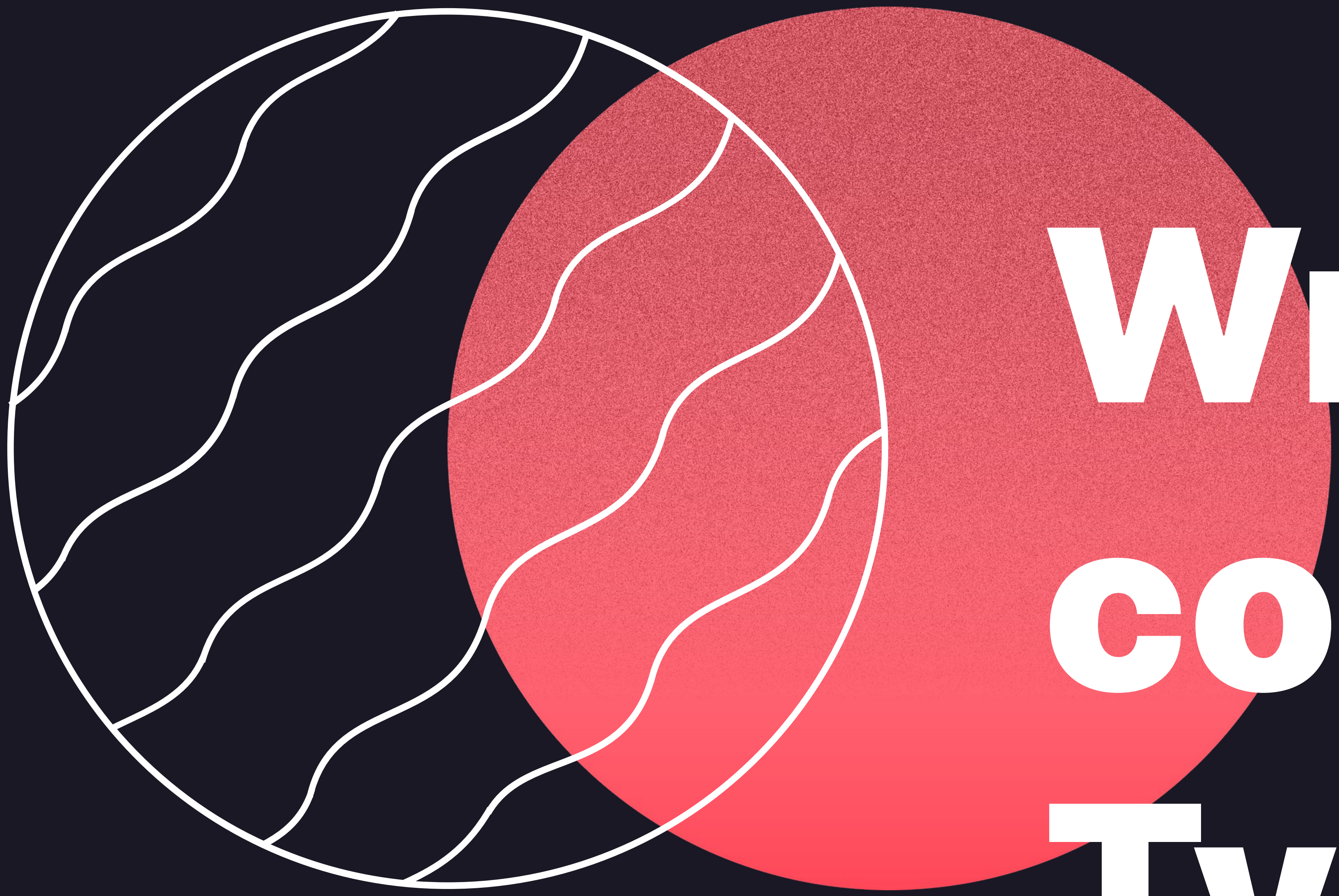
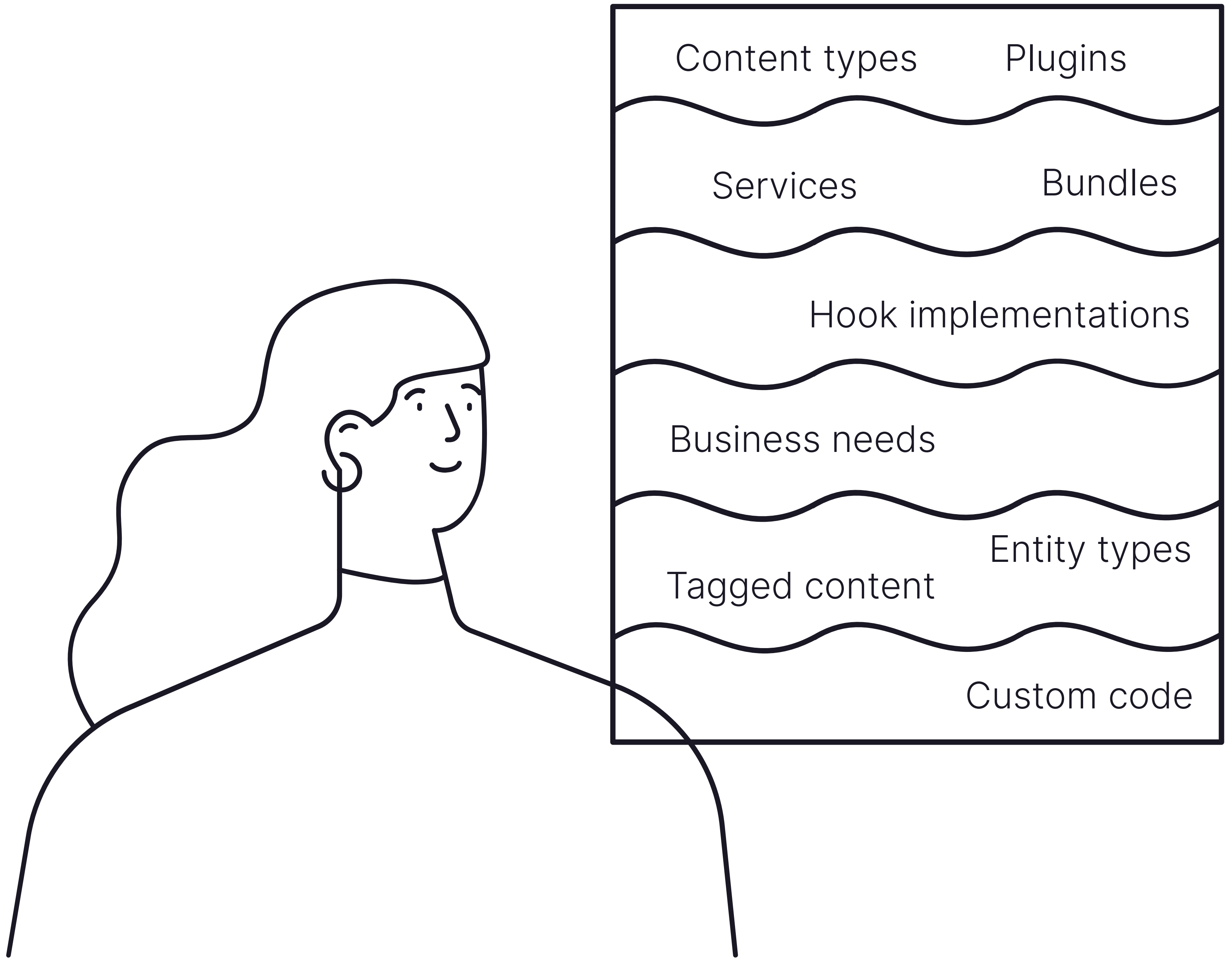
Write better code with Typed Entity

The path to maintainable custom code in Drupal



Mateu - e0ipso

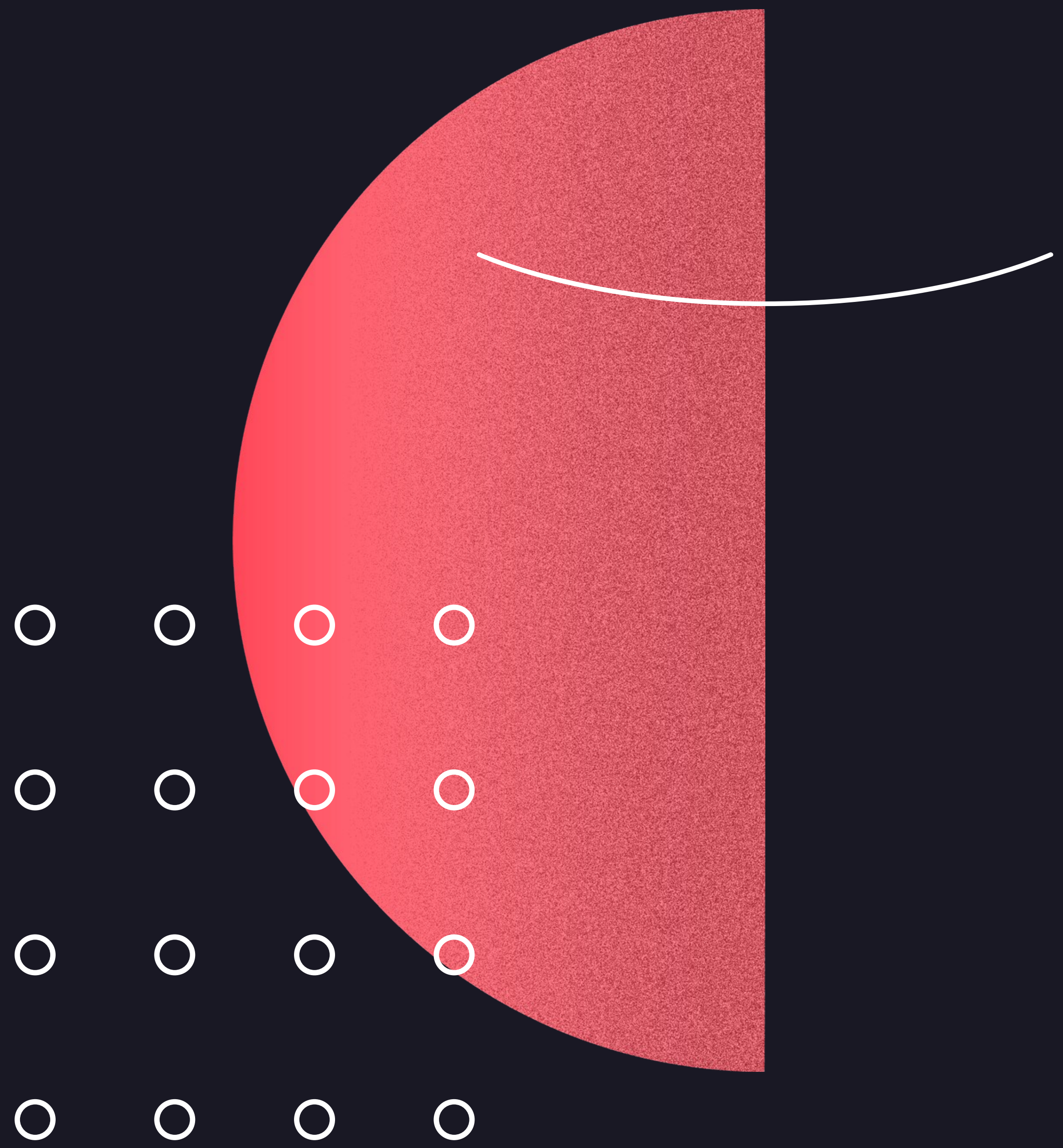
Coding Drupal projects can be challenging.



~~FRAMEWORK LOGIC~~

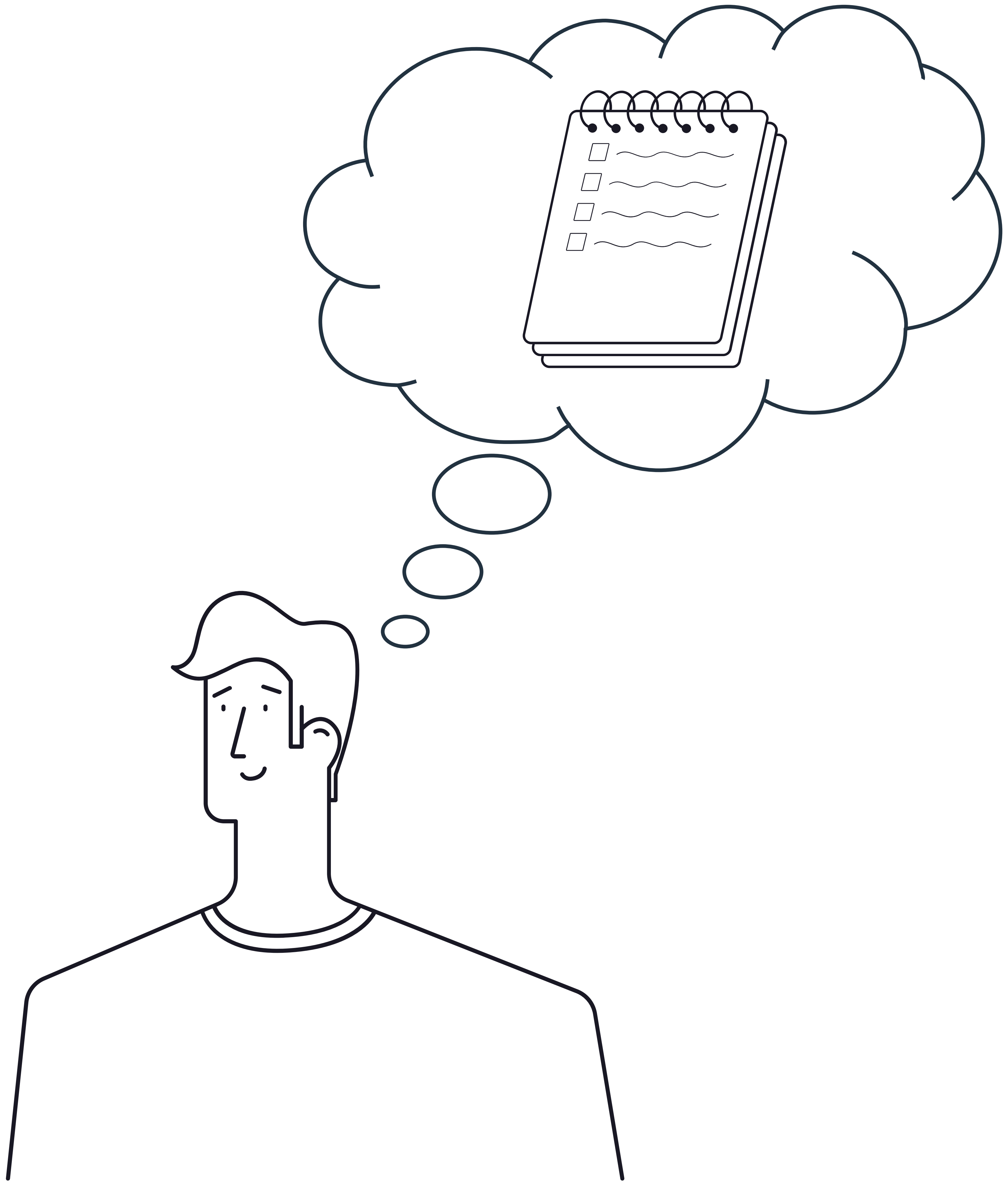
ACME
CORPORATION
"Quality First!"

