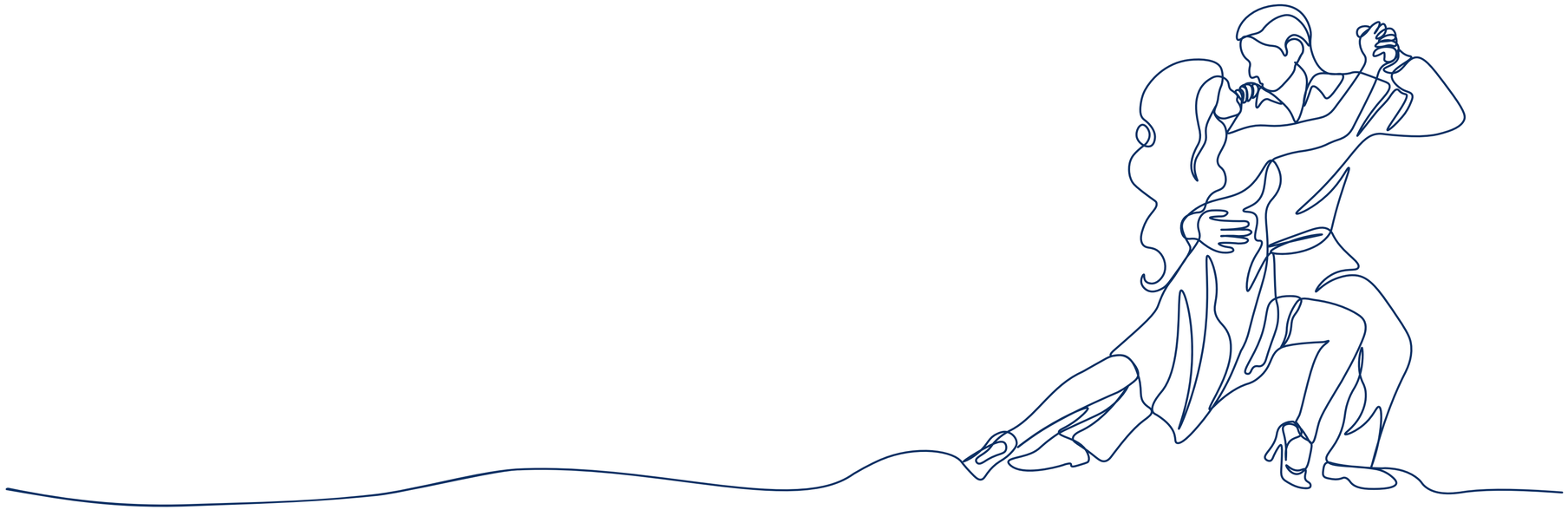


IT TAKES TWO TO TANGO

DESIGNING MODULE INTERACTIONS IN MODULITHIC SPRING APPLICATIONS

Oliver Drotbohm odrotbohm oliver.drotbohm@broadcom.com

github.com/odrotbohm

Oliver Drotbohm

odrotbohm · he/him

Frameworks & Architecture Engineering @ VMware, OpenSource enthusiast, all things Spring, Java, data, DDD, REST, software architecture, drums & music.

Edit profile

3.6k followers · **32** following

- VMware by Broadcom, Inc.
- Dresden, Germany
- 17:34 (UTC +02:00)
- info@odrotbohm.de
- www.odrotbohm.de
- @odrotbohm
- @odrotbohm@chaos.social
- odrotbohm
- in/odrotbohm

Pinned

Customize your pins

spring-projects/spring-modulith Public
Modular applications with Spring Boot
Java 737 116

xmolecules/jmolecules-integrations Public
Technology integration for jMolecules
Java 74 19

spring-playground Public
A collection of tiny helpers for building Spring applications
Java 100 11

lectures
Lecture scripts course at TU Dr
Java 7

Single sign-on to see contributions within the **pivotal** organization.

1,499 contributions in the last year

Aug Sep Oct Nov Dec Jan Feb Mar Apr
Mon
Wed
Fri

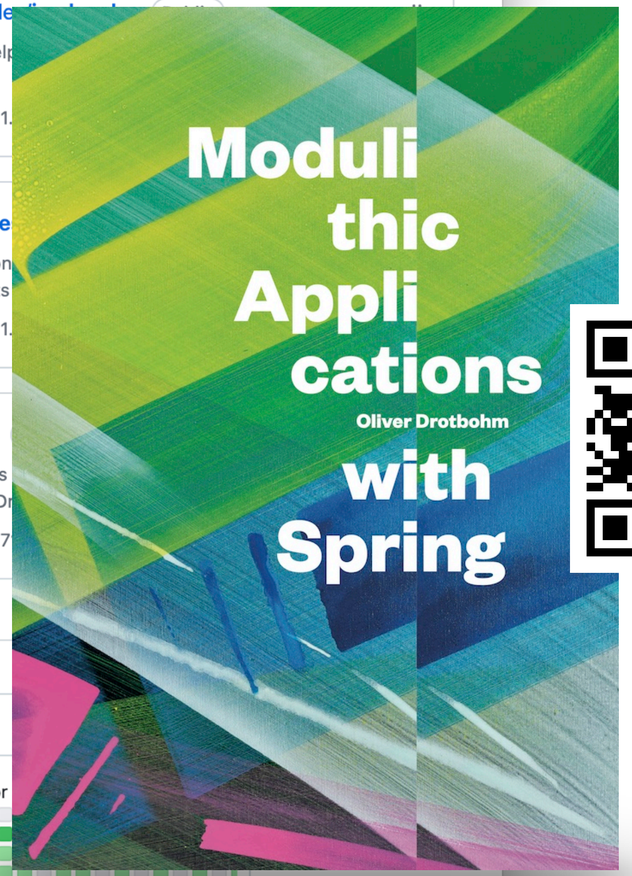
Learn how we count contributions
Less More

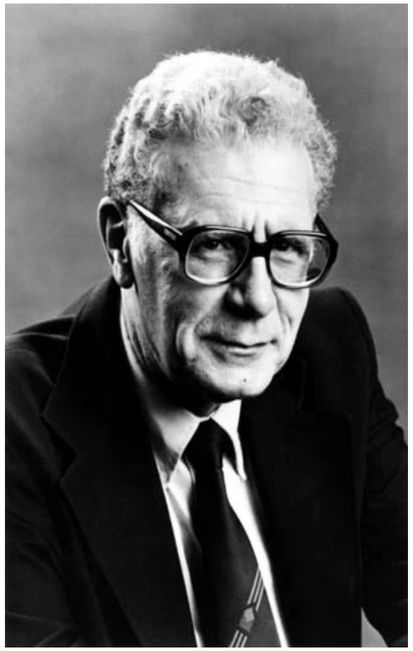
2020
2019
2018
2017

@spring-projects @xmolecules @spring-io More

Activity overview

1%
Code review

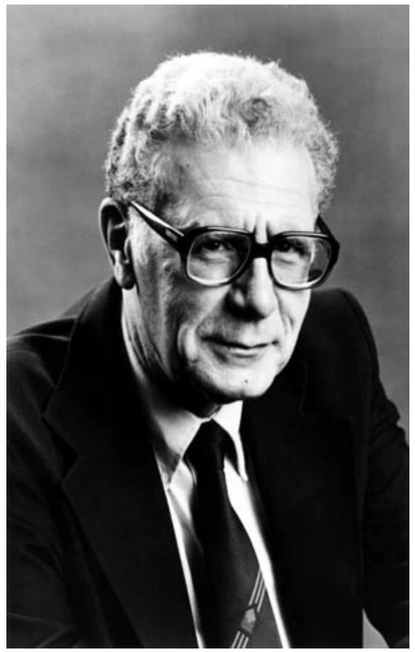




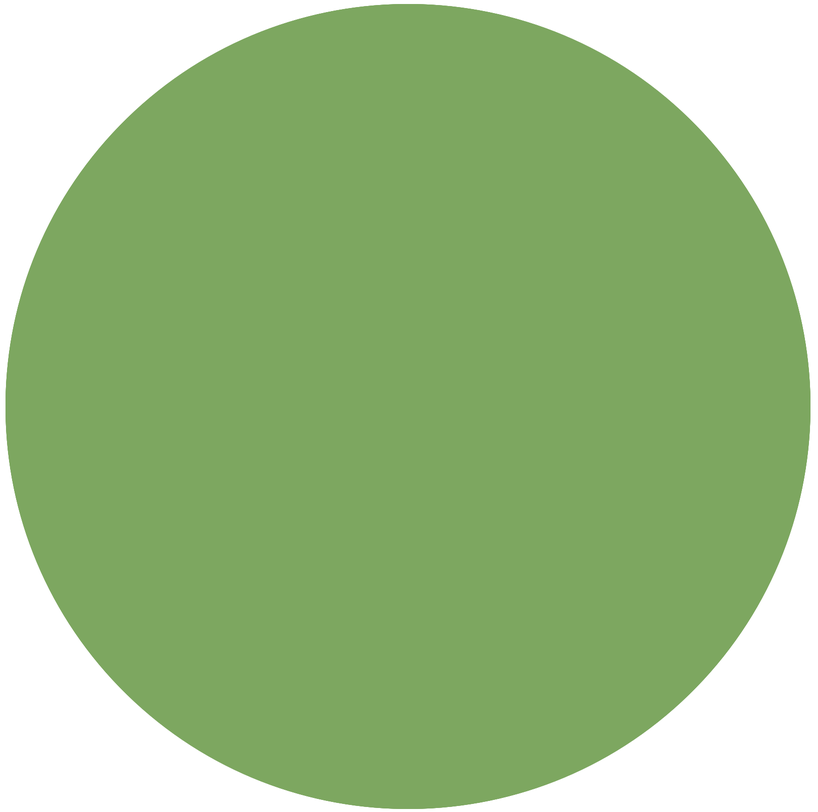
Systems Thinking, Learning and Problem Solving

