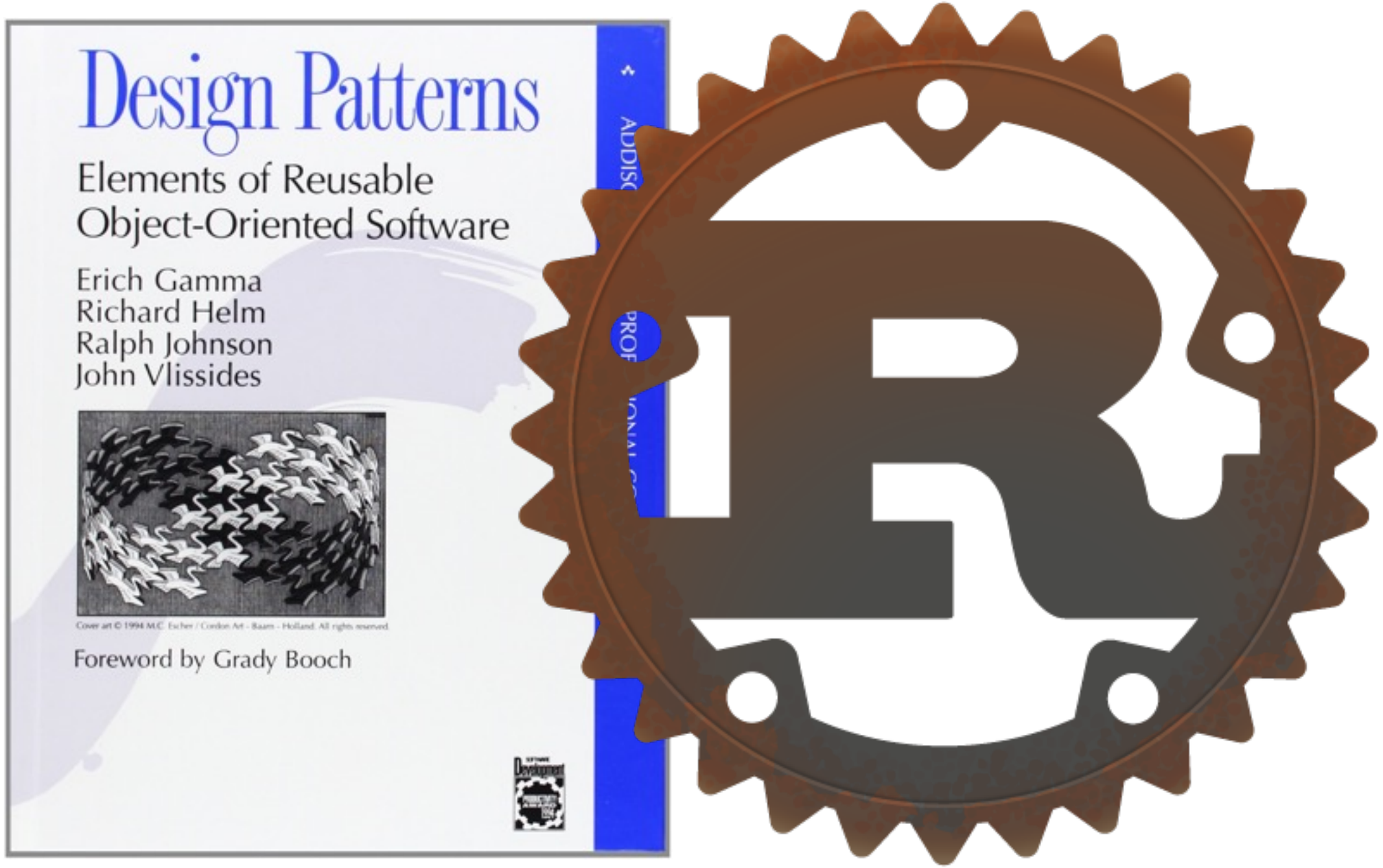


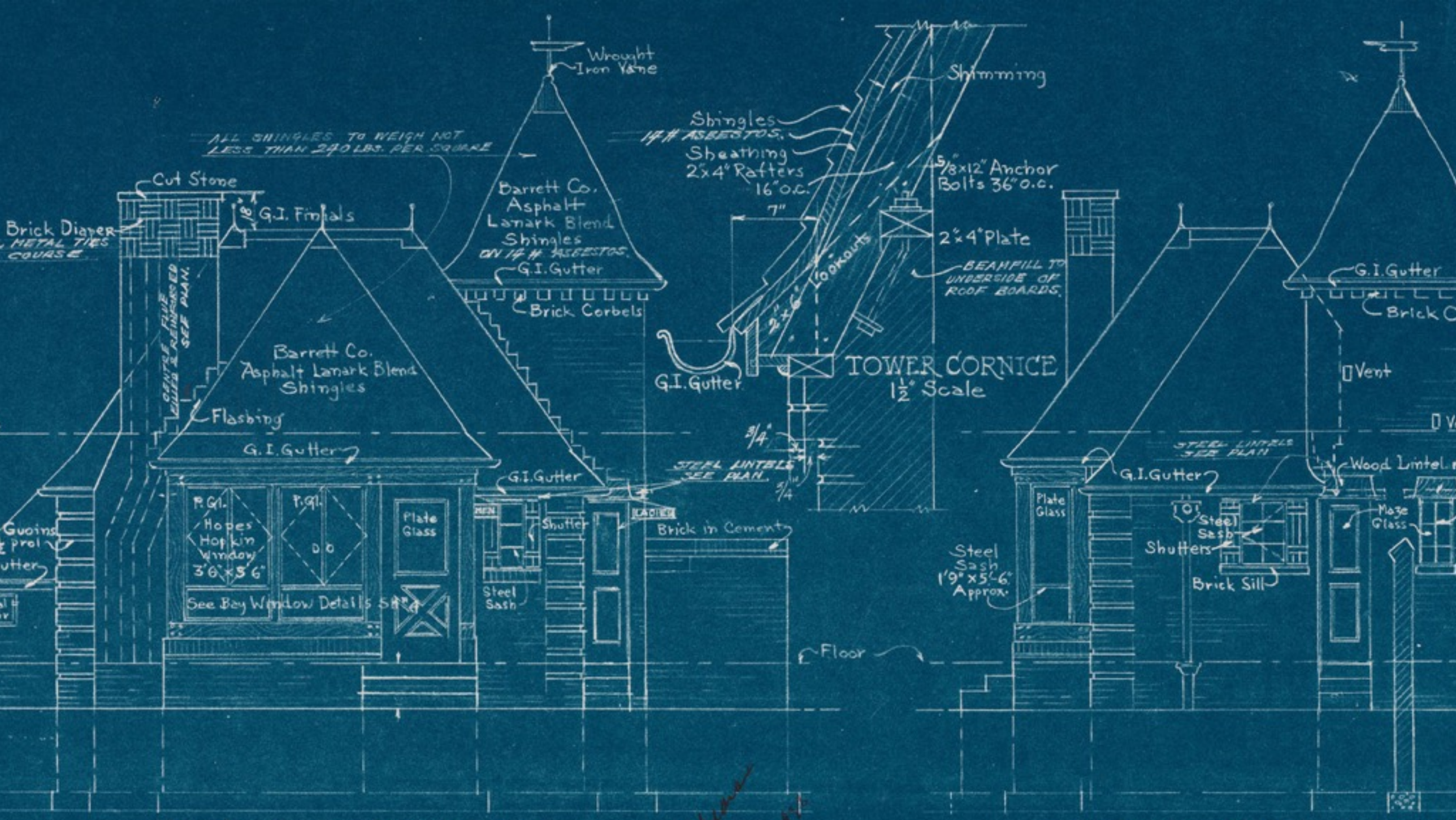
Design Patterns in Rust

Nicholas Cameron
Mozilla Research

Why talk about design patterns?

Beyond Rust 101

Tribal knowledge



The actual design patterns

Constructor

```
struct Foo {
  f: u32,
}

impl Foo {
  fn new(f: u32) -> Foo {
    Foo { f: f }
  }
}
```

Builder

```
struct Foo {
  f: u32,
  g: u32,
}

struct FooBuilder {
  f: Option<u32>,
  g: Option<u32>,
}

impl FooBuilder {
  fn new() -> FooBuilder { … }
  …
}
```

```
fn main() {
    let foo = FooBuilder::new().f(0).g(42).finalise();
}
```

