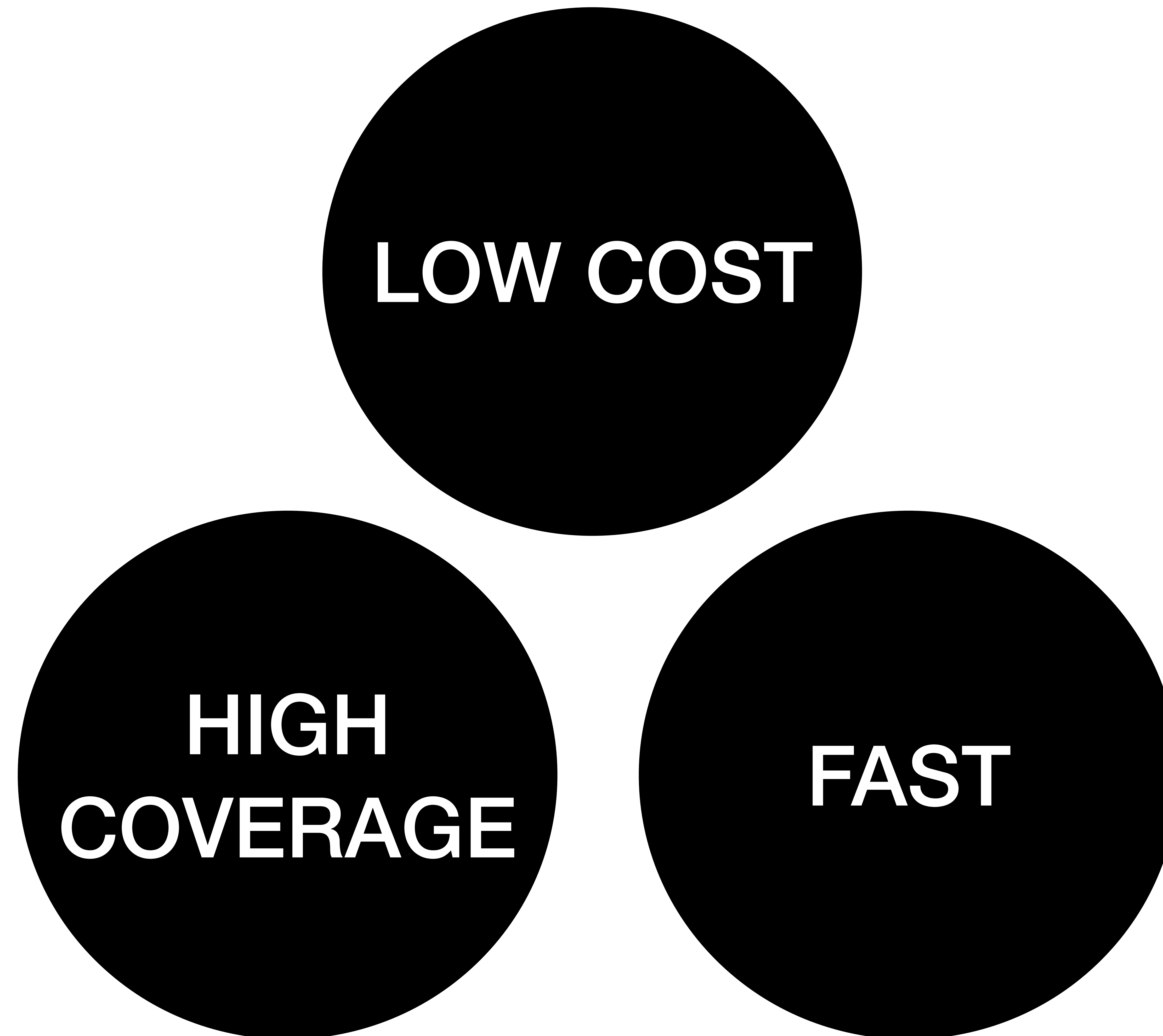
The Test Suite Holy Trinity

Dave Liddament

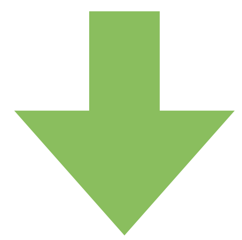
Testing is good,
but…

- 4 Projects
- 10+ years in production
- Change
- Made many mistakes



Everything is a compromise.

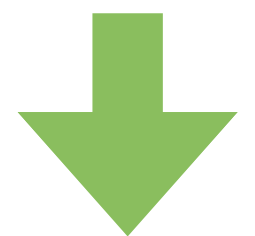
Testing continuum



Unit

System

Testing continuum



Unit

System

Unit		System
Fast	**Execution speed**	Slow
Low	**Effort to write**	High
Easy	**Debugging ease**	Hard
Abstract	**Realism**	Concrete
Easy/Hard	**Ability to change**	Easy/Hard
High/Low	**Coverage**	High/Low

Cost negative tests?

```
class TemplateParserTest extends TestCase
{
    public static function happyPath(): iterable
    {
        return [
          'No substitutions' => ["Hello World!", 'Hello World!',[]],
          '1 substitutions' => ["Hello World!", 'Hello {0}!', ['World']],
          '2 substitutions' => ["Hello World!", '{0} {1}!',['Hello', 'World']],
          'reverse order' => ["PHP is great", '{1} is {0}',['great', ‘PHP']],
          … and so on …
        ];
    }

    #[DataProvider(‘happyPath')]
    public function testTemplate(string $expected, string $input, array $data): void
    {
        $actual = TemplateParser::parse($input, $data);
        Assert::assertSame($expected, $actual);
    }
}
```

Cost negative tests?

```
class TemplateParserTest extends TestCase
{
    public static function happyPath(): iterable
    {
        return [
          'No substitutions' => ["Hello World!", 'Hello World!',[]],
          '1 substitutions' => ["Hello World!", 'Hello {0}!', ['World']],
          '2 substitutions' => ["Hello World!", '{0} {1}!',['Hello', 'World']],
          'reverse order' => ["PHP is great", '{1} is {0}',['great', ‘PHP']],
          … and so on …
        ];
    }

    #[DataProvider(‘happyPath')]
    public function testTemplate(string $expected, string $input, array $data): void
    {
        $actual = TemplateParser::parse($input, $data);
        Assert::assertSame($expected, $actual);
    }
}
```

Test harness

```
class TemplateParserTest extends TestCase
{
    public static function happyPath(): iterable
    {
        return [
          'No substitutions' => ["Hello World!", 'Hello World!',[]],
          '1 substitutions' => ["Hello World!", 'Hello {0}!', ['World']],
          '2 substitutions' => ["Hello World!", '{0} {1}!',['Hello', 'World']],
          'reverse order' => ["PHP is great", '{1} is {0}',['great', ‘PHP']],
          … and so on …
        ];
    }

    #[DataProvider('happyPath')]
    public function testTemplate(string $expected, string $input, array $data): void
    {
        $actual = TemplateParser::parse($input, $data);
        Assert::assertSame($expected, $actual);
    }
}
```

4 tests!

```
class TemplateParserTest extends TestCase
{
    public static function happyPath(): iterable
    {
        return [
          'No substitutions' => ["Hello World!", 'Hello World!',[]],
          '1 substitutions' => ["Hello World!", 'Hello {0}!', ['World']],
          '2 substitutions' => ["Hello World!", '{0} {1}!',['Hello', 'World']],
          'reverse order' => ["PHP is great", '{1} is {0}',['great', ‘PHP']],
          … and so on …
        ];
    }

    #[DataProvider('happyPath')]
    public function testTemplate(string $expected, string $input, array $data): void
    {
        $actual = TemplateParser::parse($input, $data);
        Assert::assertSame($expected, $actual);
    }
}
```

This is a test!

```
'1 substitution' => [
    "Hello World!",
    'Hello {0}!',
    ['World']
],
```