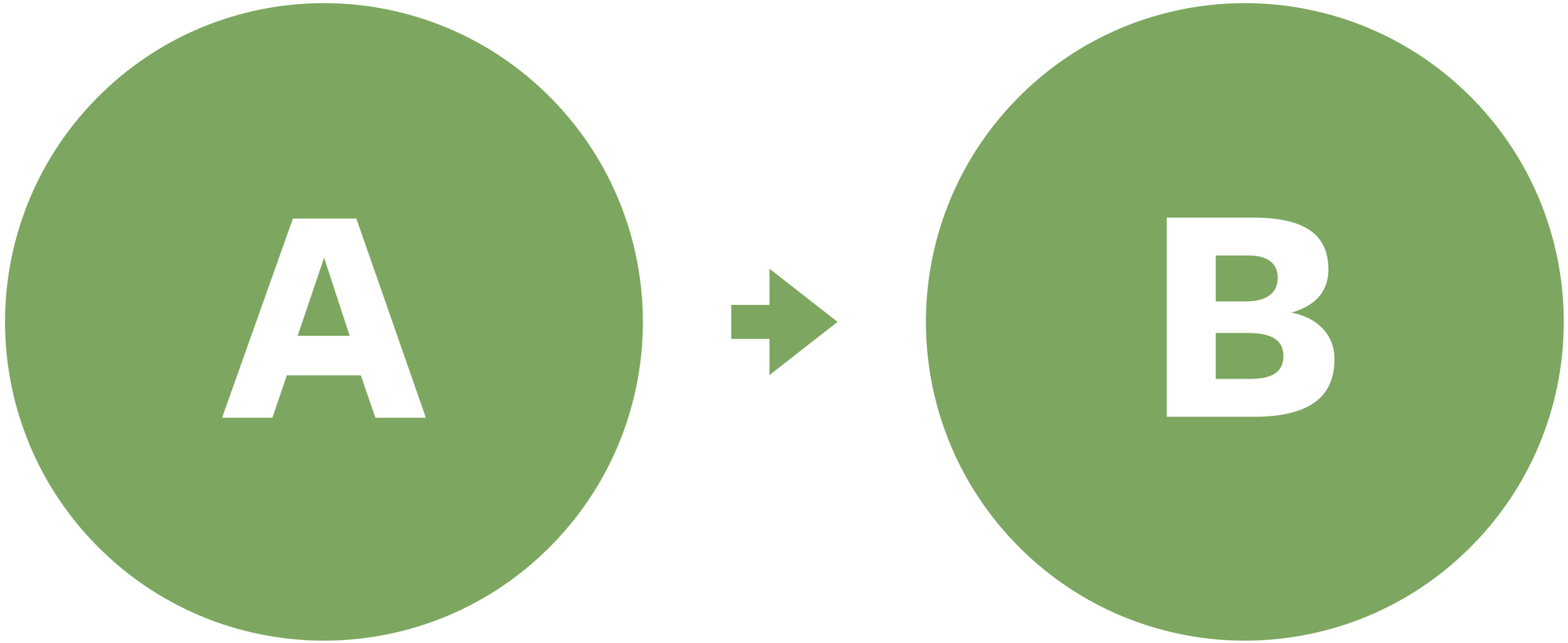
Long version | *Short version*

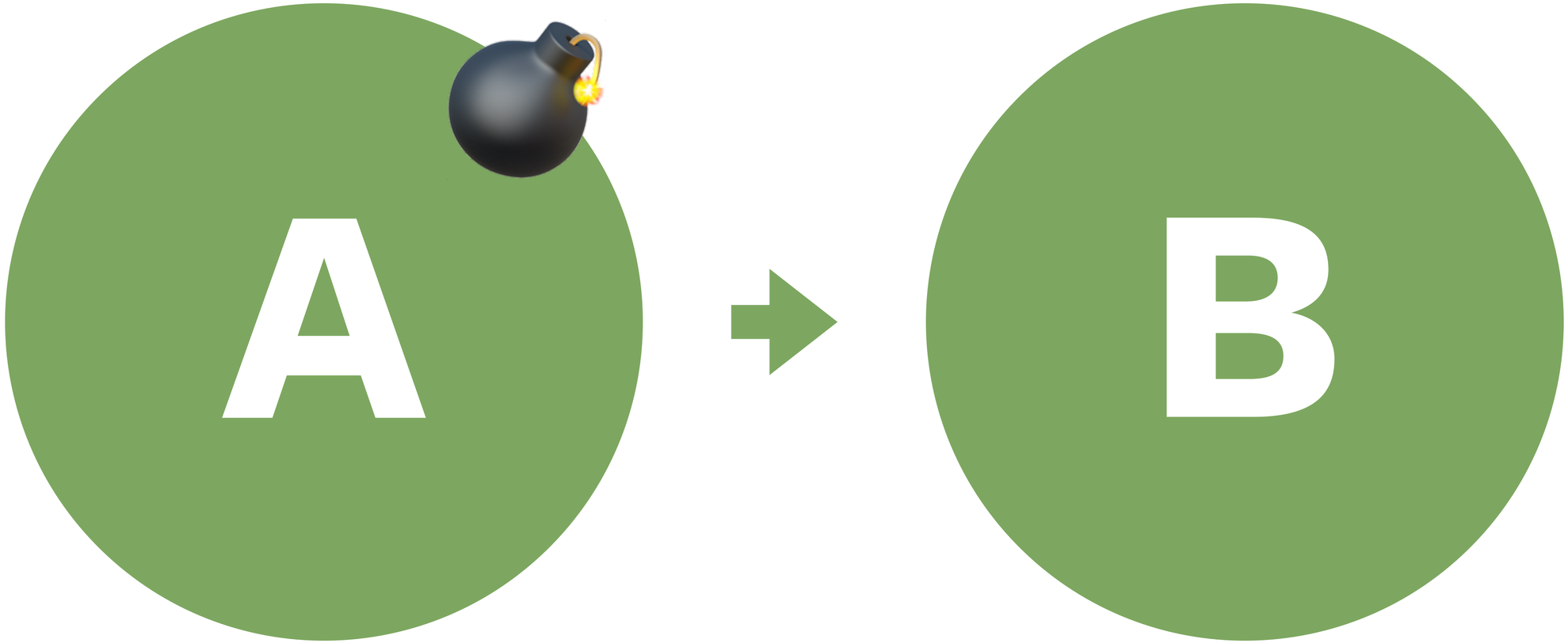
Russel L. Ackoff



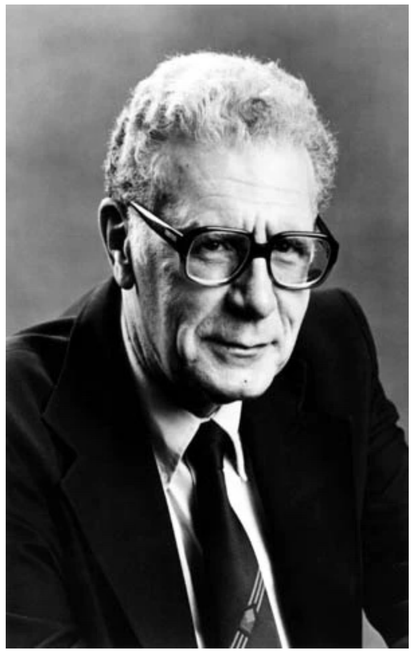
”A system is a whole that consists of *parts*, each of which can affect its behavior and properties. … Each part of the system is dependent on some other part.”




A
B

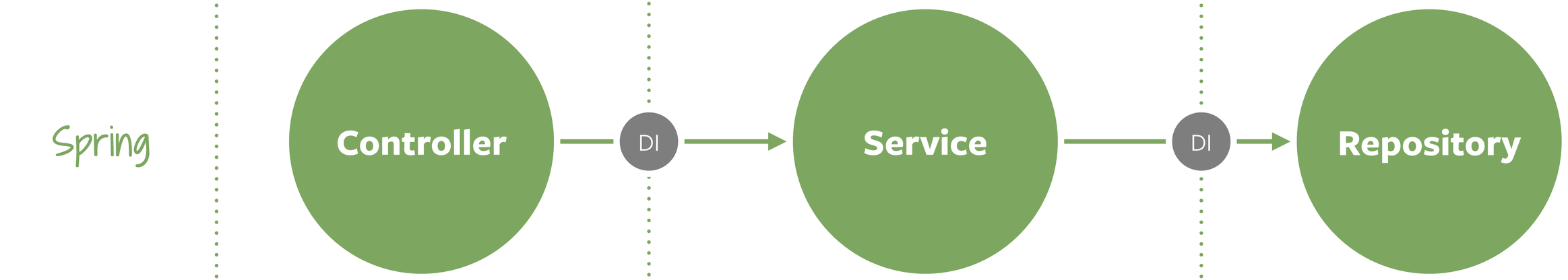


Risk of Change



”A system is never the sum of its *parts*, it’s the product of their *interactions*.”

How do we establish code interaction in a Spring application?



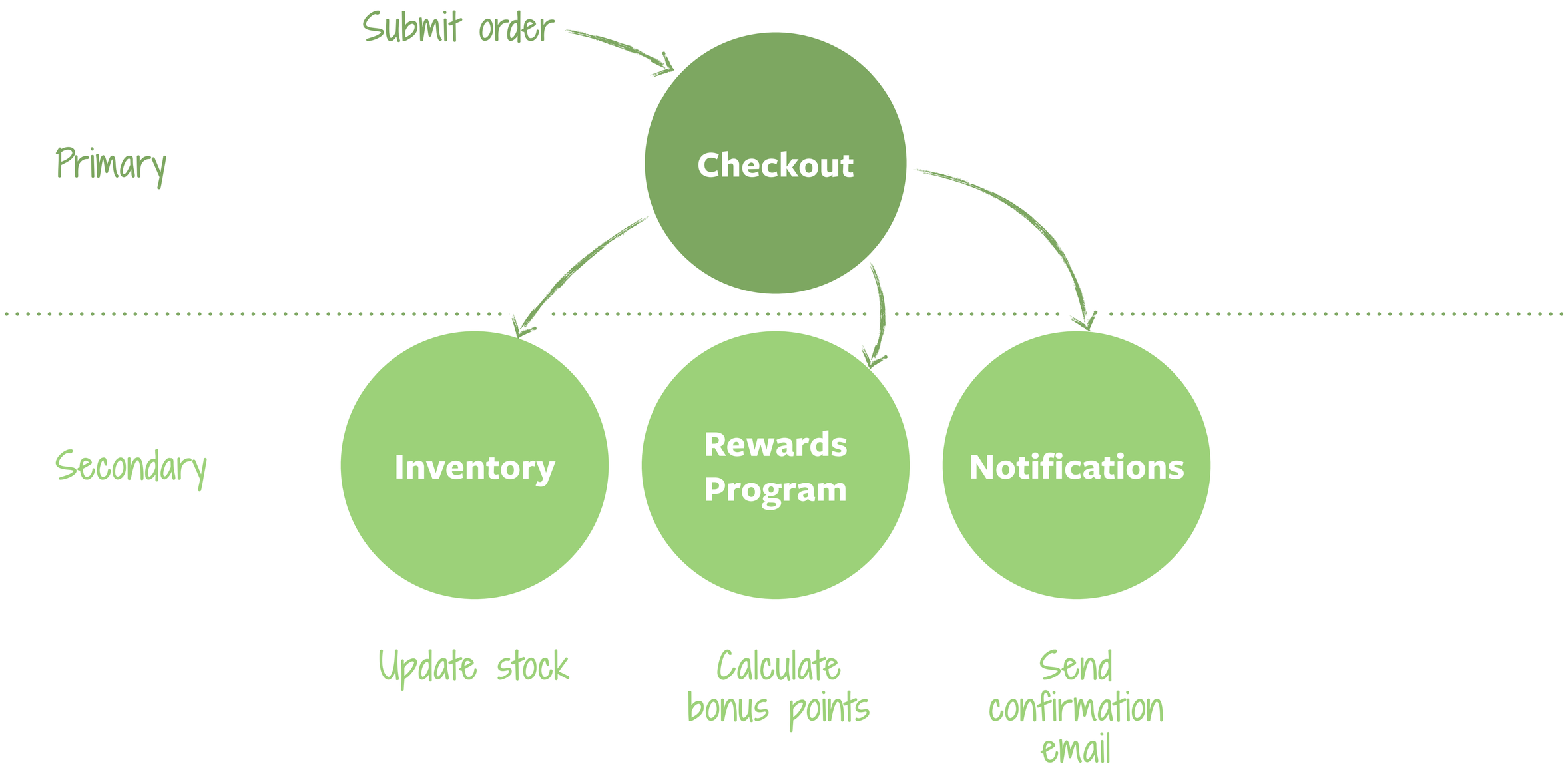
Layered	Presentation →	Business →	Persistence
Hexagonal	Driving Adapter →	Port ← Application → Port	← Driven Adapter
Onion	Infrastructure →	Application / Domain	← Infrastructure

