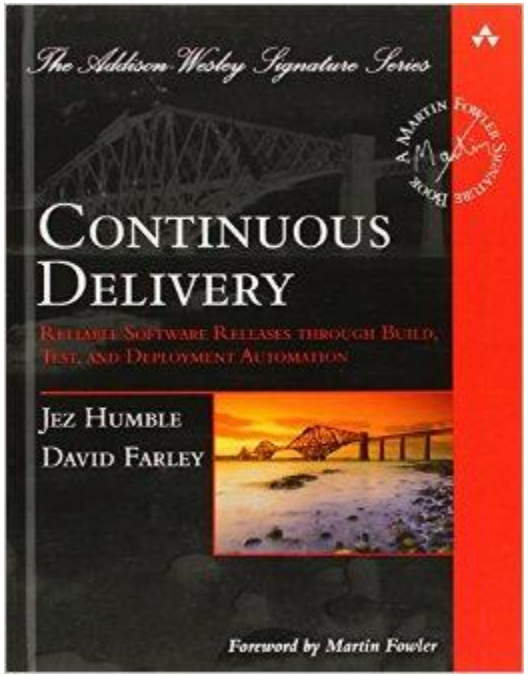
Automated Functional Acceptance Tests

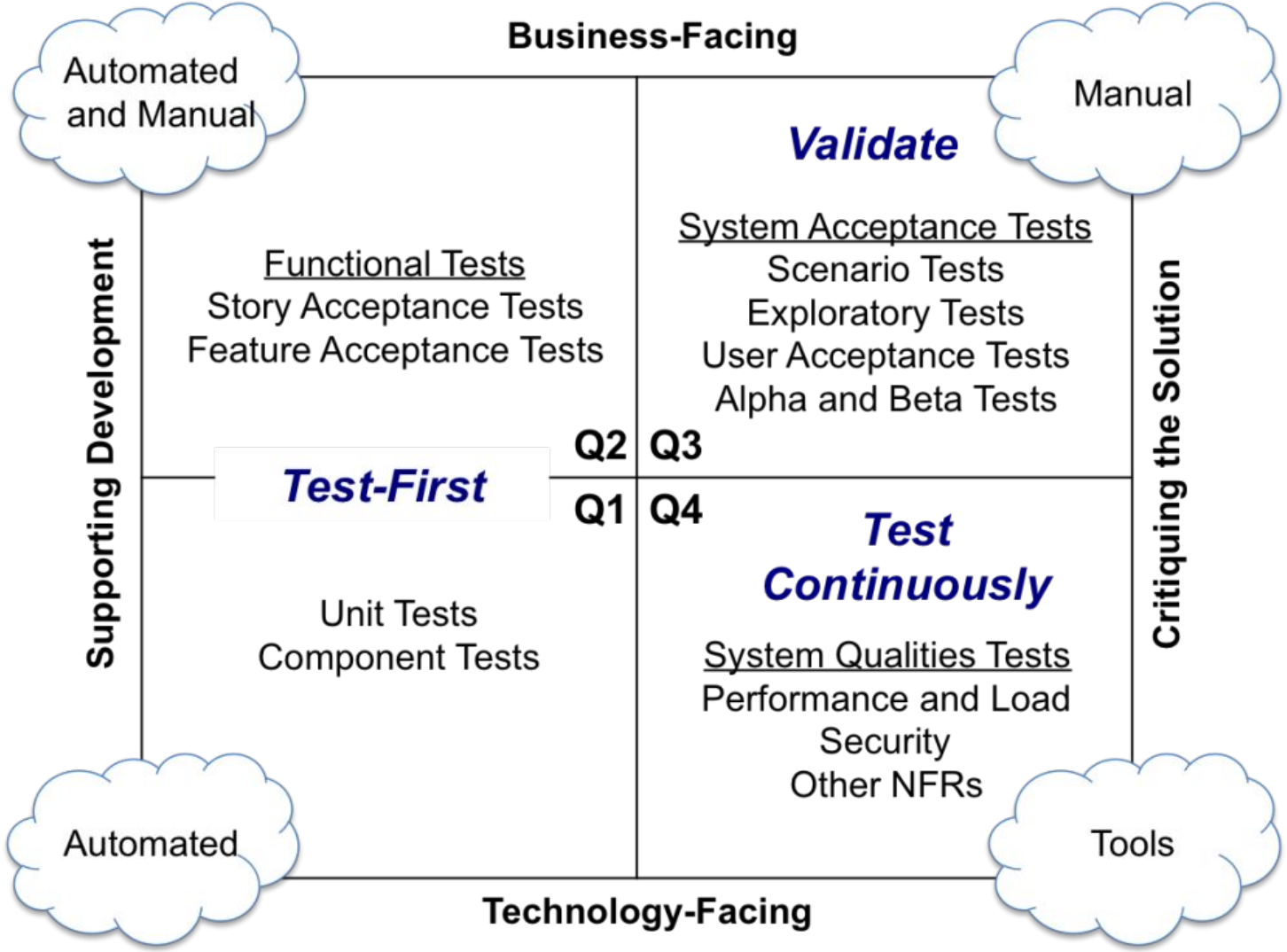


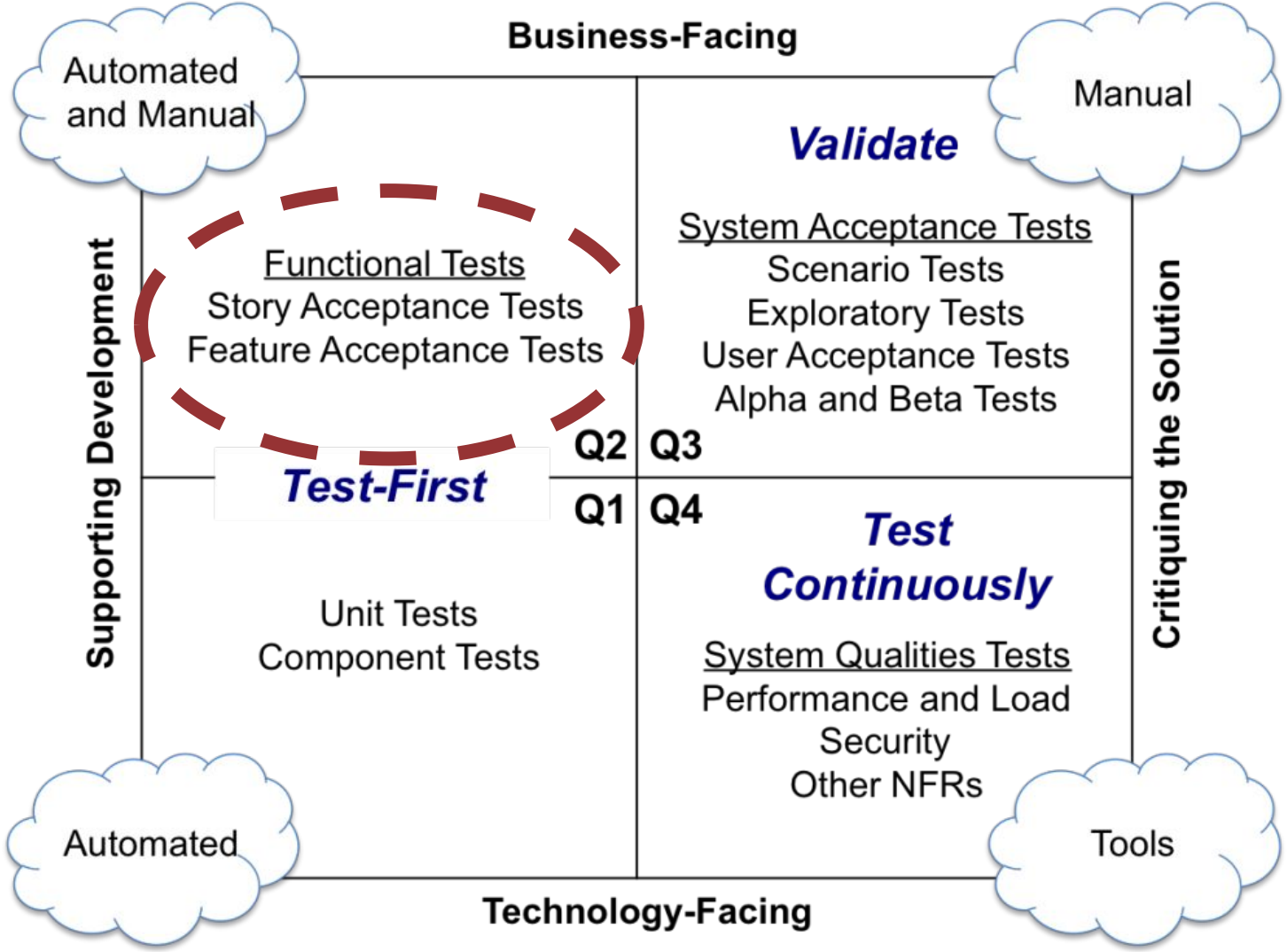
Content

- Focus on automated functional acceptance tests
- Briefly on `alarmservice`
- Tools: docker, docker-compose
- Status tests DNA-M

Goal

- Learn more about automated functional acceptance tests in practice
- Test implementation using simple microservice (`alarmservice`)
- Implement simple story, “Add node” for DNA-M


Business-Facing
Automated and Manual
Manual
Validate
System Acceptance Tests
Scenario Tests
Exploratory Tests
User Acceptance Tests
Alpha and Beta Tests
Functional Tests
Story Acceptance Tests
Feature Acceptance Tests
Supporting Development
Critiquing the Solution
Q2
Q3
Test-First
Q1
Q4
Test Continuously
Unit Tests
Component Tests
System Qualities Tests
Performance and Load
Security
Other NFRs
Automated
Tools
Technology-Facing