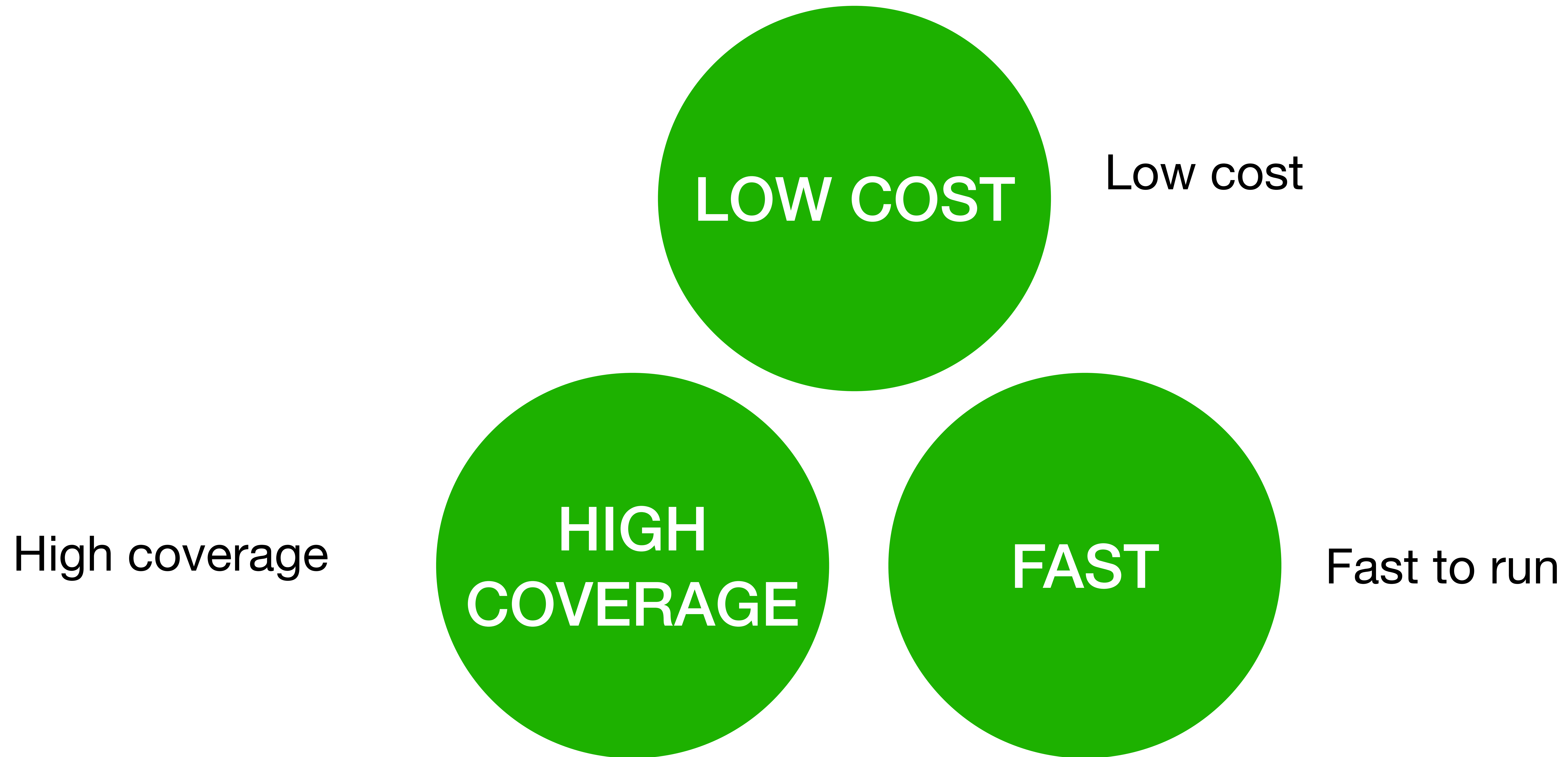
Readability

```
'1 substitution' => [
    'expected' => "Hello World!",
    'input'    => 'Hello {0}!',
    'data'     => ['World']
],
```

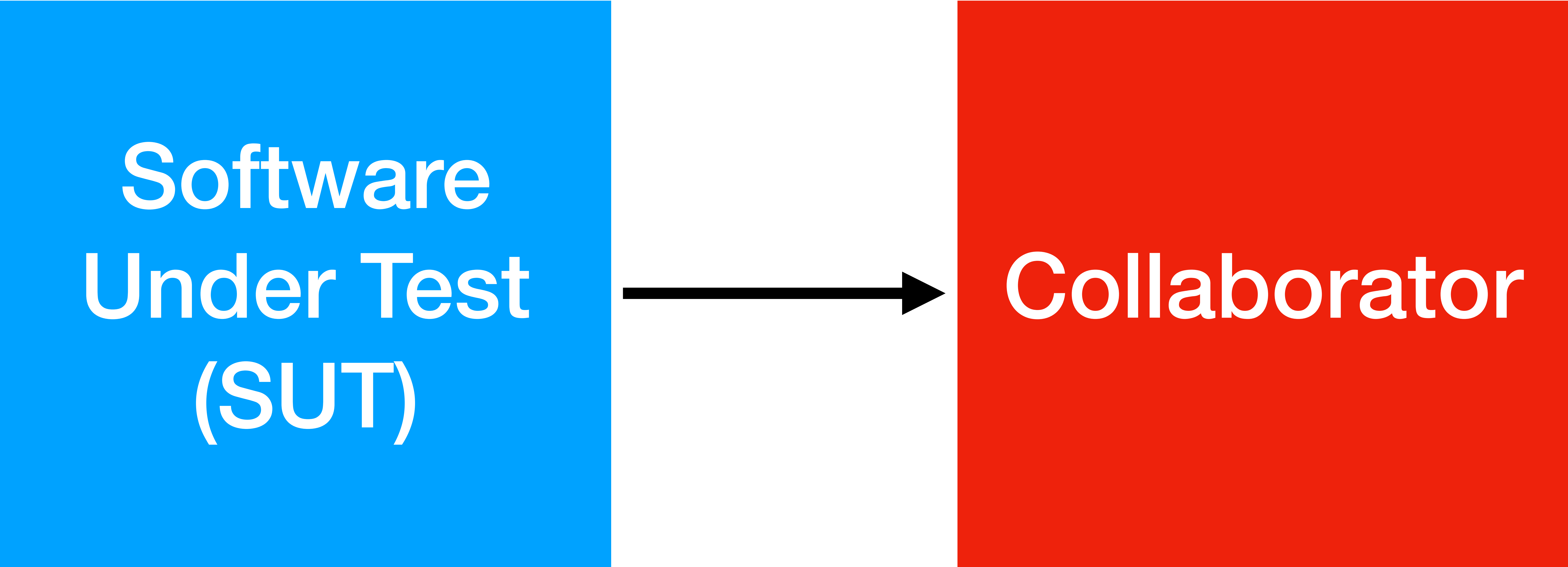
Data Providers ~ Good Architecture

```
class TemplateParserTest extends TestCase
{
    public static function happyPath(): iterable
    {
        return [
          'No substitutions' => ["Hello World!", 'Hello World!',[]],
          '1 substitutions' => ["Hello World!", 'Hello {0}!', ['World']],
          '2 substitutions' => ["Hello World!", '{0} {1}!',['Hello', 'World']],
          'reverse order' => ["PHP is great", '{1} is {0}',['great', ‘PHP']],
          … and so on …
        ];
    }

    #[DataProvider('happyPath')]
    public function testTemplate(string $expected, string $input, array $data): void
    {
        $actual = TemplateParser::parse($input, $data);
        Assert::assertSame($expected, $actual);
    }
}
```

Why is this a good test?



But reality…

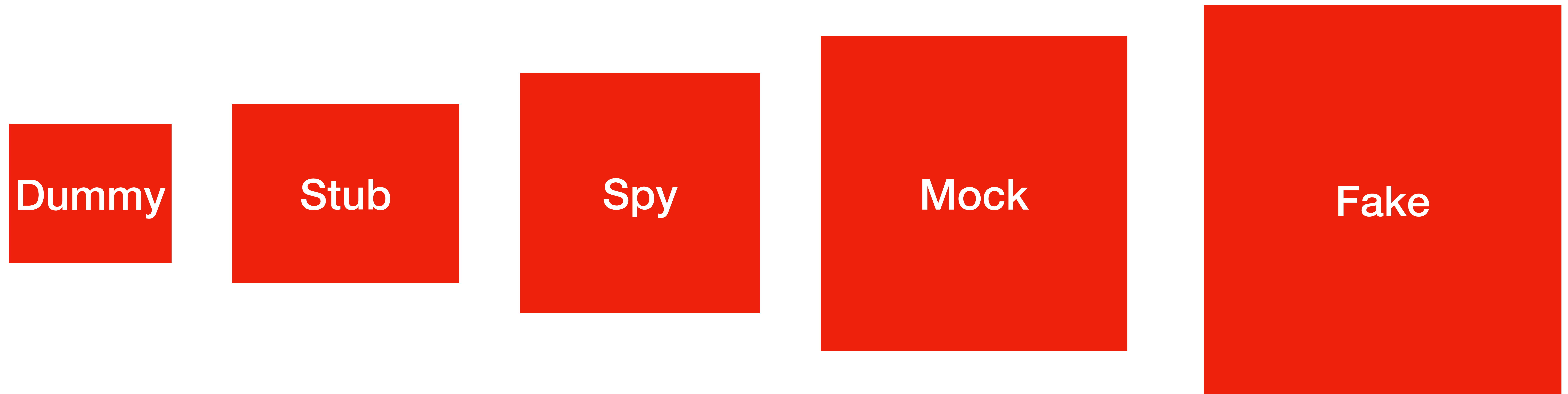

Software
Under Test
(SUT)
Collaborator

Must use Double	It Depends	Don’t use Double
Payment	Database Repository	Value objects
Email	File System	DTOs
External API	Time	
	Random	