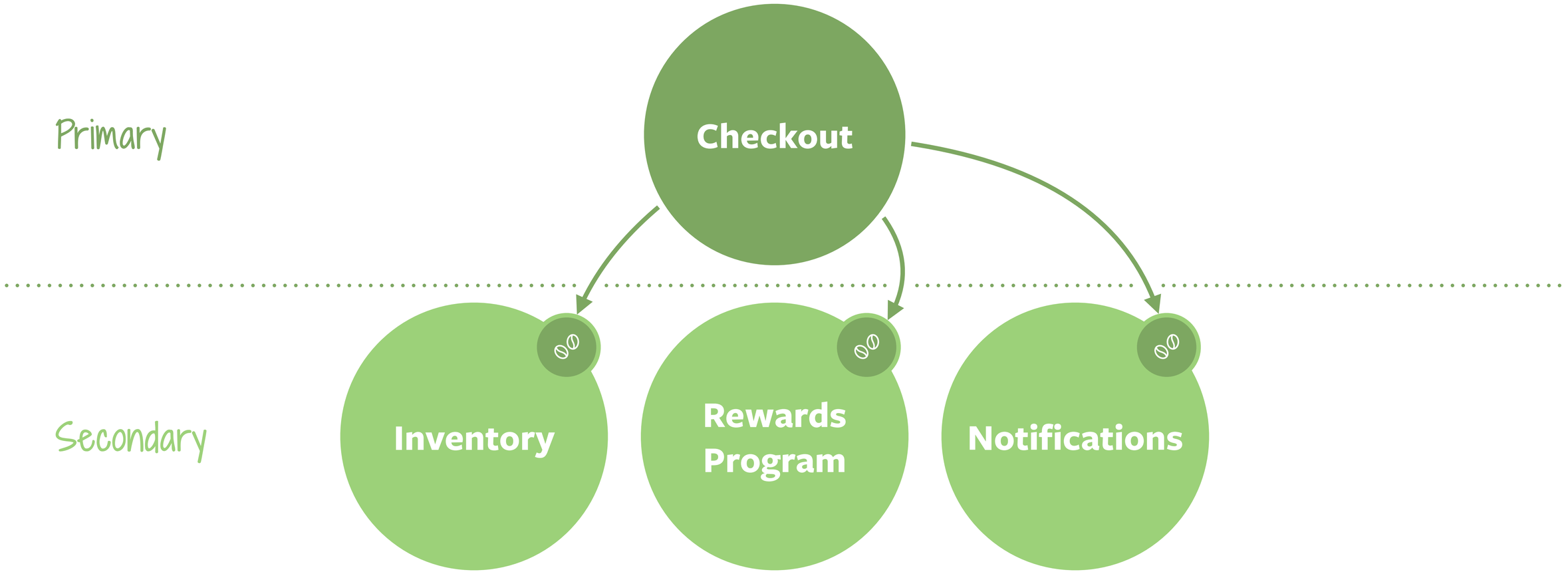
Technical Decomposition

VS.

Functional Decomposition

A software system's structure is essentially a *formalized bet* on *change patterns* you anticipate having to deal with in the future.


Submit order
Primary
Checkout
Secondary
Inventory
Rewards Program
Notifications
Update stock
Calculate bonus points
Send confirmation email



Integration via DI

```java
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;

  @Transactional
  void complete(Order order) {

    orders.save(order.complete());

  }
}
```

Primary

Internal, technical decomposition

State transition

```java
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;



  @Transactional
  void complete(Order order) {

    orders.save(order.complete());



  }
}
```

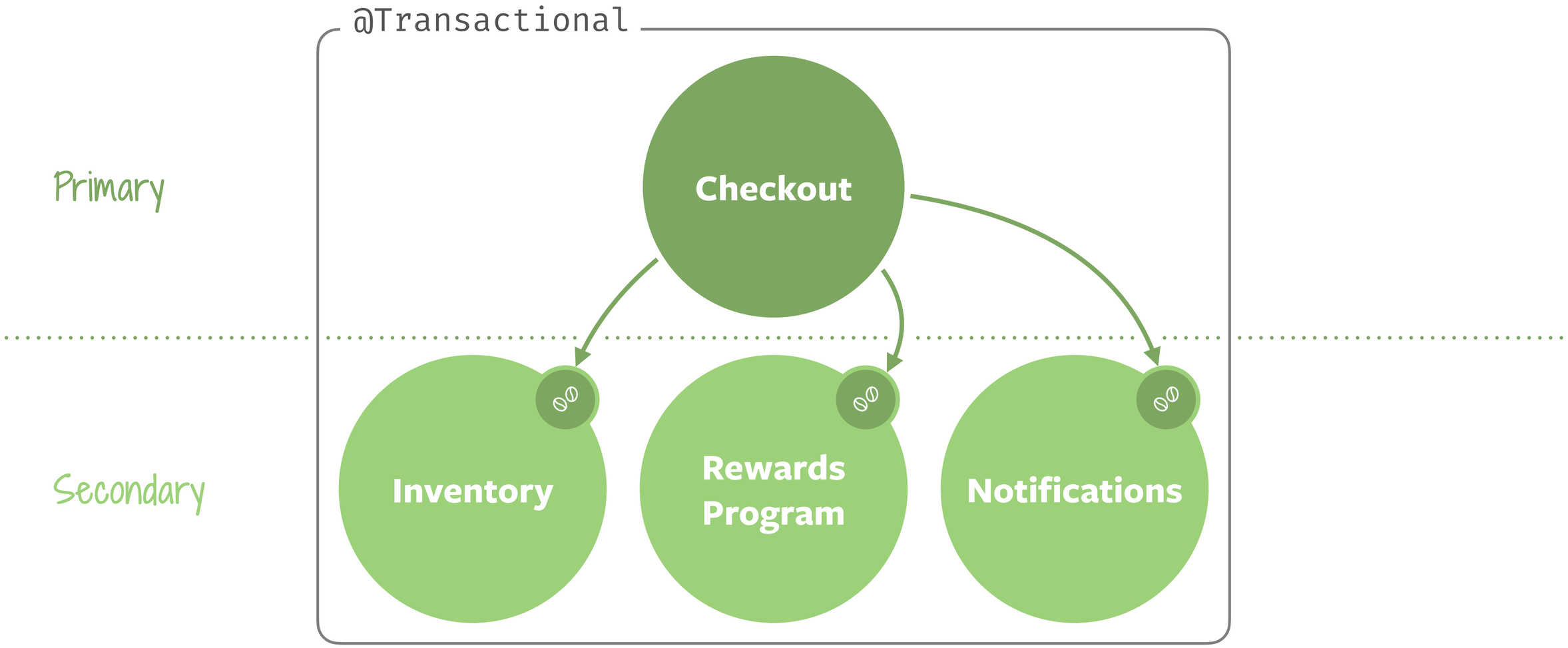
Primary

Secondary

Required at startup / in tests

```java
@Service
class Inventory { }

@Service
class RewardsProgram { }

@Service
class Notifications { }
```

Interaction expands transaction



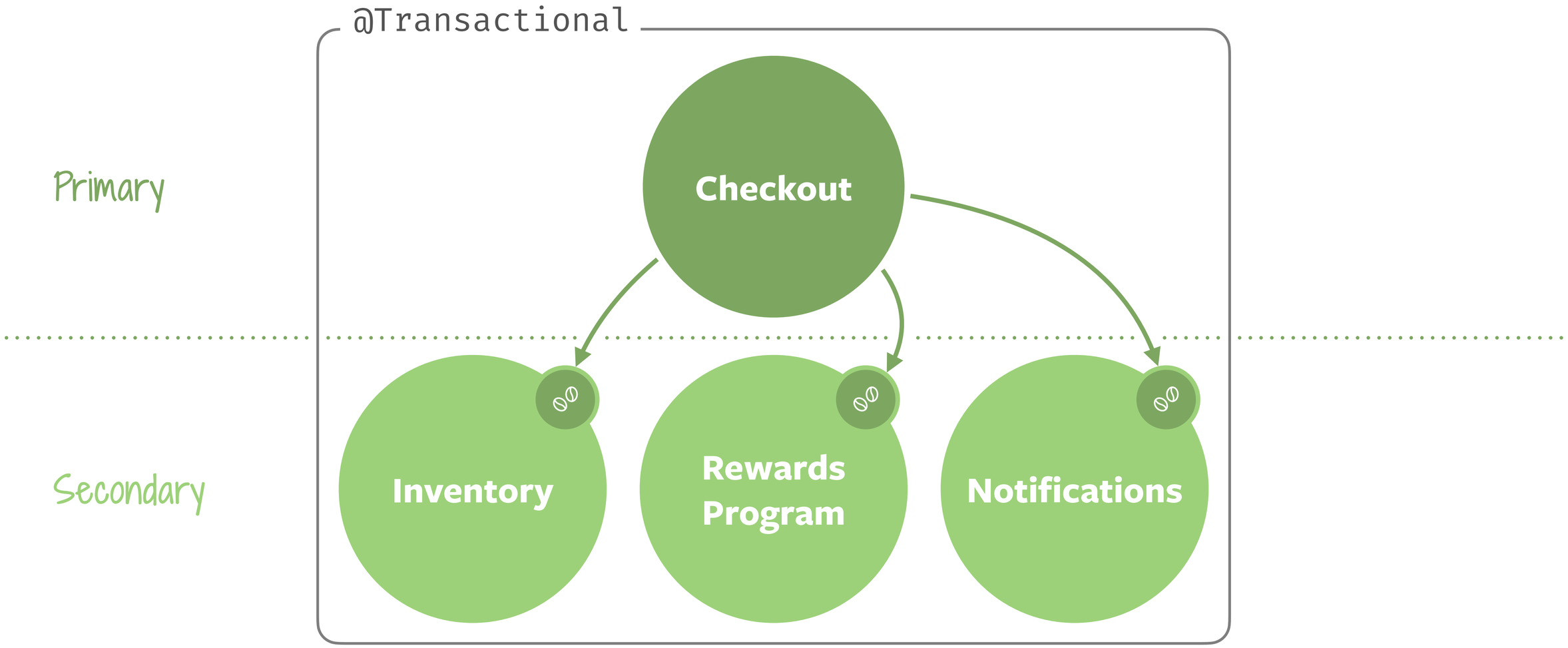
Scope of Consistency

```java
@SpringBootTest
class CheckoutTests {

  @Autowired Checkout checkout;

  @MockBean Inventory inventory;
  // Other mocks

  @Test
  void completesOrder() {

    var order = new Order(…);

    checkout.complete(order);

    verify(inventory).updateStock(…);
  }
}
```