<u>BUSINESS LOGIC</u>

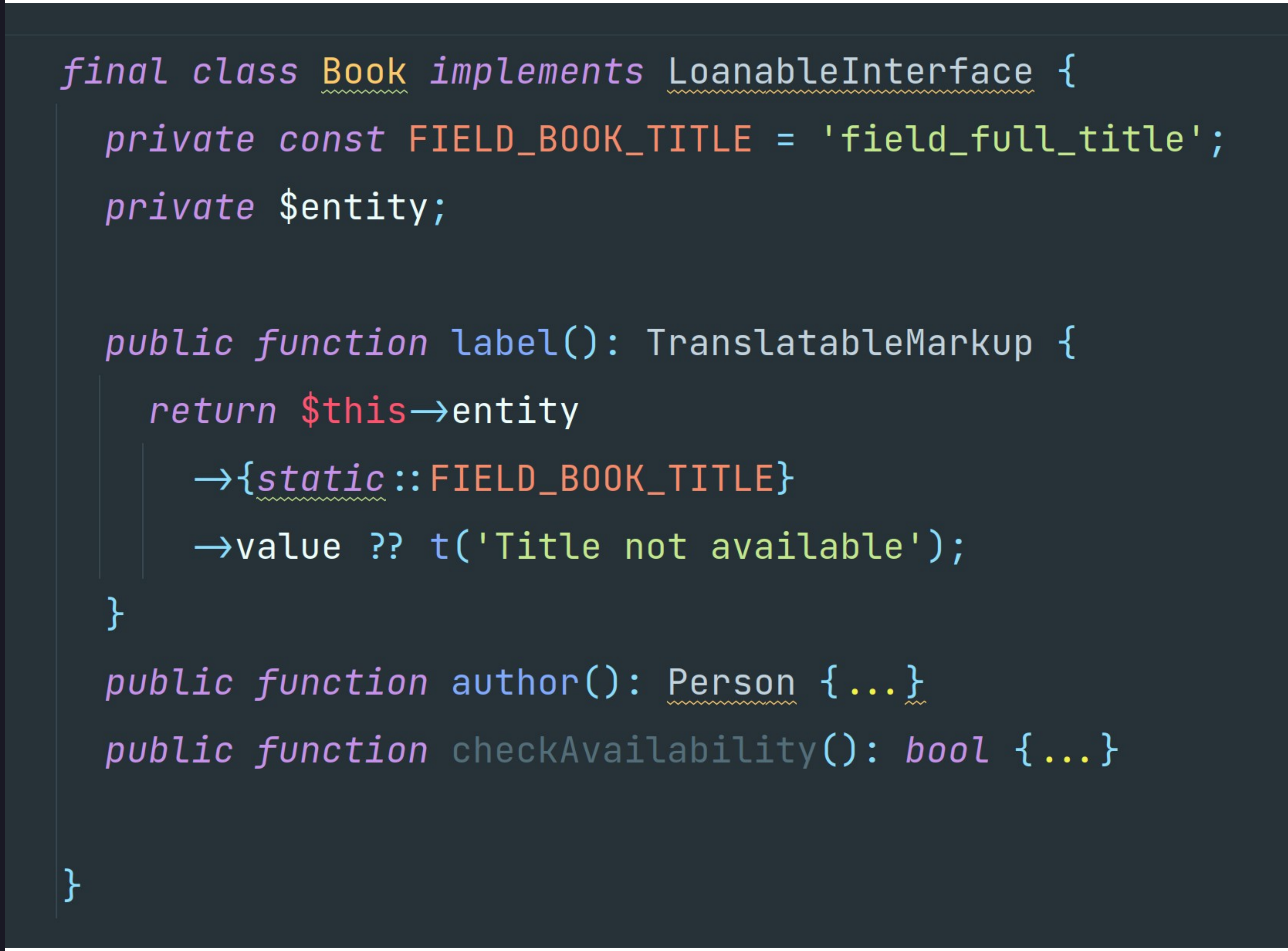


Complexity is a feature, and it needs to be contained.

THERE’S A BETTER WAY TO ORGANIZE YOUR CODE, WITH **TYPED ENTITY**



PUT LOGIC CLOSE TO THE ENTITY, NOT SCATTERED IN HOOKS

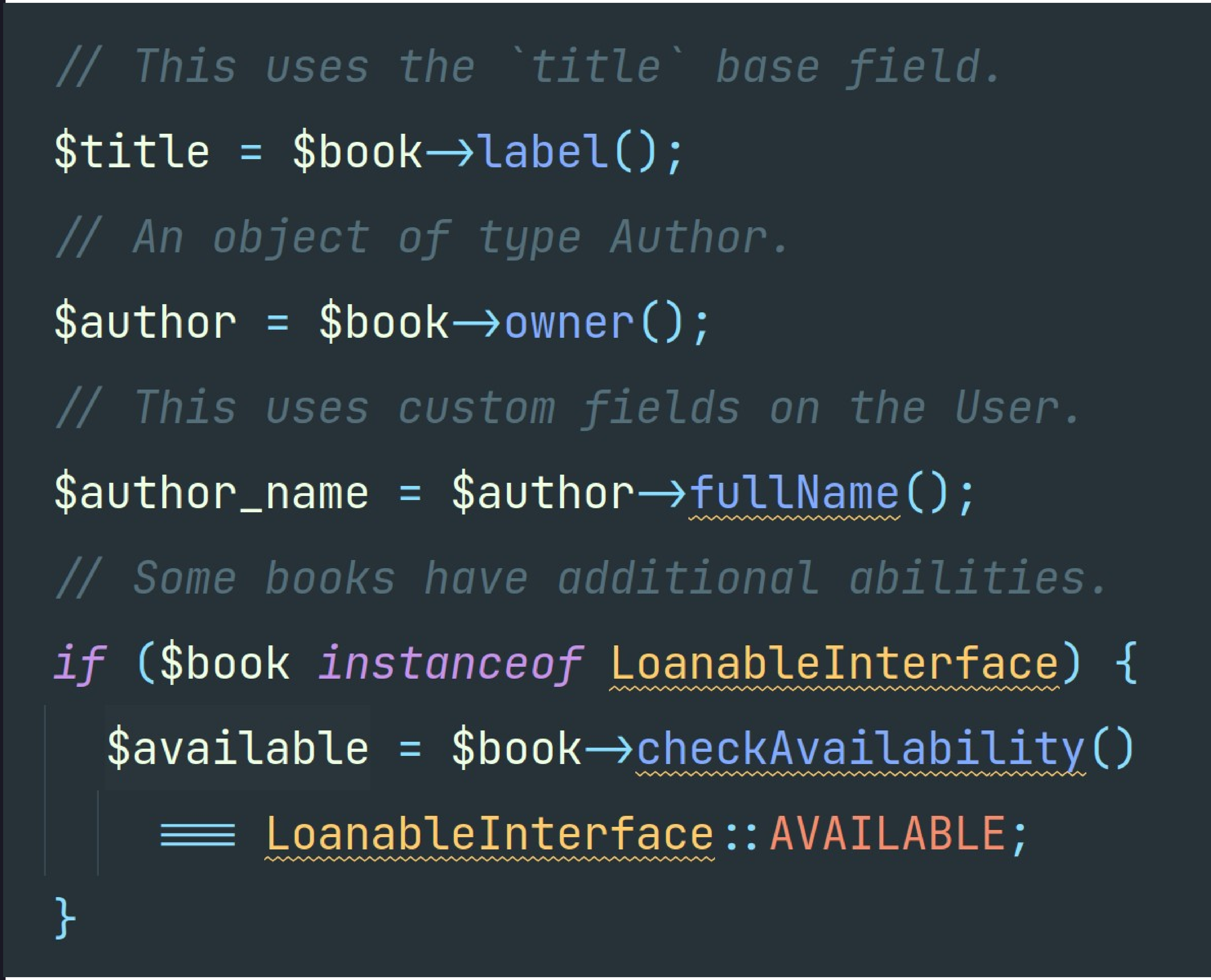


```php
final class Book implements LoanableInterface {
  private const FIELD_BOOK_TITLE = 'field_full_title';
  private $entity;

  public function label(): TranslatableMarkup {
    return $this→entity
      →{static::FIELD_BOOK_TITLE}
      →value ?? t('Title not available');
  }
  public function author(): Person {...}
  public function checkAvailability(): bool {...}

}
```

PUT LOGIC CLOSE TO THE ENTITY, NOT SCATTERED IN HOOKS

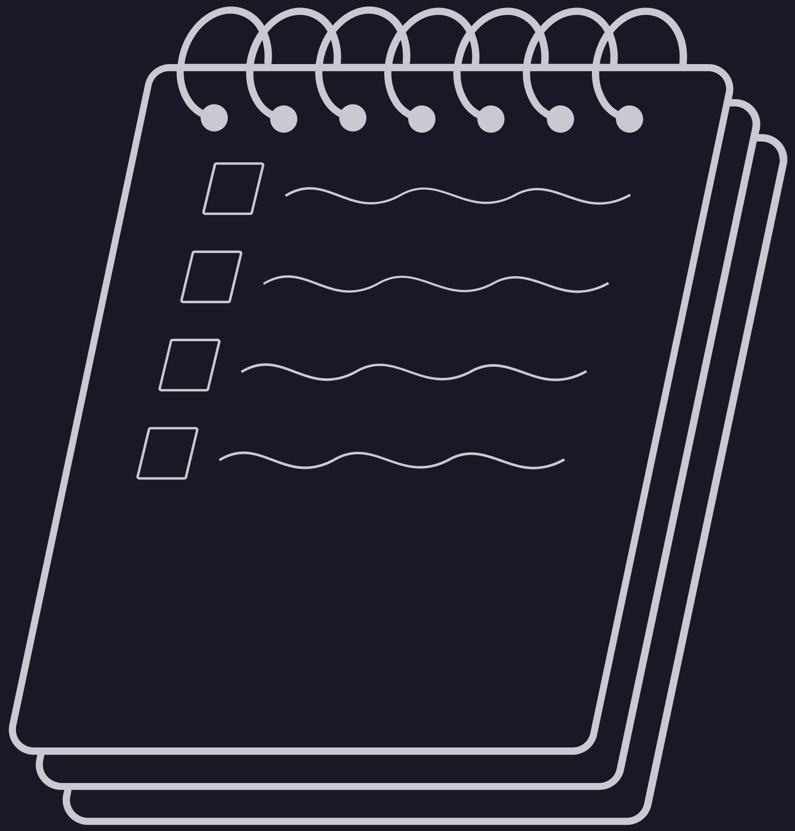


```php
// This uses the `title` base field.
$title = $book->label();
// An object of type Author.
$author = $book->owner();
// This uses custom fields on the User.
$author_name = $author->fullName();
// Some books have additional abilities.
if ($book instanceof LoanableInterface) {
  $available = $book->checkAvailability()
    === LoanableInterface::AVAILABLE;
}
```

ARE YOU ACCESSING FIELD DATA ANYWHERE?

```
$entity->field_foo->value
```

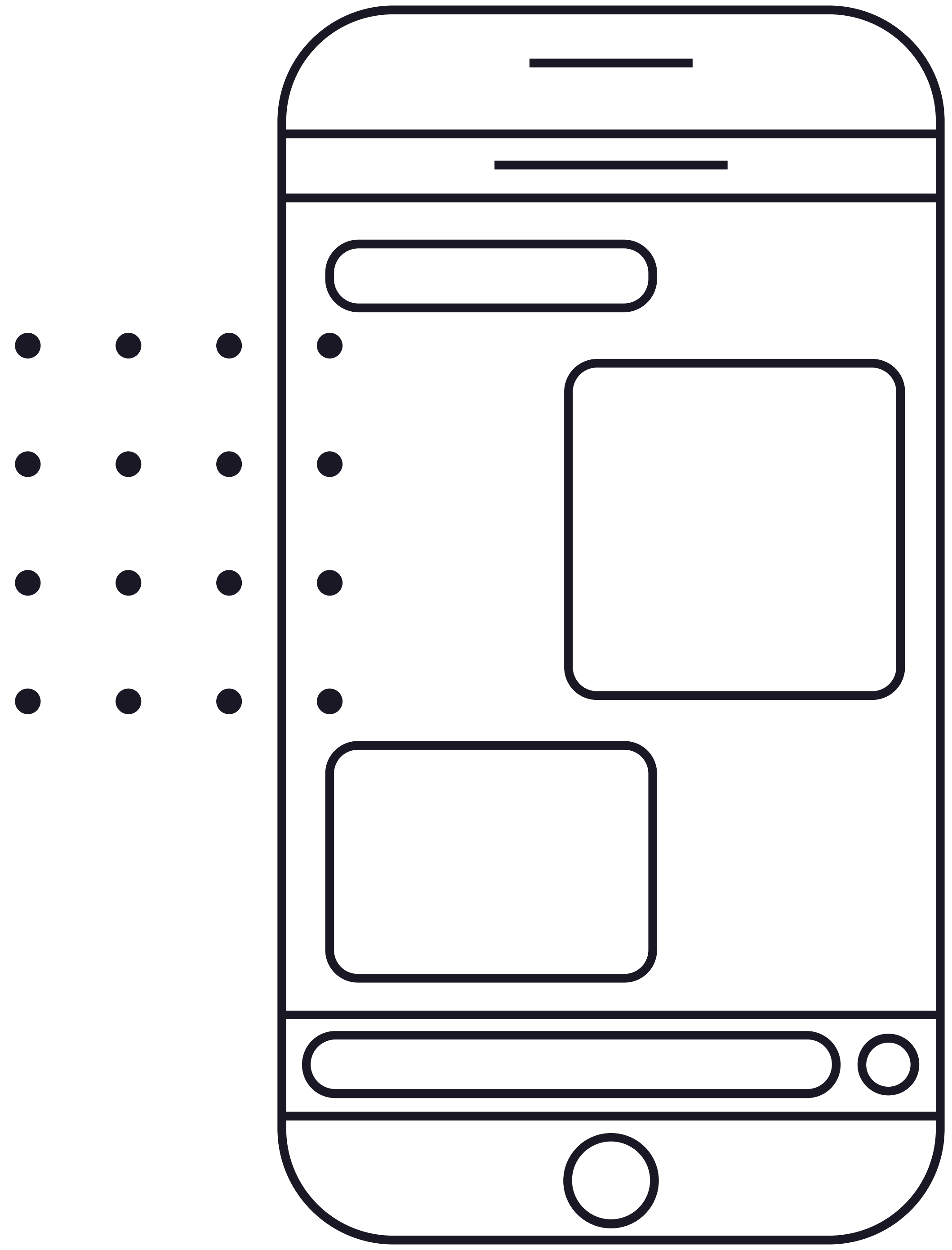
This is a red flag that indicates you need an entity wrapper.



