RAWDATA SECTION 2

Troels Andreasen & Henrik Bulskov

AGENDA

- Restful Web Services
- Testing with dependencies

WEB SERVICES

SIX CONSTRAINTS OF REST

Client-Server

client and server are separated

(client and server can evolve separately)

Statelessness

state is contained within the request

Cacheable

each response message must explicitly state if it can be cached or not

Layered System

client cannot tell what layer it's connected to

Code on Demand (optional)

server can extend client functionality

Uniform Interface

API and consumers share one single, technical interface: URI, Method, Media Type

UNIFORM INTERFACE SUBCONSTRAINTS

- Identification of resources
  - A resource is conceptually separate from its representation
  - Representation media types: application/json, application/xml, custom, …
- Manipulation of resources through representations
  - Representation + metadata should be sufficient to modify or delete the resource
- Self-descriptive message
  - Each message must include enough info to describe how to process the message
- Hypermedia as the Engine of Application State (HATEOAS)
  - Hypermedia is a generalization of Hypertext (links)
  - Drives how to consume and use the API
  - Allows for a self-documenting API

UNIFORM INTERFACE API

STRUCTURING THE OUTER FACING CONTRACT

Resource Identifier

HTTP Method

Payload

(representation: media types)

OUTER FACING CONTRACT

- The Outer Facing Contract is NOT your entity/domain model
  - Even if they are identical in structure, they are semantically VERY different
  - Decoupling between layers in the system
  - Outer Facing Contract is what users of your API will know and use

THE IMPORTANCE OF STATUS CODES

Level 200 - Success
200 – Ok
201 – Created
204 – No content

Level 400 – Client Mistakes
400 – Bad request
401 – Unauthorized
403 – Forbidden
404 – Not found
405 – Method not allowed
406 – Not acceptable
409 - Conflict
415 – Unsupported media type
422 – Unprocessable entity

Level 500 Server Mistakes
500 – Internal server error

UPDATING A RESOURCE

- HTTP PUT updates full resource
- HTTP PATCH is for partial updates
- The request body of a patch request is described by RFC 6902 (JSON Patch)
  - https://tools.ietf.org/html/rfc6902
- Patch requests should be sent with media type application/json-patch+json
  - But most APIs accept also application/json

JSON PATCH OPERATIONS

Add

```
{"op": "add",
"path": "/a/b",
"value": "foo"}
```

Remove

```
{"op": "remove",
"path": "/a/b"}
```

Replace

```
{"op": "replace",
"path": "/a/b",
"value": "foo"}
```

Copy

```
{"op": "copy",
"from": "/a/b",
"path": "/a/c"}
```

Move

```
{"op": "move",
"from": "a/b",
"path": "/a/c"}
```

Test

```
{"op": "test",
"path": "/a/b",
"value": "foo"}
```

PATCH EXAMPLE

```
[
    {
        "op": "replace",
        "path": "/title",
        "value": “new title"
    }
    ,
    {
        "op": “remove",
        "path": "/description"
    }
]
```

- array of operations
- “replace” operation
- “title” property gets value “new title”
- “remove” operation
- “description” property is removed (set to its default value)

ENTITY TAGS (ETAGS)

- Header to support smart server caching
  - Strong and Weak Caching Support
  - Returned in the Response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Date: Thu, 23 May 2013 21:52:14 GMT
ETag: W/"489302394209 8"
Content-Length: 639
```

ENTITY TAGS (ETAGS)

- Client Should Send ETag back to see if new version is available
  - Request with If-None-Match

```
GET /api/games/2 HTTP/1.1
Accept: application/json, text/xml
Host: localhost:8863
If-None-Match: "4893023942098"
```

  - Use 304 to indicate that it hasn't changed

```
HTTP/1.1 304 Not Modified
```

ENTITY TAGS (ETAGS)

- Client Should Send ETag back to see if new version is available
  - For PUT, use If-Match

```
PUT /api/games/2 HTTP/1.1
Accept: application/json, text/xml
Host: localhost:8863
If-Match: "4893023942098"
...
```

  - Use Status Code if it doesn't match