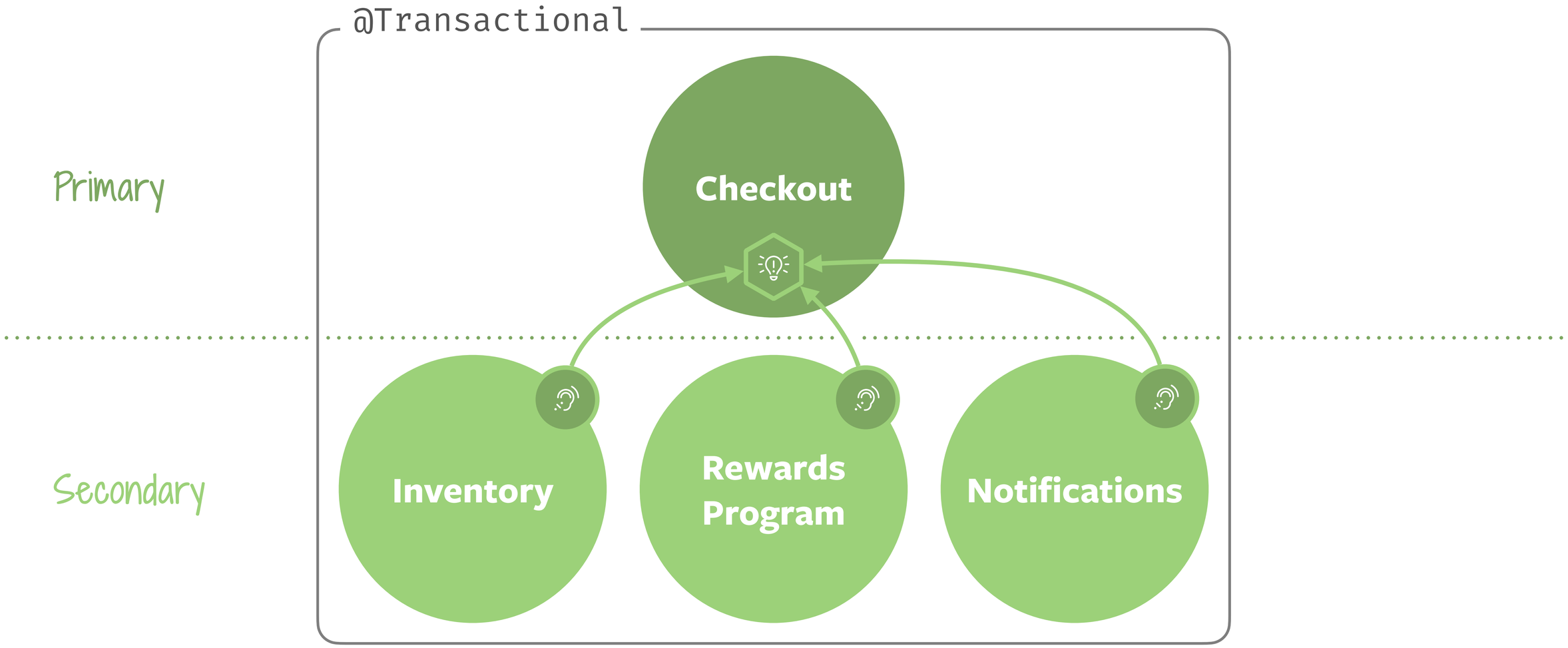
Collaborators are mocked

Given / When / Then

Testing focussed on orchestration



Scope of Consistency



Integration via Events

```java
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;
  private final ApplicationEventPublisher events;

  @Transactional
  void complete(Order order) {

    orders.save(order.complete());

    events.publishEvent(
        OrderCompleted.of(order.getId()));
  }
}
```

Primary

Secondary

Invokes

Only internal dependencies

```java
@Service
class Inventory {

  @EventListener
  void on(OrderCompleted event) {}
}

…
```

```java
@RecordApplicationEvents
/* or */
@ApplicationModuleTest
class CheckoutTests {

  private final Checkout checkout;

  @Test
  void completesOrder(AssertableApplicationEvents events) {

    var order = new Order(…);

    checkout.complete(order);

    assertThat(events)
      .contains(OrderCompleted.class)
      .matching(OrderCompleted::id, order.getId());
  }
}
```

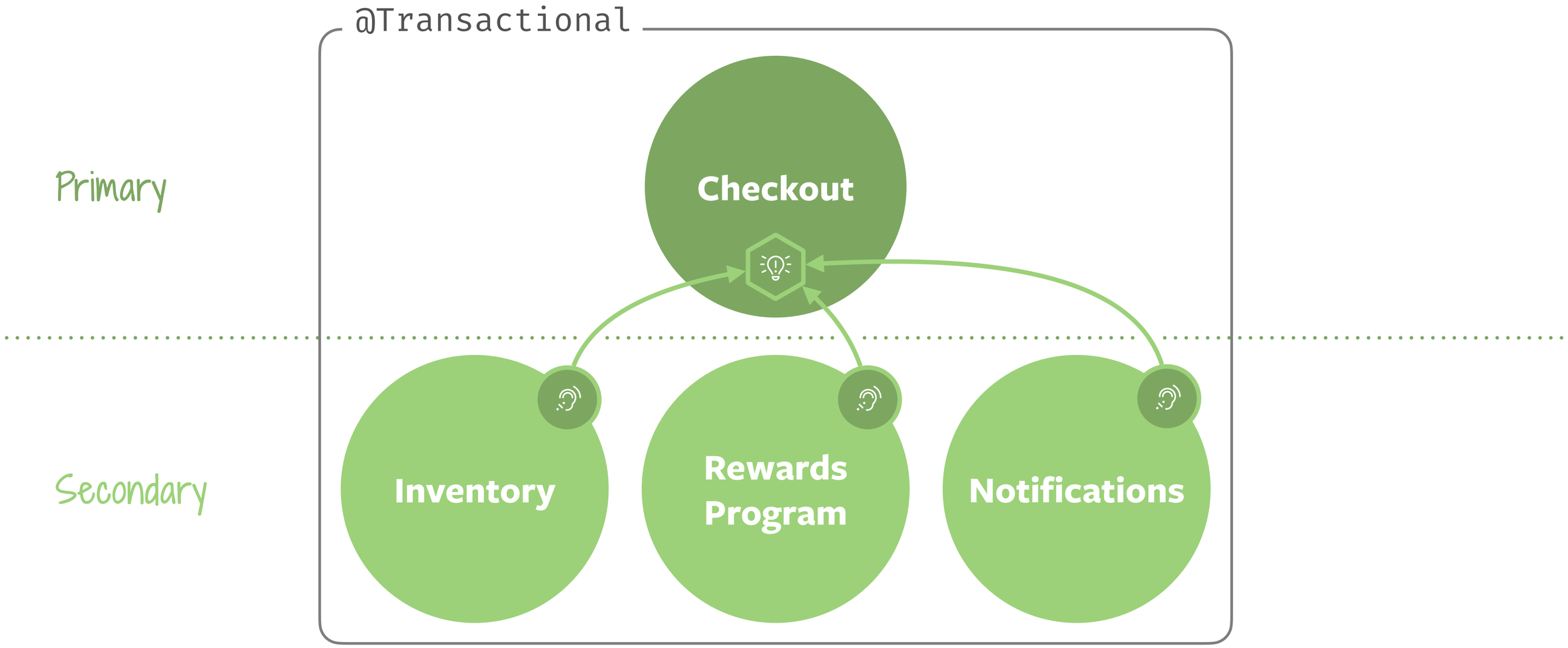
Spring Framework

Spring Modulith

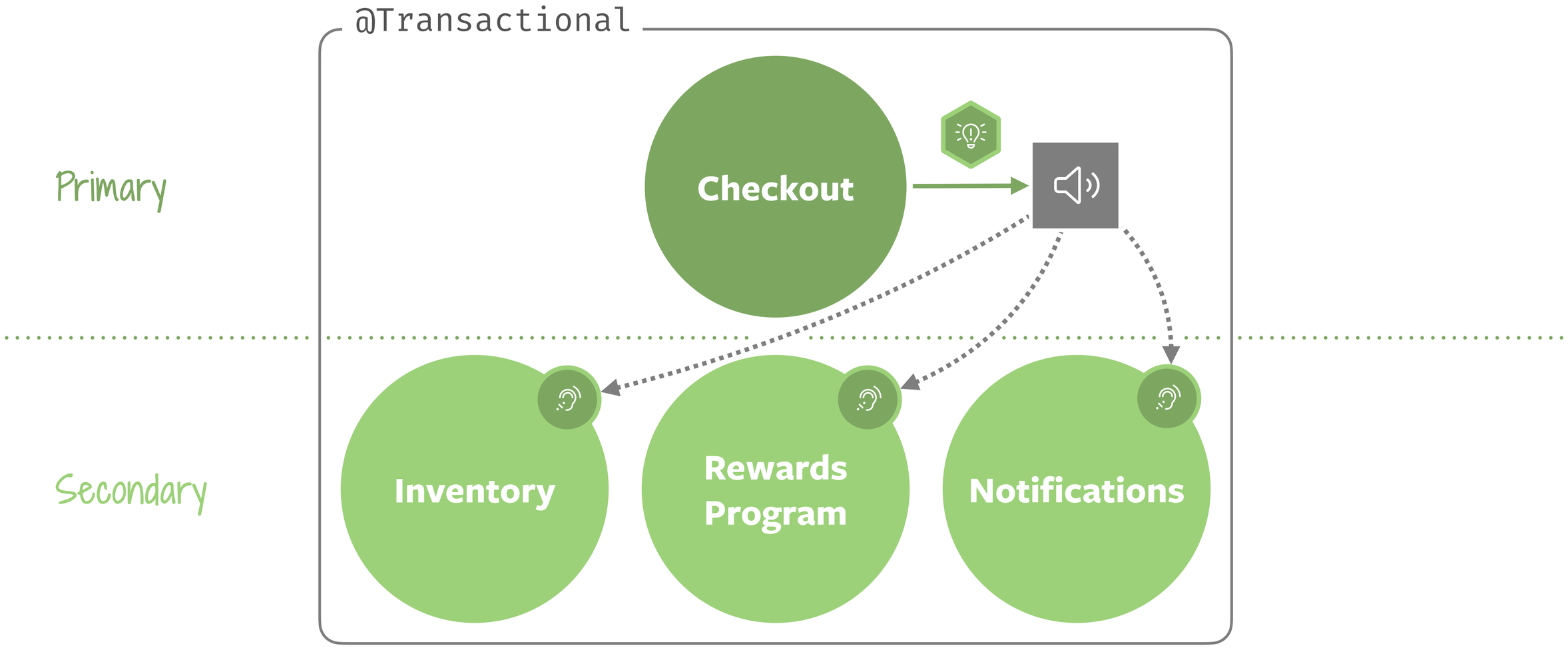
Records events published during test execution