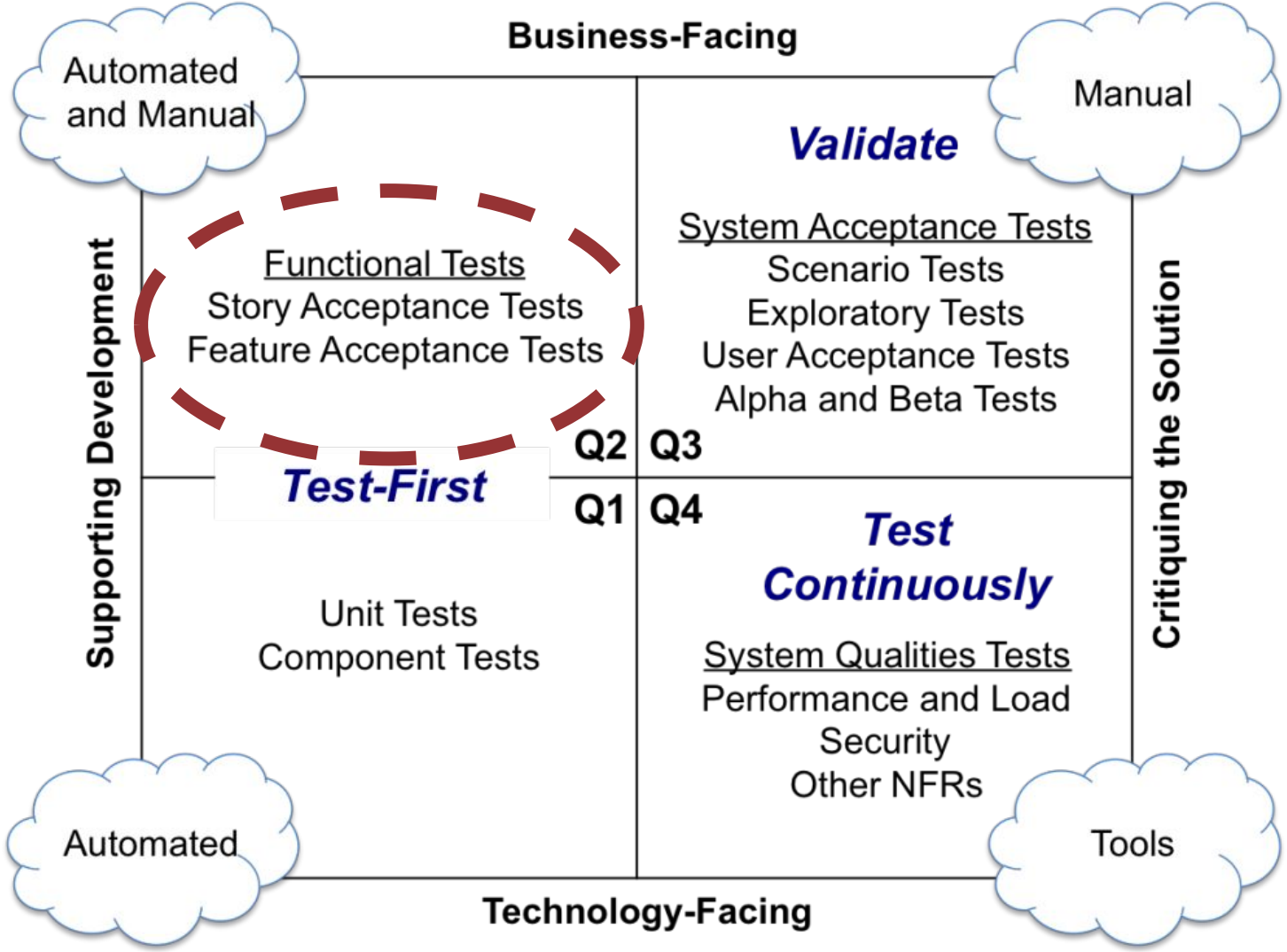

Business-Facing
Automated and Manual
Manual
Validate
System Acceptance Tests
Scenario Tests
Exploratory Tests
User Acceptance Tests
Alpha and Beta Tests
Functional Tests
Story Acceptance Tests
Feature Acceptance Tests
Supporting Development
Critiquing the Solution
Q2
Q3
Test-First
Q1
Q4
Test Continuously
Unit Tests
Component Tests
System Qualities Tests
Performance and Load
Security
Other NFRs
Automated
Tools
Technology-Facing

Q2: Business facing + Dev support

- Story Acceptance Tests
  - Written i “Business Domain Language”
  - Created before or simultaneously as impl of story
  - Created by developers, testers and productowners together
  - Blackbox-tests: Only verifies output

Q2: Business facing + Dev support

- Story Acceptance Tests
  - For developers, answers question:
    How to I know when I am done?
  - For product owner/user, answers question:
    Did I get what I asked for?


