ThoughtWorks®

Micro Services

Java, the Unix way

ThoughtWorks®

Principal Consultant

Senior Engineer

story teller

@boicy http://bovon.org jalewis@thoughtworks.com

WHAT I DID LAST SUMMER

Or how we designed and built a Resource Oriented, Event Driven System out of applications about 1000 lines long...

In the beginning…

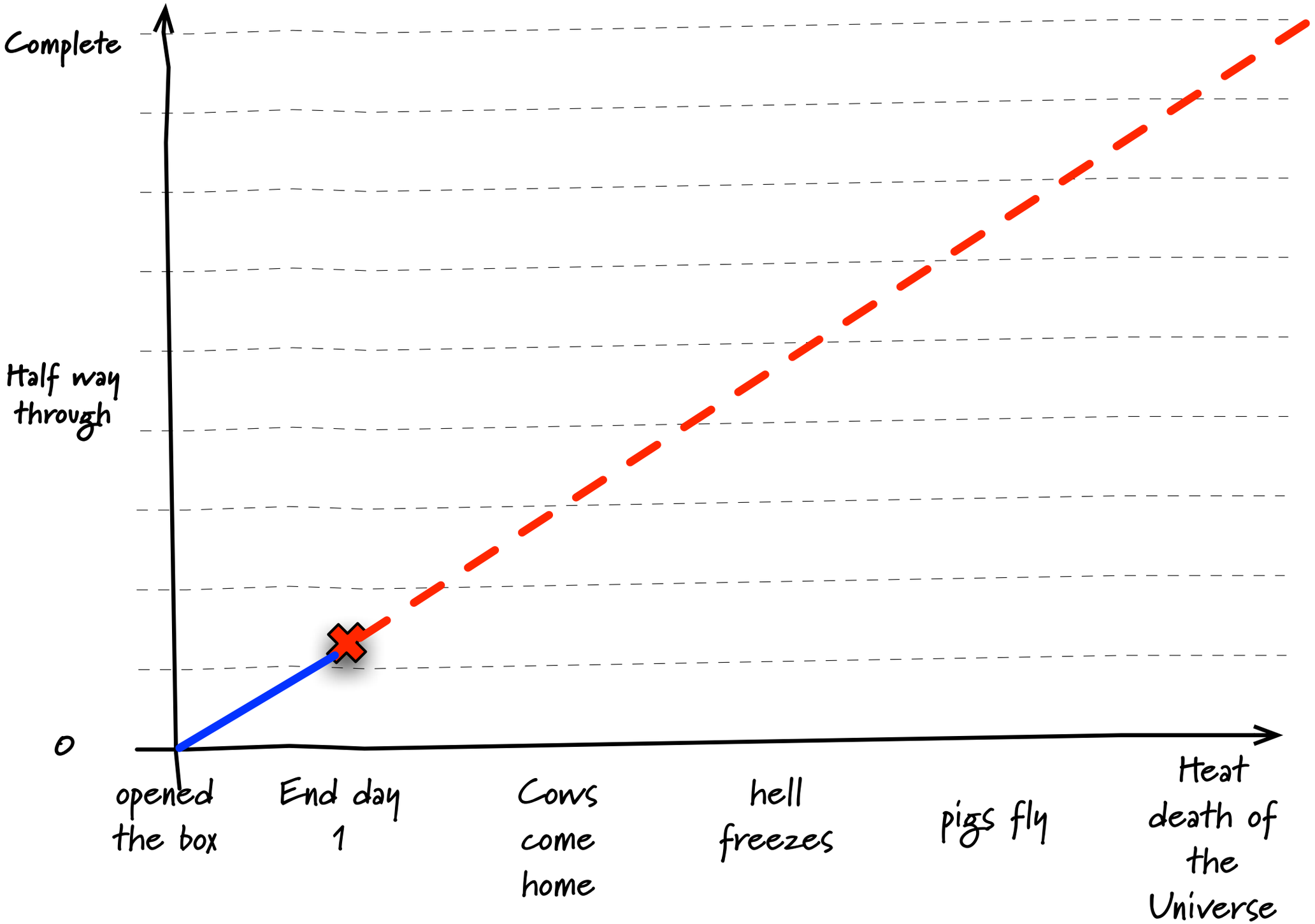
- There was a new product being developed by an organisation in London