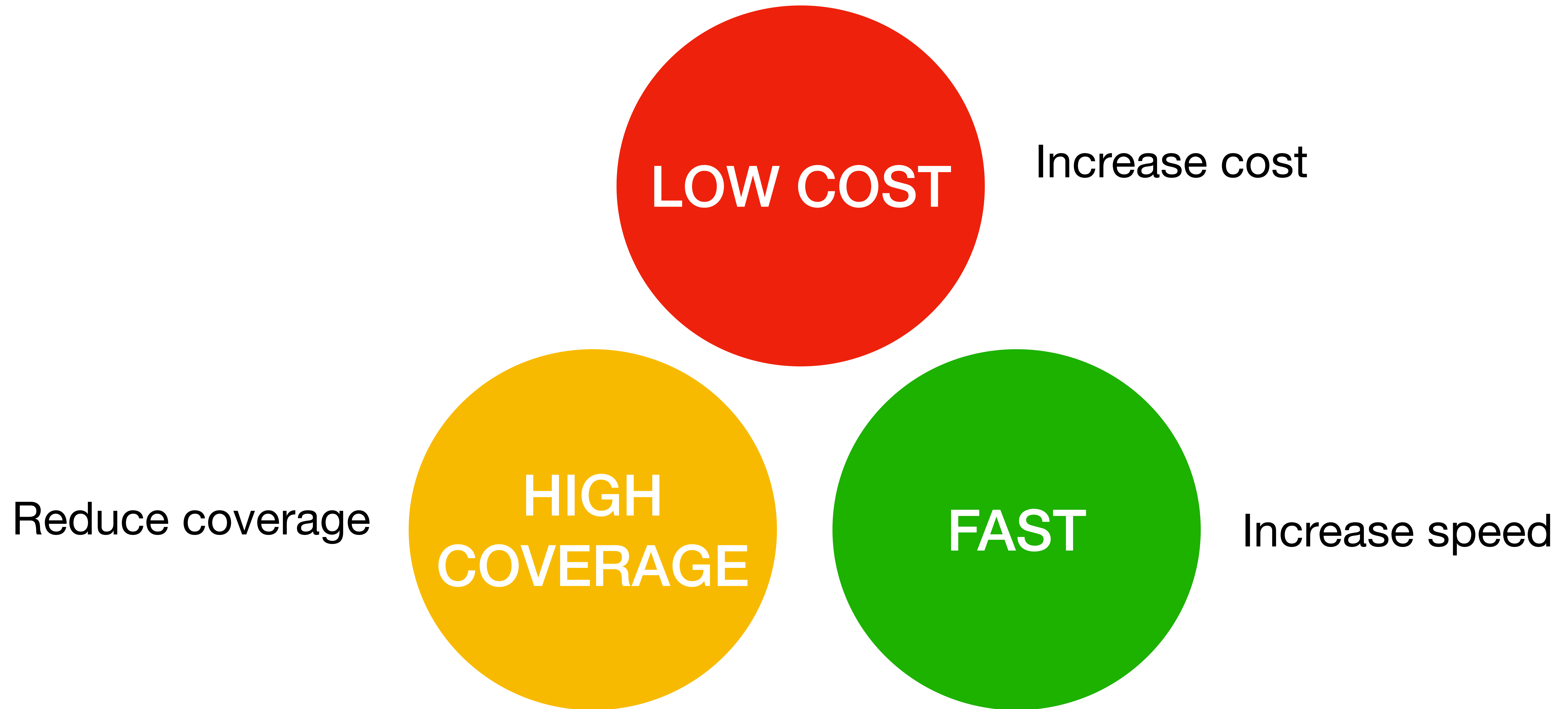
How Risky Is Your Double?

Interface width	Example	Double safety
Narrow (data orientated)	Email sending	Safe
Wide (behaviour rich)	Database	Risky

Types of Doubles



Test Doubles

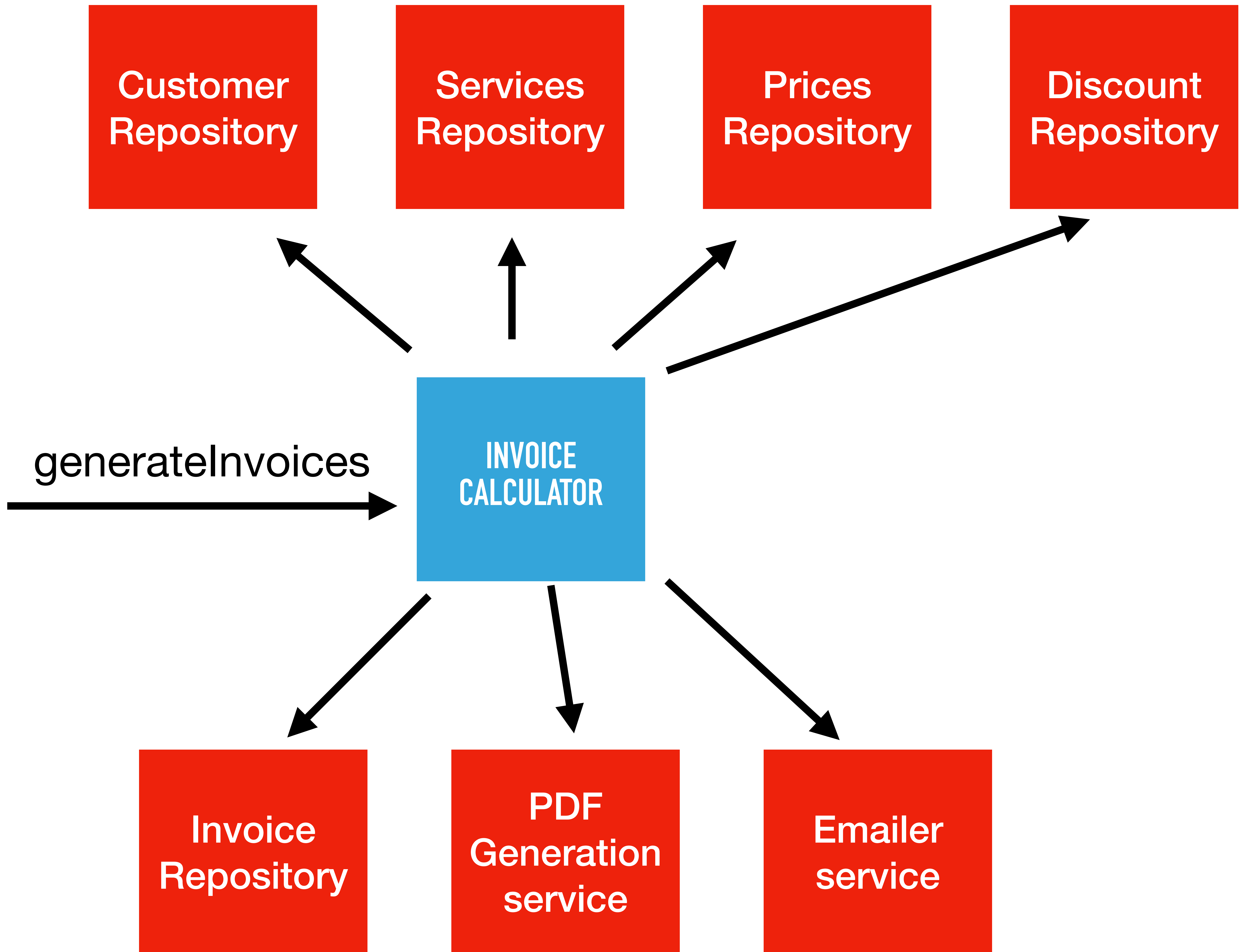


Everything is a compromise.

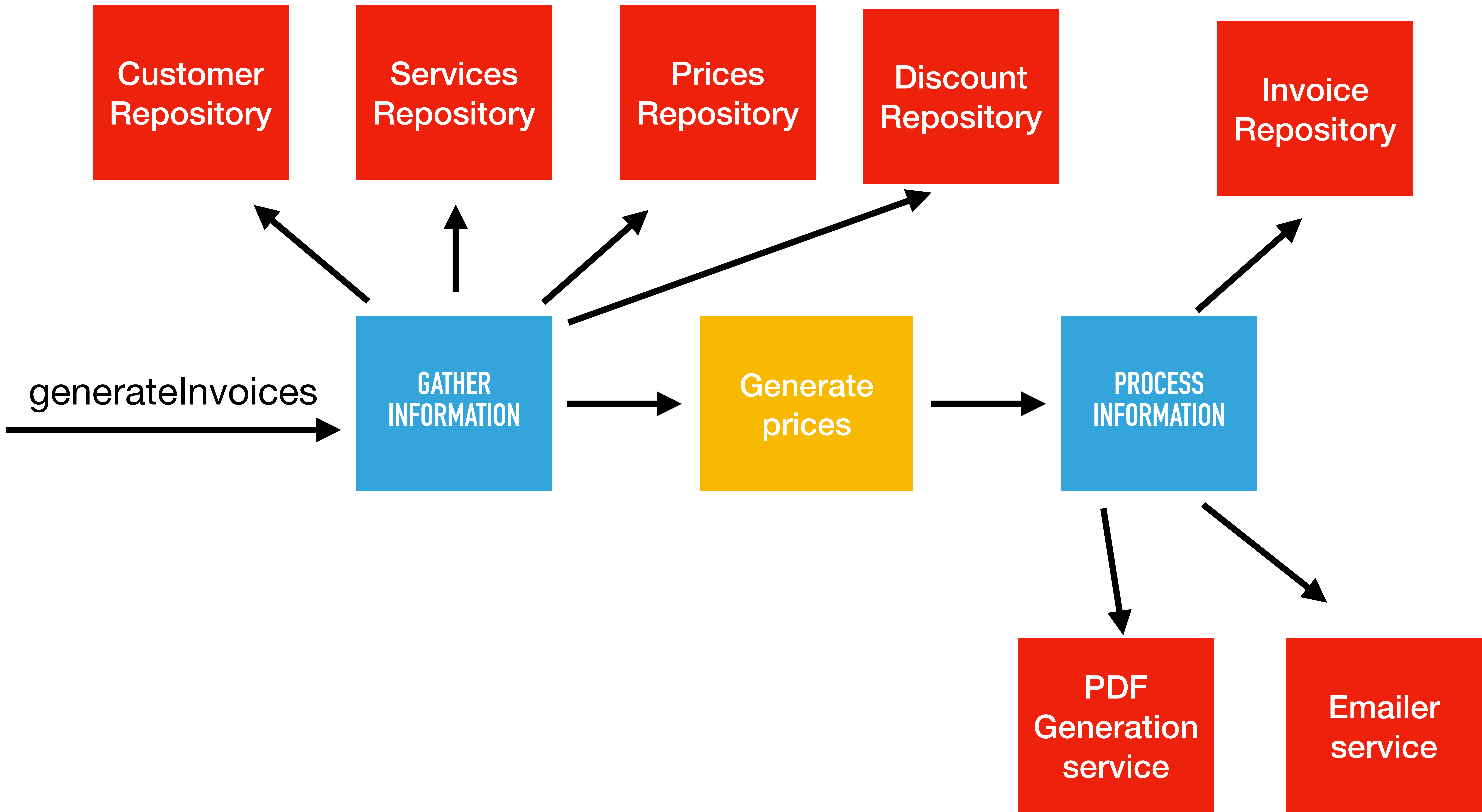
Can we avoid the complexity of doubles?

