining a centralized repository of methods, each object could instead know about how to handle particular operations by name

- Table-based dispatch is what we see in C++, for example, whereas message passing is what we see in smalltalk/objective-c



TEST PREVIEW



TEST ON 4/14

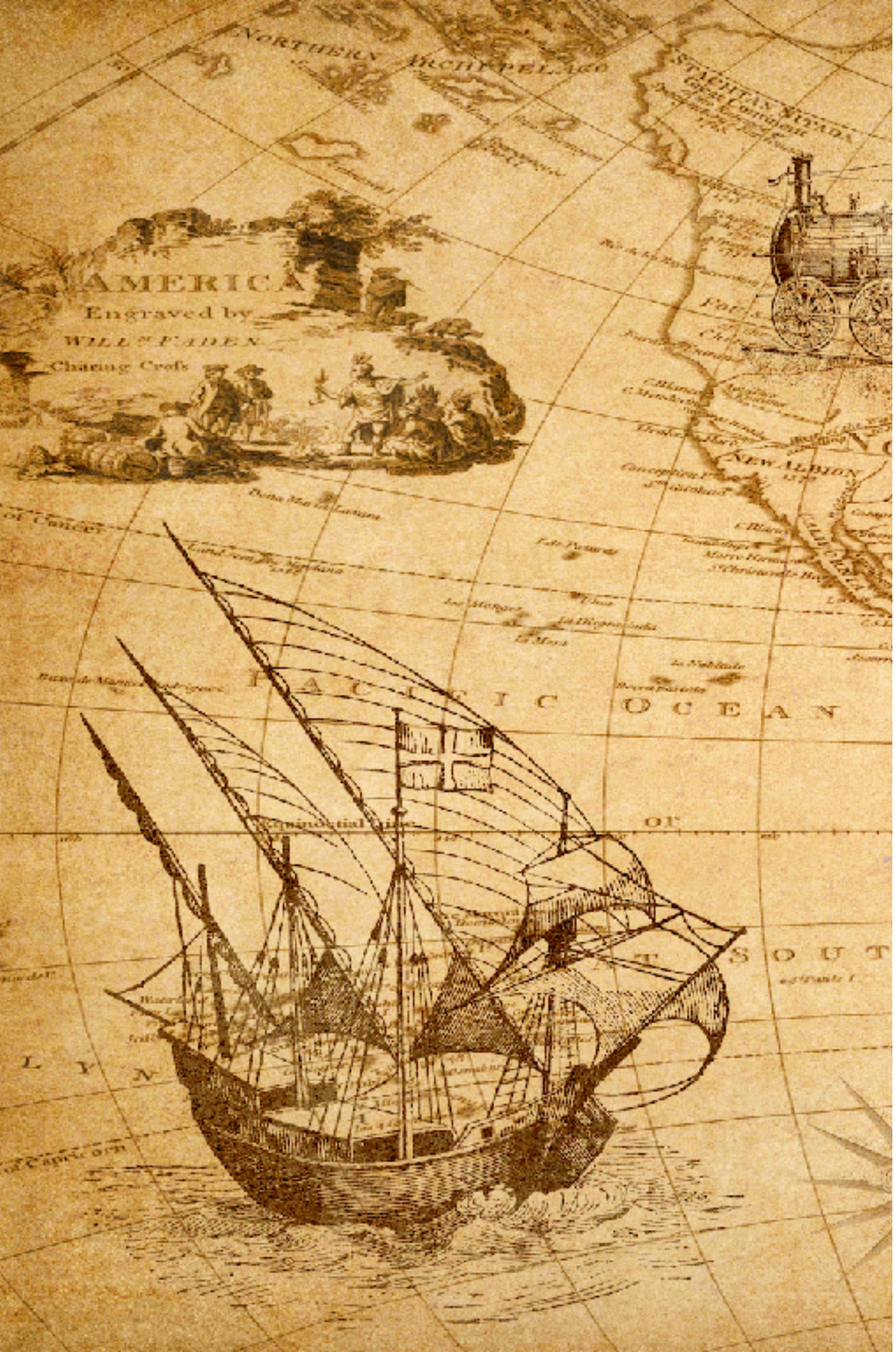
- Data structures in scheme
- Unit Tests in Rust
- Lists in Scheme
- Rust hashmaps / environment
- Symbolic data in Scheme
- Sets in Scheme
- Binary Trees
- Information Retrieval
- Abstract data representation
- Dispatch on Type

> What format will the exam take?

*-This is the #1 question everyone has had,
for some reason…*

HOMEWORK

NORTH AMERICA
NEW ALBION
NEW MEXICO
LOUISIANA
GULF OF MEXICO
HUDSONS BAY
CANADA
SOUTH SEA
OCEAN



TAKE THE SAMPLE TEST

- I need to research how the quizzes in BrightSpace work
- Take the quiz named "sample-quiz-one". Get some questions right and some questions wrong.
- Email me if anything about the quiz seems impossible.