Business-Facing
Automated and Manual
Manual
Validate
System Acceptance Tests
Scenario Tests
Exploratory Tests
User Acceptance Tests
Alpha and Beta Tests
Functional Tests
Story Acceptance Tests
Feature Acceptance Tests
Supporting Development
Critiquing the Solution
Q2
Q3
Test-First
Q1
Q4
Test Continuously
Unit Tests
Component Tests
System Qualities Tests
Performance and Load
Security
Other NFRs
Automated
Tools
Technology-Facing

Q2: Business facing + Dev support

- Story Acceptance Tests
- Feature Acceptance Tests
  - Higher level of abstraction compared to Story Acceptance Tests
  - Made of several combined Story Acceptance Tests

Q2: Business facing + Dev support

- Story Acceptance Tests
- Feature Acceptance Tests

Hit the UI or not?

Q2: Business facing + Dev support

- Aut. acceptance tests are too expensive?

  Well written unit tests, component tests, pair-programming, refactoring and exploratory testing is enough?

Q2: Business facing + Dev support

- Aut. acceptance tests are too expensive?
- Flaws in argumentation
  - Only functional acceptance tests verify that application delivers business value
  - Functional acceptance tests protects the application when making major changes
  - Without functional acceptance tests the burdon on testers increases

Q2: Functional acceptance tests

Acceptance Criteria “Happy Path”

Given	A specified state of a system
When	An action or event occurs
Then	The state of the system has changed or an output has been produced

- Alternative Paths, Sad Paths

Q2: Functional acceptance tests

Acceptance Criteria “Happy Path”

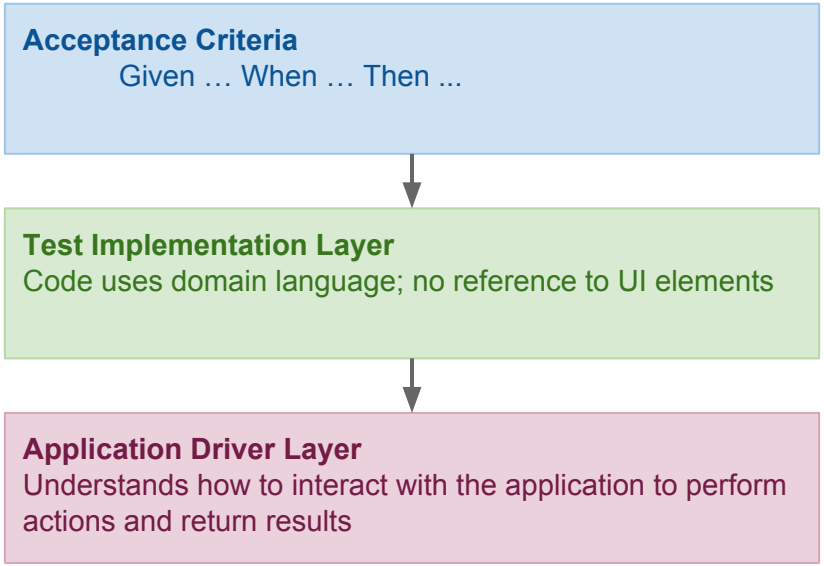
Given	Book that has not been checked out User who is registered on the system
When	User checks out a book
Then	Book is marked as checked out

Creating maintainable accept. tests

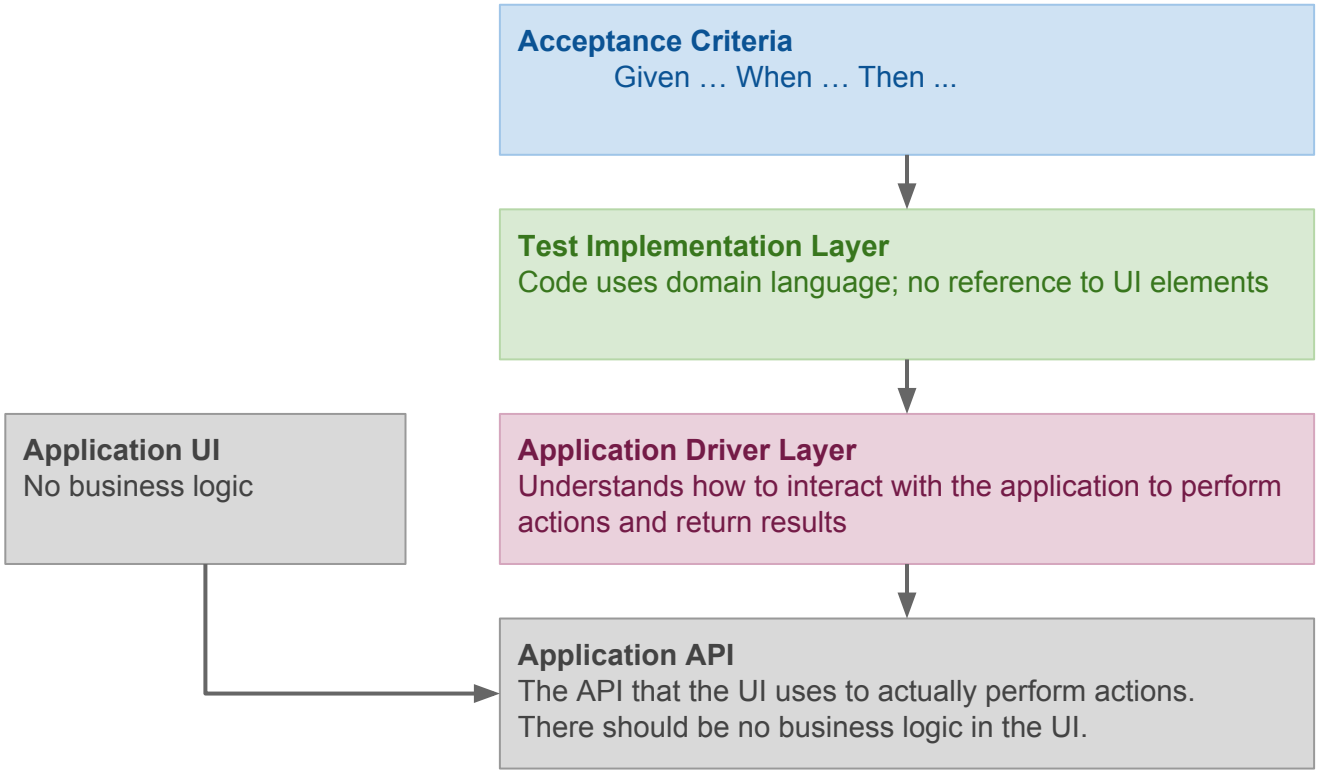
- Acceptance criteria
  - Written with automation in mind
  - Following INVEST principles

    Independent, Negotiable, Valuable, Estimable, Small and Testable
  - Valuable to the end user
  - Testable
- Acceptance tests should always be layered

