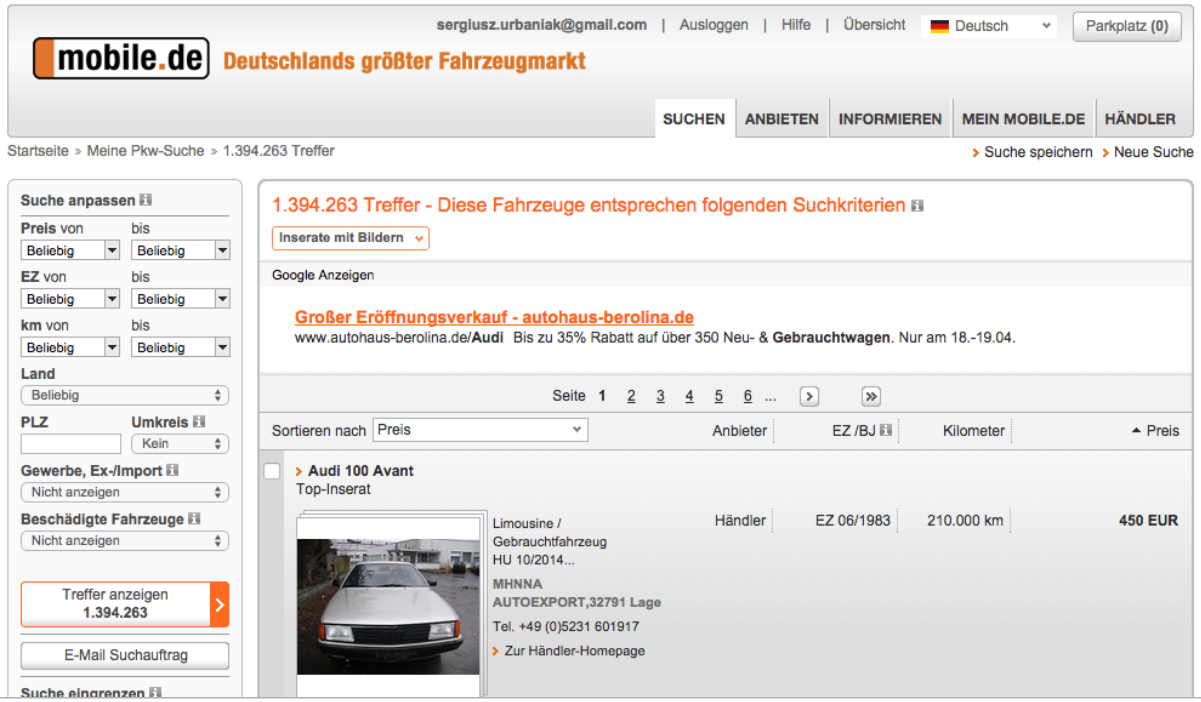
# Programming the web like a game

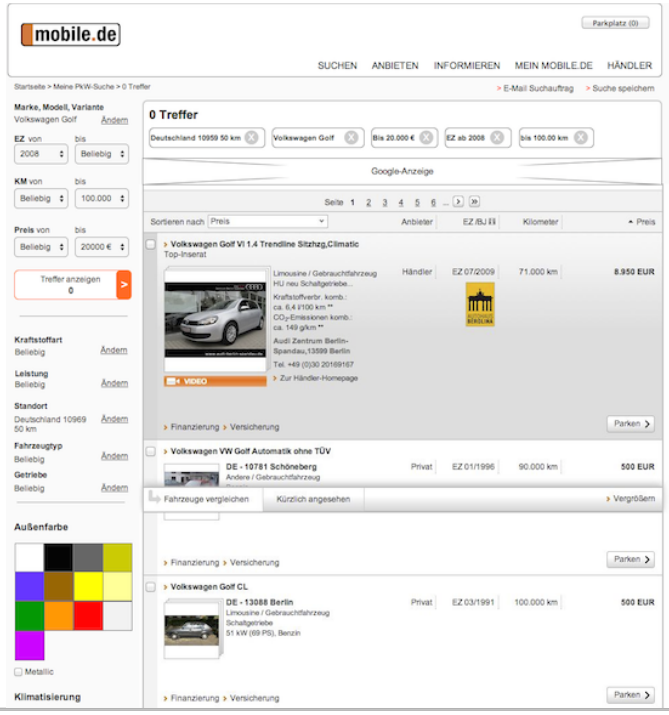
## Applying ECS at mobile.de.