Entity Types are the main integration point for custom business logic.

ENTITIES HAVE MANY RESPONSIBILITIES

- **We render them as content in the screen**
- They are used for navigation purposes
- They hold SEO metadata
- We add decorative hints to them
- We use their fields to group content
- They can be embedded
- ...



SIMILAR SOLUTIONS?

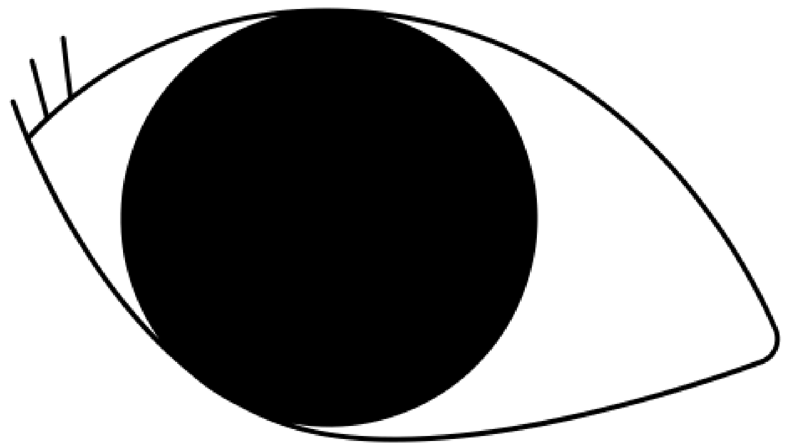
There is a core patch to allow having custom classes for entity bundles.

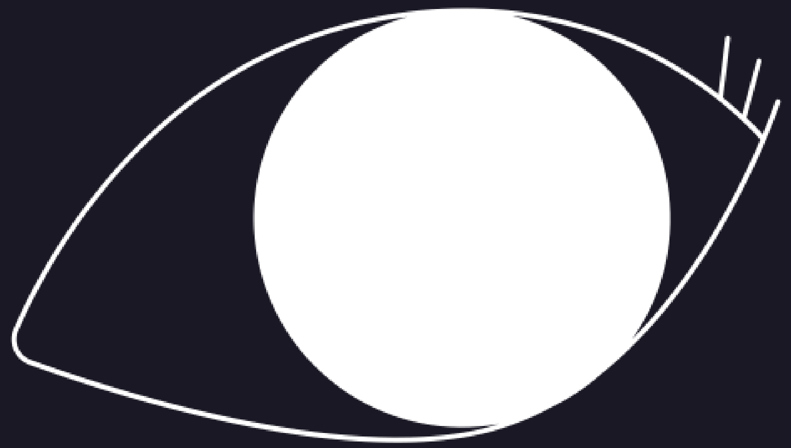
[#2570593]

```
Node::load(12) → Book
```

The Bundle Override module does the same as the core patch.

(seeking co-maintainer)





DRAWBACKS WITH THAT APPROACH

Increments API surface of entity objects.

A method added to Node can collide with your Book class.

Unit testing carries over all the storage complexity.

Solves the solution only partially.

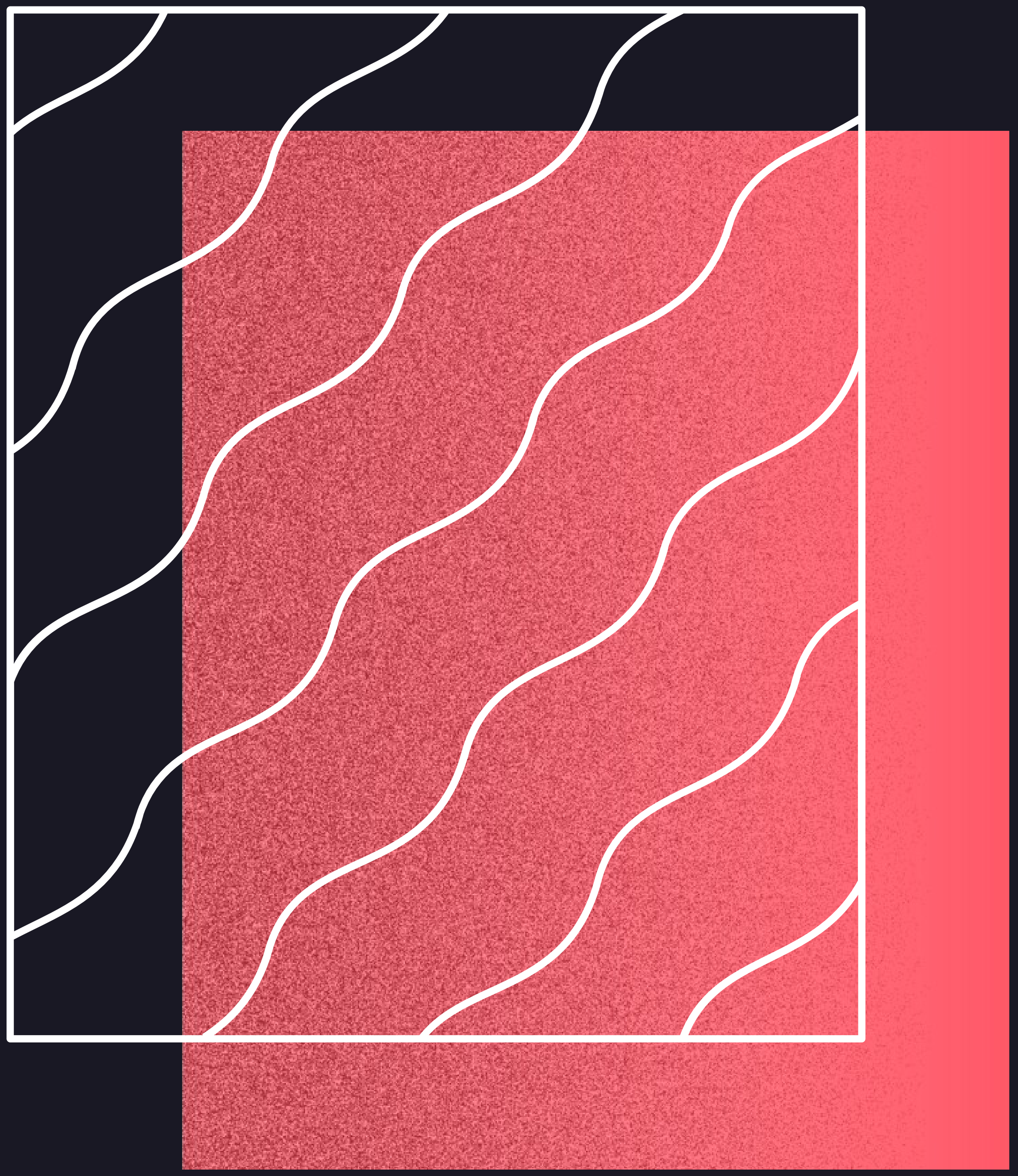
How about methods that apply to many books?

How can `SciFiBook`, `HistoryBook`, and `Book`, coexist?

Perpetuates inheritance, even into application space.

We should favor composition over inheritance.

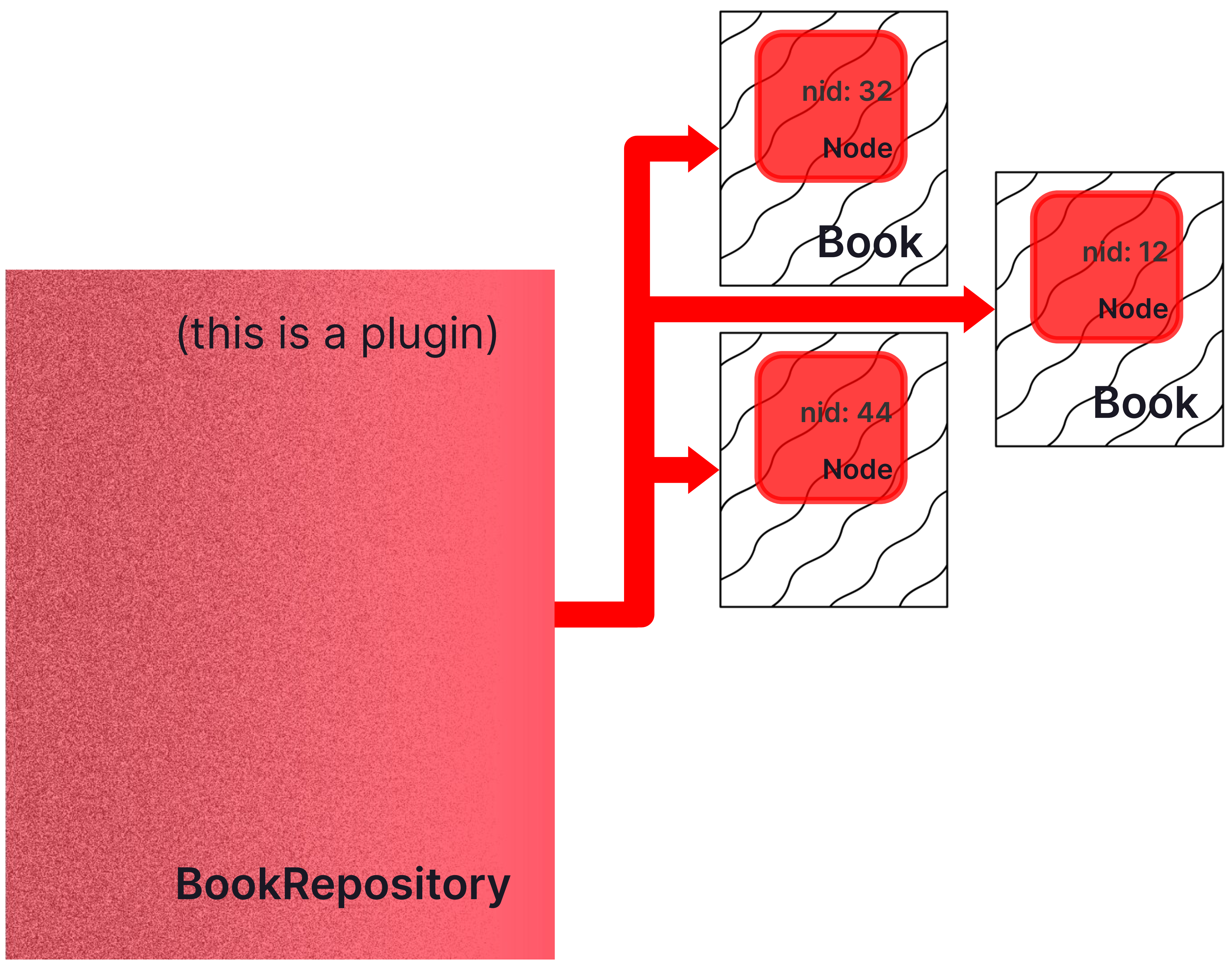
Can we separate framework logic from application logic?

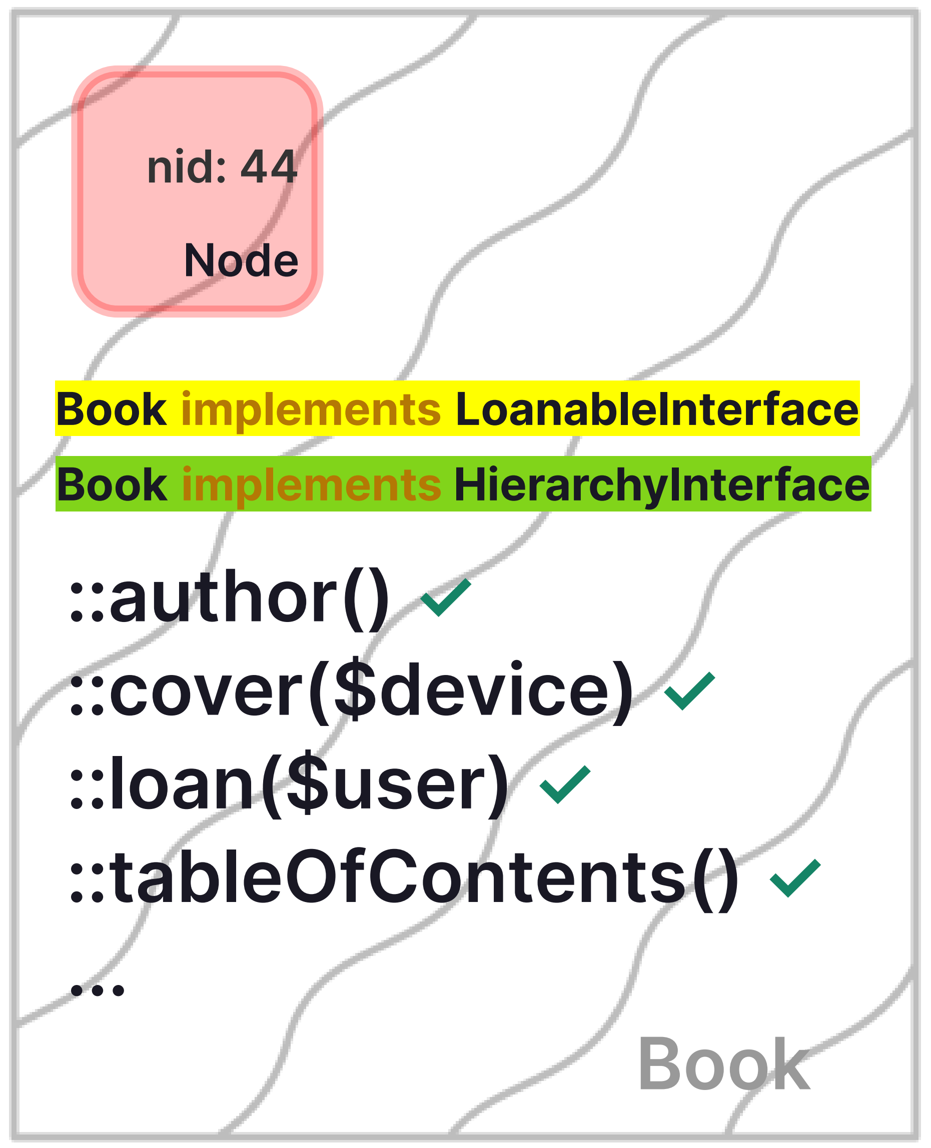


TYPED ENTITY'S APPROACH

Create a **plugin** and associate it to Entity Type [and Bundle]. This operates at the entity type level, great for things like `findTaggedWith()`. We call these **TYPED REPOSITORIES**.