Creating maintainable accept. tests



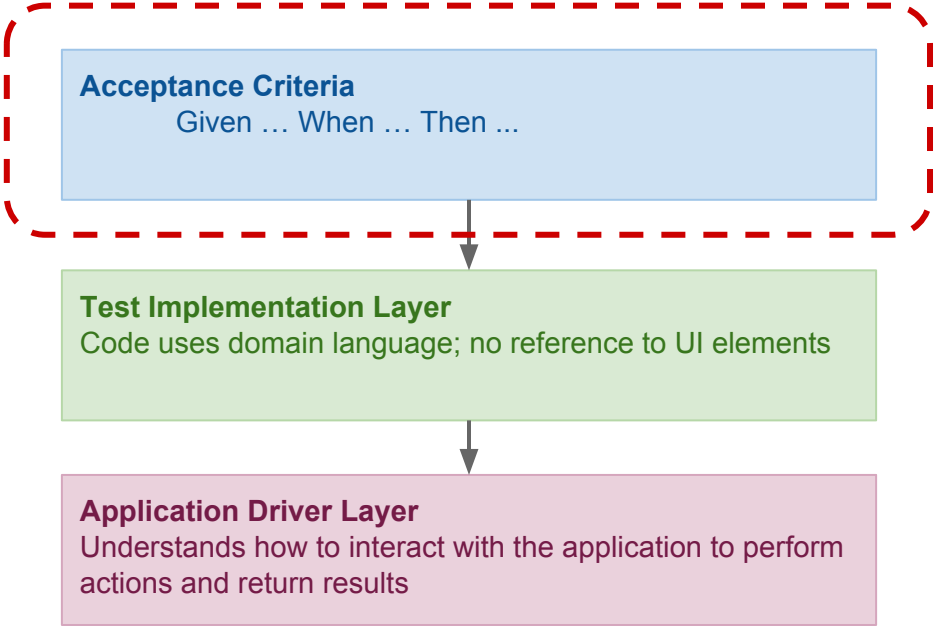
Creating maintainable accept. tests



The process of creating executable specifications

- Automated acceptance not only about testing
- Spawned Behavior Driven Development
- Specifications does not get out of date

The process of creating executable specifications



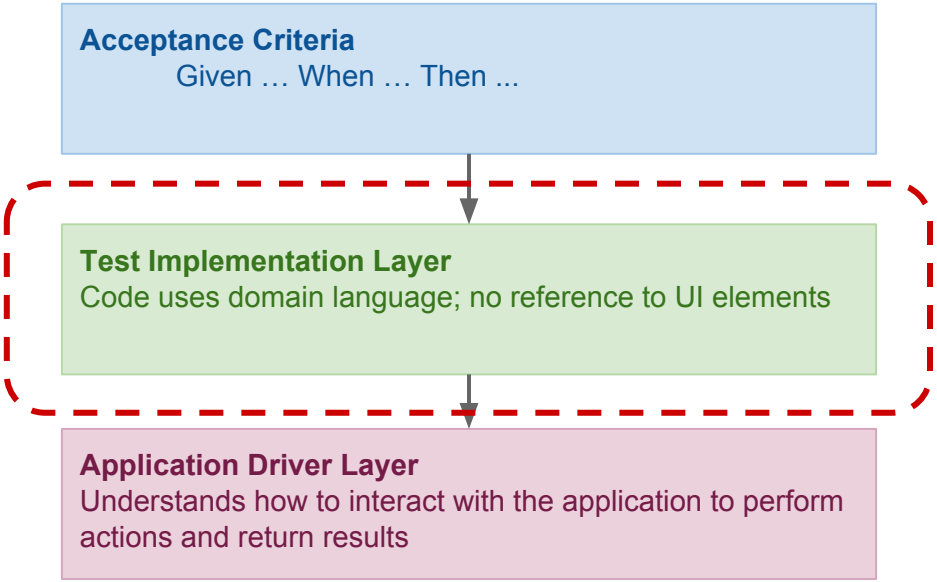
The process of creating executable specifications

1. Discuss acceptance criteria for your story with your customer
2. Write them down in executable format:

```
Scenario: User order should debit account correctly

Given there is an instrument called bond
And there is a user called Dave with 50 dollars in his account

When I log in as Dave
And I select instrument bond
And I place an order to buy 4 at 10 dollars each
And the order is successful

Then I have 10 dollars left on my account
```

Acceptance criterion in file called `placing_an_order.feature` (Cucumber)

The process of creating executable specifications



The process of creating executable specifications

3. Write an implementation for the test which uses only the domain language, accessing the application driver layer

```
Given /there is an instrument called (\w+)$/ do |instrument |
    @admin_api.create_instrument(instrument)
Given ...
    @admin_api.create_user(user, amount)
When /I log in as (\w+)$ do |user |
    trading_api.login(user)
When ...
    trading_api.select_instrument(bond)
    trading_api.place_order(quantity, amount)
    trading_api.confirm_order_success(instrument, quantity, amount)
Then
    trading_api.confirm_account_balance(balance)
```

Test implementation in simplfied Ruby file `placing_an_order_steps.rb`

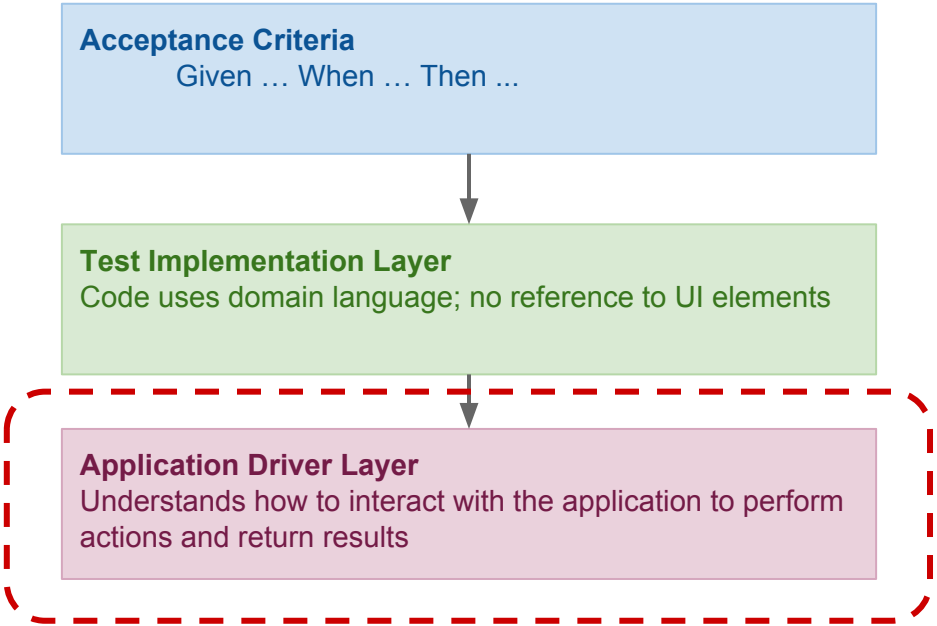
The process of creating executable specifications

3. Write an implementation for the test which uses only the domain language, accessing the application driver layer

```
Given /there is an instrument called (\w+)$/ do |instrument |
    @admin_api.create_instrument(instrument)
Given ...
    @admin_api.create_user(user, amount)
When /I log in as (\w+)$ do |user |
    trading_api.login(user)
When ...
    trading_api.select_instrument(bond)
    trading_api.place_order(quantity, amount)
    trading_api.confirm_order_success(instrument, quantity, amount)
Then
    trading_api.confirm_account_balance(balance)
```

Test implementation in simplfied Ruby file `placing_an_order_steps.rb`

The process of creating executable specifications



The process of creating executable specifications

4. Create an application driver layer which talks to the system under test

```
admin_api.create_user(user, amount) {
    Interaction with application/system under test
    ….
}

trading_api.login(user) {
   Interaction with application/system under test
    ….
}
```

The process of creating executable specifications

```
public class PlacingAnOrderAcceptanceTest {
    @Test
    public void userOrderShouldDebitAccountCorrectly() {
        //Given
        adminApi.createInstrument("name: bond")
        adminApi.createUser("Dave", "balance: 50.00")
        tradingApi.login("Dave")
        //When
        tradingApi.selectInstrument("bond")
        tradingApi.placeOrder("price: 10.00", "quantity: 4")
        tradingApi.confirmOrderSuccess("instrument: bond", "price: 10.00", "quantity: 4")
        //Then
        tradingApi.confirmBalance("balance: 10.00")
    }
}
```

`adminApi` and **`tradingApi`** belongs to the Application Driver Layer.