Sergiusz Urbaniak

## "Old" public search



## Envisioned publich search




## New public search (aka PubSe)

suchen.mobile.de/fz/search.html (http://suchen.mobile.de/fz/search.html)

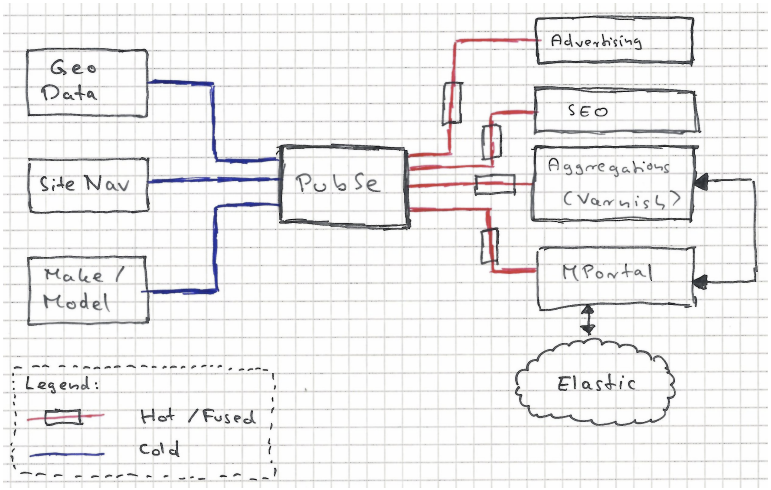
Goals:

- Break the monolith
- Sane service architecture (say microservices one more time...)
- Stack "not too far" from the existing one
- Move towards a twelve-factor-app [1]

Biggest goal: KISS (Keep it Super Simple)

[1] http://12factor.net/ (http://12factor.net/)

## System boundaries




Most of the "logic" is outside PubSe, how hard can it be?

## Attempt #1

*KISS* (Keep it Super Simple)

1. Controller → Service → RemoteSearch

2. Soy ← Map<String, ?> ← JSON

Internal "service" classes:

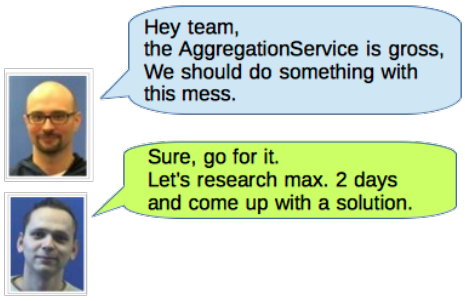
- **SearchService**: Maps request parameter and invokes our search backend
- **RefDataService**: Loads reference data
- **AggregationService**: Aggregates data
- **Magellan**: Renders the "magellan"
- **Dutchman**: Renders the "dutchman"
- **PaginationViewModelBuilder**: Builds paging model
- ...

## Problems with attempt #1

- Services have too many responsibilies, have too many LOC
- Concrete concerns (i.e. the "logic" for handling `minPrice/maxPrice` parameter) are spread all over the place

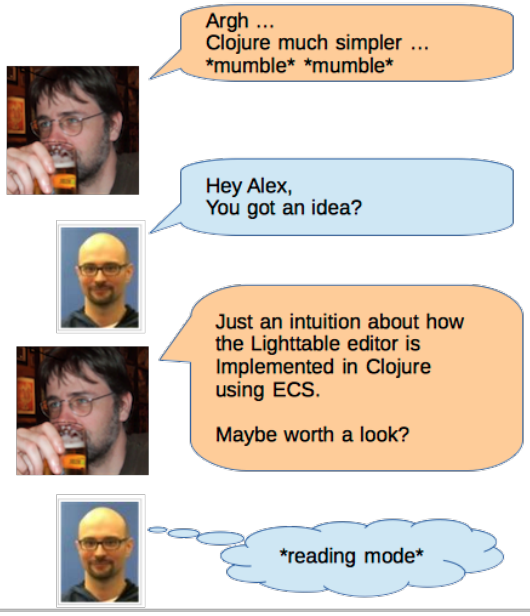
search.html?...&minPrice=5000&maxPrice=10000 (http://suchen.mobile.de/fz/search.html?isSearchRequest=true&minPrice=5000&maxPrice=10000)