Given / When / Then

Testing focussed on signal



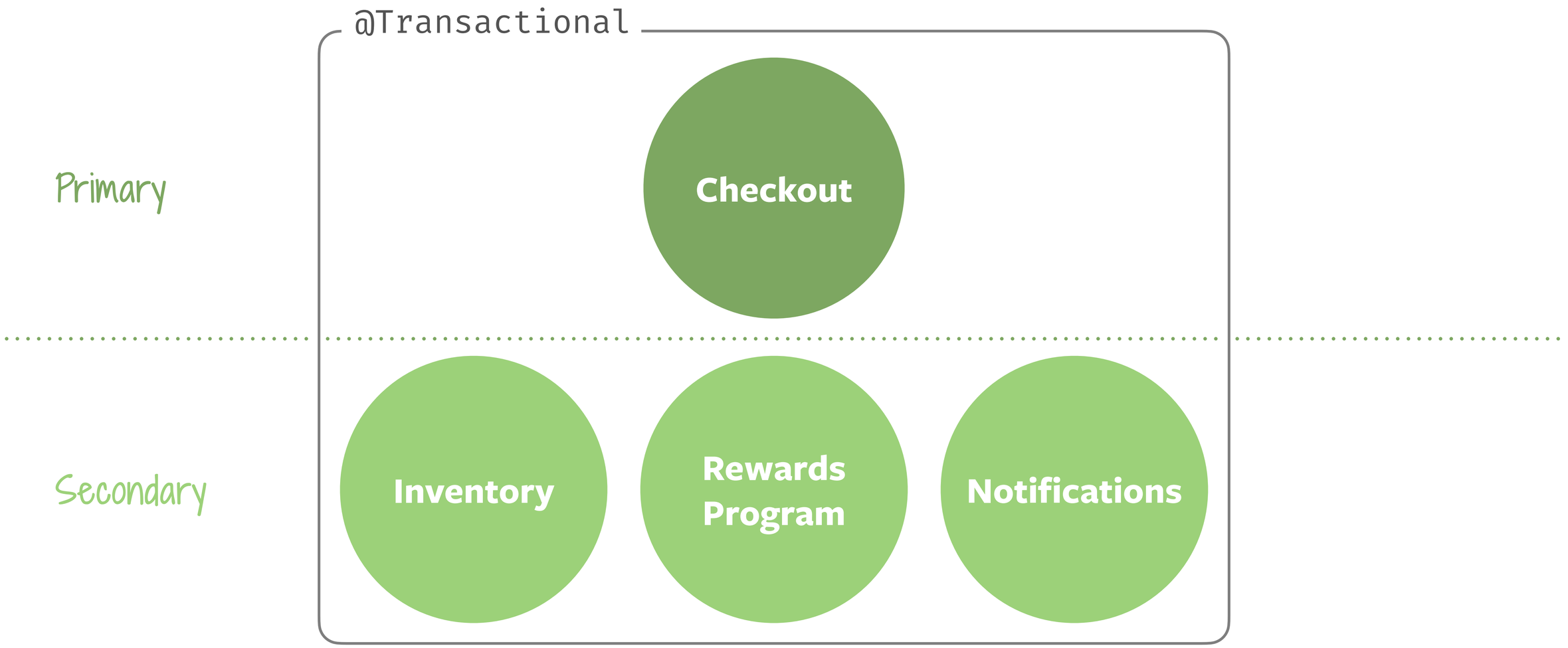
Integration via Events

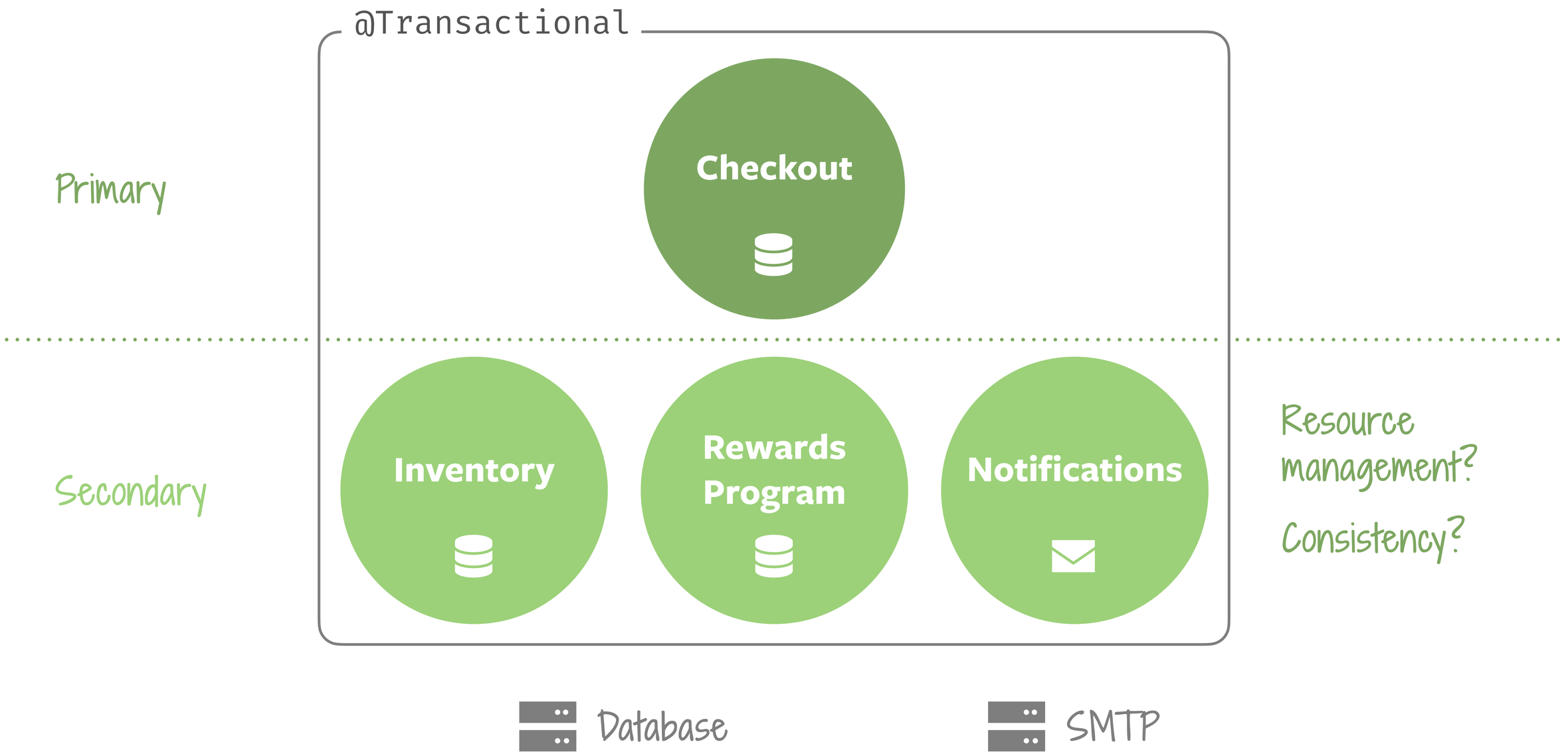


Invocation Flow for Events

Summary

- Change in structural arrangement
- Change in approach to testing
- ***No*** change to consistency arrangement


@Transactional
Primary
Checkout
Secondary
Inventory
Rewards Program
Notifications


@Transactional
Primary
Checkout
Secondary
Inventory
Rewards Program
Notifications
Resource management?
Consistency?
Database
SMTP

```java
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;
  private final ApplicationEventPublisher events;

  @Transactional
  void complete(Order order) {

    orders.save(order.complete());

    events.publishEvent(
        OrderCompleted.of(order.getId()));
  }
}
```

Primary

Secondary

```java
@Service
class Notifications {

  @EventListener
  void on(OrderCompleted event) {
    // Interact with SMTP server
  }
}
```

What if this takes long?

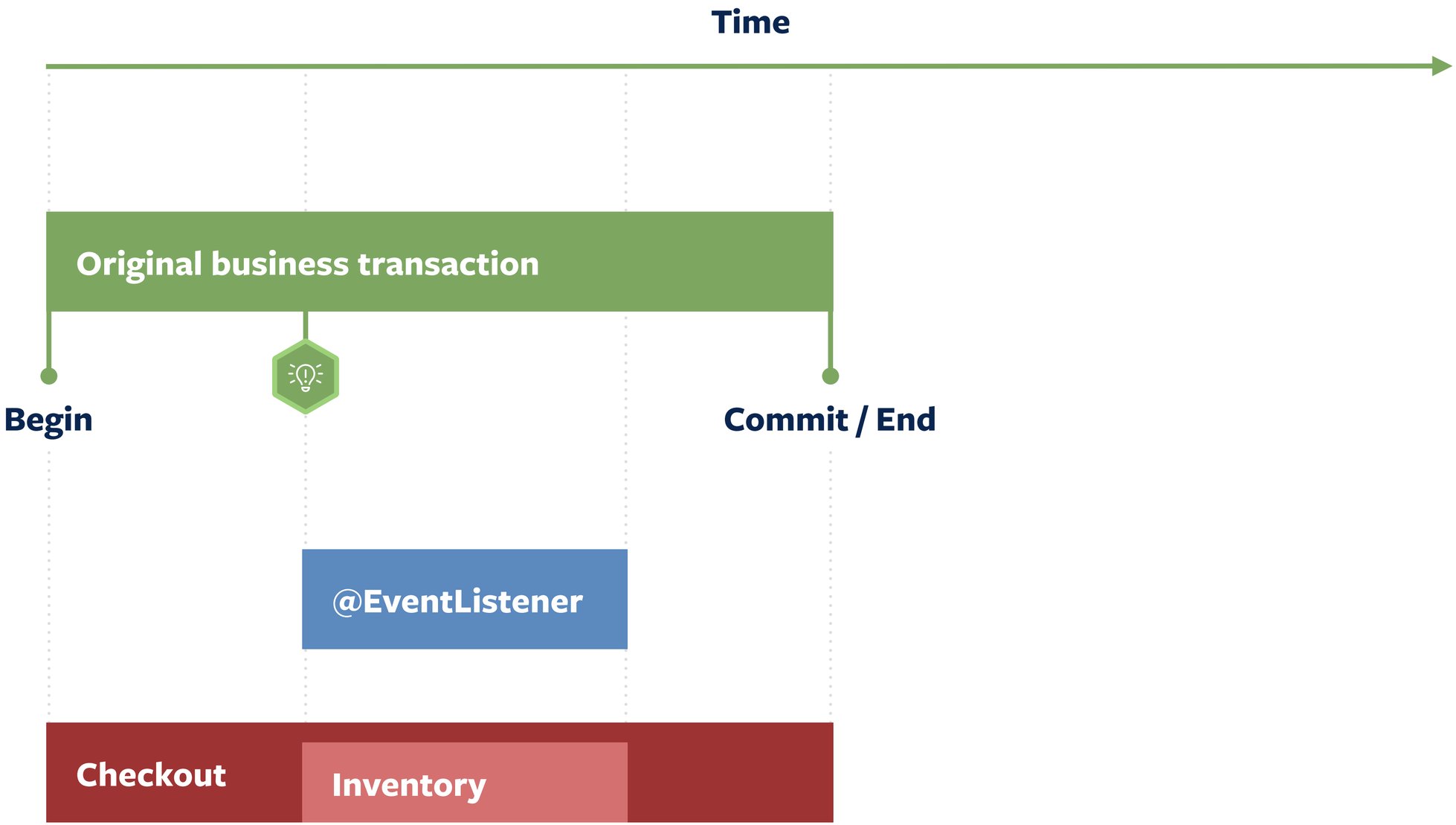
```java
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;
  private final ApplicationEventPublisher events;

  @Transactional
  void complete(Order order) {

    orders.save(order.complete());

    events.publishEvent(
        OrderCompleted.of(order.getId()));
  }
}
```