Written to mirror Domain Language (in this case trading)

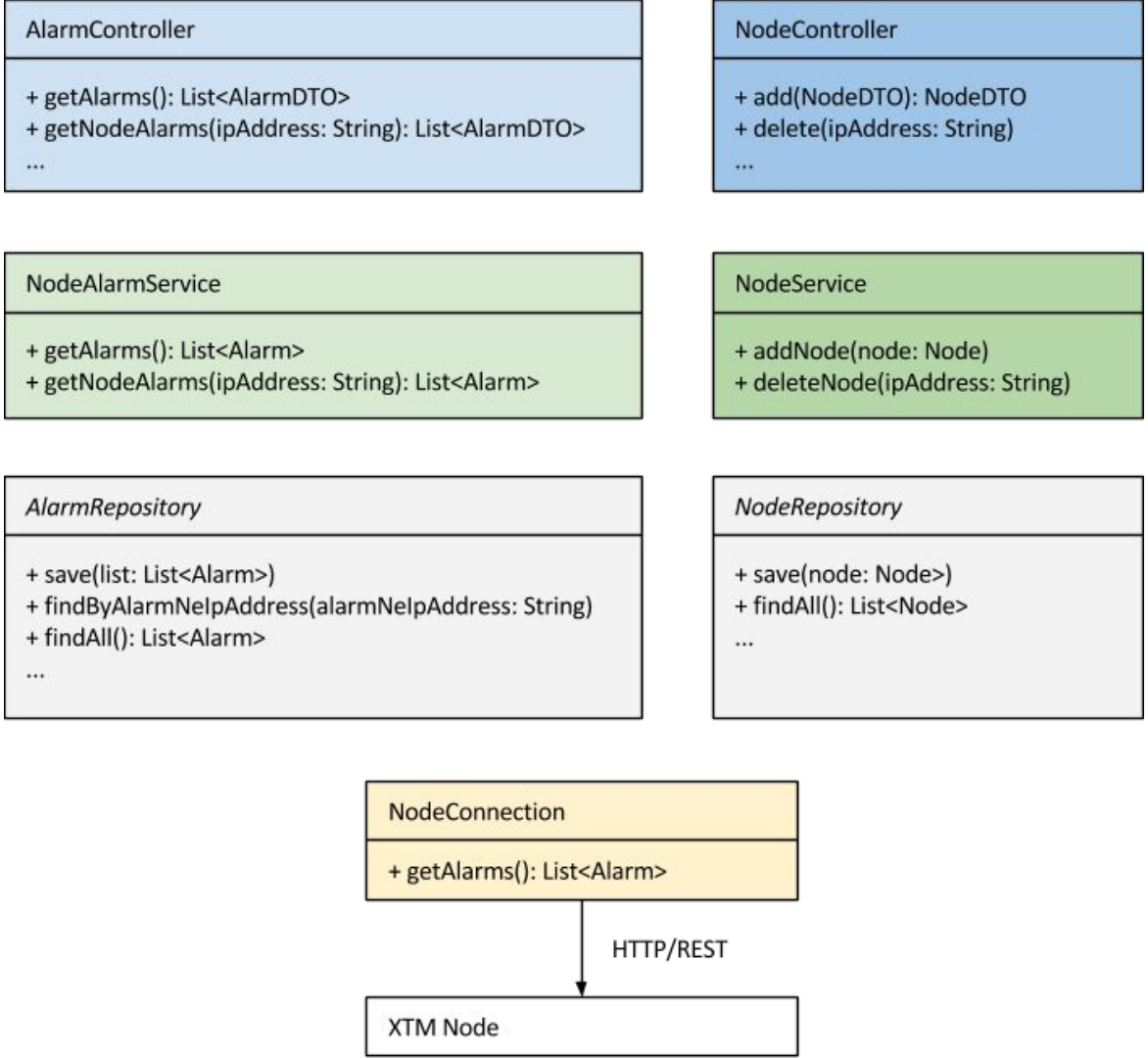
Can be thought of as DSL in it's own right

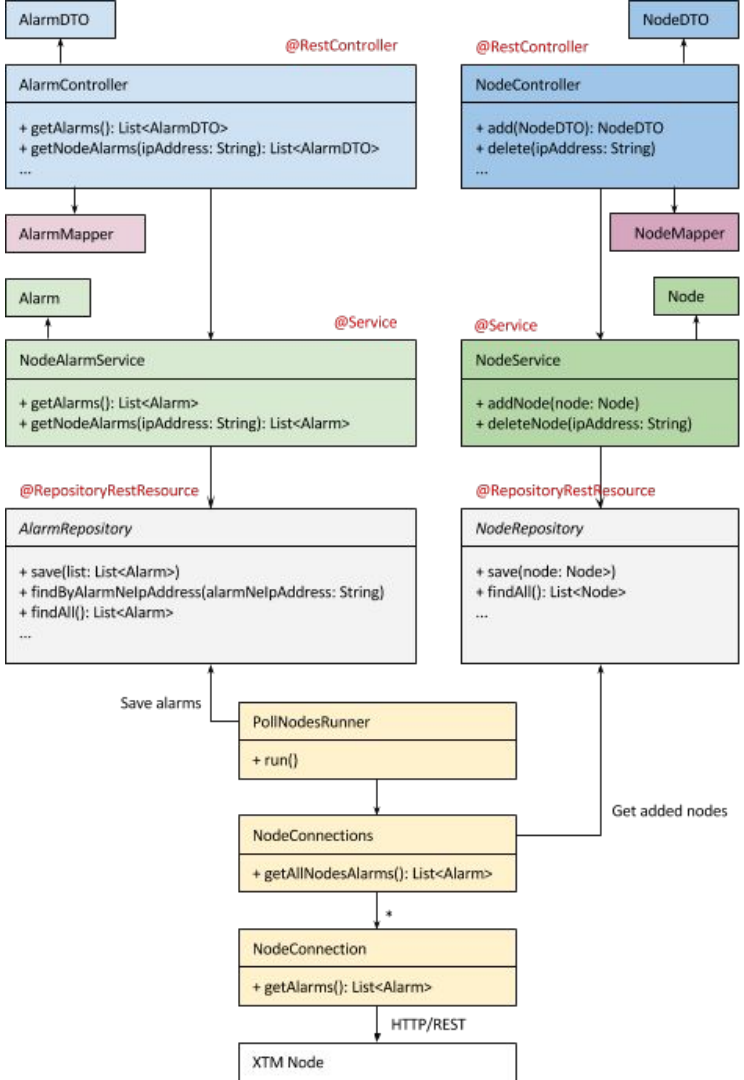
They abstract away (possibly) complex interactions with the application