```
HTTP/1.1 412 Precondition Failed
```

PAGING

- Lists should always support paging
  - Query String parameters to request paging information
  - Object Wrapper to indicate next/prev links

```
{
 "totalResults": 1598,
 "nextPage": "http://.../api/games/?page=3",
 "prevPage": "http://.../api/games/?page=1",
 "results": [...]
}
```

- Can use different page sizes too
  - But limit size of page to limit server load

```
http://.../api/games?page=5&pageSize=50
```

VERSIONING THE API

- Why Version
  - Once you publish an API, it's set in stone
  - Publishing an API is not a trivial move
  - Users/Customers rely on the API not changing
  - But requirements will change
  - Need a way to evolve the API without breaking existing clients
  - API Versioning isn't Product Versioning
    - Don't tie them together

SOME EXAMPLES

- Tumblr
  - Uri Path
    - http://api.tumblr.com/v2/user/
- Netflix
  - Uri Parameter
    - http://api.netflix.com/catalog/titles/series/70023522?v=1.5
- GitHub API
  - Content Negotiation
    - Content Type: application/vnd.github.1.param+json
- Azure
  - Request Header
    - x-ms-version: 2011-08-18

VERSIONING IN THE URI PATH

- Using Part of Your Path to Version
  - Allows you to drastically change the API
    - Everything below the version is open to change

```
http://.../api/v1/Customers?type=Current&id=123
http://.../api/v2/CurrentCustomers/123
```

- Pro(s):
  - Simple to segregate old APIs for backwards compatibility
- Con(s):
  - Requires lots of client changes as you version
  - E.g. version # has to change in every client
  - Increases the size of the URI surface area you have to support

VERSIONING IN THE URI PARAMETERS

- Version as Query String Parameter
  - Optional Parameter

```
http://.../api/Customers
http://.../api/Customers?v=2.1
```

- Pro(s):
  - Without version, users always get latest version of API
  - Little client change as versions mature
- Con(s):
  - Can surprise developers with unintended changes

VERSIONING WITH CONTENT NEGOTIATION

- Versioning with Content Type in Accept Header
  - Instead of using standard MIME types, use custom