Typed Repositories know what object to create, given an entity. These are objects that contain the entity, instead of replacing `Node`. We call these **WRAPPED ENTITIES**.

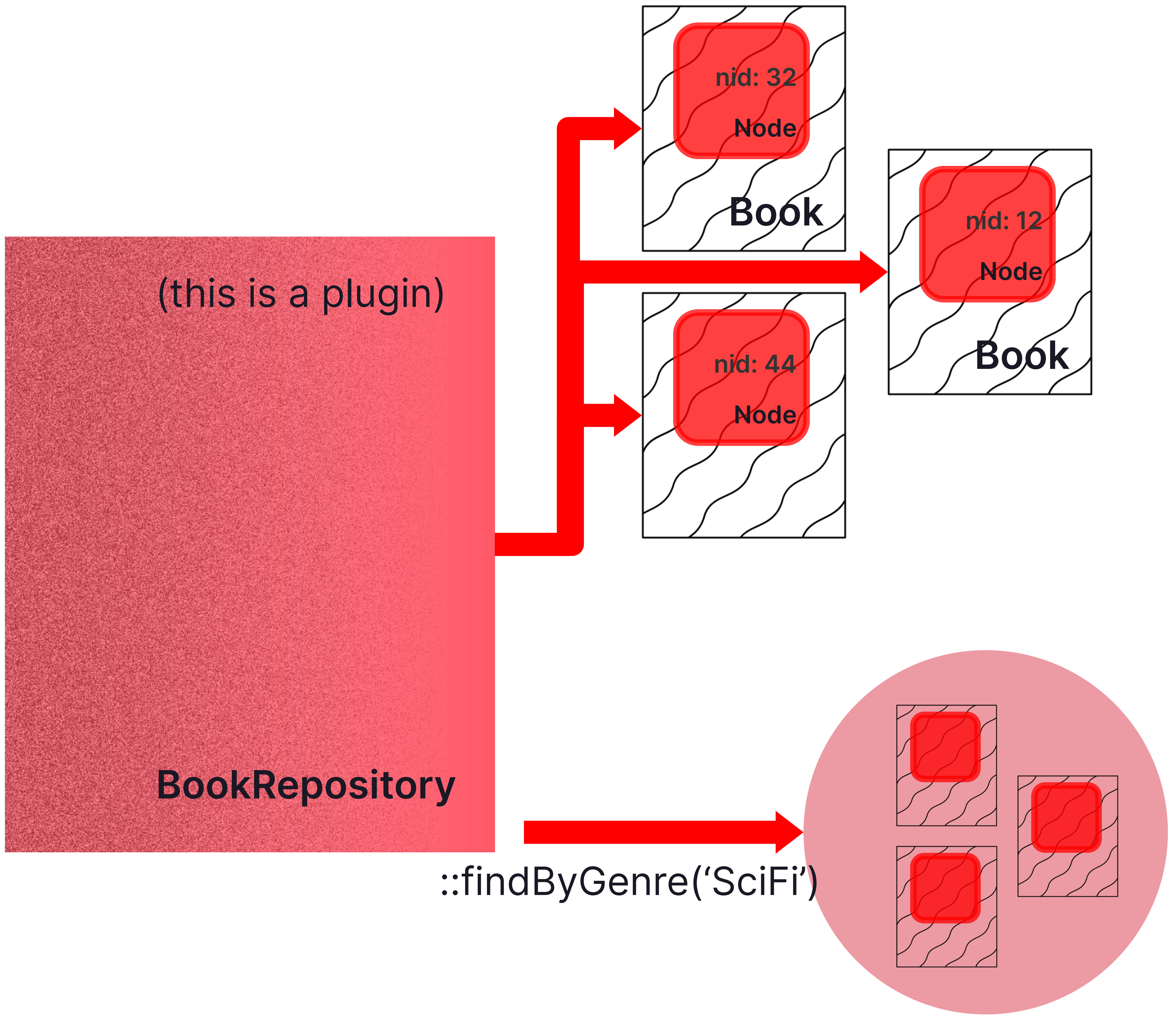


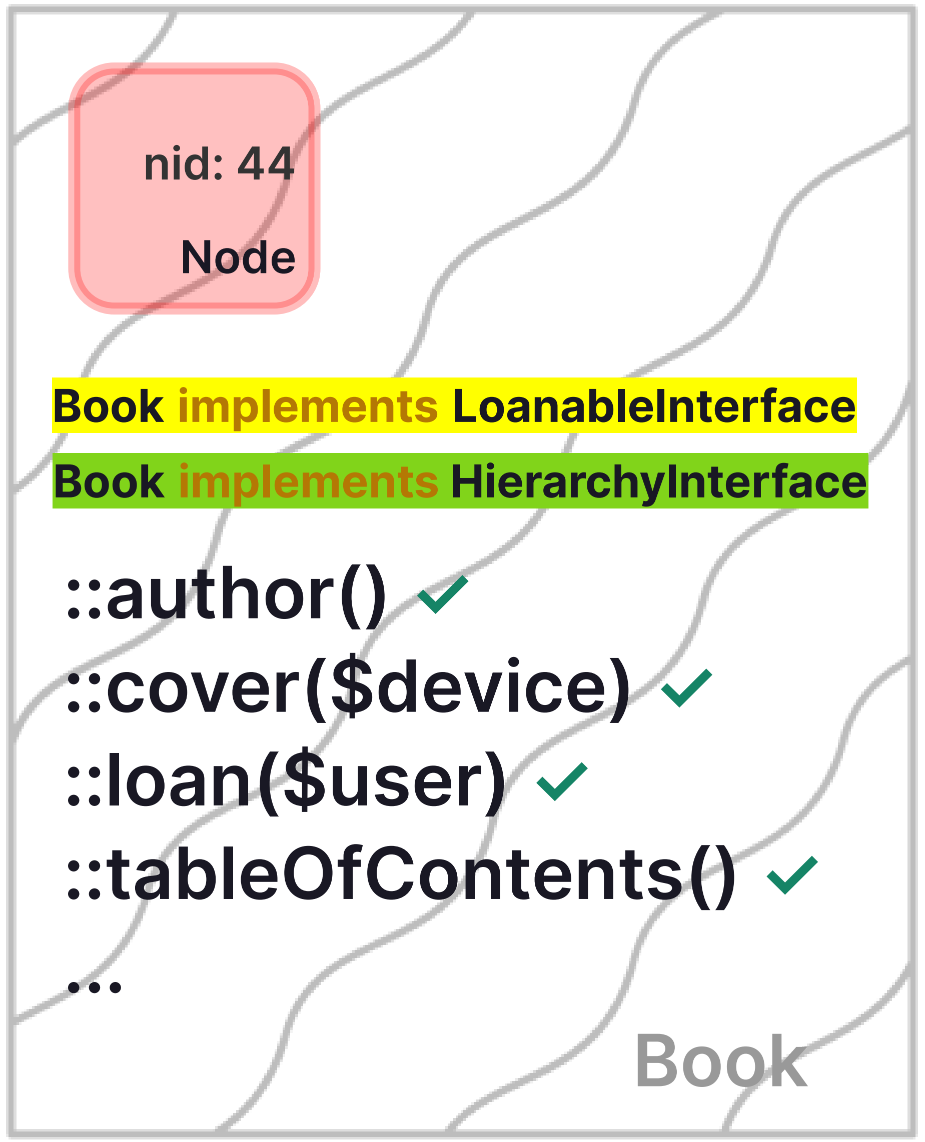


Book implements LoanableInterface

Book implements HierarchyInterface

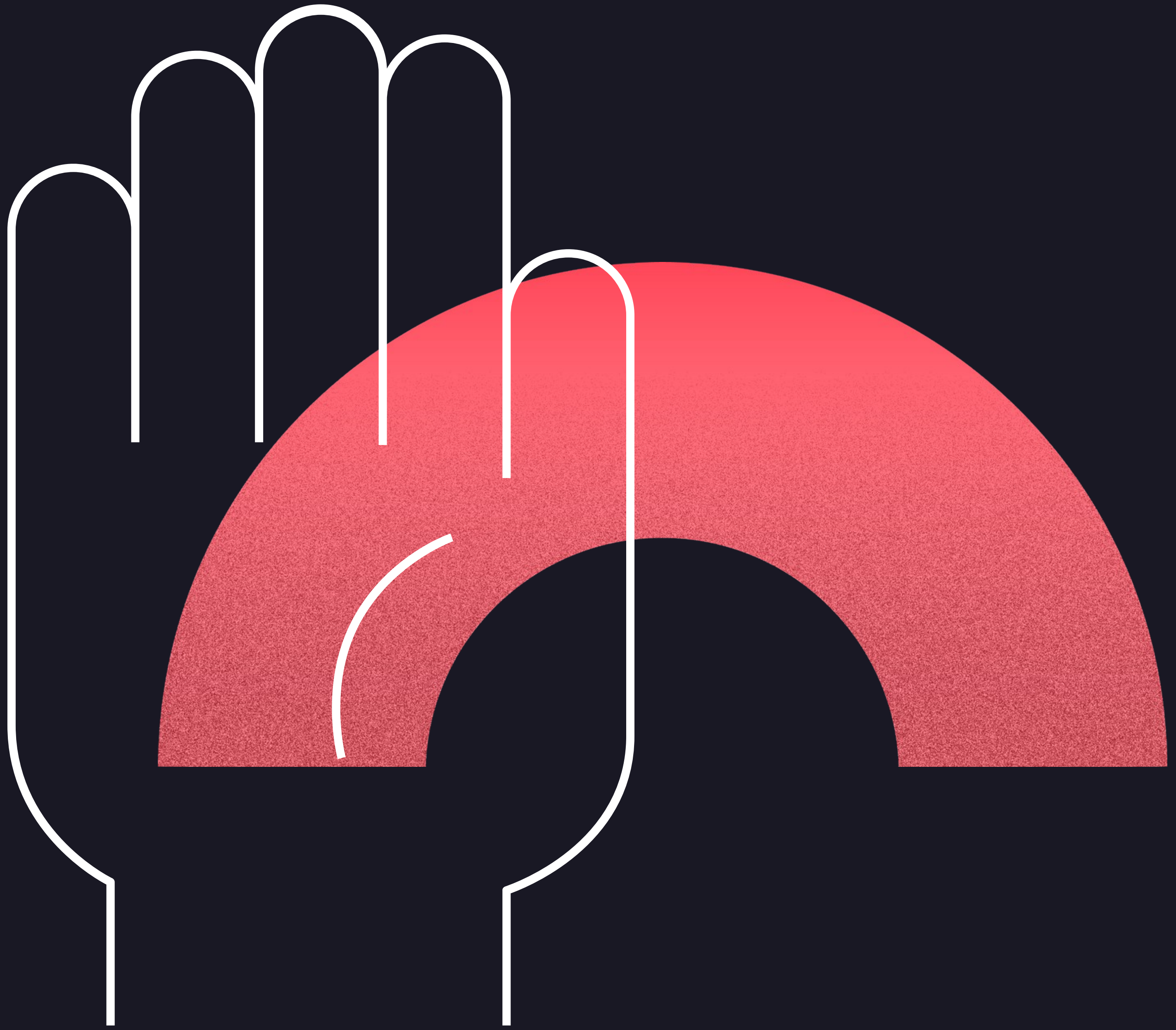
```
::author() ✓
::cover($device) ✓
::loan($user) ✓
::tableOfContents() ✓
...
```





Book implements LoanableInterface
Book implements HierarchyInterface

::author() ✓
::cover($device) ✓
::loan($user) ✓
::tableOfContents() ✓
...