Entry

```
struct Hashmap<K, V> { … }
```

```
if map.contains_key(&key) {
  *map.find_mut(&key) += 1;
} else {
  map.insert(key, 1);
}
```

```
impl<K, V> Hashmap<K, V> {
  fn find_with_or_insert_with<'a, A>
    (&'a mut self,
     k: K,
     a: A,
     found: |&K, &mut V, A|,
     not_found: |&K, A| -> V)
    -> &'a mut V
  { … }
}
```

```
impl<K, V> Hashmap<K, V> {
    fn entry(&mut self, k: K) -> Entry<K, V> {
        ...
    }
}
```

```
enum Entry<K, V> {
  Occupied(…),
  Vacant(…),
}
```

```
enum Entry<K, V> {
  Occupied(…),
  Vacant(…),
}
```

```
match map.entry(key) {
  Occupied(e) => e.get_mut() += 1,
  Vacant(e) => e.insert(1),
}
```

```
enum Entry<K, V> {
  Occupied(…),
  Vacant(…),
}
```

```
match map.entry(key) {
  Occupied(e) => e.get_mut() += 1,
  Vacant(e) => e.insert(1),
}
```

```
map.entry(key).or_insert(0) += 1;
```

```
enum Entry<'a, K, V> {
   Occupied(…),
   Vacant(…),
}
```

```
enum Entry<'a, K, V> {
  Occupied(…),
  Vacant(…),
}
```

```
fn entry(&mut self, k: K) -> Entry<K, V>
```

```
enum Entry<'a, K, V> {
   Occupied(…),
   Vacant(…),
}
```

```
fn entry(&mut self, k: K) -> Entry<K, V>
```

```
fn entry(&'a mut self, k: K) -> Entry<'a, K, V>
```

Guarded RAII

```
struct Mutex<T> { … }
struct MutexGuard<T> { … }

impl<T> Mutex<T> {
  fn lock(&self) -> MutexGuard<T> {
    …
  }
}

impl<T> Drop for MutexGuard {
  fn drop(&mut self) {
    // Unlock the mutex
  }
}
```

```
struct MutexGuard<T> { … }

impl<T> Deref for MutexGuard<T> {
  type Target = T;

  fn deref(&self) -> &T {
    &self.data
  }
}
```

```
fn print(s: &String) {
   …
}
```

```
fn foo(x: Mutex<String>) {
   let s = x.lock()
   print(&s);
}
```

```
fn print(s: &String) {
  SOME_GLOBAL.f = s;
}
```

```
fn foo(x: Mutex<String>) {
  let s = x.lock()
  print(&s);
}
```