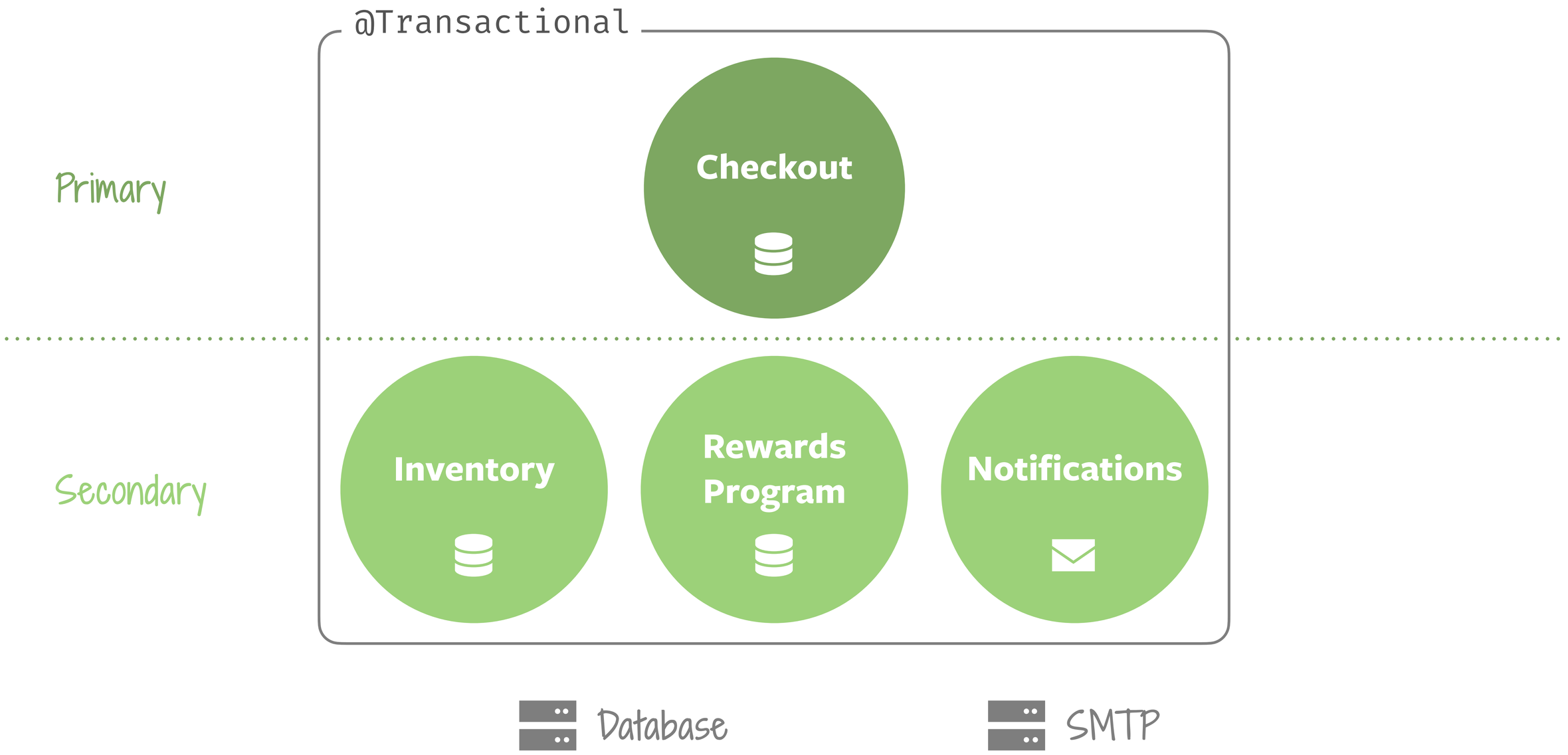
Primary

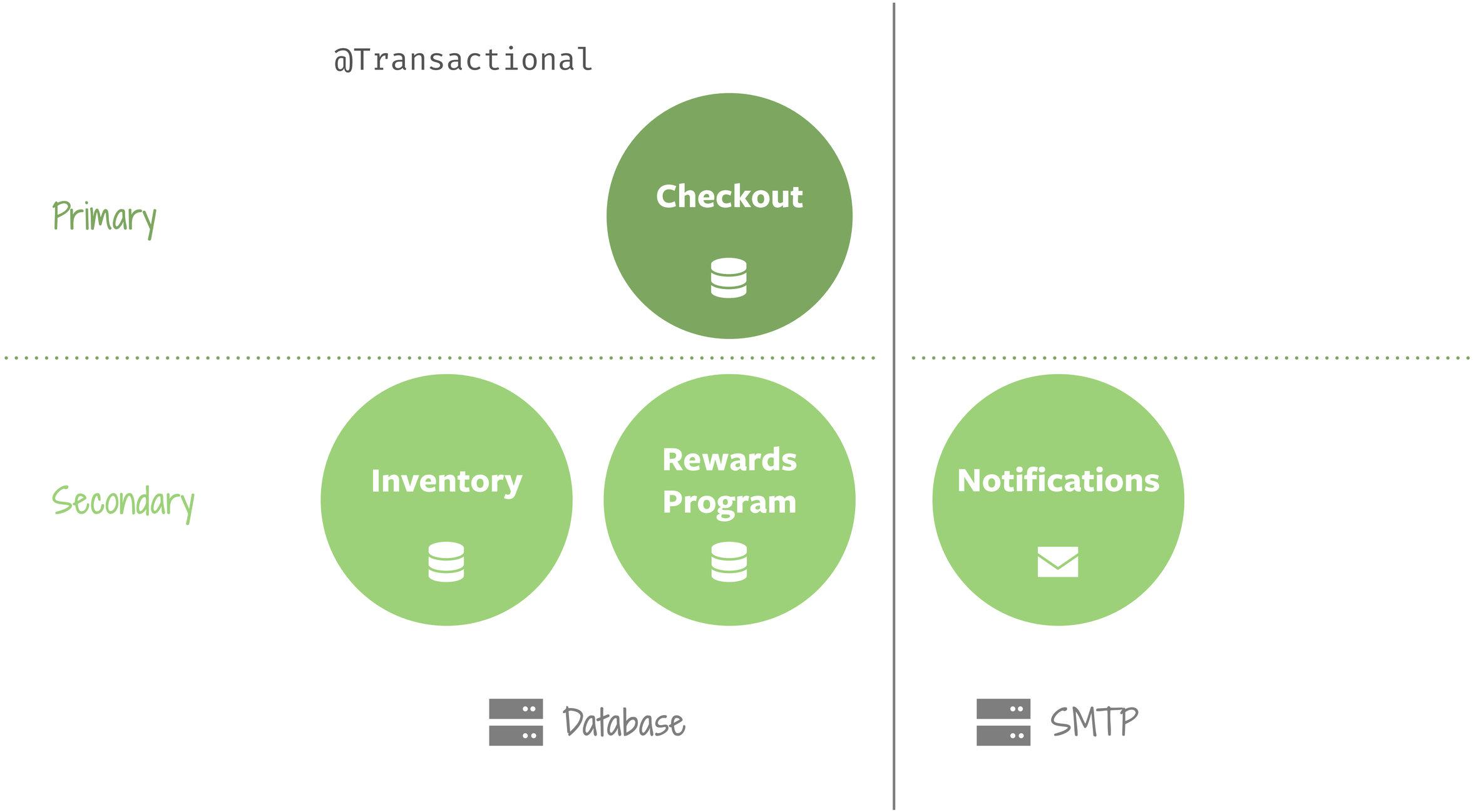
Secondary

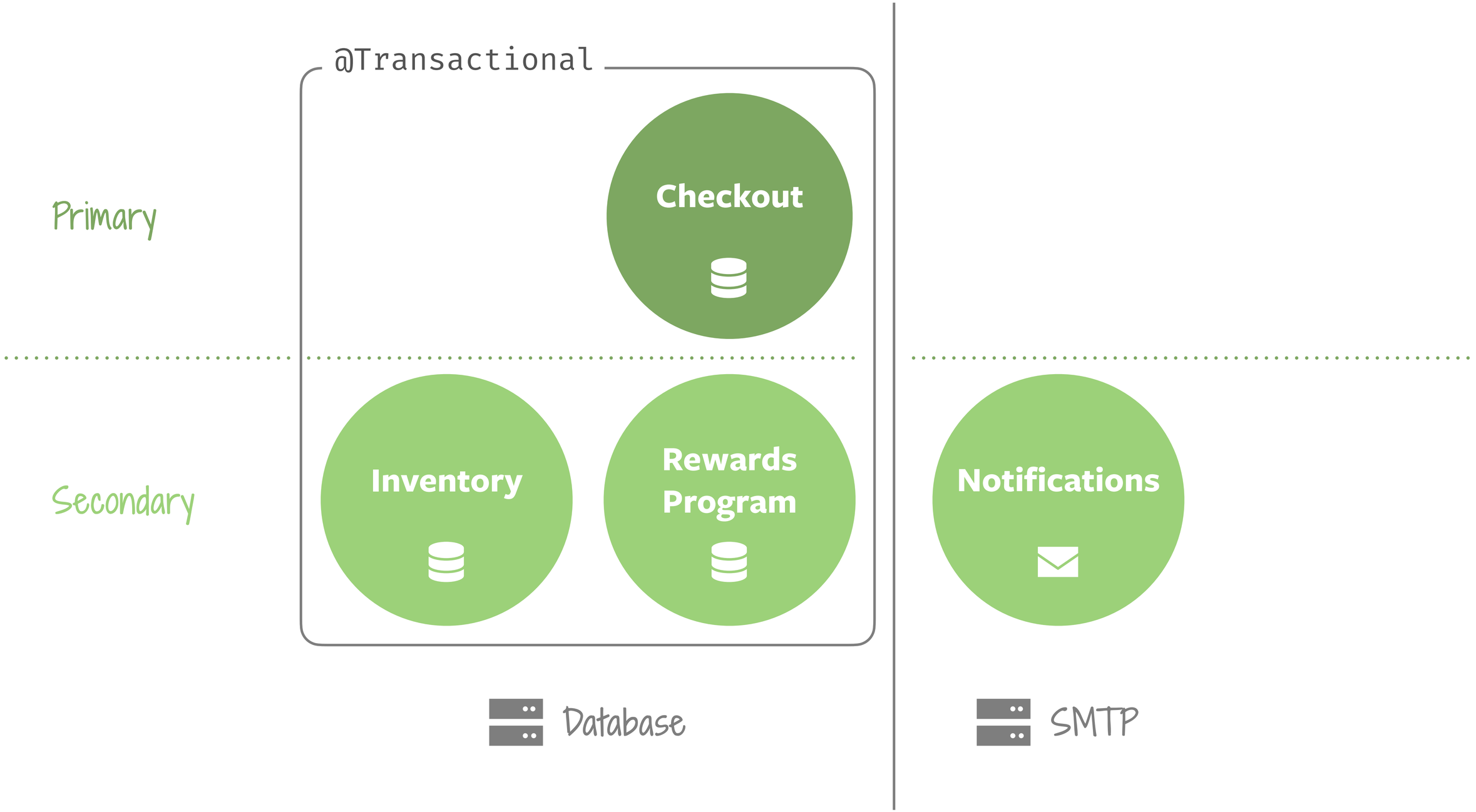
```java
@Service
class Notifications {

  @EventListener
  void on(OrderCompleted event) {
    // Interact with SMTP server
  }
}
```

What if this succeeds but this rolls back?