```
GET /api/customer/123
HOST: http://.../
Accept: application/myapp.v1.customer
```

- Pro(s):
  - Packages API and Resource Versioning in one
  - Removes versioning from API so clients don't have to change
- Con(s):
  - Adds complexity - adding headers isn't easy on all platforms
  - Can encourage increased versioning which causes more code churning

VERSIONING WITH CUSTOM HEADER

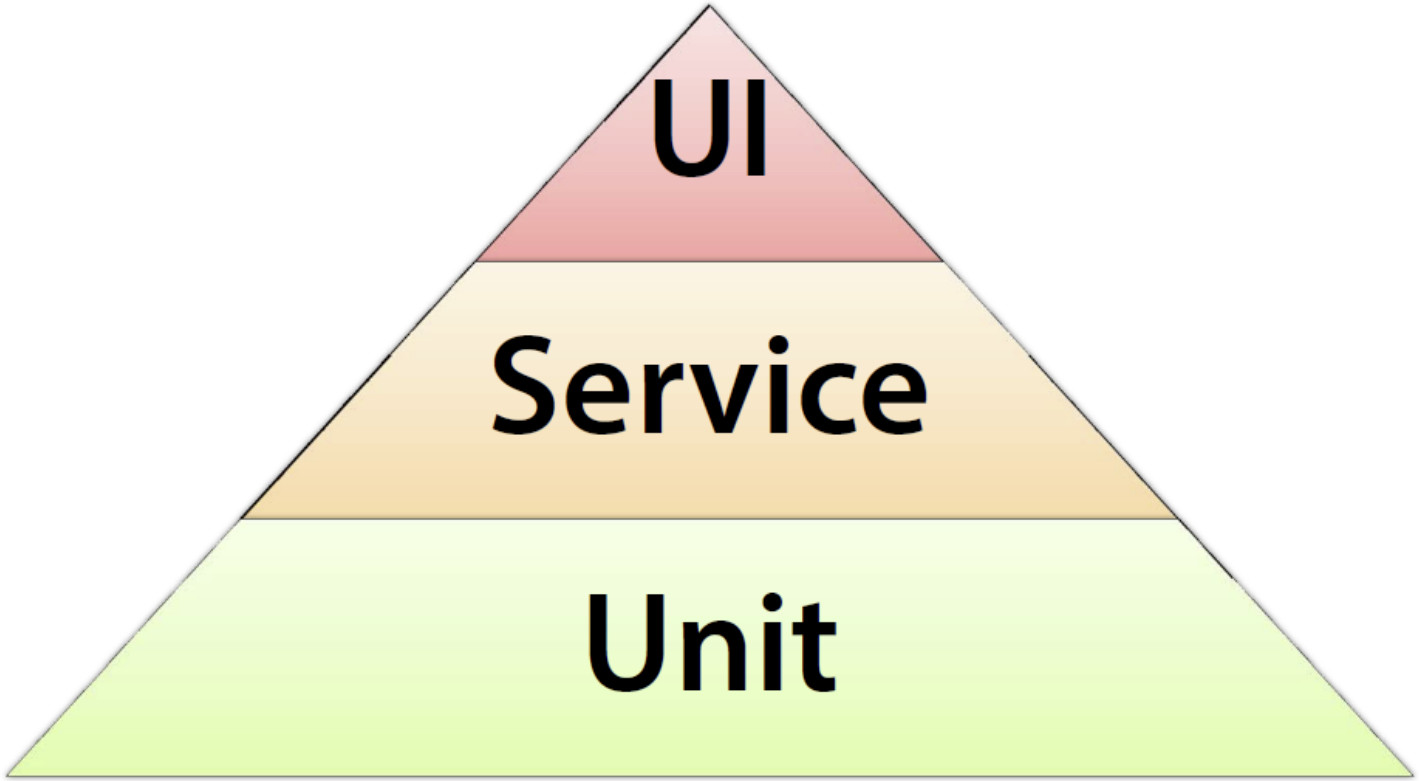
- Using a Custom Header to Version API calls
  - Should be a header value that is only of value to your API