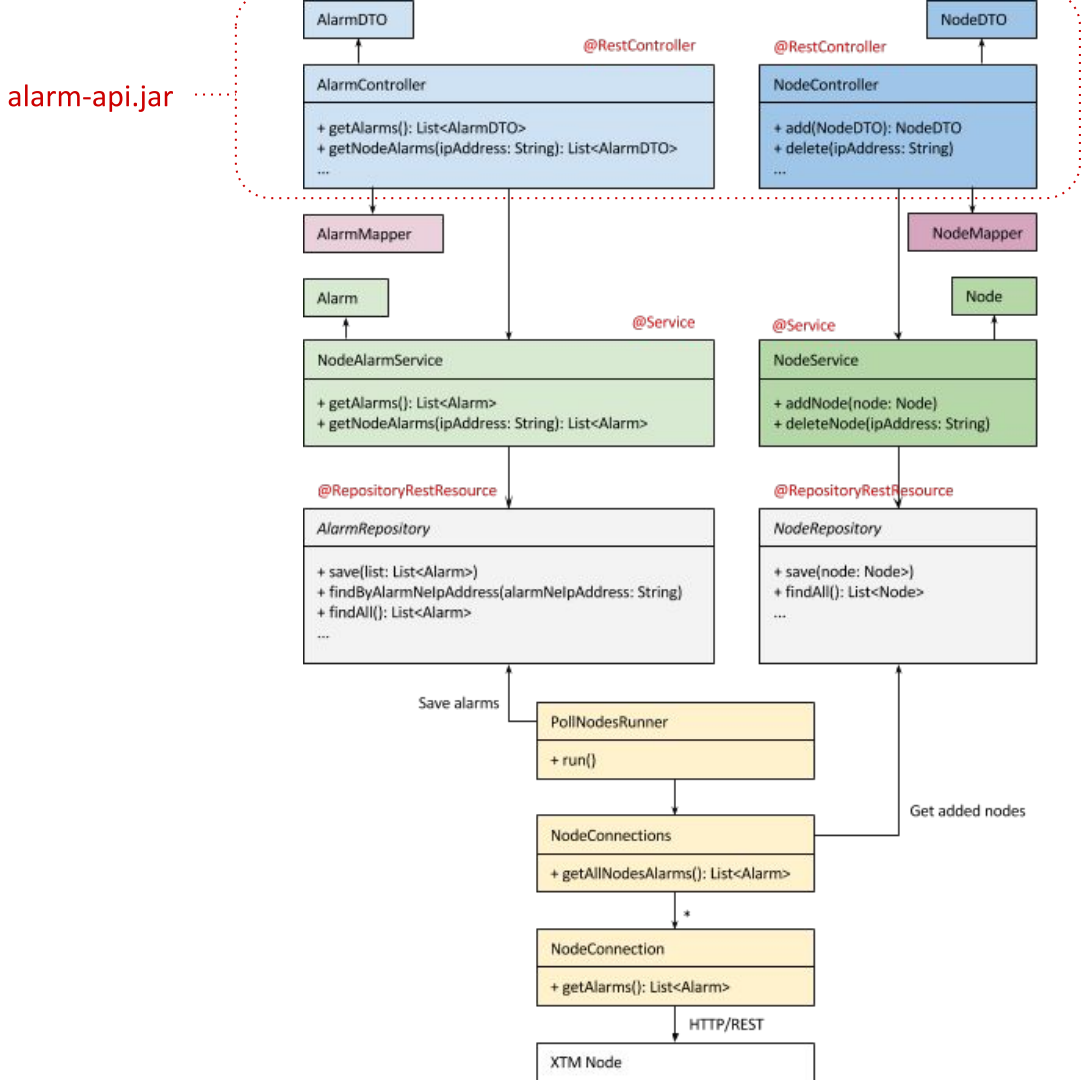
`Dave` and **`bond`** are aliases. Behind the scenes application driver creates real instruments and users