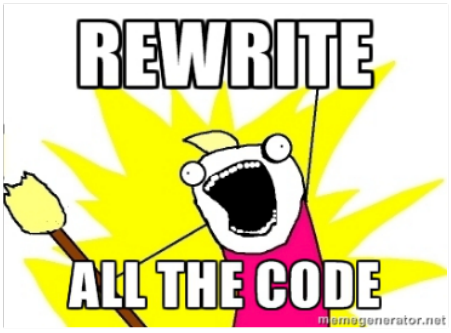

## PubSea team@retro

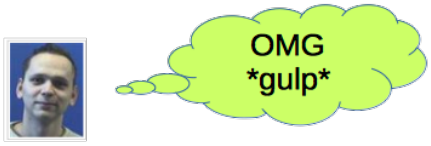


## Back at the desk

Light Table: Chris Granger - The IDE as a value (http://www.chris-granger.com/2013/01/24/the-ide-as-data/)



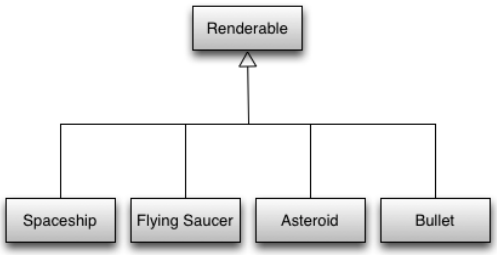
## Programming is hard, let's talk about games!


Source www.arcade-museum.com (http://www.arcade-museum.com/game_detail.php?game_id=7146)

Ingredients

- Spaceship, Flying Saucer
- Asteroids, Bullet
- Static objects, Force Field

## A classical OO model

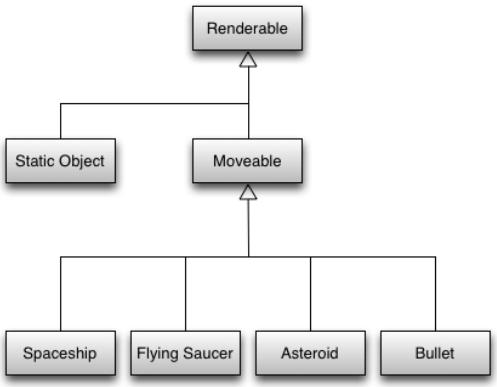



Source Richard Lord - What is an entity framework (http://www.richardlord.net/blog/what-is-an-entity-framework)

Nice, but ...

- the spaceship, flying saucer and asteroid are all movable too!
- Plus there are static objects

## A classical OO model

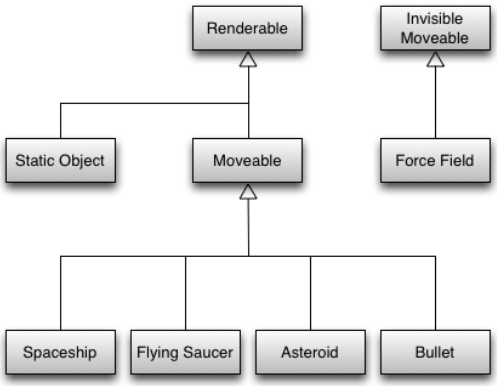



Source Richard Lord - What is an entity framework (http://www.richardlord.net/blog/what-is-an-entity-framework)

Nice, but ...

- What if we want an invisible game object? (Moveable but not Renderable)

## A classical based OO model




Source Richard Lord - What is an entity framework (http://www.richardlord.net/blog/what-is-an-entity-framework)

- Our type hierarchy becomes complex
- We're missing multiple inheritance

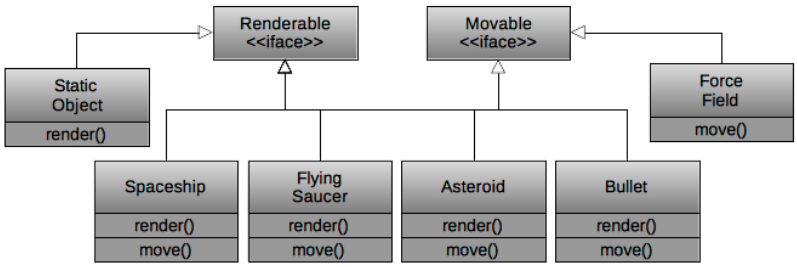
## Hah, we have interfaces!