Example: Invoice Generation

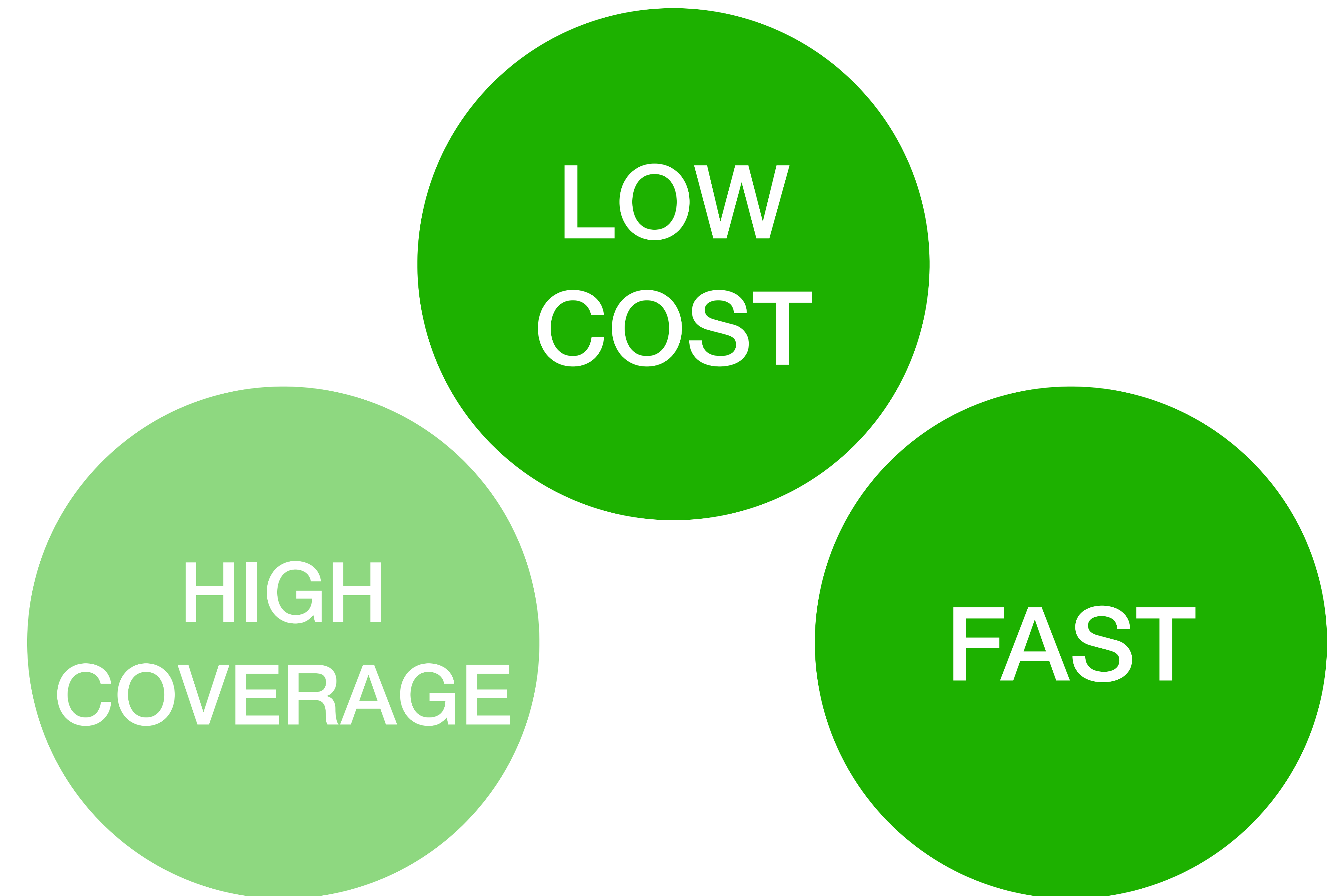
- Get all customers. For each customer: [Customer repository]
  - Get Services they used during invoice month [Services repository]
  - Lookup prices [Price repository]
  - Lookup discounts [Discount repository]
  - Calculate prices (complex logic)
  - Generate Invoice [Invoice Repository]
  - Create PDF [PDF Generation service]
  - Email PDF [Email service]

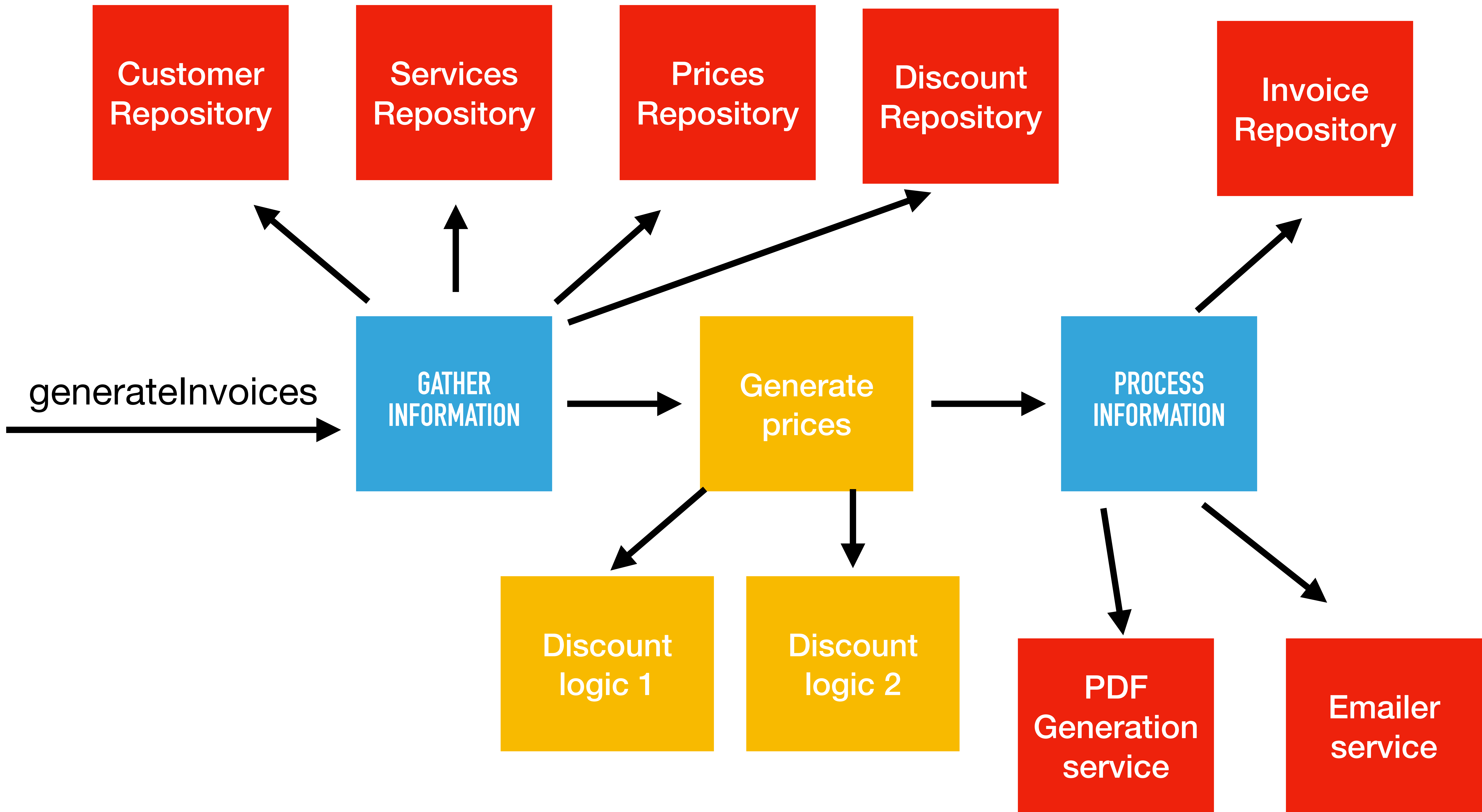


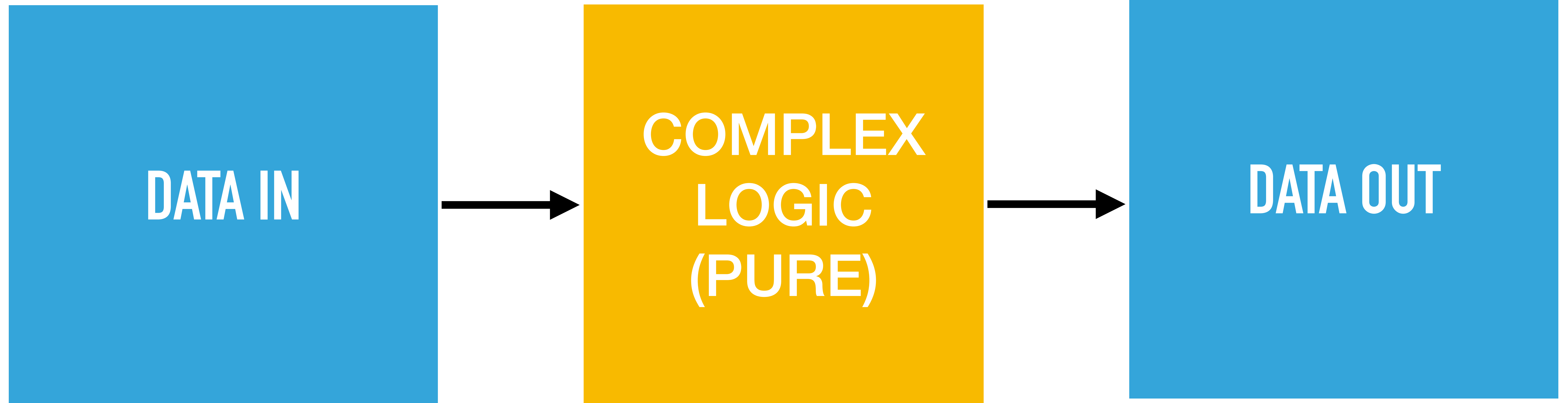
Customer Repository
Services Repository
Prices Repository
Discount Repository
generateInvoices
INVOICE CALCULATOR
Invoice Repository
PDF Generation service
Emailer service