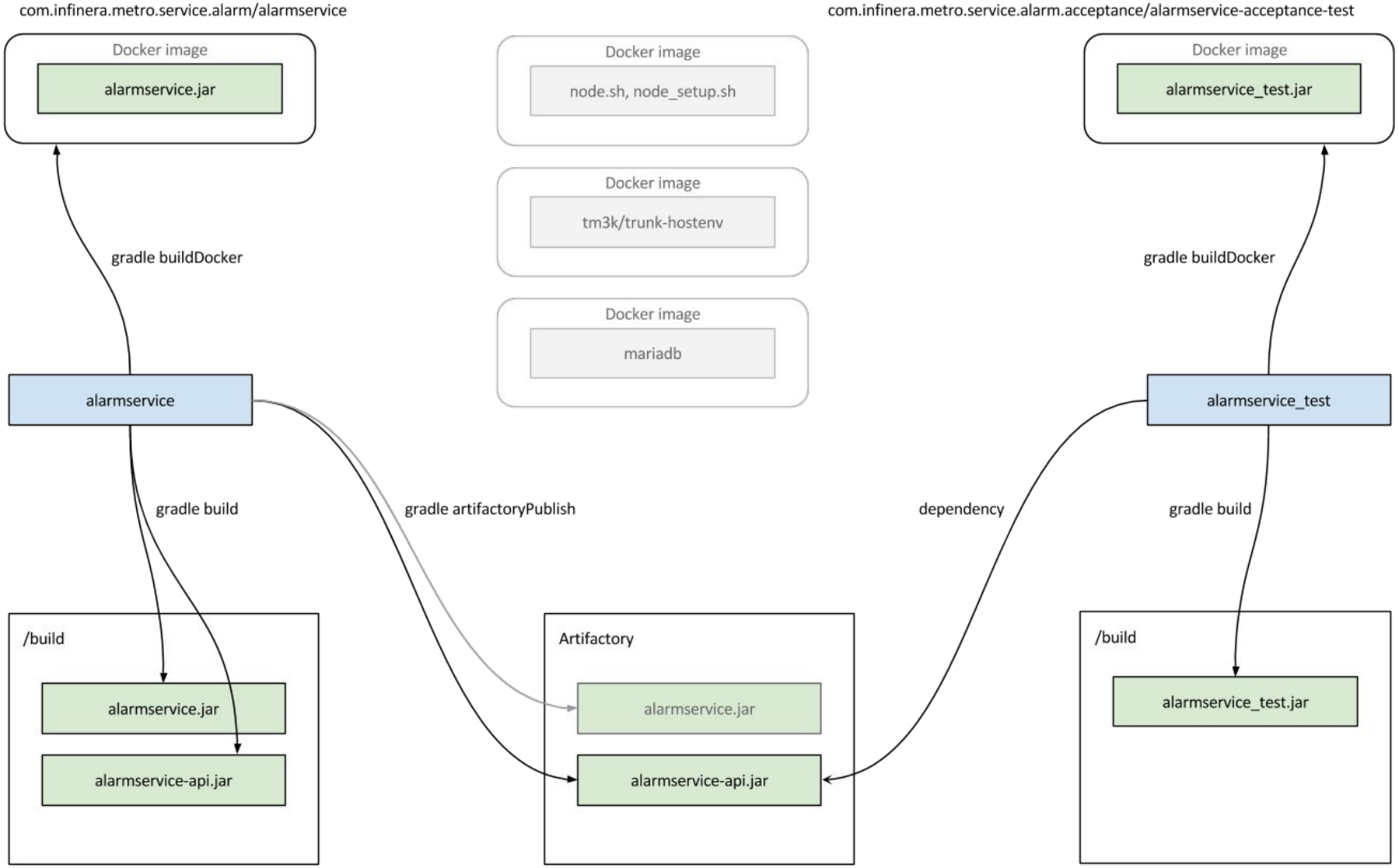
Alarmservice

- Aggregates alarms from XTM nodes
- Utilizes the REST api of XTM nodes


AlarmController
+ getAlarms(): List<AlarmDTO>
+ getNodeAlarms(ipAddress: String): List<AlarmDTO>
...
NodeController
+ add(NodeDTO): NodeDTO
+ delete(ipAddress: String)
...
NodeAlarmService
+ getAlarms(): List<Alarm>
+ getNodeAlarms(ipAddress: String): List<Alarm>
NodeService
+ addNode(node: Node)
+ deleteNode(ipAddress: String)
AlarmRepository
+ save(list: List<Alarm>)
+ findByAlarmNeIpAddress(alarmNeIpAddress: String)
+ findAll(): List<Alarm>
...
NodeRepository
+ save(node: Node>)
+ findAll(): List<Node>
...
NodeConnection
+ getAlarms(): List<Alarm>
HTTP/REST
XTM Node


AlarmDTO
NodeDTO
@RestController
@RestController
AlarmController
+ getAlarms(): List<AlarmDTO>
+ getNodeAlarms(ipAddress: String): List<AlarmDTO>
...
NodeController
+ add(NodeDTO): NodeDTO
+ delete(ipAddress: String)
...
AlarmMapper
NodeMapper
Alarm
Node
@Service
@Service
NodeAlarmService
+ getAlarms(): List<Alarm>
+ getNodeAlarms(ipAddress: String): List<Alarm>
NodeService
+ addNode(node: Node)
+ deleteNode(ipAddress: String)
@RepositoryRestResource
@RepositoryRestResource
AlarmRepository
+ save(list: List<Alarm>)
+ findByAlarmNeIpAddress(alarmNeIpAddress: String)
+ findAll(): List<Alarm>
...
NodeRepository
+ save(node: Node>)
+ findAll(): List<Node>
...
Save alarms
PollNodesRunner
+ run()
Get added nodes
NodeConnections
+ getAllNodesAlarms(): List<Alarm>
*
NodeConnection
+ getAlarms(): List<Alarm>
HTTP/REST
XTM Node


alarm-api.jar
AlarmDTO
NodeDTO
@RestController
@RestController
AlarmController
+ getAlarms(): List<AlarmDTO>
+ getNodeAlarms(ipAddress: String): List<AlarmDTO>
...
NodeController
+ add(NodeDTO): NodeDTO
+ delete(ipAddress: String)
...
AlarmMapper
NodeMapper
Alarm
Node
@Service
@Service
NodeAlarmService
+ getAlarms(): List<Alarm>
+ getNodeAlarms(ipAddress: String): List<Alarm>
NodeService
+ addNode(node: Node)
+ deleteNode(ipAddress: String)
@RepositoryRestResource
@RepositoryRestResource
AlarmRepository
+ save(list: List<Alarm>)
+ findByAlarmNeIpAddress(alarmNeIpAddress: String)
+ findAll(): List<Alarm>
...
NodeRepository
+ save(node: Node>)
+ findAll(): List<Node>
...
Save alarms
PollNodesRunner
+ run()
Get added nodes
NodeConnections
+ getAllNodesAlarms(): List<Alarm>
*
NodeConnection
+ getAlarms(): List<Alarm>
HTTP/REST
XTM Node


com.infinera.metro.service.alarm/alarmservice
com.infinera.metro.service.alarm.acceptance/alarmservice-acceptance-test
Docker image
alarmservice.jar
Docker image
node.sh, node_setup.sh
Docker image
alarmservice_test.jar
Docker image
tm3k/trunk-hostenv
Docker image
mariadb
gradle buildDocker
gradle buildDocker
alarmservice
alarmservice_test
gradle build
gradle artifactoryPublish
dependency
gradle build
/build
alarmservice.jar
alarmservice-api.jar
Artifactory
alarmservice.jar
alarmservice-api.jar
/build
alarmservice_test.jar

Docker containers

- Service images
  Long running processes: Web servers, databases e.t.c.
- Executable images
  Short lived processes: Compilers, build tools, tests
- Containers = Immutable

Gradle Docker plugin

transmode/gradle-docker

Makes it possible to build and publish Docker images with Gradle

docker-compose

Tool for defining and running multi-container applications

1. Define Dockerfiles for each sub application
2. Define services that make up your application i `docker-compose.yml`
3. Run

   ```
   $ docker-compose up
   ```

docker-compose

docker-compose.yml example:

```
version: '2'
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
```

docker-compose

docker-compose.yml example:

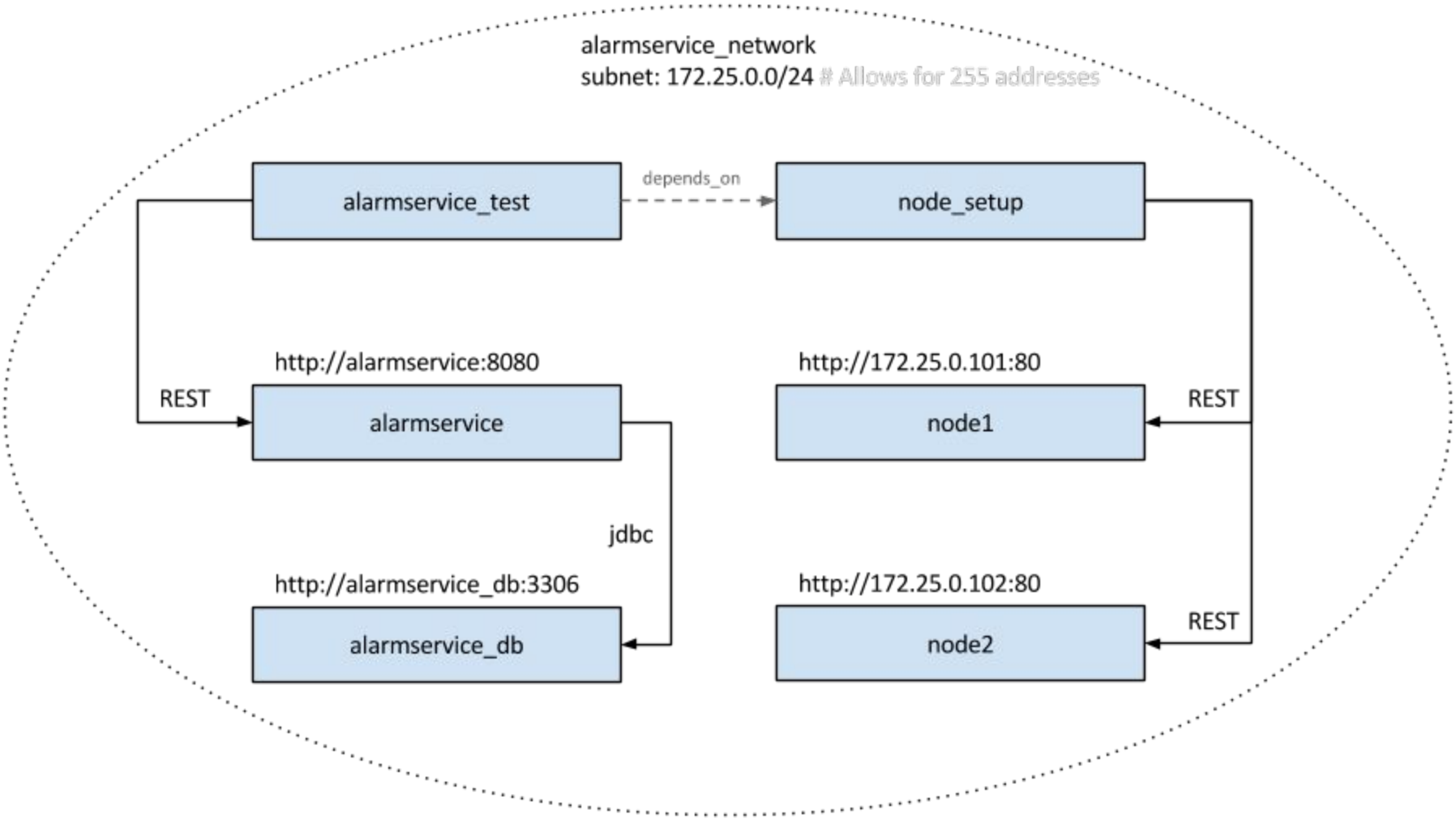
```
version: '2'
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
```