```
GET /api/customer/123
HOST: http://.../
x-MyApp-Version: 2013-08-13
```

- Pro(s):
  - Separates Versioning from API call signatures
  - Not tied to resource versioning (e.g. Content Type)
- Con(s):
  - Adds complexity - adding headers isn't easy on all platforms

TESTING WITH DEPENDENCIES

THE TEST PYRAMID



PROPERTIES OF A GOOD UNIT TEST

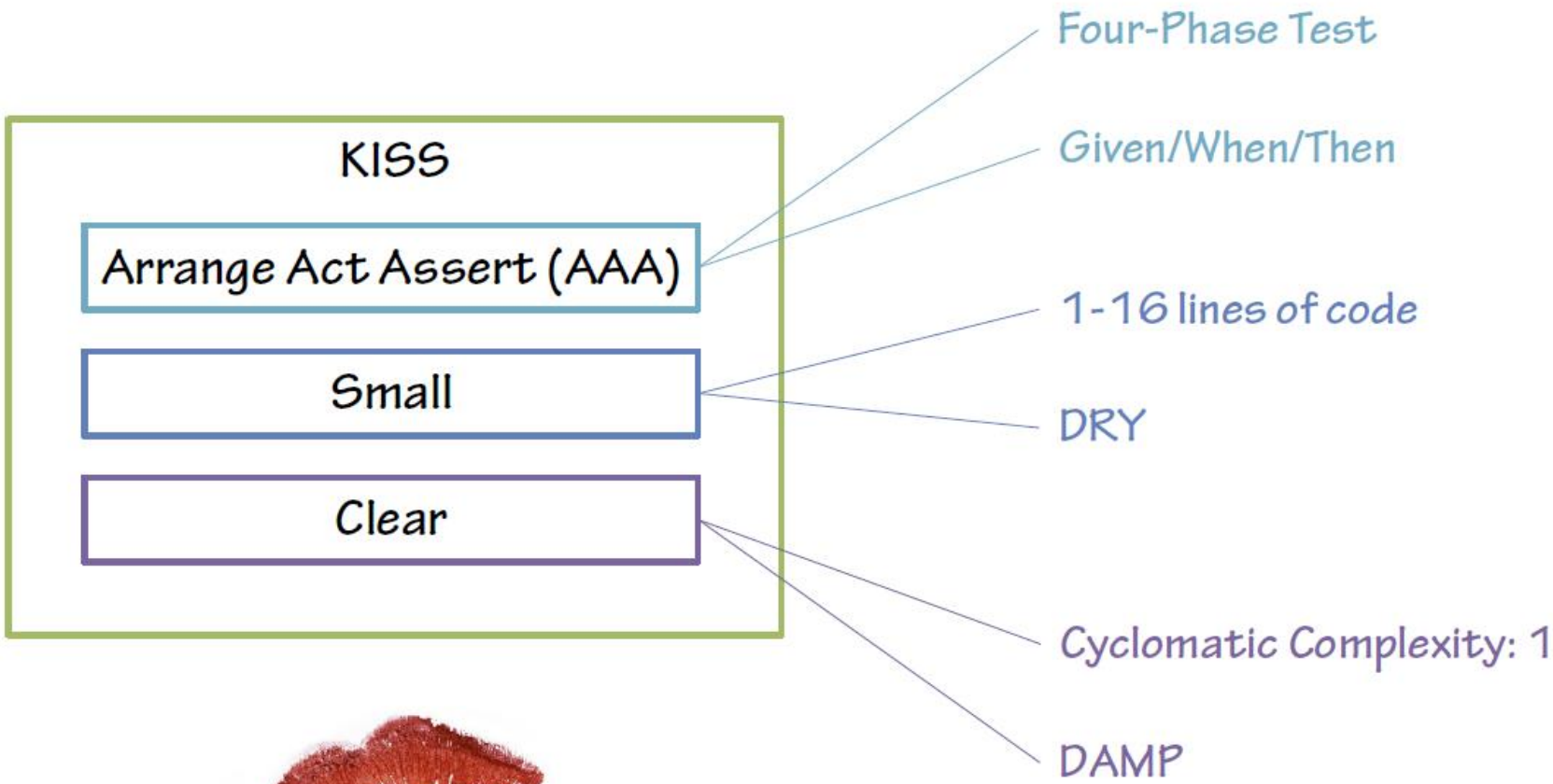
- Atomic
- Deterministic
- Repeatable
- Order Independent & Isolated
- Fast
- Easy to Setup

TESTING STRUCTURE

- AAA
  - Arrange
  - Act
  - Assert

```
[TestFixture]
public class TellerTests
{
    [Test]
    public void CheckBalance_WithNoTransactions_Returns0Balance()
    {
        var teller = new Teller();

        var accountBalance = teller.CheckBalance();