Problem:

- `render()` methods are redundant

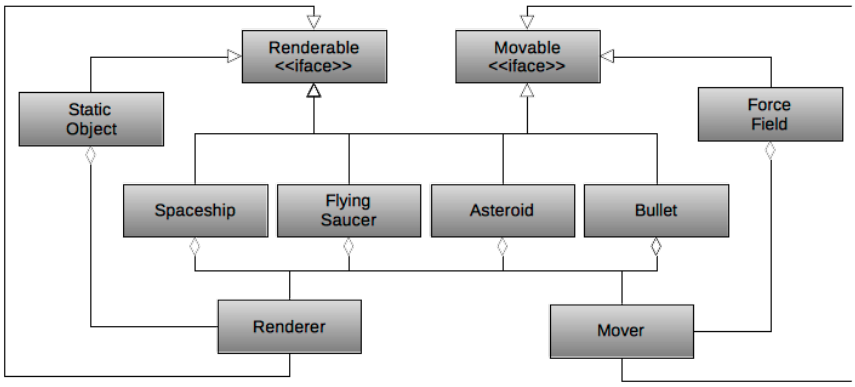
Solution:

- Favor composition over inheritance




## An interface based OO model

Hmm ... that's quit a lot of arrows ...

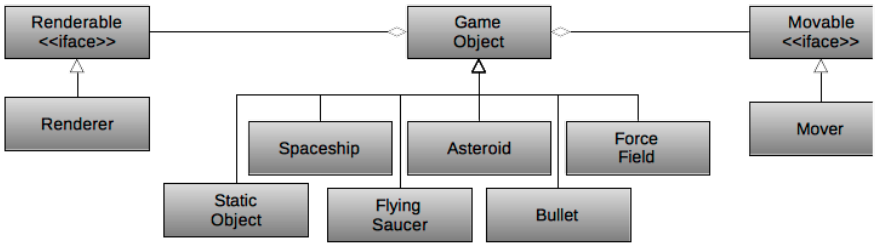


## Patterns, patterns everywhere

Let's use bridges!

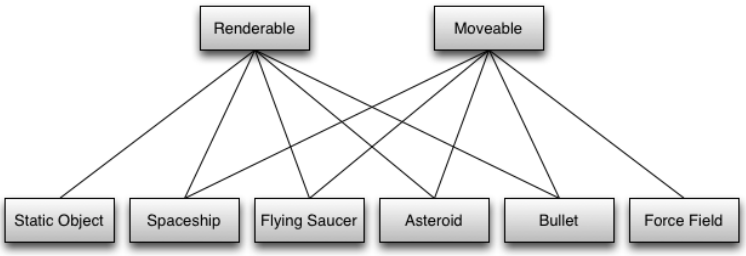
"decouple an abstraction from its implementation so that the two can vary independently"

But wait .. we have **three** hierarchies now!




## A fresh start ...

- Stop shuffling around hierarchies ...
- Renderable & Moveable are **not** superclasses
- Renderable & Moveable are simply data structures (**components**)
- The spaceship, asteroids, flying saucers, the bullet are **entities**




Source Richard Lord - What is an entity framework (http://www.richardlord.net/blog/what-is-an-entity-framework)

## Components, Entities

```
struct Moveable {
    Integer x, y
    Float velocity
}
```

```
struct Renderable {
    Bitmap mesh
}
```

```
struct Entity {
    UUID id
}
```

## World

Who stores the association between a **component** and an **entity**?

→ the **world**

The world should be considered the "database" of our engine:

- Add new entities
- Query for entities having components

Examples:

- Hello world, give me all entities having a `Renderable` component
- Hello world, create a new entity having a `Moveable` component
- Hello world, please update a given entity with a new `Renderable` component instance

## Interfaces/Classes

```
public interface World {
    Entity newEntity(Component... cs);

    <C extends Component> void createOrUpdate(Entity e, C component);

    <C extends Component> Optional<C> component(Entity e, Class<C> clazz);

    <C extends Component, T> Stream<T> map(Class<C> clazz, BiFunction<Entity, C, T> func);
}
```

```
public final class Entity {
    Entity() {}
}
```

```
public interface Component {}
```

```
public interface Sys {
    void apply(World w);
}
```

## Systems

Who speaks to the **world**?