Customer Repository
Services Repository
Prices Repository
Discount Repository
Invoice Repository
generateInvoices
GATHER INFORMATION
Generate prices
PROCESS INFORMATION
PDF Generation service
Emailer service

```php
final readonly class PriceCalculator
{
    /** @return array<Invoice> */
    public function calculatePrices(

        array $customers,
        array $services,
        array $prices,
        array $discounts,

    ): array {


    }
}
```

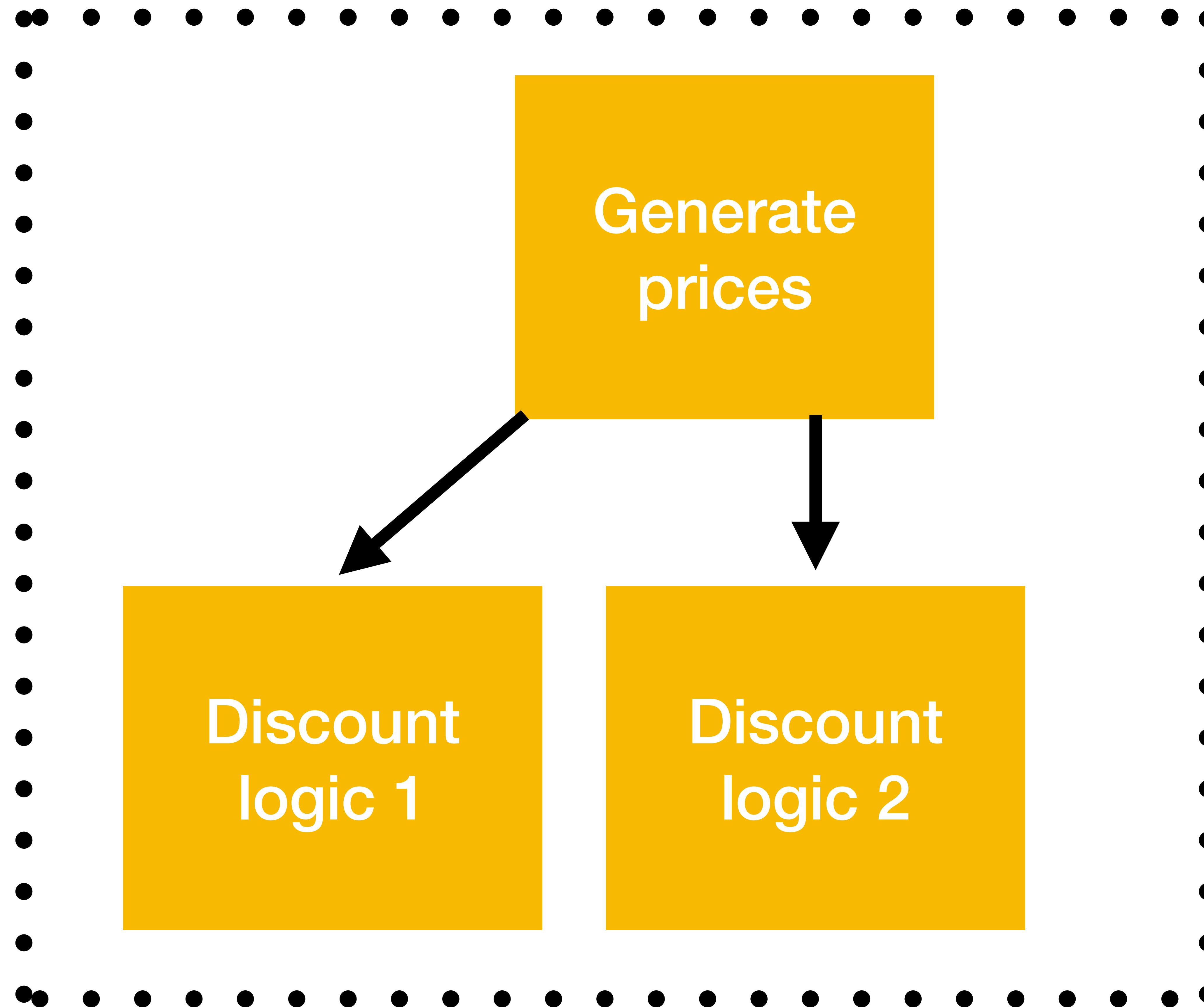



Customer Repository
Services Repository
Prices Repository
Discount Repository
Invoice Repository
generateInvoices
GATHER INFORMATION
Generate prices
PROCESS INFORMATION
Discount logic 1
Discount logic 2
PDF Generation service
Emailer service


DATA IN
COMPLEX
LOGIC
(PURE)
DATA OUT

A layered approach

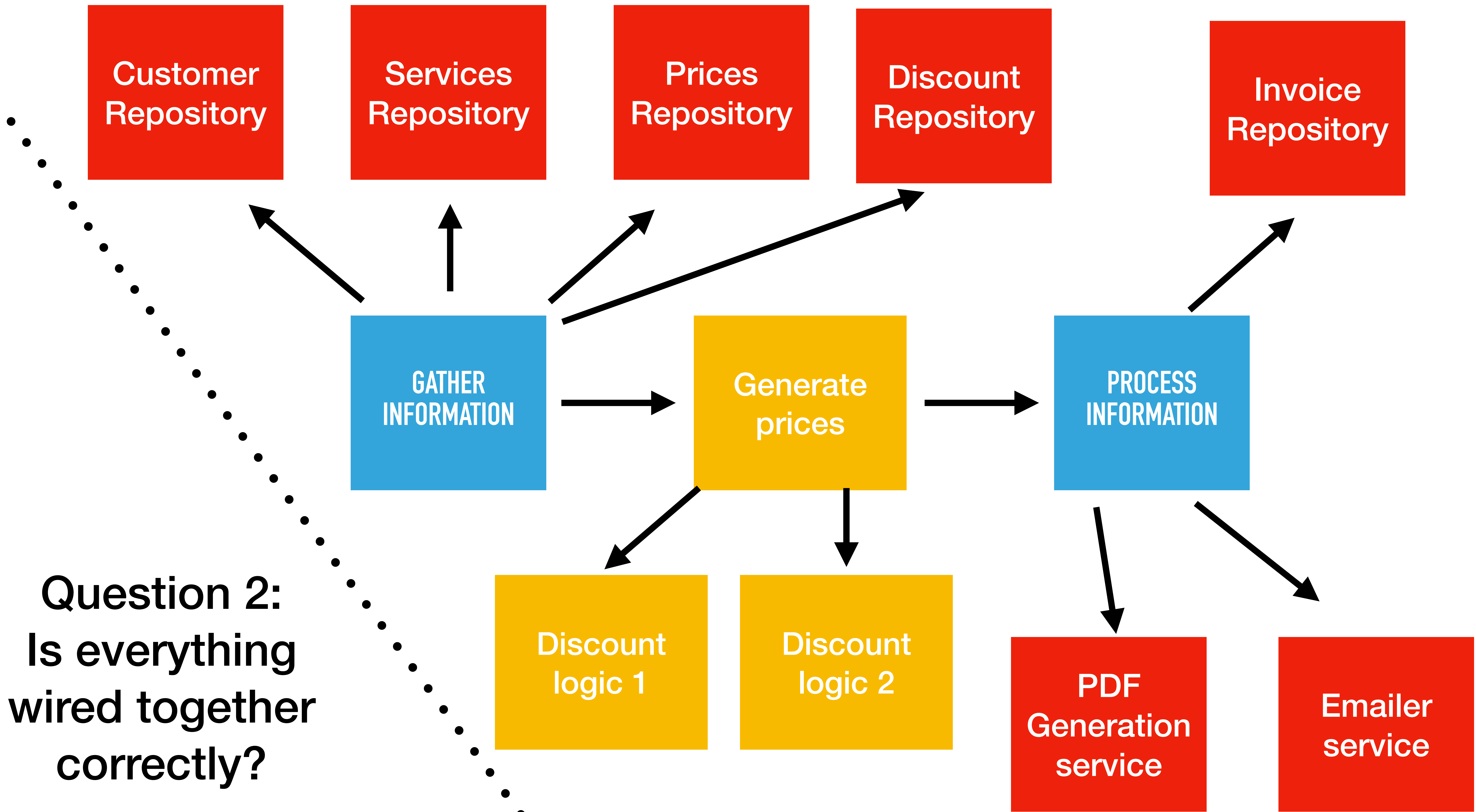
Question 1:
Is all the pricing logic correct?