KEEP

CALM

AND

CARRY

ON

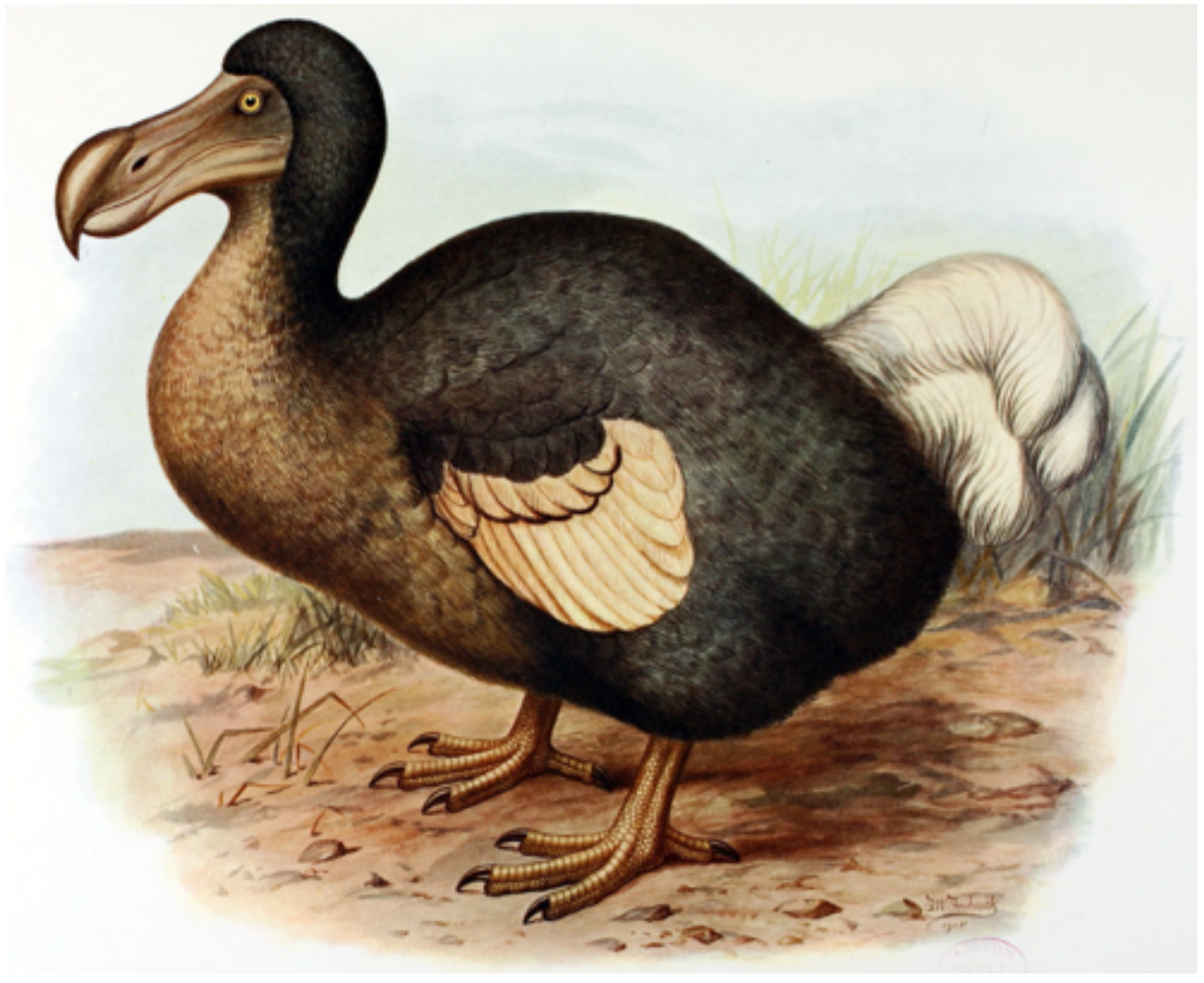
```
struct Mutex<T> { … }
struct MutexGuard<'a, T> {
  data: &'a T
}
```

```
struct Mutex<T> { … }
struct MutexGuard<'a, T> {
  data: &'a T
}

impl<T> Mutex<T> {
  fn lock(&'b self) -> MutexGuard<'b, T> {
    …
  }
}
```

```
struct Mutex<T> { … }
struct MutexGuard<'a, T> {
  data: &'a T
}

impl<T> Mutex<T> {
  fn lock(&'b self) -> MutexGuard<'b, T> {
    …
  }
}

impl<'a, T> Deref for MutexGuard<'a, T> {
  type Target = T;

  fn deref<'c>(&'c self) -> &'c T {
    &self.data
  }
}
```

```
fn foo(x: Mutex<String>) {
  let s: MutexGuard<'a, String> = x.lock()
  print(&s);
}
```

```
fn print<‘b>(s: &’b String) {
  SOME_GLOBAL.f = s;
}

fn foo(x: Mutex<String>) {
  let s: MutexGuard<‘a, String> = x.lock()
  print(&s);
}
```



An anti-pattern

`Deref` polymorphism

```
class Foo {
  void foo() { … }
}

class Bar extends Foo {}

void qux(Bar b) {
  b.foo();
}
```

```
struct Foo;

impl Foo {
  fn foo(&self) { … }
}

struct Bar {
  foo: Foo,
}
```

```
struct Foo;

impl Foo {
  fn foo(&self) { … }
}

struct Bar {
  foo: Foo,
}

impl Deref for Bar {
  type Target = Foo;
  fn deref(&self) -> &Foo {
    &self.foo
  }
}
```

```
struct Foo;

impl Foo {
  fn foo(&self) { … }
}

struct Bar {
  foo: Foo,
}
```

```
impl Deref for Bar {
  type Target = Foo;
  fn deref(&self) -> &Foo {
    &self.foo
  }
}

fn qux(b: &Bar) {
  b.foo();
}
```

Some kind of conclusion

`https://github.com/nrc/patterns`

@nick_r_cameron
nrc@mozilla.com