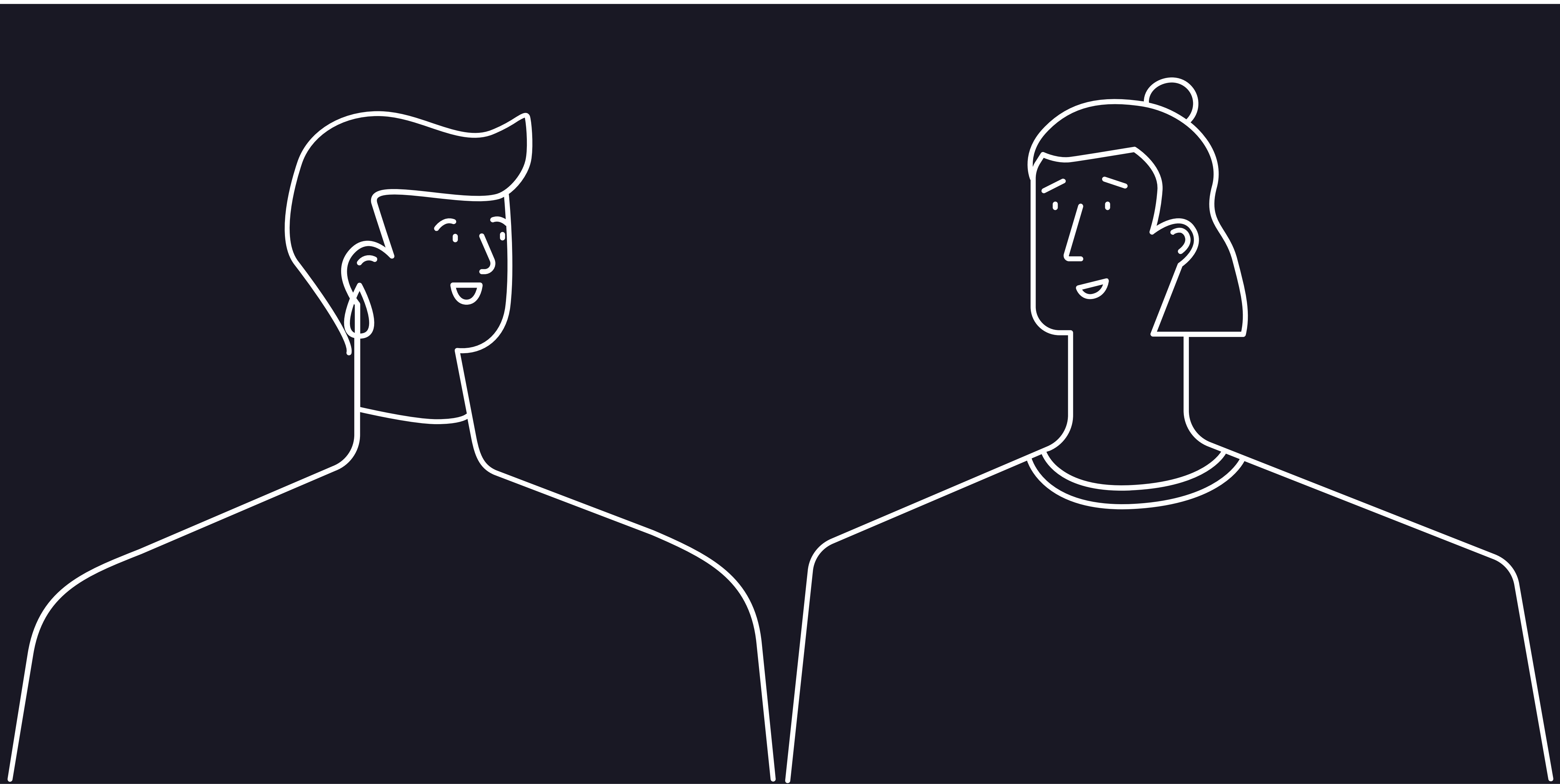
LET'S SEE SOME CODE

REMEMBER: Typed Entity is for your project's **custom** code. It is optimized to improve DX while working on business logic.

NEW REQUIREMENT

“One important detail is that books located in Area 51 are considered off limits.”

- Your stakeholder



FIRST APPROACH

```php
/**
 * Implements hook_node_access().
 */
function physical_media_node_access(NodeInterface $node, $op, AccountInterface $account) {
  if ($node->getType() !== 'book') {
    return;
  }
  $book = \Drupal::service(RepositoryManager::class)->wrap($node);
  assert($book instanceof FindableInterface);
  $location = $book->getLocation();
  if ($location->getBuilding() === 'area51') {
    return AccessResult::forbidden('Nothing to see.');
  }
  return AccessResult::neutral();
}
```

MOVE ENTITY LOGIC CLOSER TO THE ENTITY

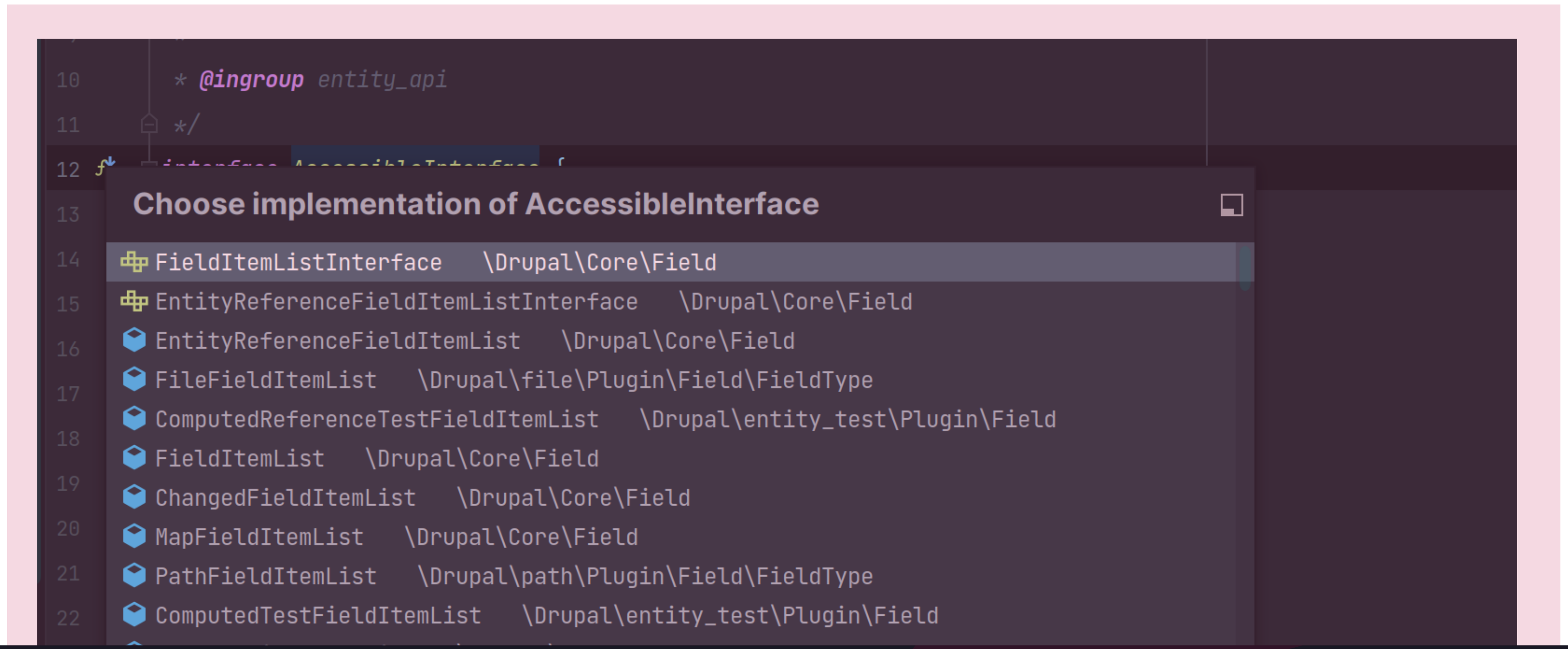
We have logic about "book" in a hook inside of `physical_media.module`. We should bring it into the `Book` class.

That should leave our *access hook* to check on any wrapped entity: "*does this entity support access checks? If so, check it. If not, carry on*"

MORE REFINED APPROACH

```
function physical_media_node_access($node, $op, $account) {
  try {
    $wrapped_node = typed_entity_repository_manager()->wrap($node);
  }
  catch (RepositoryNotFoundException $exception) {
    return AccessResult::neutral();
  }
  return $wrapped_node instanceof AccessibleInterface
    ? $wrapped_node->access($op, $account, TRUE)
    : AccessResult::neutral();
}
```

“WHAT TYPES SUPPORT ACCESS?”



```
 * @ingroup entity_api
 */
```

Choose implementation of AccessibleInterface

```
FieldItemListInterface   \Drupal\Core\Field
EntityReferenceFieldItemListInterface   \Drupal\Core\Field
EntityReferenceFieldItemList   \Drupal\Core\Field
FileFieldItemList   \Drupal\file\Plugin\Field\FieldType
ComputedReferenceTestFieldItemList   \Drupal\entity_test\Plugin\Field
FieldItemList   \Drupal\Core\Field
ChangedFieldItemList   \Drupal\Core\Field
MapFieldItemList   \Drupal\Core\Field
PathFieldItemList   \Drupal\path\Plugin\Field\FieldType
ComputedTestFieldItemList   \Drupal\entity_test\Plugin\Field
```

This leads to better:

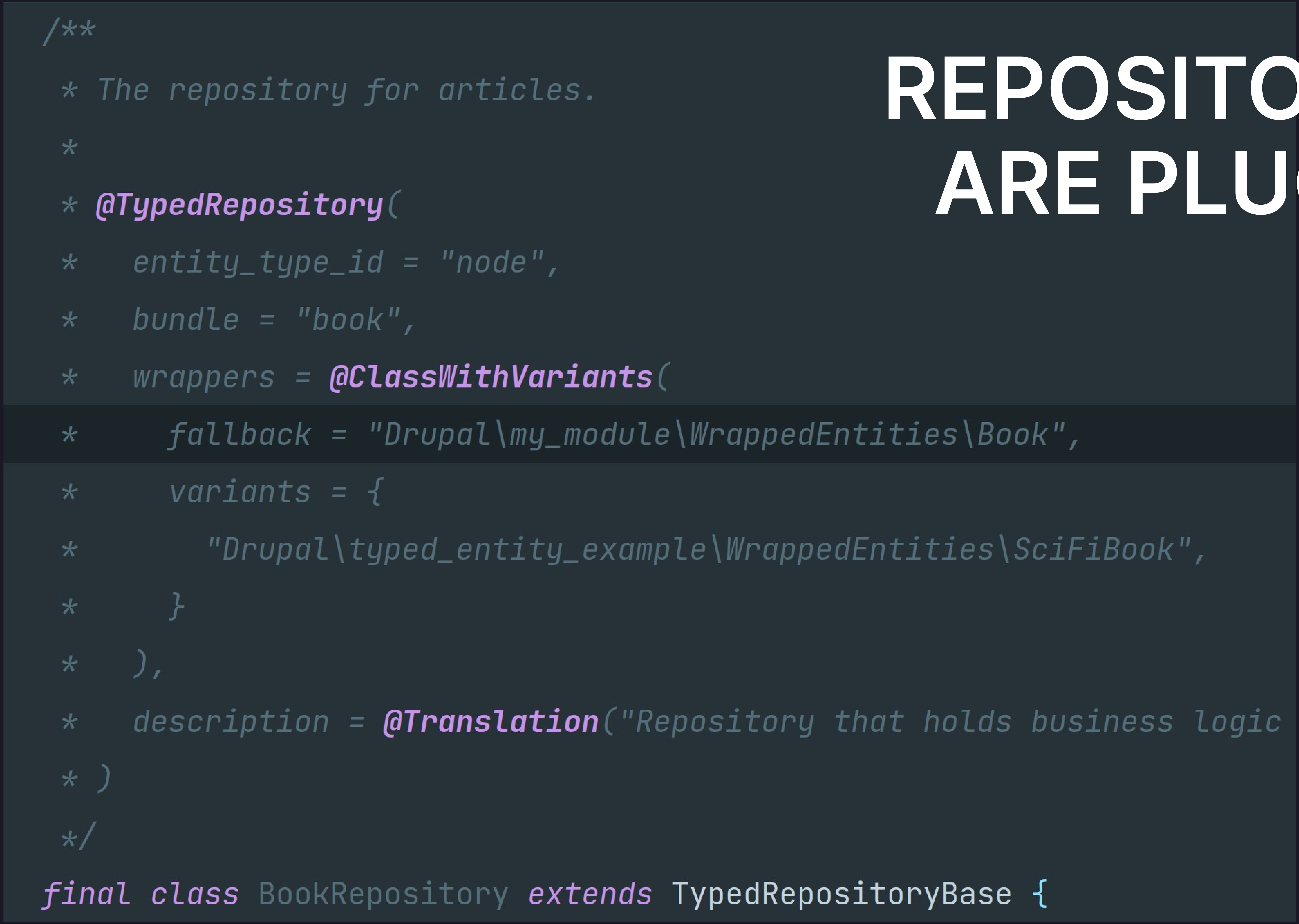
- Code organization
- Readability
- Code authoring/discovery
- Class testability
- Static analysis
- Code reuse

WAIT! HOW DOES IT WORK?

```
typed_entity_repository_manager()->wrap($entity);
```

Returns an object of type Book… but how?

REPOSITORIES ARE PLUGINS



```
/**
 * The repository for articles.
 *
 * @TypedRepository(
 *   entity_type_id = "node",
 *   bundle = "book",
 *   wrappers = @ClassWithVariants(
 *     fallback = "Drupal\my_module\WrappedEntities\Book",
 *     variants = {
 *       "Drupal\typed_entity_example\WrappedEntities\SciFiBook",
 *     }
 *   ),
 *   description = @Translation("Repository that holds business logic
 * )
 */
final class BookRepository extends TypedRepositoryBase {
```

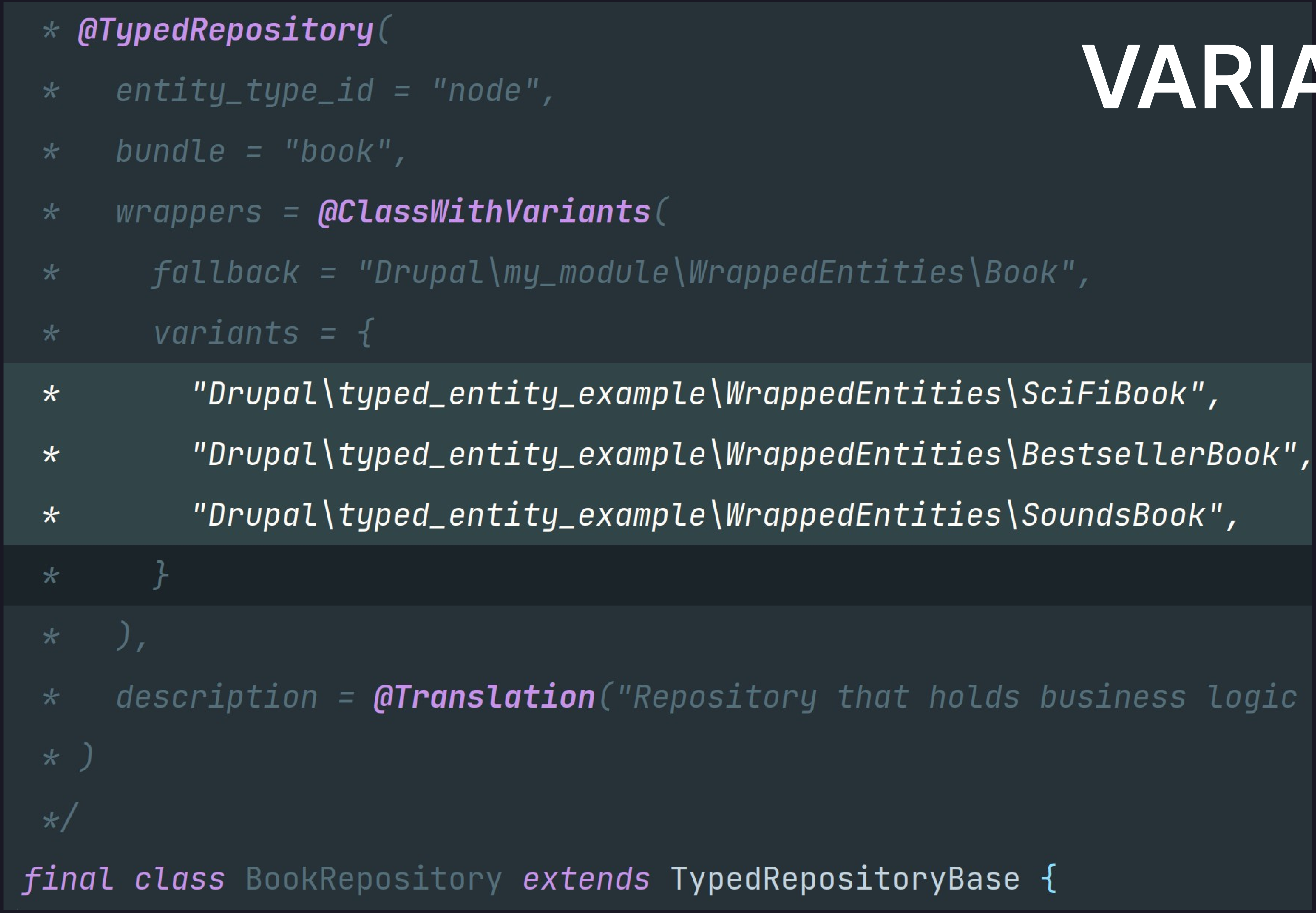
We often attach special behavior to entities with certain data