Time
Original business transaction
Begin
Commit / End
@EventListener
Checkout
Inventory


@Transactional
Primary
Checkout
Secondary
Inventory
Rewards Program
Notifications
Database
SMTP


@Transactional
Primary
Checkout
Secondary
Inventory
Rewards Program
Notifications
Database
SMTP


@Transactional
Primary
Checkout
Secondary
Inventory
Rewards Program
Notifications
Database
SMTP

Primary

Secondary

```java
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;
  private final ApplicationEventPublisher events;

  @Transactional
  void complete(Order order) {

    orders.save(order.complete());

    events.publishEvent(
        OrderCompleted.of(order.getId()));
  }
}
```

```java
@Service
class Notifications {

  @EventListener
  void on(OrderCompleted event) {
    // Interact with SMTP server
  }
}
```

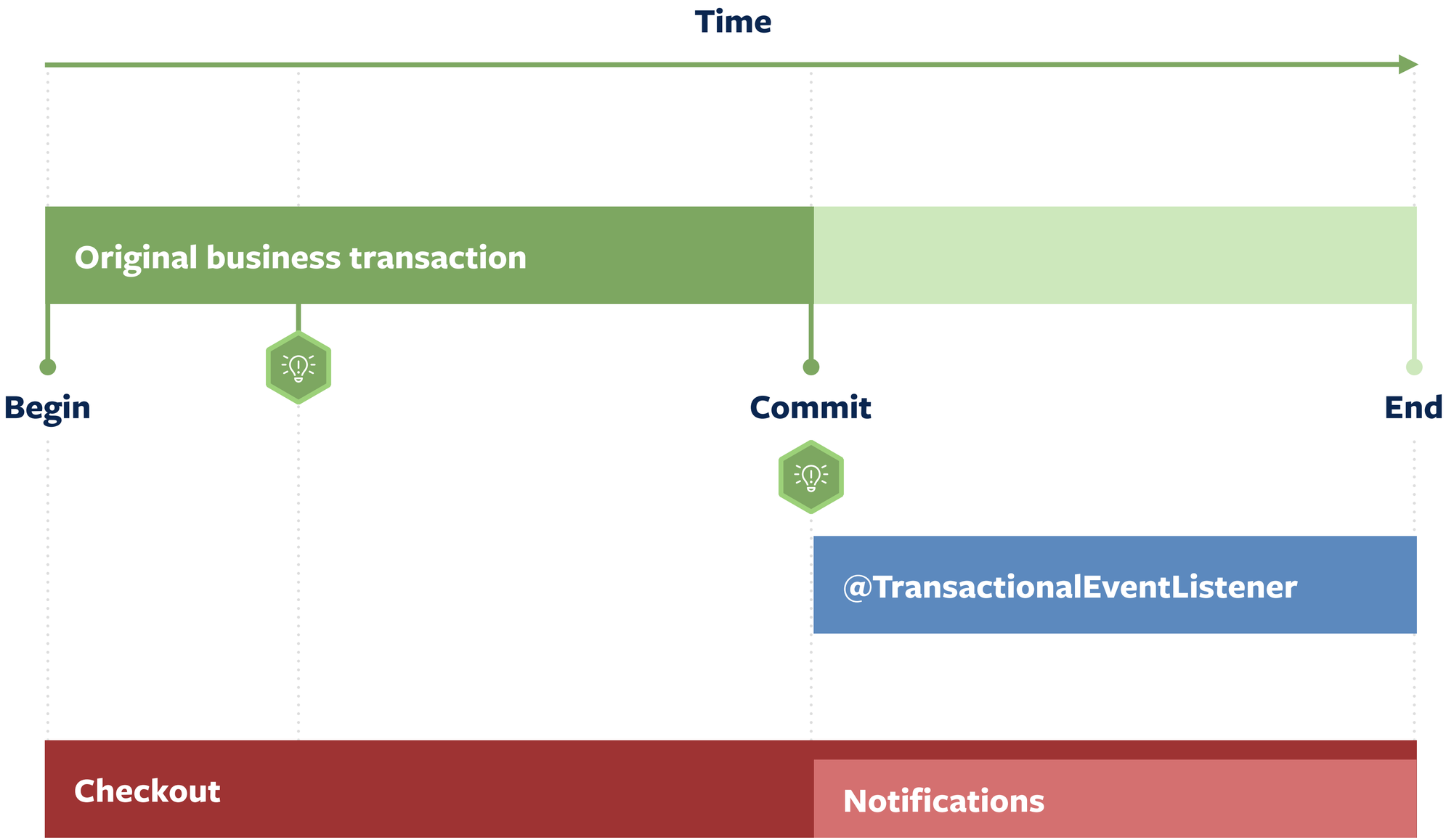
Triggered on transaction commit

Primary

Secondary

```
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;
  private final ApplicationEventPublisher events;

  @Transactional
  void complete(Order order) {

    orders.save(order.complete());

    events.publishEvent(
        OrderCompleted.of(order.getId()));
  }
}
```

```
@Service
class Notifications {

  @TransactionalEventListener
  void on(OrderCompleted event) {
    // Interact with SMTP server
  }
}
```

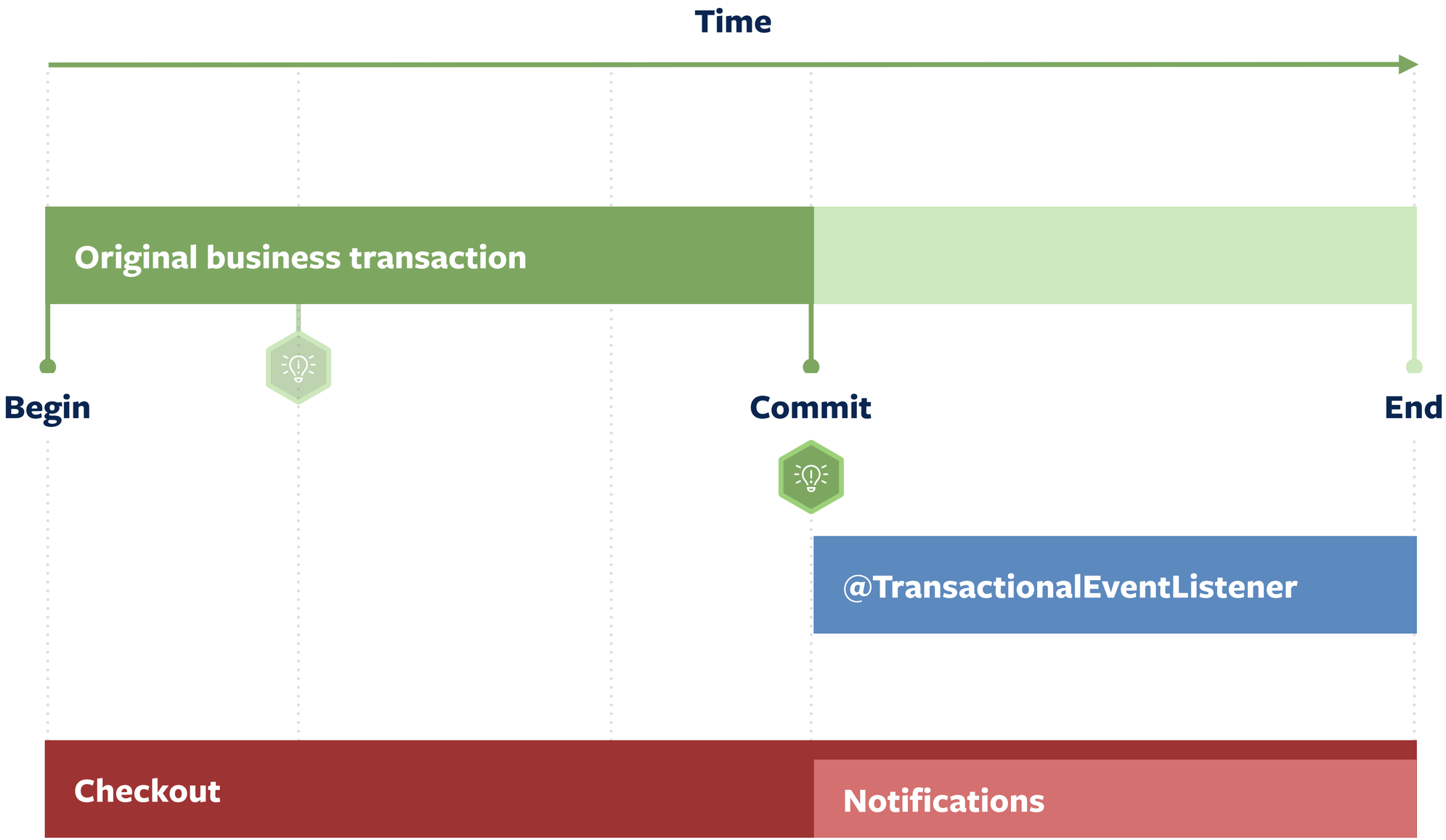

Time
Original business transaction
Begin
Commit
End
@TransactionalEventListener
Checkout
Notifications