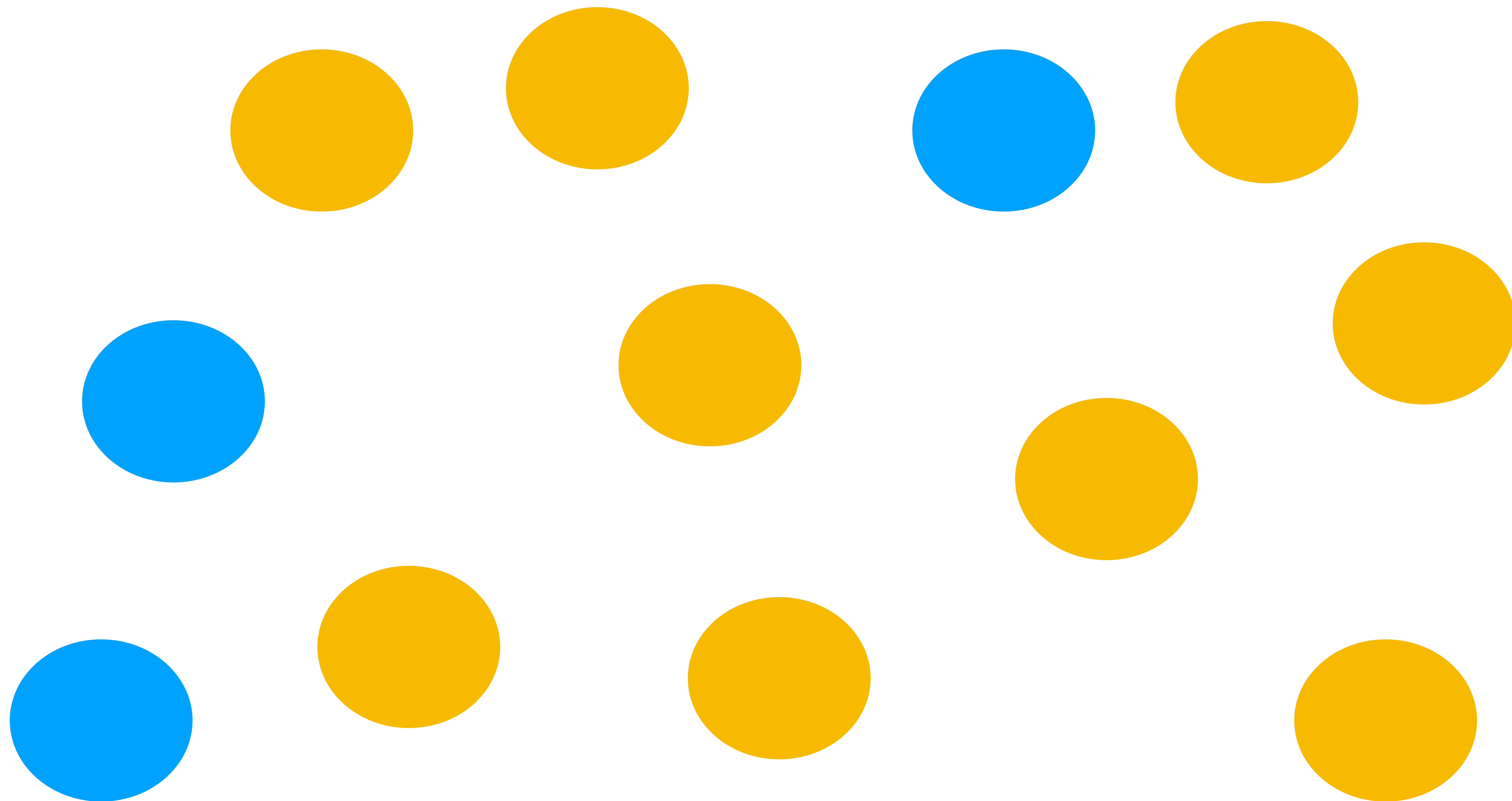


Test continuum

Pure logic

(Lots of tests)


Customer Repository
Services Repository
Prices Repository
Discount Repository
Invoice Repository
GATHER INFORMATION
Generate prices
PROCESS INFORMATION
Discount logic 1
Discount logic 2
PDF Generation service
Emailer service
Question 2:
Is everything wired together correctly?



Tracking tests

```
class PriceCalculationTest extends TestCase
{

    #[Ticket('FEAT-006')]
    public function testDiscount(): void
    {
    }
}
```

```
phpunit --group 'FEAT-006' --list-tests
```

Bonus 1: If only I had...

```
File: specs/FEAT-006.md

# Price calculation

Calculate price using <description of the feature>.

Discounts are applied using <further description>.
```

If only I had...

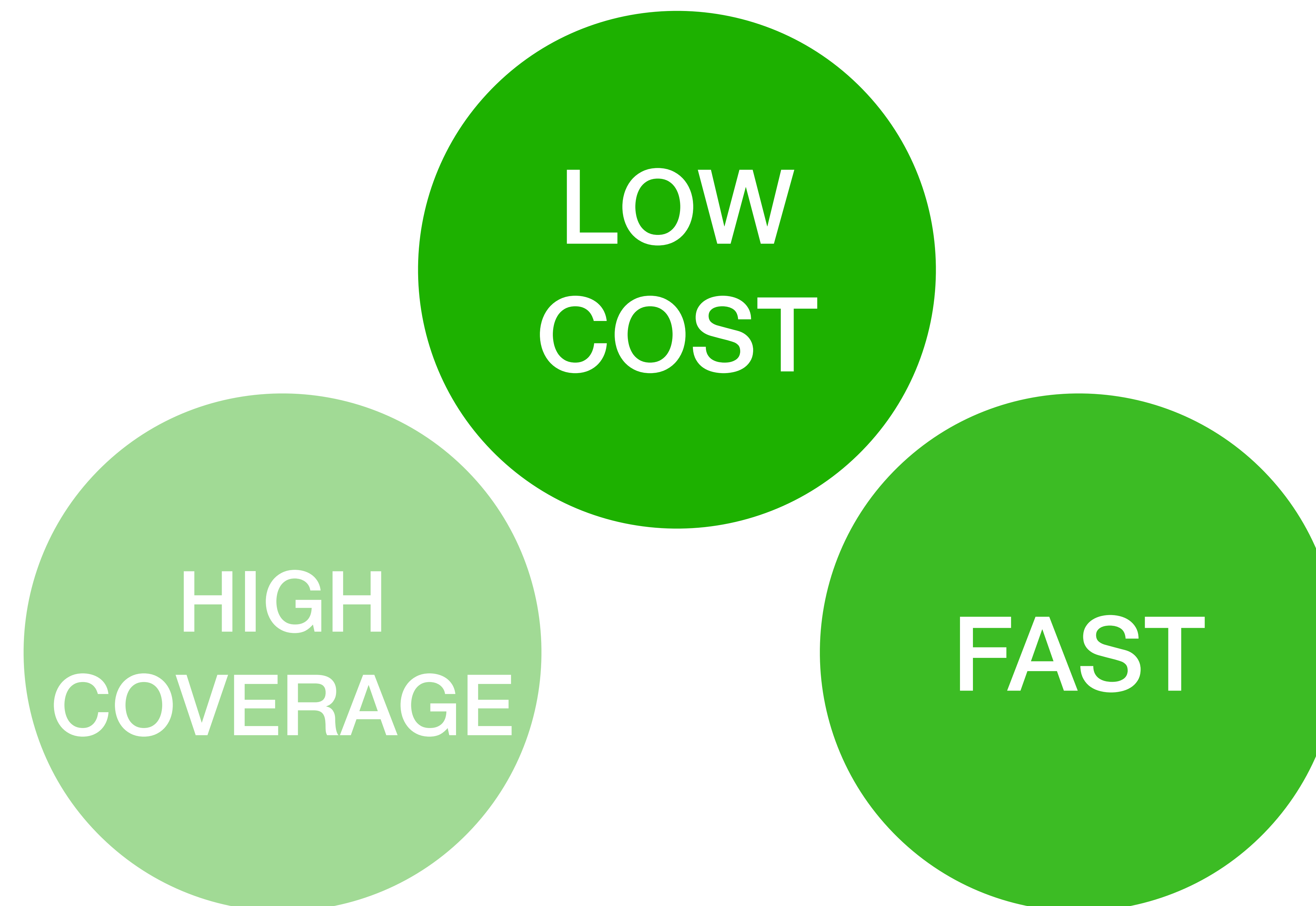
```
File: specs/FEAT-006.md

# Price calculation

Calculate price using <description of the feature>.

Discounts are applied using <further description>.
```

✅ What happens when unrelated test fails

✅ Onboarding

✅ AI (understanding codebase + review)

A layered approach and pure functions give you…



Decouple your tests from the SUT

Your tests should only change if the behaviour they are testing changes *

Test setup

```
class Company
{
    public function __construct(
        string $name,
        String $website,
        string $subdomain,
    ) {}

}
```

No longer valid...

```
$company = new Company(“Company 1”,“www.company1.com”);
```

```
companies:
  - name: Company 1
    website: www.company1.com

  - name: Company 2
    website: www.company2.com
```

Object Mother

```
class CompanyObjectMother {

   public function company1(): Company {

      … return company if already created …

      $company = new Company( “Company 1”
                              “www.company1.com”);

     return $company;
}
```

One change required

```
class CompanyObjectMother {

   public function company1(): Company {

      … return company if already created …

      $company = new Company( “Company 1”
                              “www.company1.com”
                              “Company1”);

     return $company;
}
```