“books w/ sound” “bestsellers”

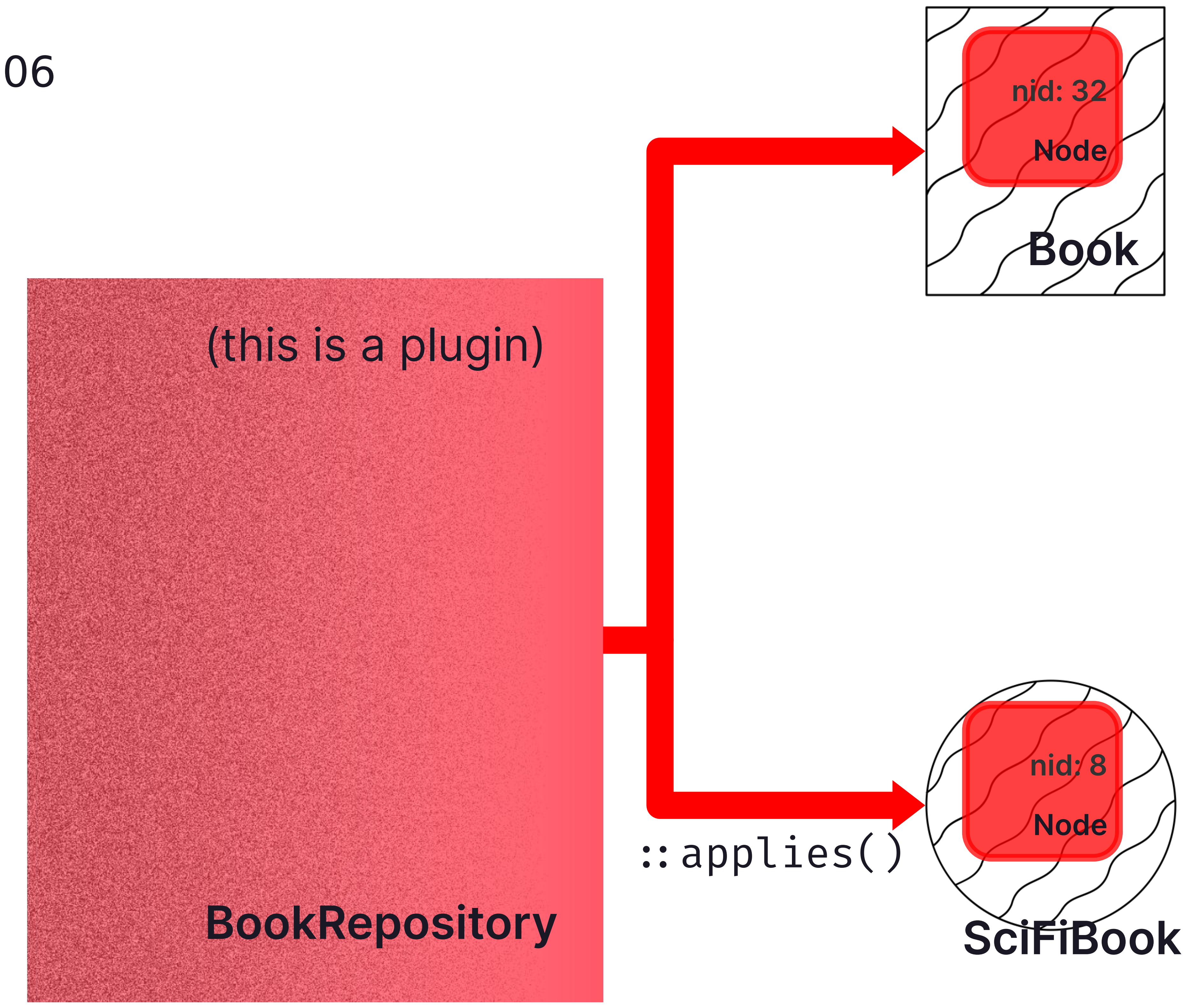
“books in a collection” “sci-fi books”

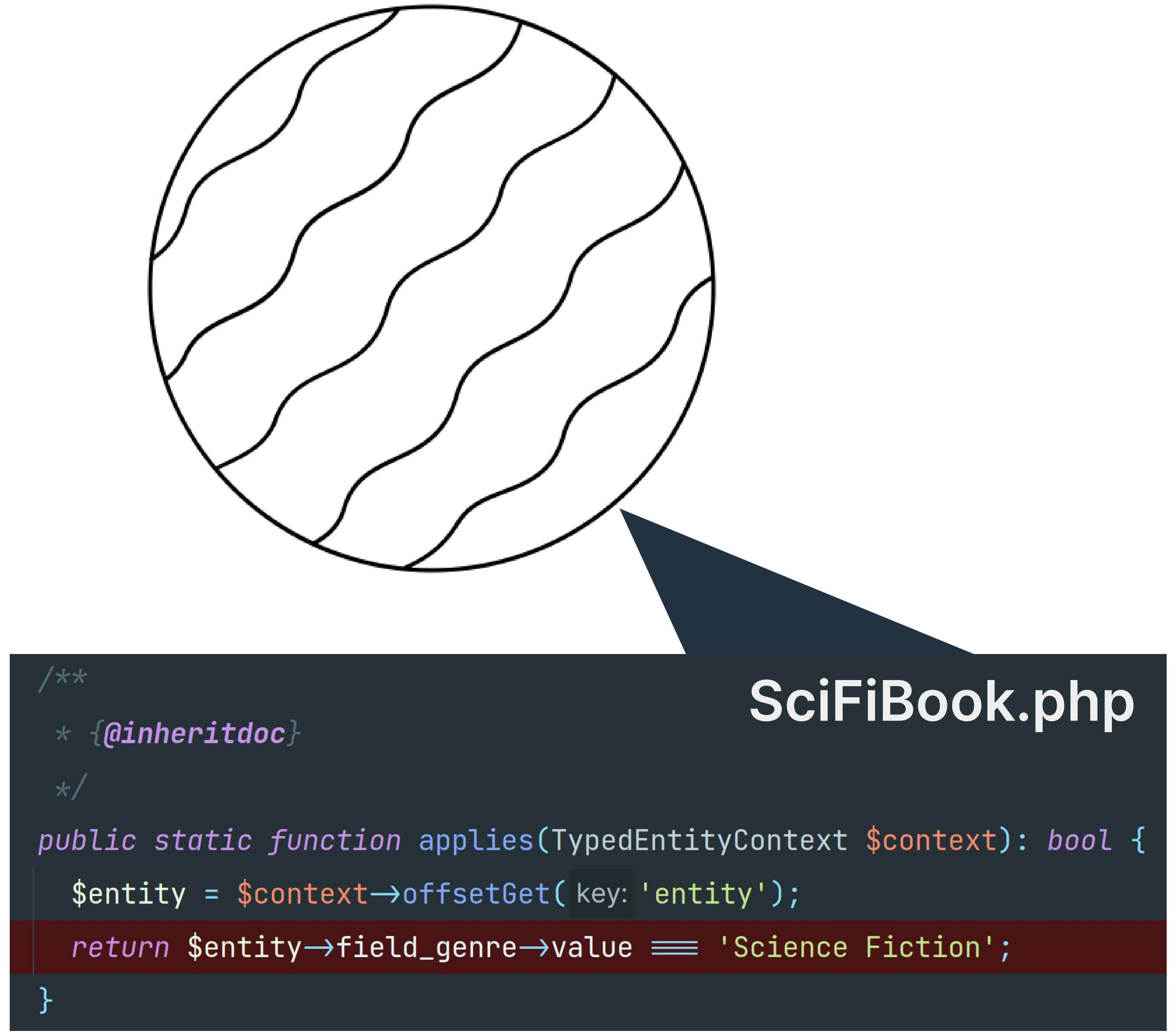
“audiobooks”

VARIANTS



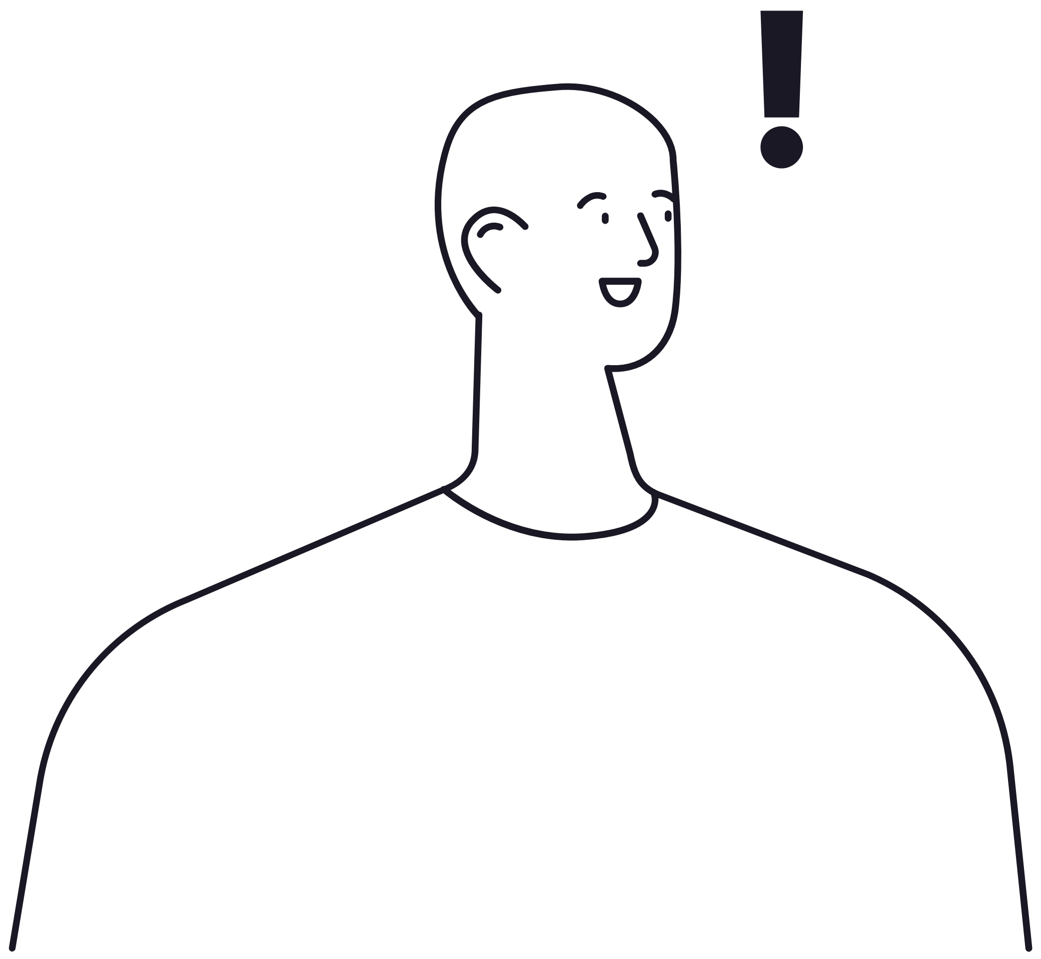
```
 * @TypedRepository(
 *   entity_type_id = "node",
 *   bundle = "book",
 *   wrappers = @ClassWithVariants(
 *     fallback = "Drupal\my_module\WrappedEntities\Book",
 *     variants = {
 *       "Drupal\typed_entity_example\WrappedEntities\SciFiBook",
 *       "Drupal\typed_entity_example\WrappedEntities\BestsellerBook",
 *       "Drupal\typed_entity_example\WrappedEntities\SoundsBook",
 *     }
 *   ),
 *   description = @Translation("Repository that holds business logic
 * )
 */
final class BookRepository extends TypedRepositoryBase {
```