→ **systems**

Systems implement the logic for an application. Given the asteroid example we can identify the following systems:

- **InputSystem**

1. Reads input devices (keyboard, mouse, joystick)
2. Updates all entities having a `Moveable` component

- **RenderSystem**

1. Iterates over all `Renderable` entities and renders them on the screen

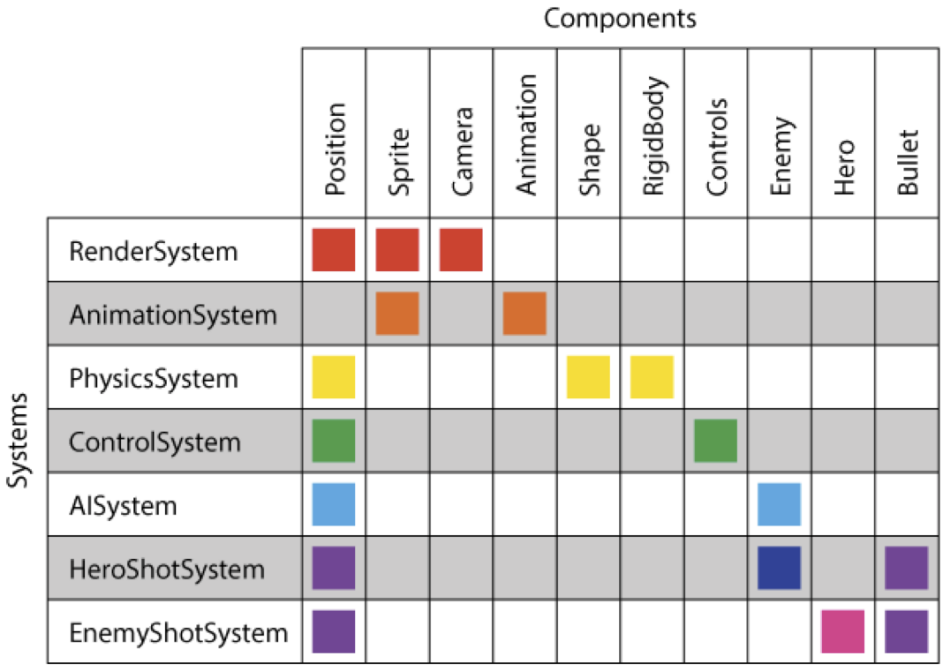
## Systems

```
class InputSystem {
    function apply(World w) {
        // get input from joystick
        newX, newY = ...

        w.forEach(Moveable, (entity, movable) -> {
            w.update(entity, new Moveable(newX, newY))
        })
    }
}
```

```
class RenderSystem {
    function apply(World w) {
        w.forEach(Renderable, (entity, renderable) -> {
            graphicsCard.display(renderable.mesh)
        })
    }
}
```

## The big picture

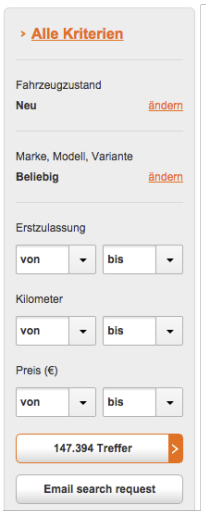


## Famous ECS engines

The game industry made a big move towards ECS ~14 years ago. Famous ECS engines:


Artemis ECS (https://github.com/junkdog/artemis-odb)