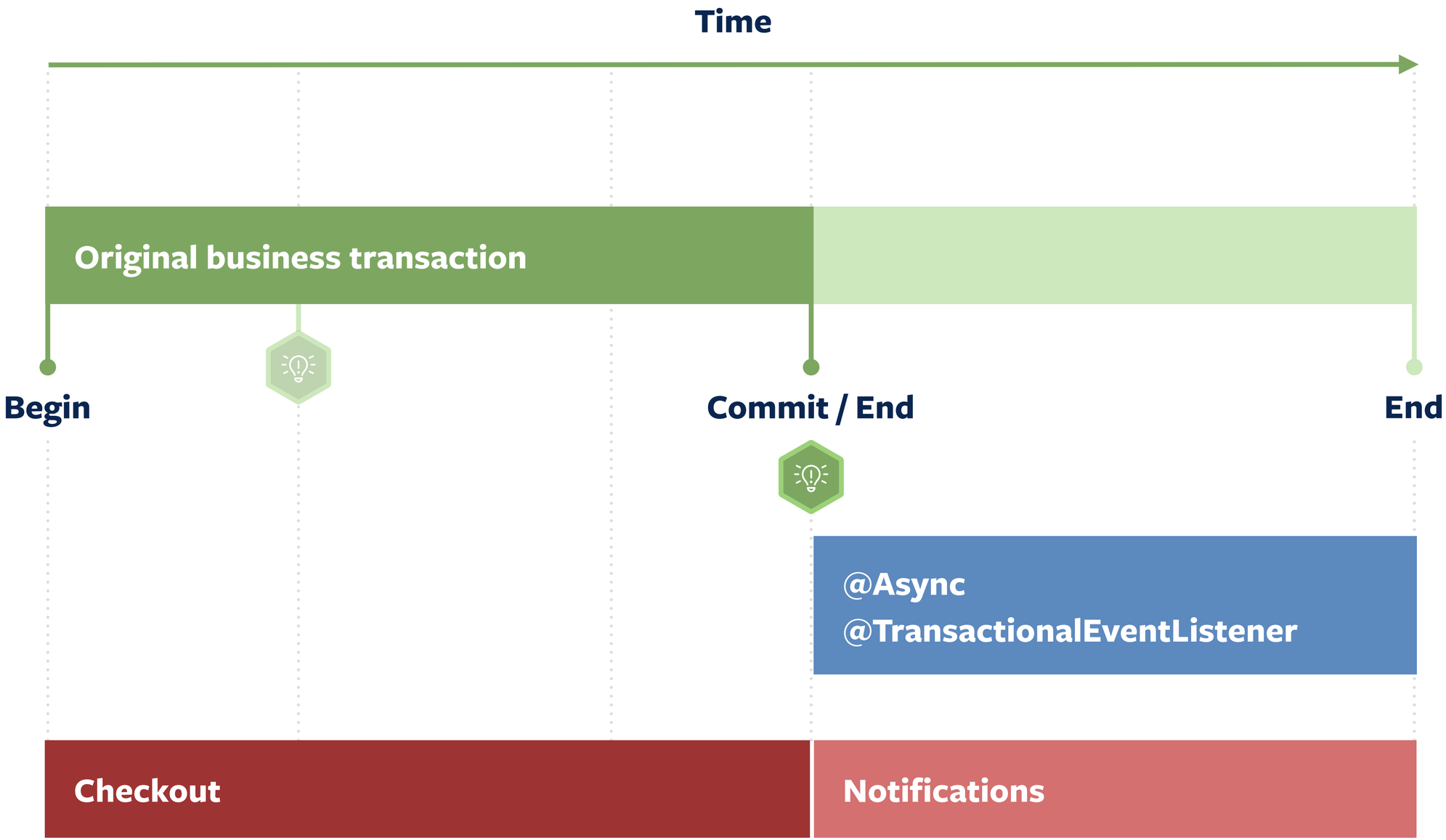
```java
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;
  private final ApplicationEventPublisher events;

  @Transactional
  void complete(Order order) {

    orders.save(order.complete());

    events.publishEvent(
        OrderCompleted.of(order.getId()));
  }
}
```

```java
@Service
class Notifications {

  @TransactionalEventListener
  void on(OrderCompleted event) {
    // Interact with SMTP server
  }
}
```

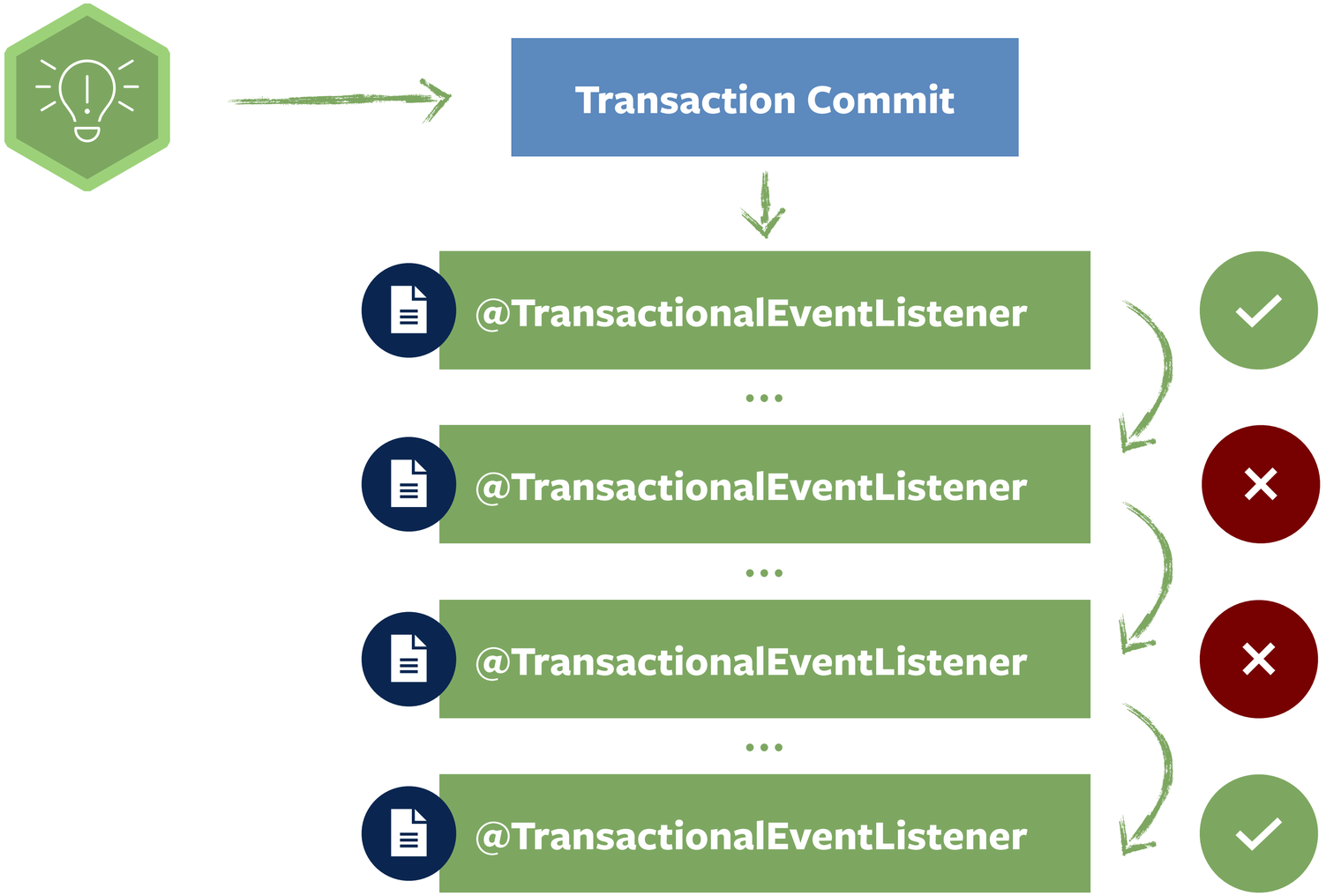
```java
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;
  private final ApplicationEventPublisher events;

  @Transactional
  void complete(Order order) {

    orders.save(order.complete());

    events.publishEvent(
        OrderCompleted.of(order.getId()));
  }
}
```

```java
@Service
class Notifications {

  @Async
  @TransactionalEventListener
  void on(OrderCompleted event) {
    // Interact with SMTP server
  }
}
```


Time
Original business transaction
Begin
Commit
End
@TransactionalEventListener
Checkout
Notifications


Time
Original business transaction
Begin
Commit / End
End
@Async
@TransactionalEventListener
Checkout
Notifications

What if a transactional event listener fails?


Transaction Commit
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener

```java
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;
  private final ApplicationEventPublisher events;

  @Transactional
  void complete(Order order) {

    orders.save(order.complete());

    events.publishEvent(
        OrderCompleted.of(order.getId()));
  }
}
```

```java
@Service
class Inventory {

  @Async
  @TransactionalEventListener
  void on(OrderCompleted event) {
    // Interact with SMTP server
  }
}
```

```java
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;
  private final ApplicationEventPublisher events;

  @Transactional
  void complete(Order order) {

    orders.save(order.complete());

    events.publishEvent(
        OrderCompleted.of(order.getId()));
  }
}
```

```java
@Service
class Inventory {

  @Async
  @Transactional
  @TransactionalEventListener
  void on(OrderCompleted event) {
    // Interact with SMTP server
  }
}
```

```java
@Service
@RequiredArgsConstructor
class Checkout {

  private final OrderRepository orders;
  private final ApplicationEventPublisher events;

  @Transactional
  void complete(Order order) {

    orders.save(order.complete());

    events.publishEvent(
        OrderCompleted.of(order.getId()));
  }
}
```

```java
@Service
class Inventory {

  @ApplicationModuleListener
  void on(OrderCompleted event) {
    // Interact with SMTP server
  }
}
```

```java
@ApplicationModuleTest
class CheckoutTests {

  private final Checkout checkout;

  @Test
  void completesOrder(Scenario scenario) {

    var order = new Order(…);

    scenario.stimulate(() → checkout.complete(order))
      .andWaitForEventOfType(InventoryUpdated.class)
      .matchingMappedValue(InventoryUpdated::getOrderId, order.getId())
      .toArrive();
  }
}
```

Spring Modulith

Given / When / Then

Summary

- Functional decomposition first, technical decomposition second
- Align consistency scope and testing with functional boundaries
- In modulithic arrangements, use spectrum of integration options and consistency levels
- Spring Modulith for advanced support

	Classic	Event Listener	Application Module Listener
Module references	via Spring beans	via `@EventListener`	via `@ApplicationModuleListener` (`@Async` `@TransactionalEventListener`)
Integration style	synchronous	synchronous	Asynchronous / Transaction-bound
Tests			
Orientation	vertical / horizontal	vertical	vertical
Focus	Module interaction	Module operation outcome signaled by an event	Module operation outcome signaled by an event
Approach	via mocks / `@MockBean`	via `AssertablePublishedEvents`	via `Scenario`
Risk	Integration tests including multiple modules	Tests exclude code that could break transaction at runtime	Asynchronous module interaction
Consistency			
Boundaries	Potentially (accidentally) spanning multiple modules	Potentially (accidentally) spanning multiple modules	Aligned with module boundaries / Eventual Consistency /
Mechanism	Transaction	Transaction	Event Publication Registry
Module coupling			

Thank you!
Questions?

Sample Code

Oliver Drotbohm odrotbohm oliver.drotbohm@broadcom.com