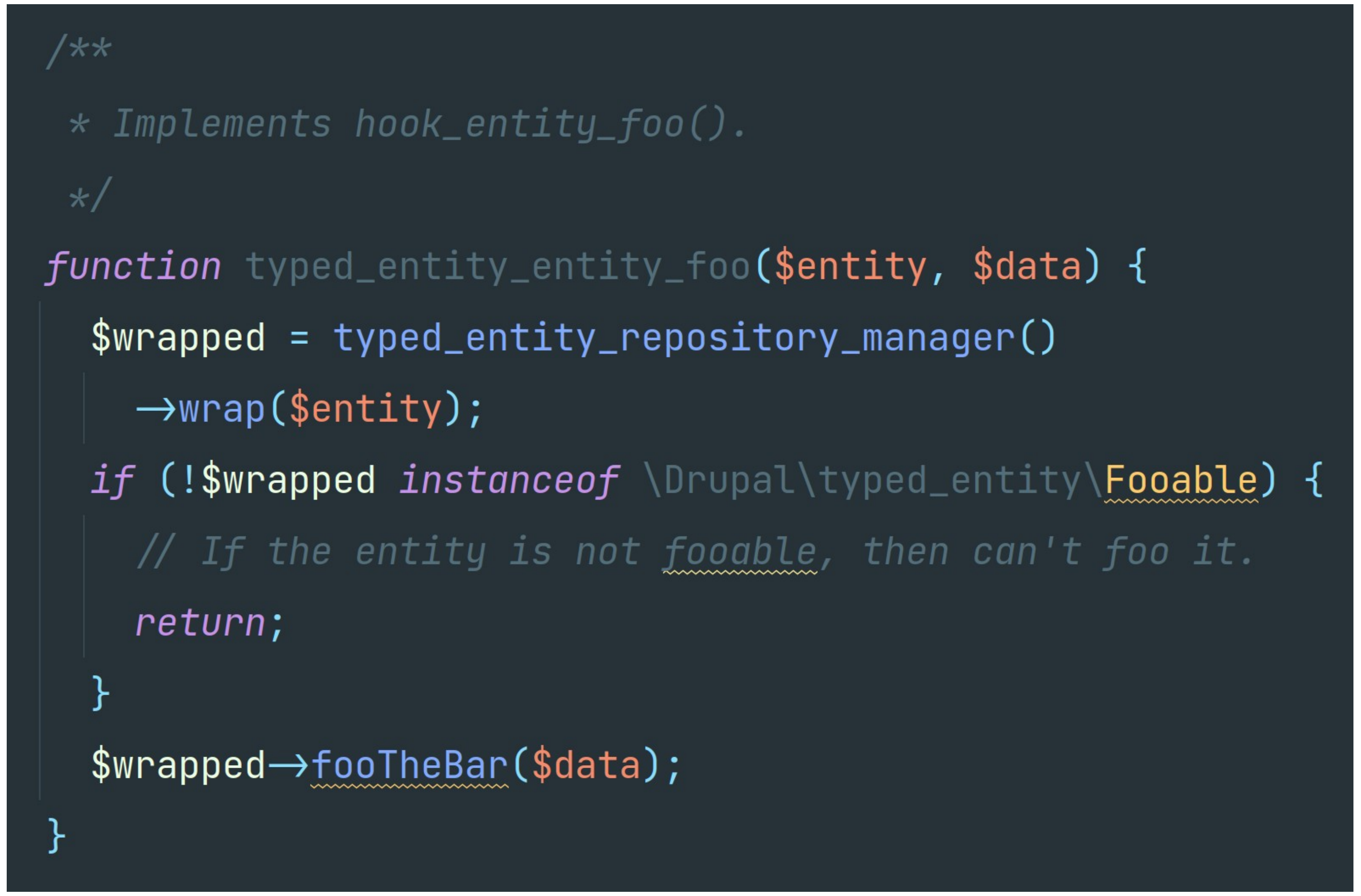
```php
/**
 * {@inheritdoc}
 */
public static function applies(TypedEntityContext $context): bool {
  $entity = $context→offsetGet( key: 'entity');
  return $entity→field_genre→value === 'Science Fiction';
}
```

CAN YOU IMPLEMENT HOOKS FOR ME?



There are many entity hooks. Typed Entity could implement them and delegate to interfaces.

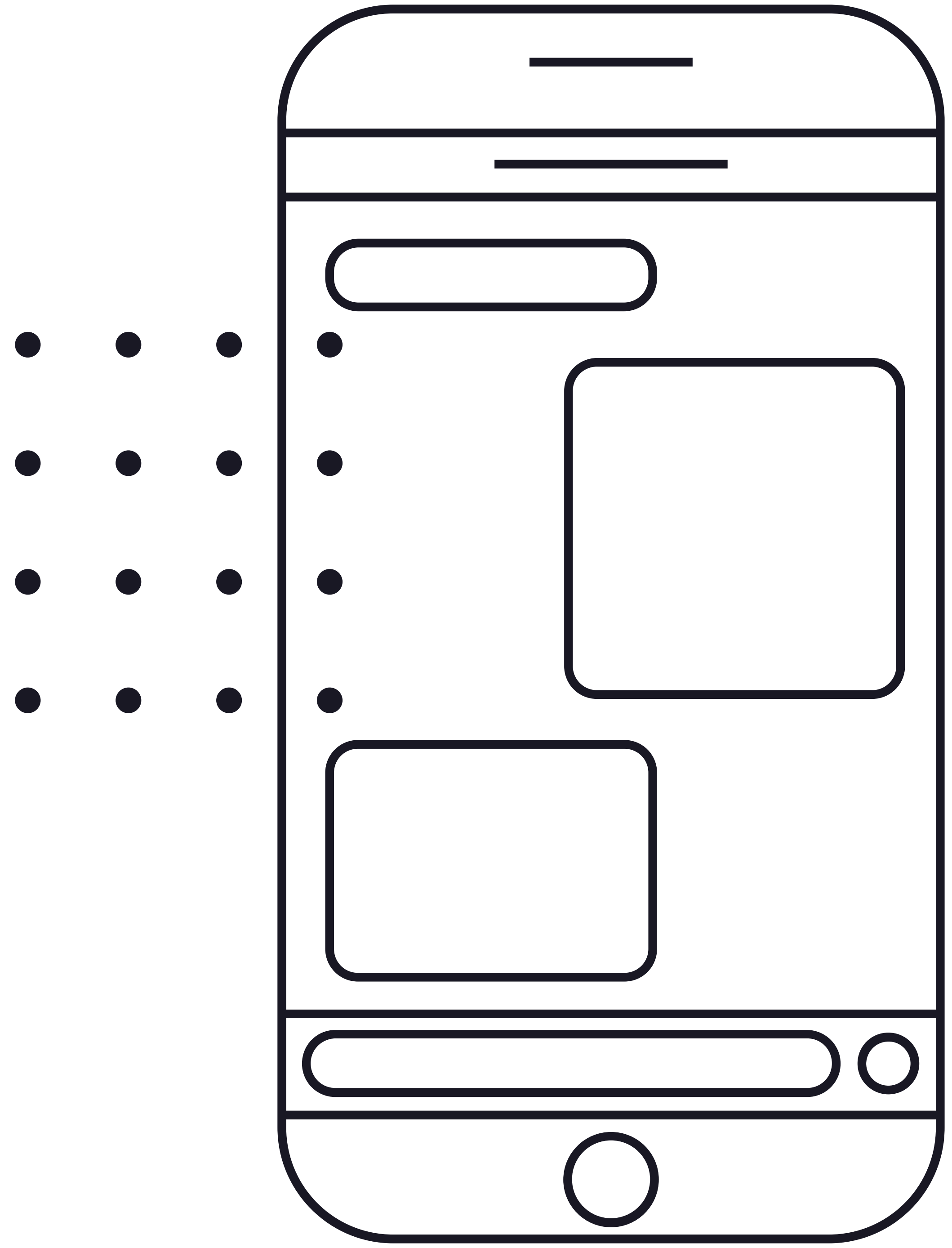
Does that happen?

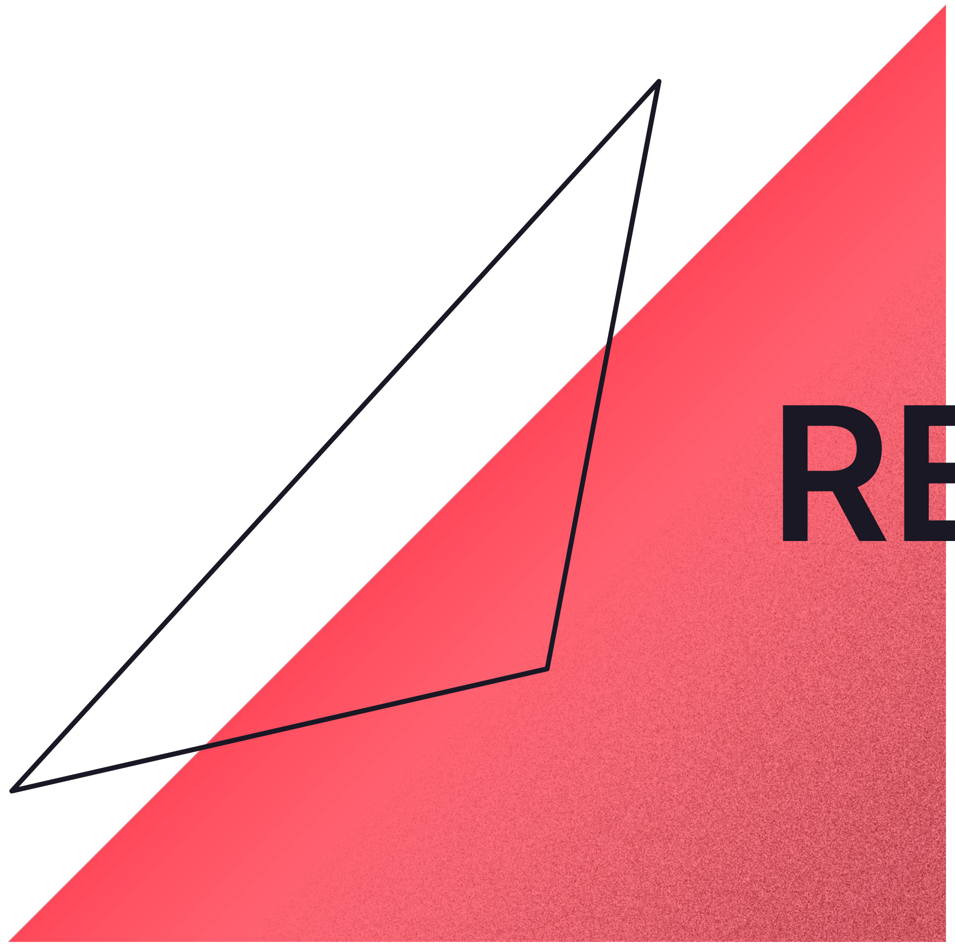


```php
/**
 * Implements hook_entity_foo().
 */
function typed_entity_entity_foo($entity, $data) {
  $wrapped = typed_entity_repository_manager()
    →wrap($entity);
  if (!$wrapped instanceof \Drupal\typed_entity\Fooable) {
    // If the entity is not fooable, then can't foo it.
    return;
  }
  $wrapped→fooTheBar($data);
}
```

ENTITIES HAVE MANY RESPONSIBILITIES

- **We render them as content in the screen**
- They are used for navigation purposes
- They hold SEO metadata
- We add decorative hints to them
- We use their fields to group content
- They can be embedded
- ...





RENDERERS

The most **common** thing we do with entities is render them.

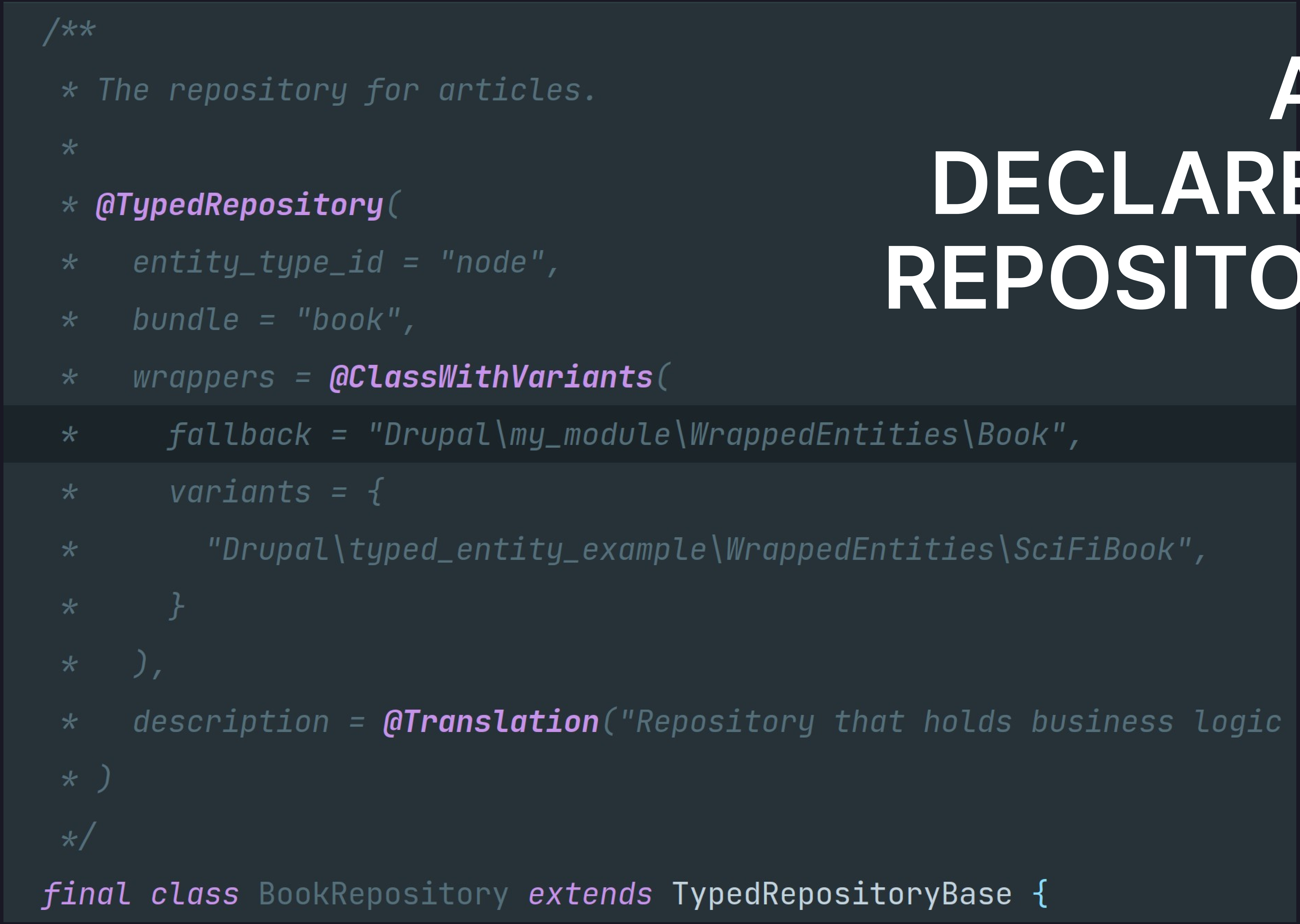
There is a natural `::applies` logic → **view modes**.

Not statistically proven.

Typed Entity let's you scope the relevant bits of your `preprocess`, `view_alter`, … in a **renderer** object.