        const int expectedBalance = 0;
        Assert.That(accountBalance, Is.EqualTo(expectedBalance),
            "Empty account should return a 0 balance.");
    }
}
```

READABLE TESTS



NAMING

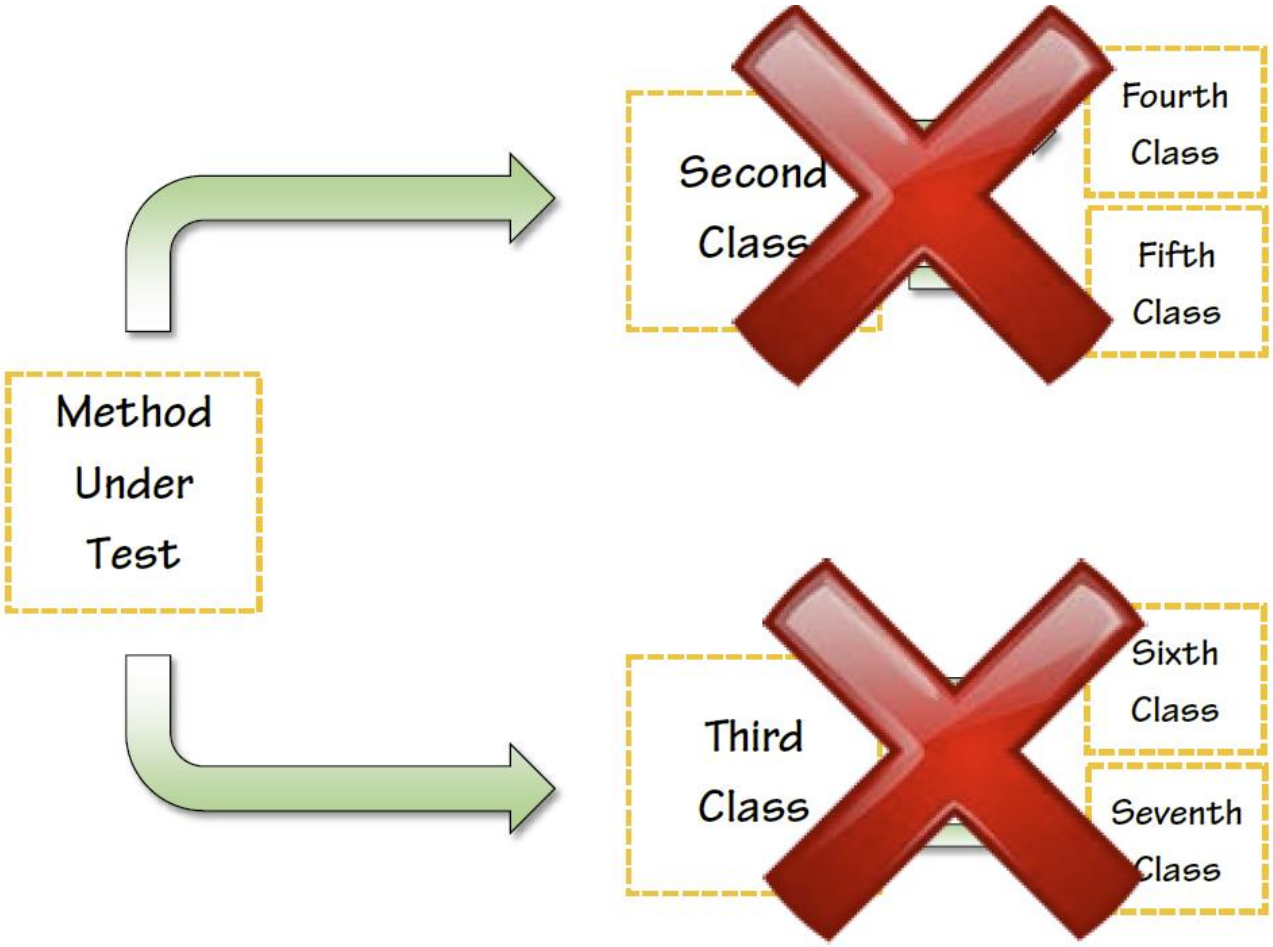
```
namespace BankManager.Tests
{
    [TestFixture]
    public class TellerTests
    {
        [Test]
        public void CheckBalance_WithNoTransactions_Returns0Balance()
        {
            var teller = new Teller();

            var accountBalance = teller.CheckBalance();

            const int expectedBalance = 0;
            Assert.That(accountBalance, Is.EqualTo(expectedBalance),
                "Empty account should return a 0 balance.");
        }
    }
}
```

{ProjectName}.Tests

{ClassName}Tests

{MethodName}_{StateOfTests}_{ExpectedResult}

A COMMON PROBLEM WITH UNIT TESTS



MOCKING

- Methods under test often leverage dependencies
- Testing with dependencies creates challenges
  - Live database needed
  - Multiple developers testing simultaneously
  - Incomplete dependency implementation
- Mocking frameworks give you control

MOCKING OPTIONS

- Implement the mocked functionality in a class
  - This approach is tedious and obscure
- Leverage a mocking framework
  - Avoid class creation
  - Leverages the proxy pattern
- Many tools – we'll use Moq