Builder

```
$company = new CompanyBuilder()->build();


$company = new CompanyBuilder()
                ->name(“Company 2”)
                ->website(“www.company2.com”)
                ->subdomain(“company2”)
                ->build();
```

Defer to other Object Mothers/Builders

```
class UserObjectMother {

   public function anna(): User {

      $company = $this->companyObjectMother()->company1();

      $user = $userService->registerUser(
                  “anna@acme.com”,
                  “Anna”,
                  “Passw0rd”
                  $company);

      return $user;
}
```

Everything changes!

```
public function testX(): void
{
  // Given
  $player = new PlayerBuilder()
                  ->title(“TEM”)
                  ->build();

  $this->sendMessage($player, “start”);

  // When

  … Rest of test…
```

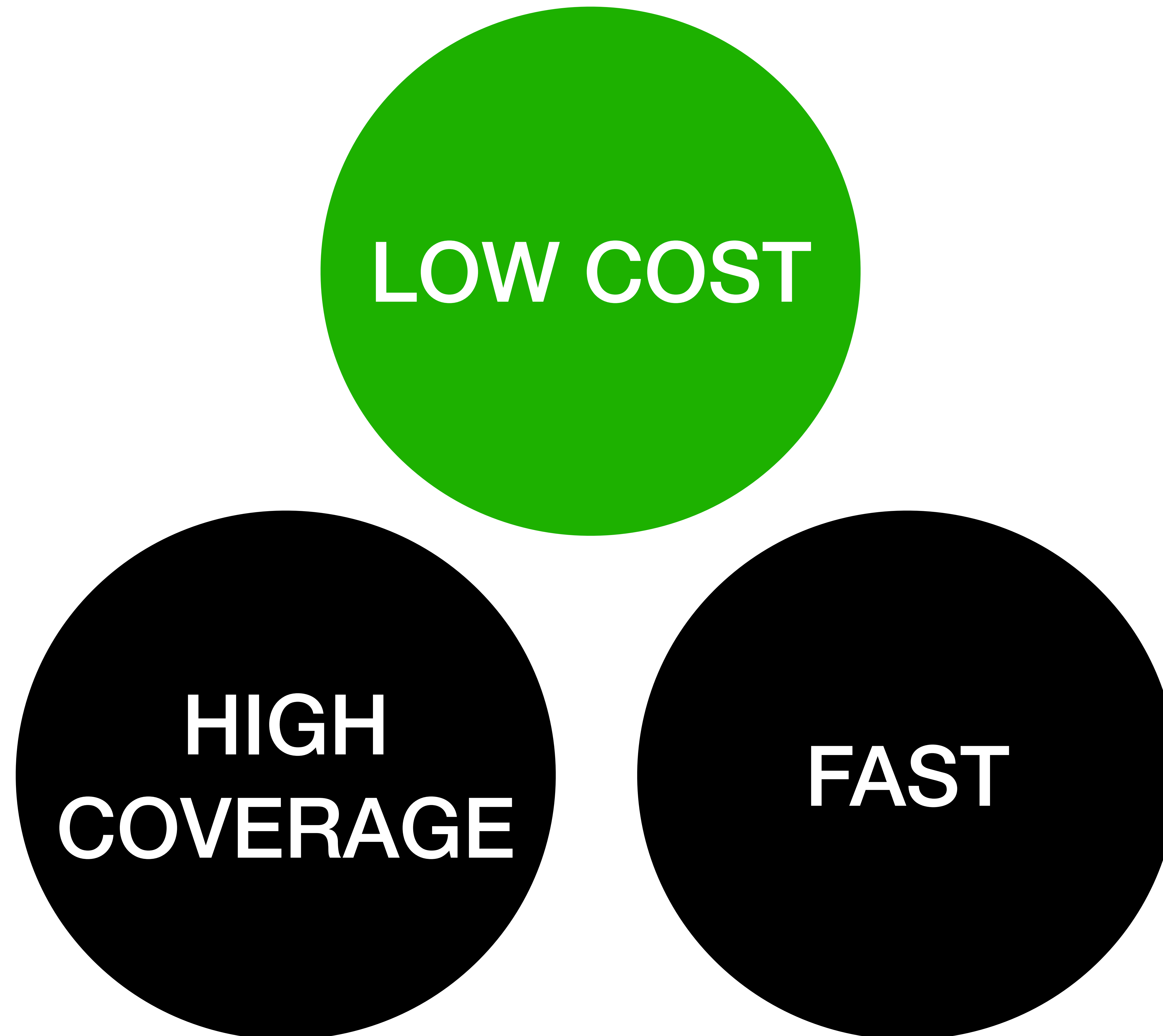
If only…

```
public function testX(): void
{
   // Given
   $player = new PlayerBuilder()
                  ->title(“TEM”)
                  ->startGame()
                  ->build();

  // When

   … Rest of test…
```