- The organisation had gathered their list of high level requirements

- And they asked ThoughtWorks if we could help them design and build it…

So we took a look at their requirements

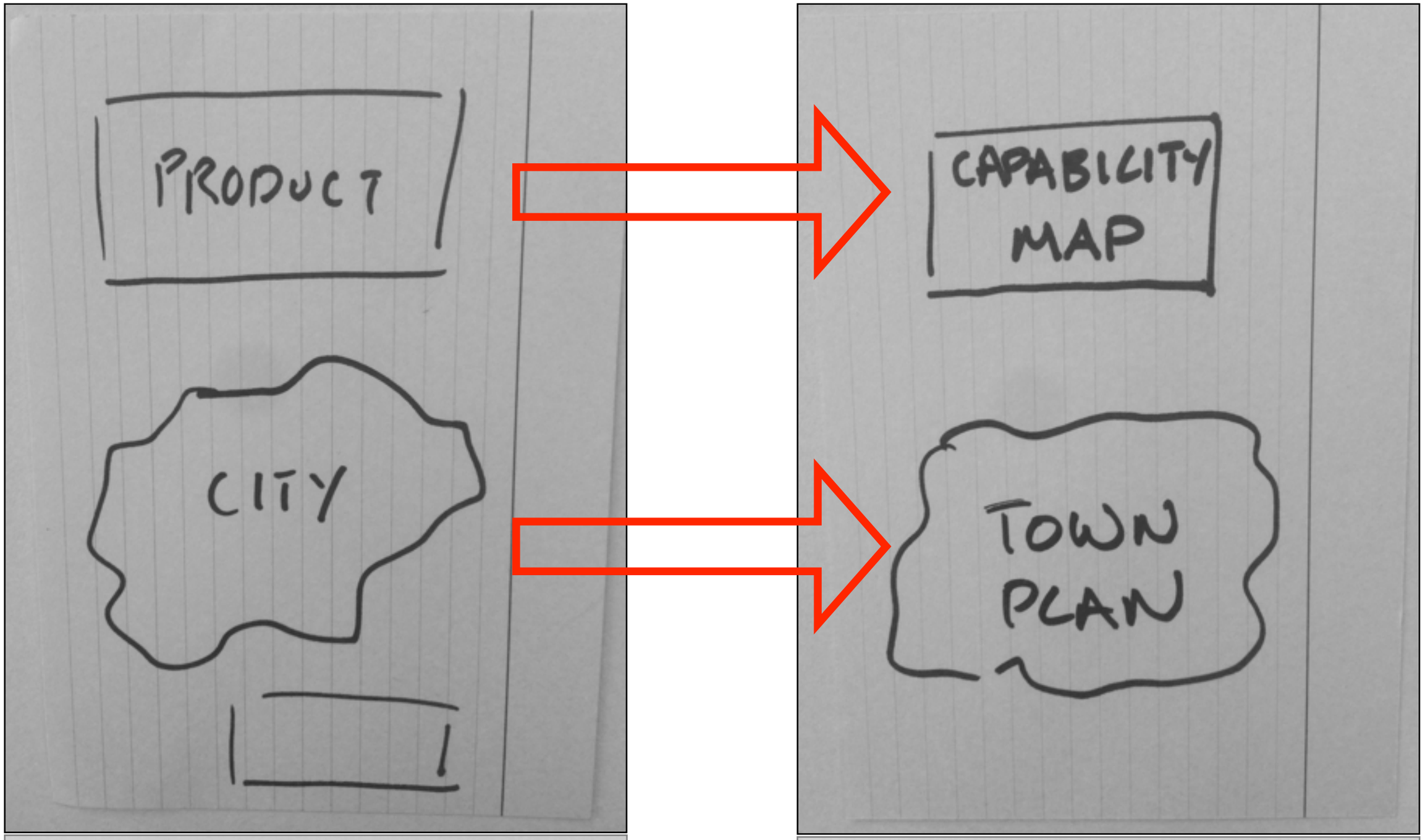
- Me and my mates at ThoughtWorks
- Worked out to be about a lot of points worth of User Stories