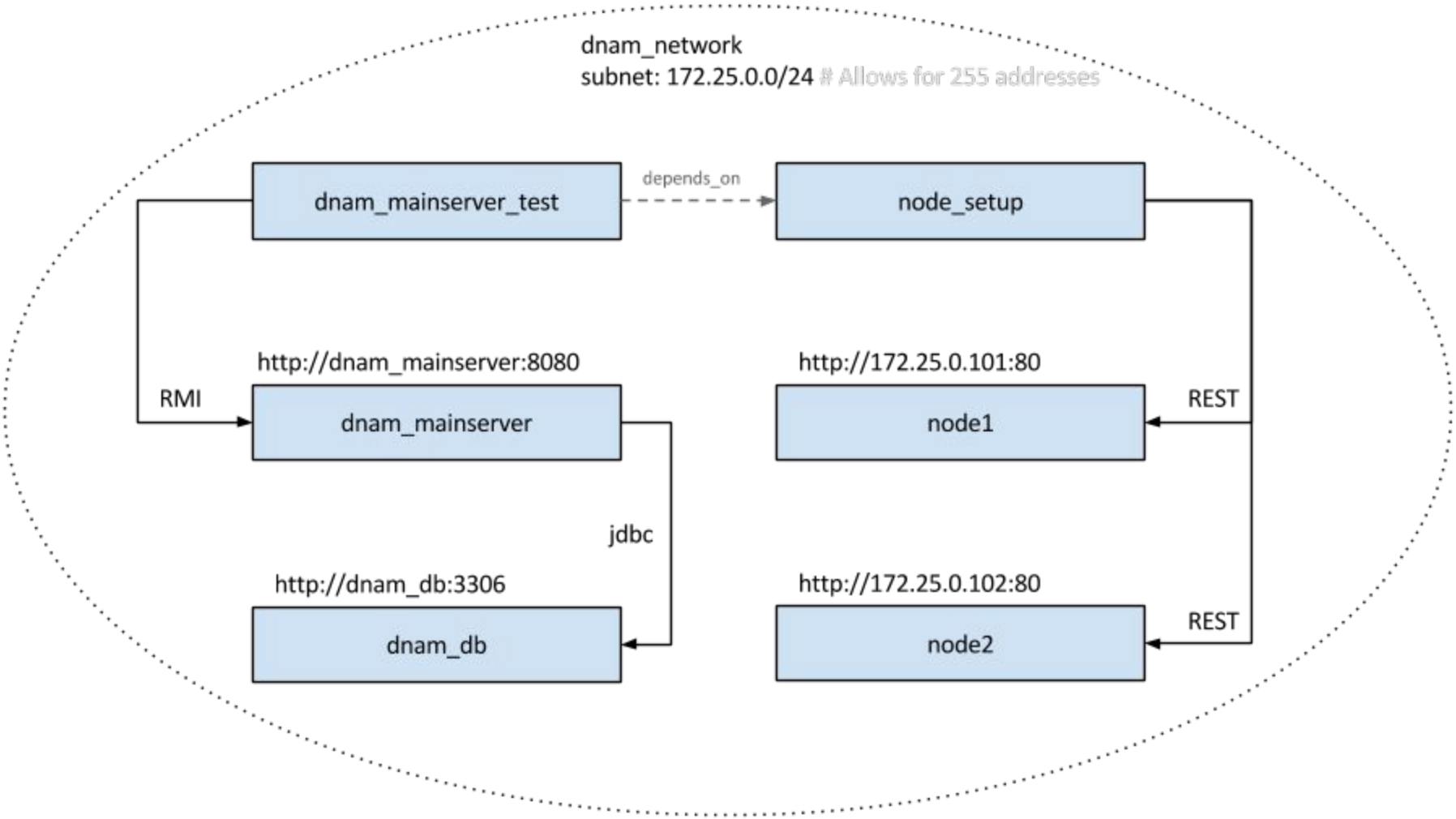
When you run docker-compose up, the following happens:

1. A network called myapp_default is created.
2. A container is created using web's configuration. It joins the network myapp_default under the name web.
3. A container is created using db's configuration. It joins the network myapp_default under the name db.

Each container can now look up the hostname web or db and get back the appropriate container's IP address.

For example, web's application code could connect to the URL postgres://db:5432 and start using the Postgres databa


alarmservice_network
subnet: 172.25.0.0/24 # Allows for 255 addresses
alarmservice_test
depends_on
node_setup
http://alarmservice:8080
REST
alarmservice
http://172.25.0.101:80
node1
REST
jdbc
http://alarmservice_db:3306
alarmservice_db
http://172.25.0.102:80
node2
REST


dnam_network
subnet: 172.25.0.0/24 # Allows for 255 addresses
dnam_mainserver_test
depends_on
node_setup
http://dnam_mainserver:8080
http://172.25.0.101:80
RMI
dnam_mainserver
node1
REST
jdbc
http://dnam_db:3306
http://172.25.0.102:80
dnam_db
node2
REST

Continuous Delivery

av Jez Humble och David Farley

http://www.continuousagile.com/

http://www.scaledagileframework.com/test-first/

http://blogs.atlassian.com/tag/cd-skeptics/

http://www.agilemanifesto.org/

In our ideal project, testers collaborate with developers and users to write automated tests from the start of the project.

These tests are written before developers start work on the features that they test.

Together, these tests form an executable specification of the behavior of the system, and when they pass, they demonstrate that the functionality required by the customer has been implemented completely and correctly.

The automated test suite is run by the CI system every time a change is made to the application—which means the suite also serves as a set of regression tests.