## PubSe entities?




Condition

MakeModel

FirstRegistration

Mileage

Price


Fuel

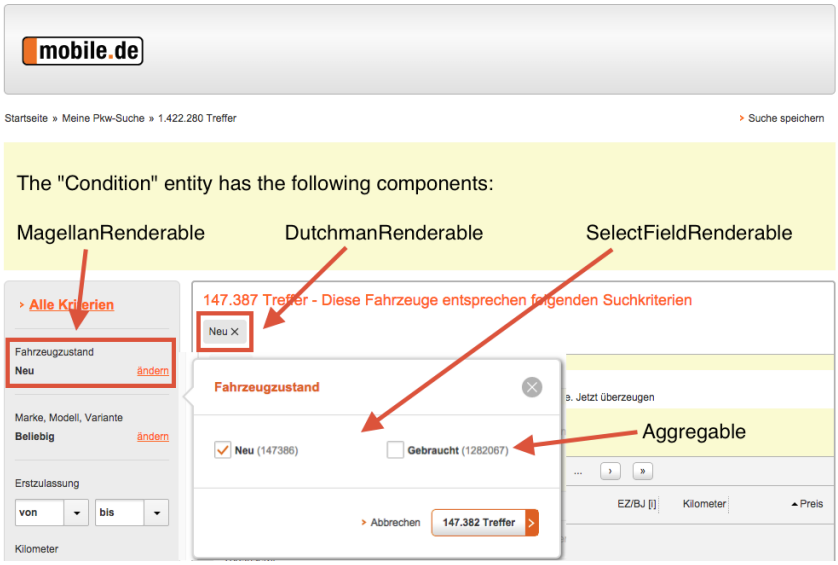
Power

AmbitAddress
AmbitCountry

Category

Transmission

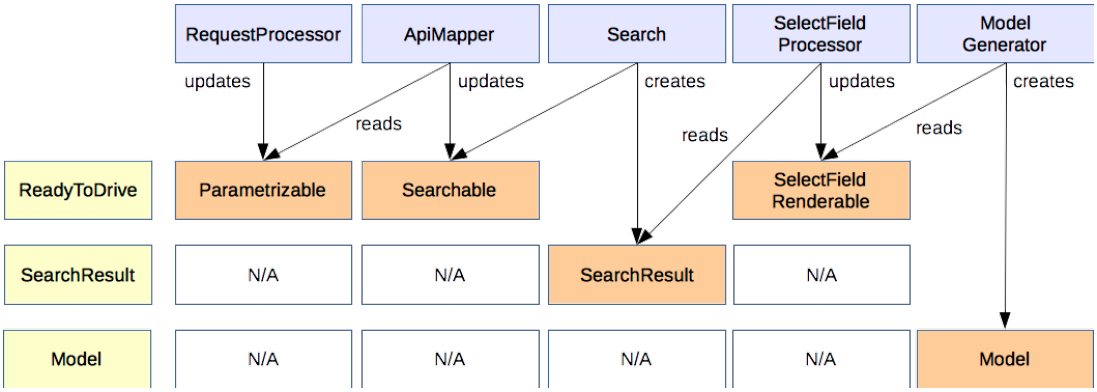
## PubSe components?




## PubSe systems?

- **RequestProcessor**: Responsible for parsing the request in flight. Updates `Parametrizable` components
- **ApiMapper**: Responsible for value mapping for the mportal API. Updates `Searchable` components
- **Search**: Responsible for invoking the mportal API. Reads `Searchable` components, Creates an entity with a `SearchResult` component.
- **SelectFieldProcessor**: Responsible for updating `SelectFieldRenderable` components.
- **DutchmanProcessor**: Responsoble for updating `DutchmanRenderable` components.
- **MagellanProcessor**: Responsoble for updating `MagellanRenderable` components.

## A partial PubSe ECS matrix




## A few numbers

In PubSe we have ...

- ~40 systems
- ~50 entities
- ~50 components

## A LEGO backend

One can stitch systems and/or worlds differently together.

Examples:

- **Search**: Needs the whole she-bang
- **HitCounter**: Doesn't need aggregations, so leave this system out
- **Tomcat**: Is blocking, thus can have a blocking async system
- **Netty**: Is non-blocking, thus needs a non-blocking async system
- **Logging**: A logging world can trace an event log
- **Tracing**: Every action in the world is recorded an can be inspected in a special trace view

## Final thoughts

Peter Norvig:

- Objects are state data with attached behavior
- Closures/Functions are behaviors with attached state data

Rob Pike:

- Sometimes data is just data and functions are just functions

ECS falls into the former category

## References


- Hall, Daniel. ECS Game Engine Design (http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1138&context=cpesp)
- http://www.richardlord.net/blog/what-is-an-entity-framework
- http://entity-systems.wikidot.com/
- http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/
- http://vasir.net/blog/game-development/how-to-build-entity-component-system-in-javascript
- http://blog.lmorchard.com/2013/11/27/entity-component-system/


## Thank you

Sergiusz Urbaniak


Source dilbert.com (http://dilbert.com/strip/2000-08-16)

## Thank you

Sergiusz Urbaniak