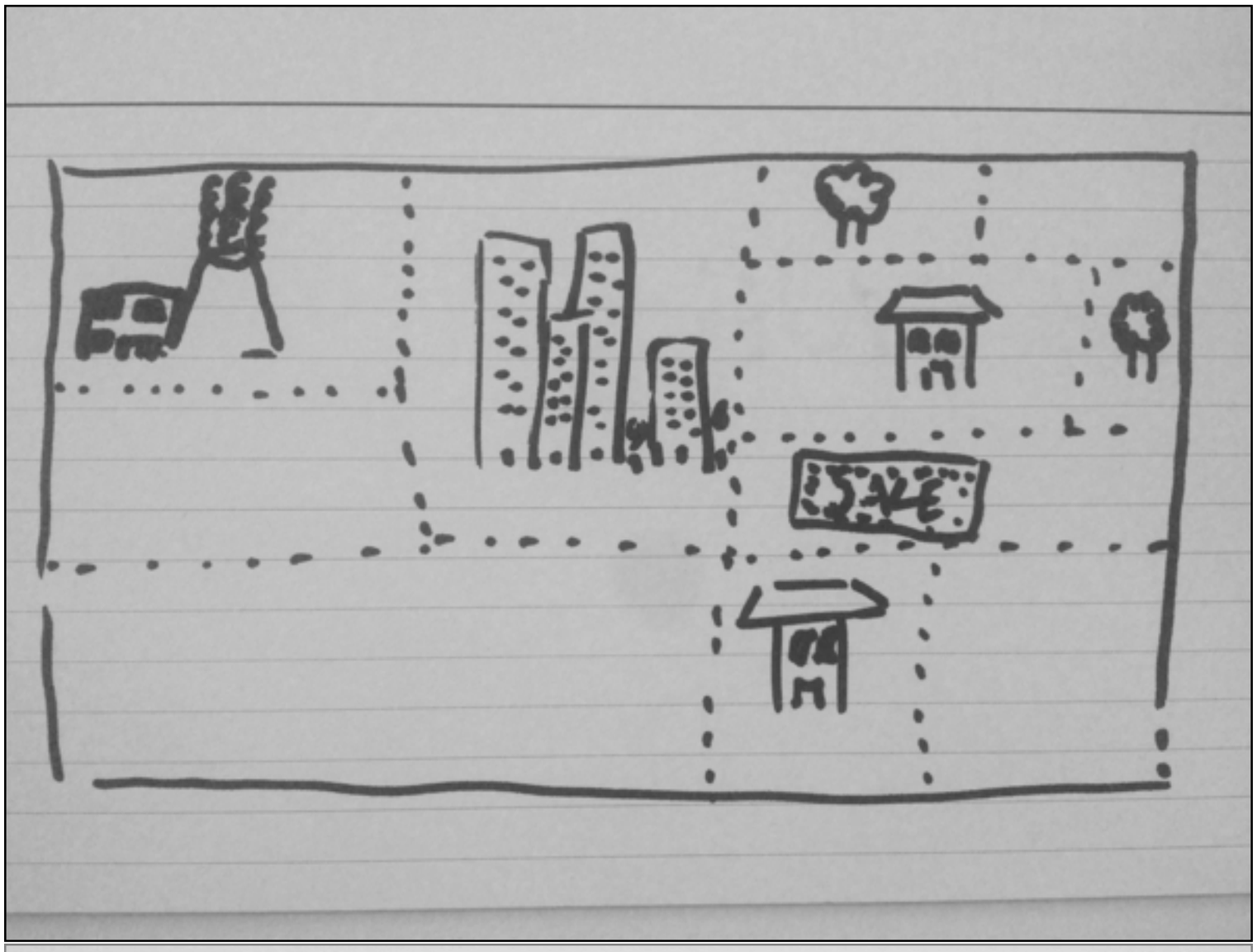

Complete
Half way through
0
opened the box
End day 1
Cows come home
hell freezes
pigs fly
Heat death of the Universe