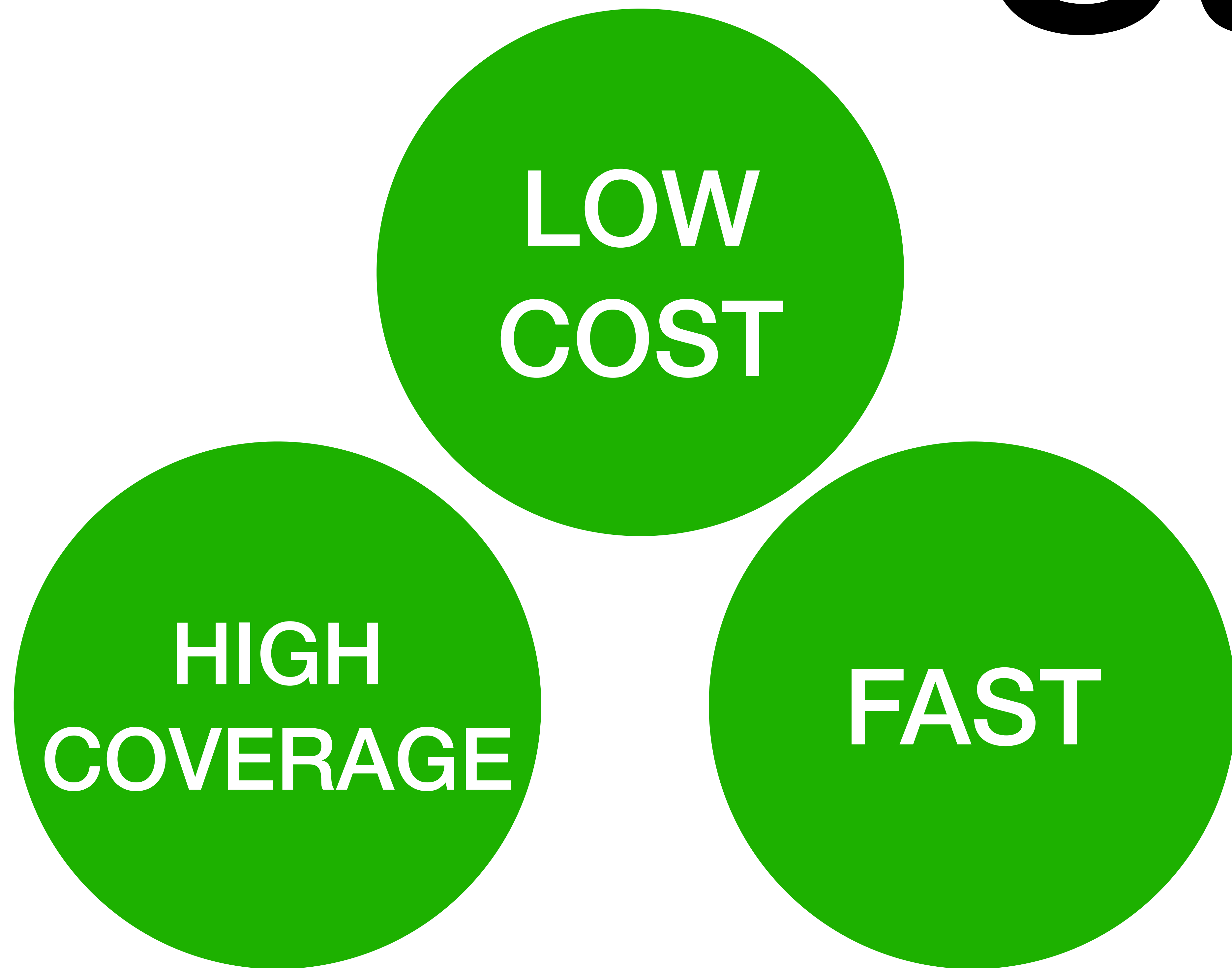
Use DSL but at a code level

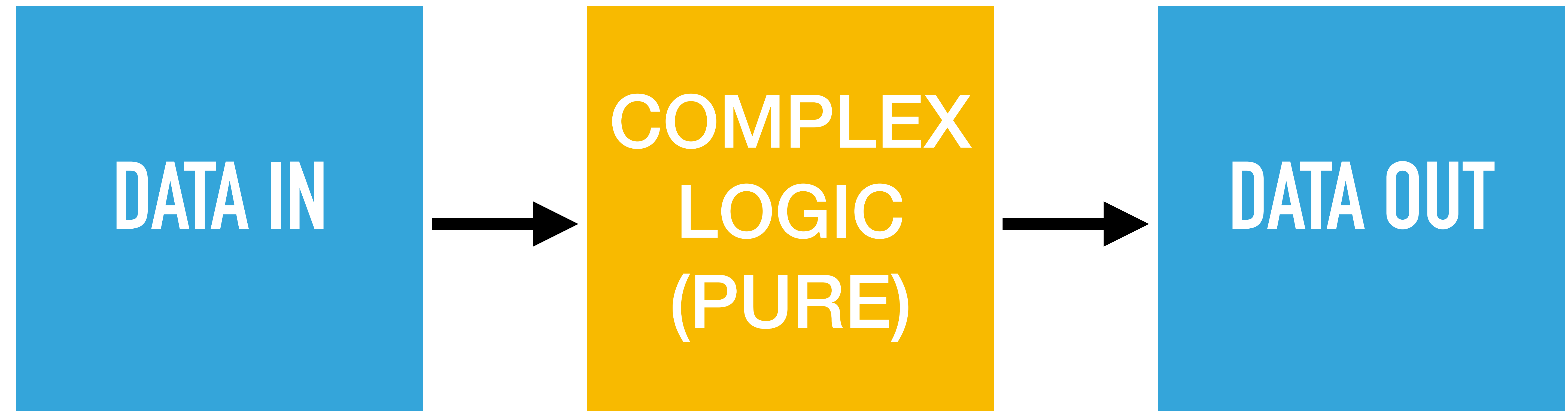
```
public function testIncorrectAnswerProcessing(): void
{

  // Setup a started game

  $this->assertClueReceived($player, “clue 1”);

  $this->sendMessage($player, “wrong answer”);

  $this->assertWrongAnswerMessageReceived($player);
}
```

Decouple from SUT



Summary





Pragmatic use of doubles

Decouple test from SUT

- Builder / Object mother patterns
- DRY in test setup
- Custom assertions and actions

Dave Liddament

@daveliddament
@daveliddament@phpc.social
@daveliddament.bsky.social
github.com/DaveLiddament
www.linkedin.com/in/daveliddament/

Me when I'm not coding!

Bonus
Verifying doubles: Contract tests (1)

```
interface UserRepository
{
    public function save(User $user): void;

    public function findByEmail(string $email): ?User;
}
```

Verifying doubles: Contract tests (2)

```
abstract class UserRepositoryContractTest extends TestCase
{
    abstract protected function createRepository(): UserRepository;

    public function testFindByEmailIsCaseInsensitive(): void
    {
        $repo = $this->createRepository();
        $user = new User(new UserId('123'), 'Alice@Example.com');

        $repo->save($user);

        $found = $repo->findByEmail('alice@example.com');
        $this->assertEquals($user, $found);
    }
}
```

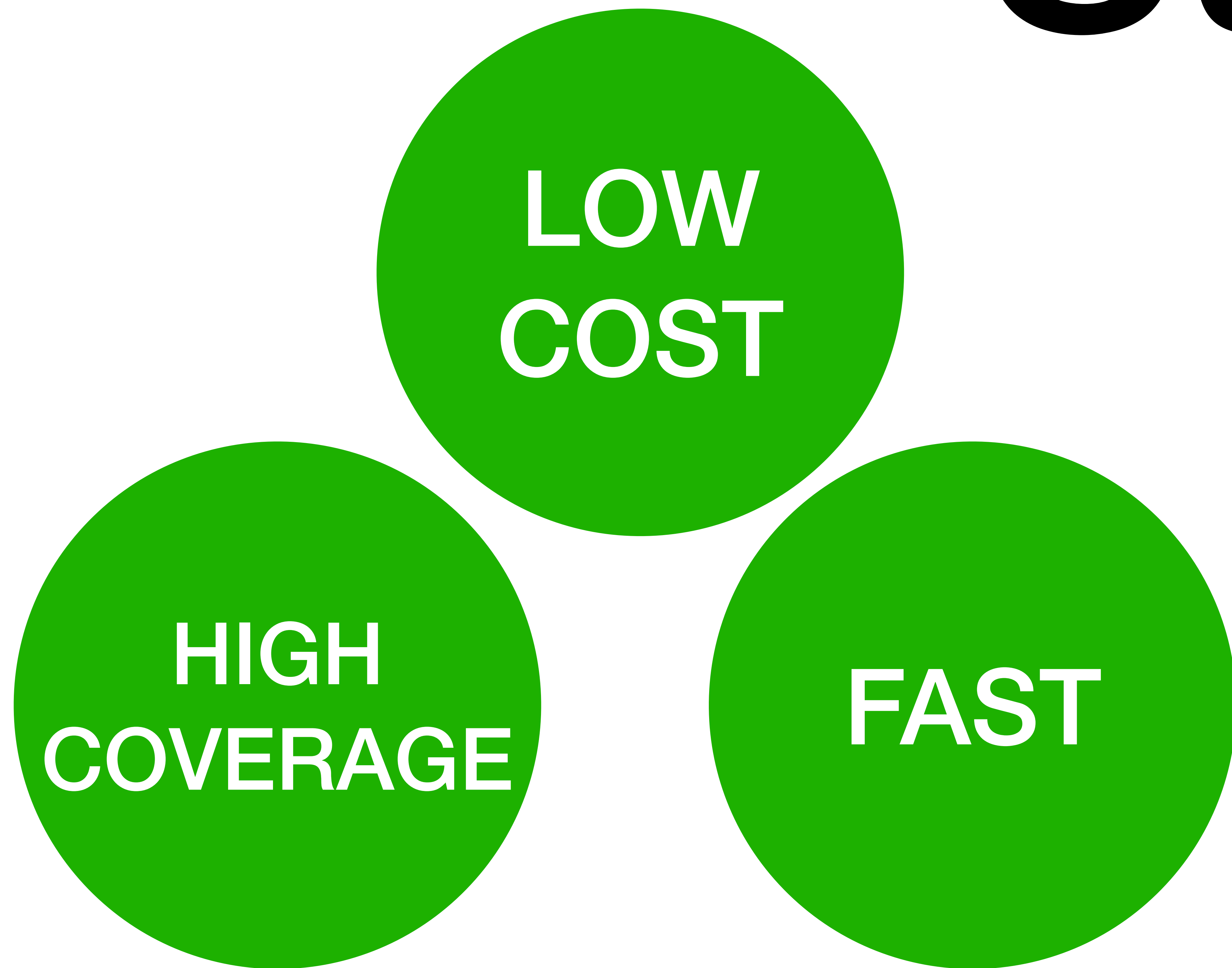
Verifying doubles: Contract tests (2)

```php
abstract class UserRepositoryContractTest extends TestCase
{
    abstract protected function createRepository(): UserRepository;

    public function testFindByEmailIsCaseInsensitive(): void
    {
        $repo = $this->createRepository();
        $user = new User(new UserId('123'), 'Alice@Example.com');

        $repo->save($user);

        $found = $repo->findByEmail('alice@example.com');
        $this->assertEquals($user, $found);
    }
}
```

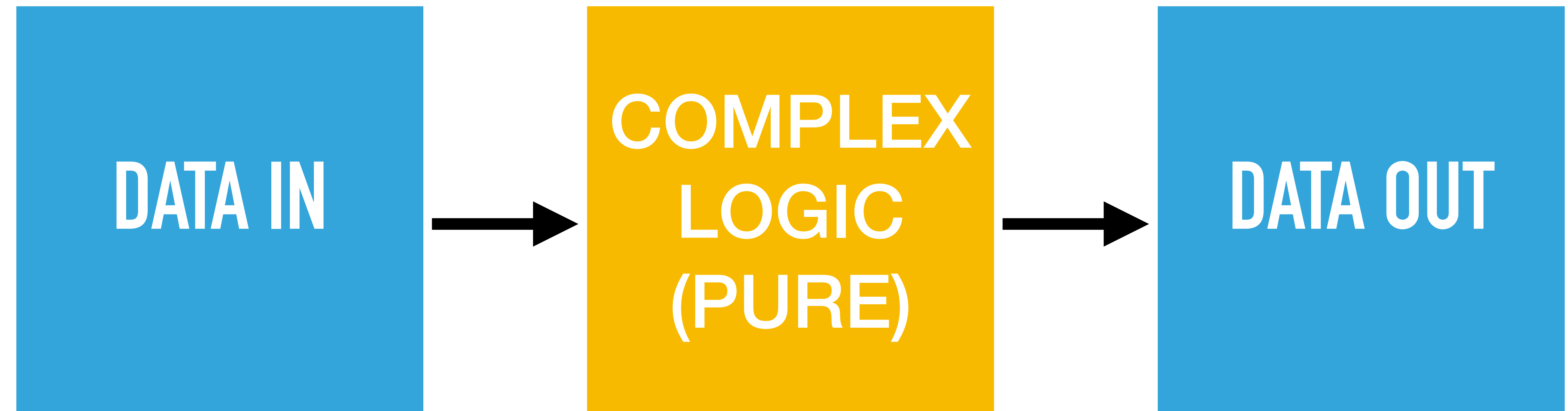
Verifying doubles: Contract tests (3)

```
class MySqlUserRepositoryTest extends UserRepositoryContractTest
{
    protected function createRepository(): UserRepository
    {
        return new MySqlUserRepository($this->getTestConnection());
    }
}

class InMemoryUserRepositoryTest extends UserRepositoryContractTest
{
    protected function createRepository(): UserRepository
    {
        return new InMemoryUserRepository();
    }
}
```

Summary



Pragmatic use of doubles



Decouple test from SUT

- Builder / Object mother patterns
- DRY in test setup
- Custom assertions and actions

Dave Liddament

@daveliddament
@daveliddament@phpc.social
@daveliddament.bsky.social
github.com/DaveLiddament
www.linkedin.com/in/daveliddament/

Me when I'm not coding!