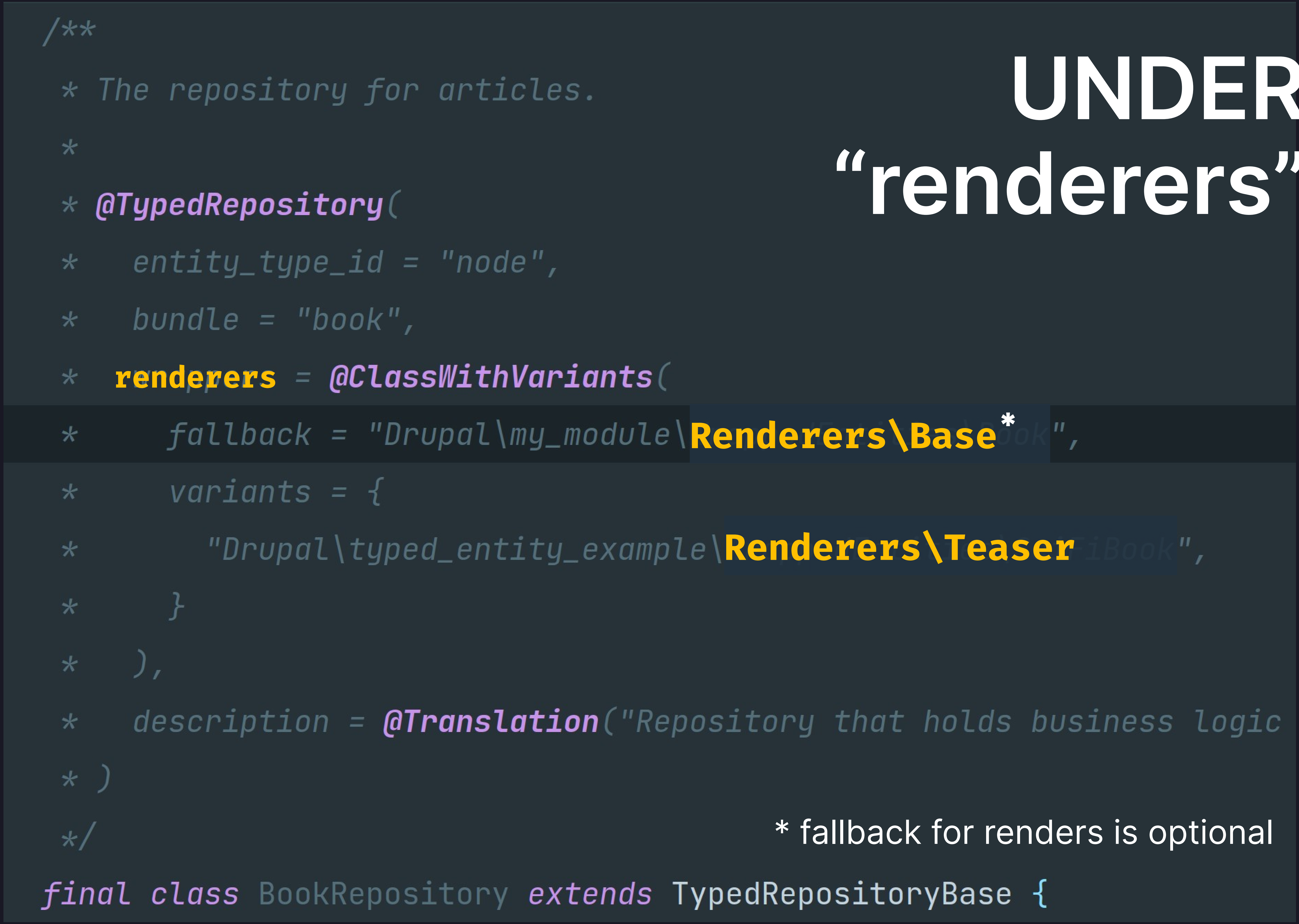
```
/**
 * Implements hook_entity_view_alter().
 */
function typed_entity_entity_view_alter(array &$build, EntityInterface $e
/**
 * Implements hook_preprocess().
 */
function typed_entity_preprocess(&$variables, $hook) {...}
/**
 * Implements hook_entity_display_build_alter().
 */
function typed_entity_entity_display_build_alter(&$build, $context) {
```

ALSO DECLARED IN REPOSITORIES



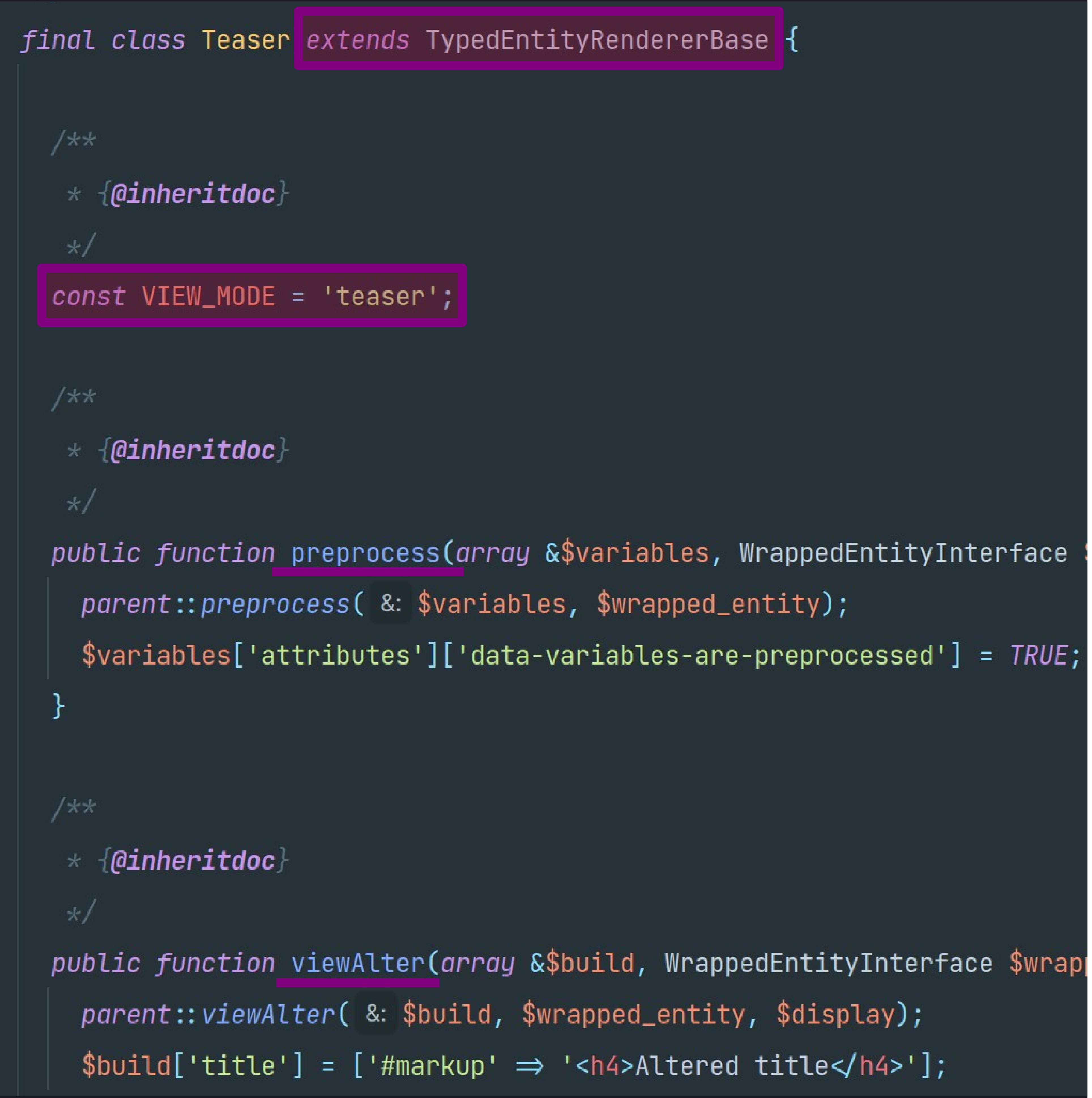
```
/**
 * The repository for articles.
 *
 * @TypedRepository(
 *   entity_type_id = "node",
 *   bundle = "book",
 *   wrappers = @ClassWithVariants(
 *     fallback = "Drupal\my_module\WrappedEntities\Book",
 *     variants = {
 *       "Drupal\typed_entity_example\WrappedEntities\SciFiBook",
 *     }
 *   ),
 *   description = @Translation("Repository that holds business logic
 * )
 */
final class BookRepository extends TypedRepositoryBase {
```

UNDER THE "renderers" KEY



```
/**
 * The repository for articles.
 *
 * @TypedRepository(
 *   entity_type_id = "node",
 *   bundle = "book",
 *  renderers = @ClassWithVariants(
 *     fallback = "Drupal\my_module\Renderers\Base*",
 *     variants = {
 *       "Drupal\typed_entity_example\Renderers\Teaser",
 *     }
 *   ),
 *   description = @Translation("Repository that holds business logic
 * )
 */
final class BookRepository extends TypedRepositoryBase {
```

* fallback for renders is optional



```
final class Teaser extends TypedEntityRendererBase {

  /**
   * {@inheritdoc}
   */
  const VIEW_MODE = 'teaser';

  /**
   * {@inheritdoc}
   */
  public function preprocess(array &$variables, WrappedEntityInterface
    parent::preprocess( &: $variables, $wrapped_entity);
    $variables['attributes']['data-variables-are-preprocessed'] = TRUE;
  }

  /**
   * {@inheritdoc}
   */
  public function viewAlter(array &$build, WrappedEntityInterface $wrap
    parent::viewAlter( &: $build, $wrapped_entity, $display);
    $build['title'] = ['#markup' => '<h4>Altered title</h4>'];
```

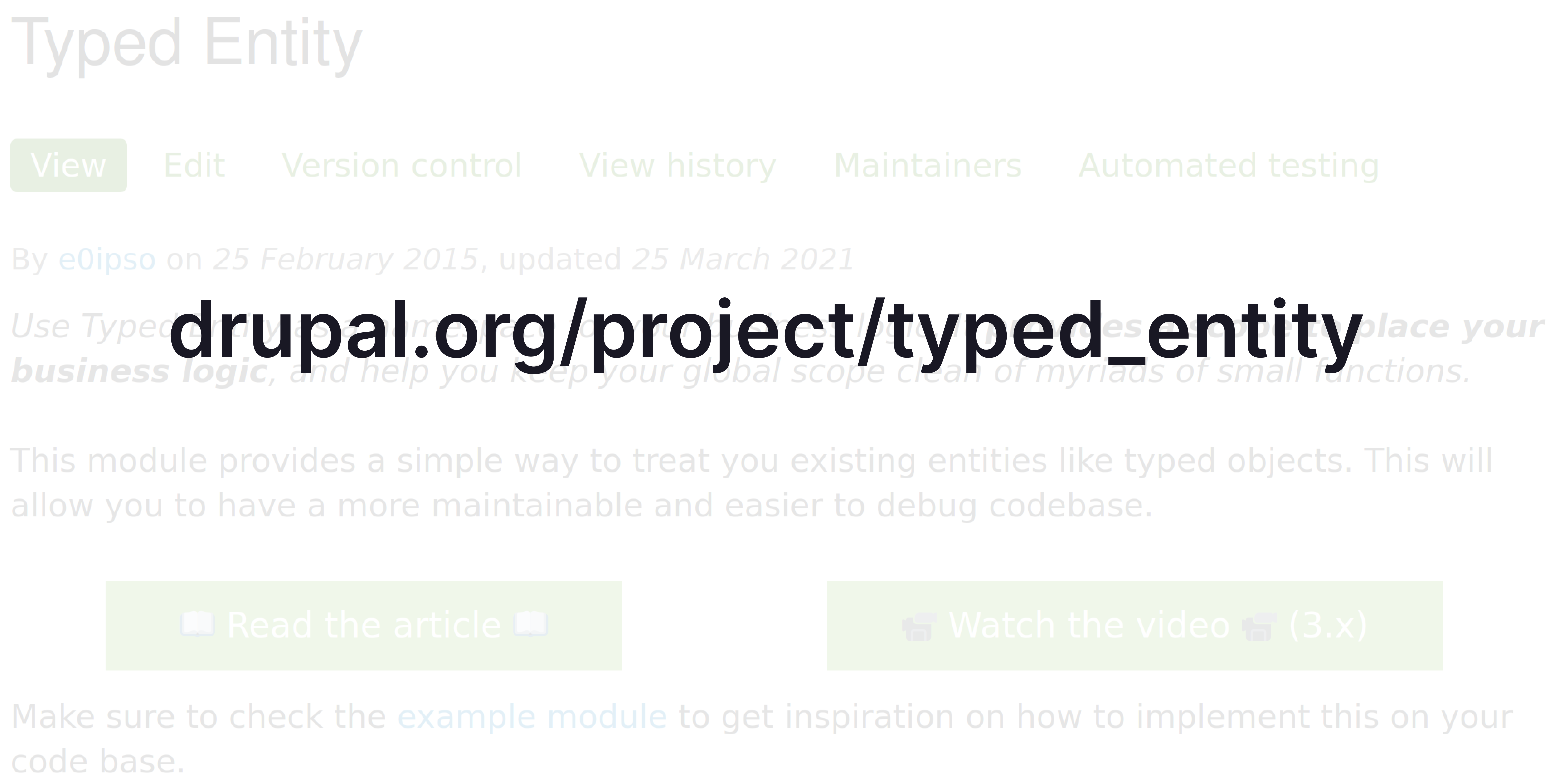
TESTABLE, DISCOVERABLE, MAINTAINABLE, AND READABLE

IN SUMMARY

- **Encapsulate** business logic in wrappers.
- Add **variants** (if needed) for specialized business logic.
- When implementing hooks/services check for wrapper **interfaces**.
- Use **renderers** instead of logic in rendering-specific hooks.
- Add variants per **view mode**.

MAKE THEM TESTABLE, DISCOVERABLE, MAINTAINABLE, AND READABLE

drupal.org/project/typed_entity



Typed Entity

By e0ipso on *25 February 2015*, updated *25 March 2021*

*Use Typed [illegible] logic. It **provides a scope to place your business logic**, and help you keep your global scope clean of myriads of small functions.*

This module provides a simple way to treat you existing entities like typed objects. This will allow you to have a more maintainable and easier to debug codebase.

Make sure to check the example module to get inspiration on how to implement this on your code base.