Tip 1

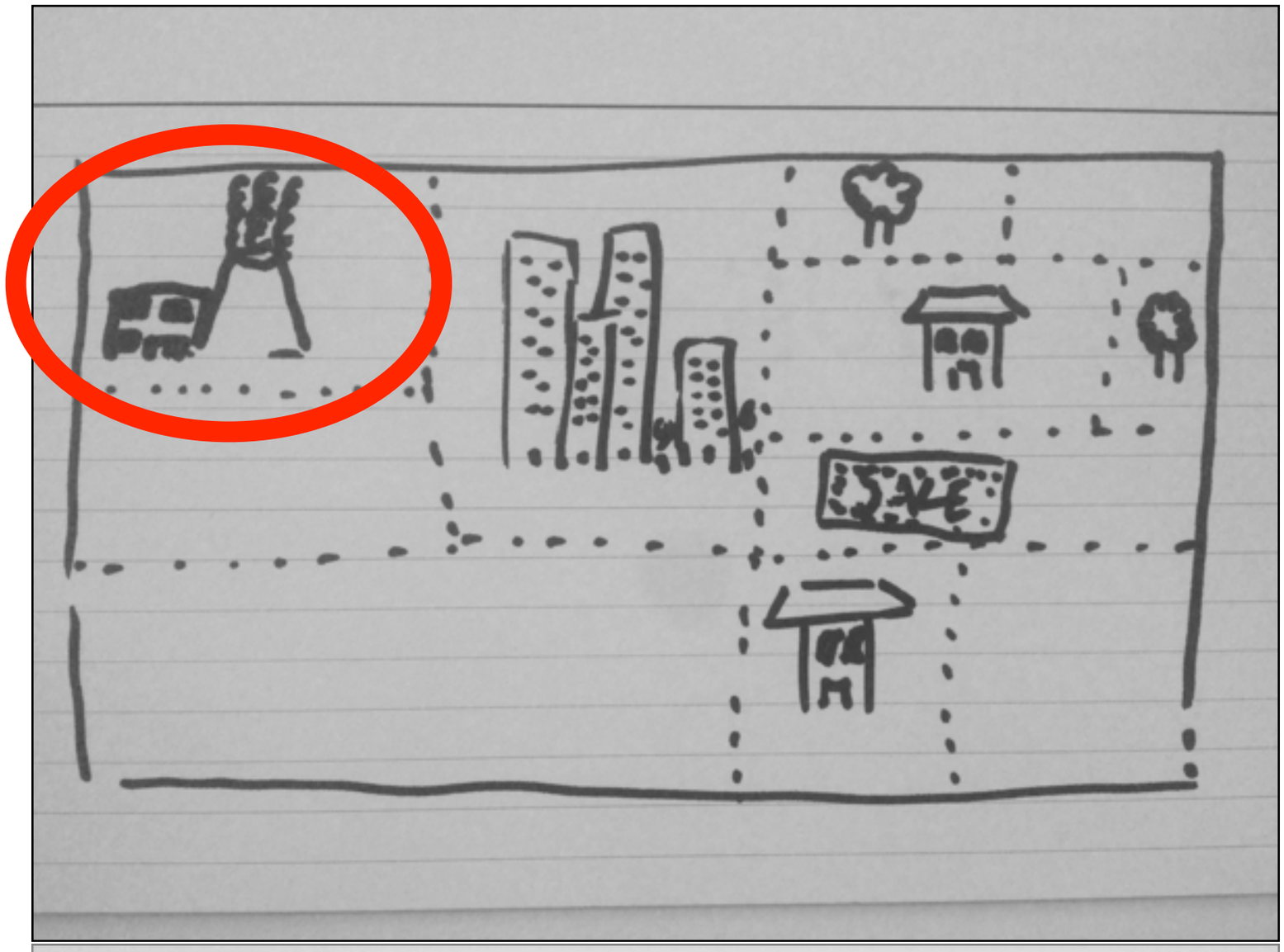
Divide and conquer

break down complex problems into smaller chunks that can be solved individually

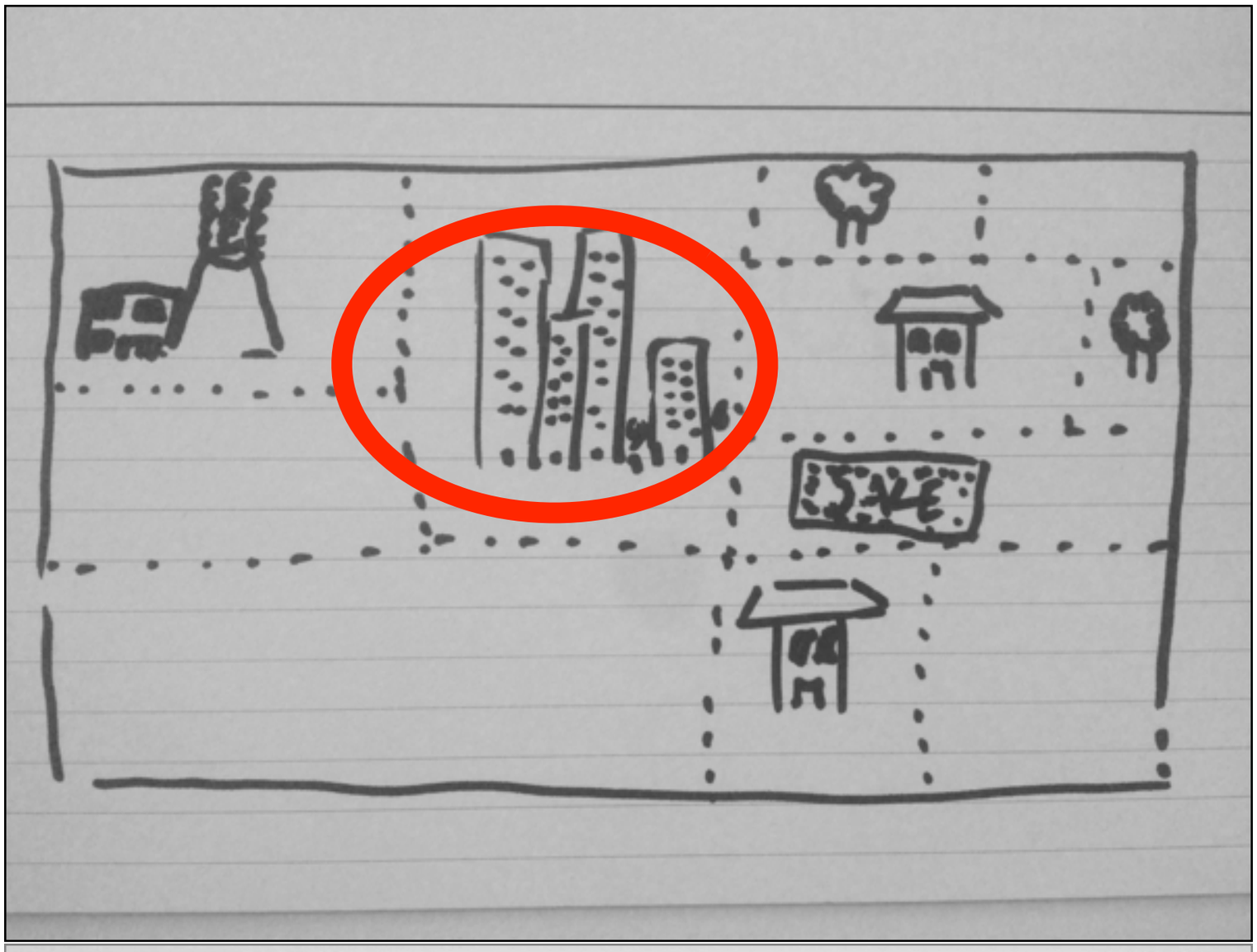

PRODUCT
CITY
CAPABILITY MAP
TOWN PLAN

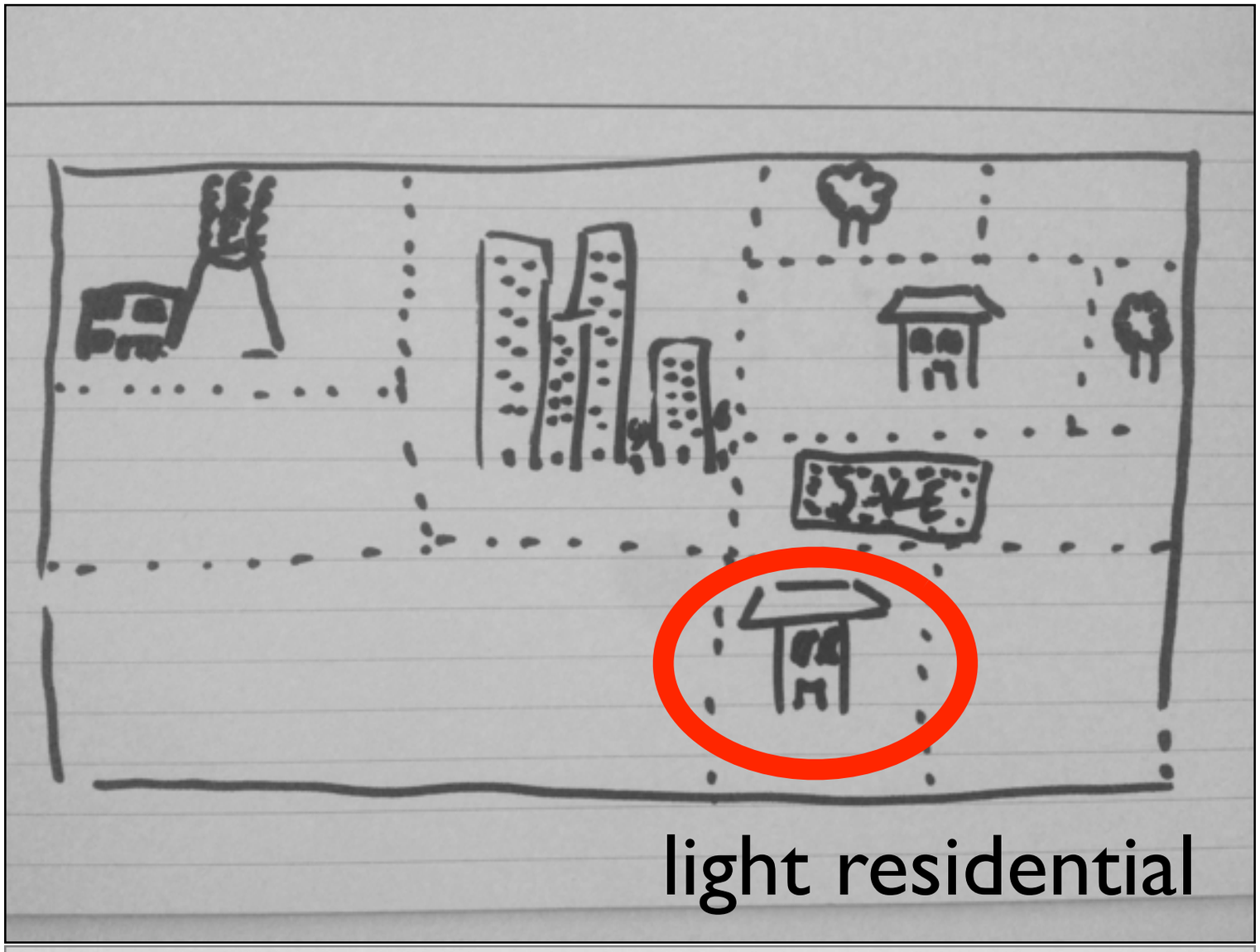


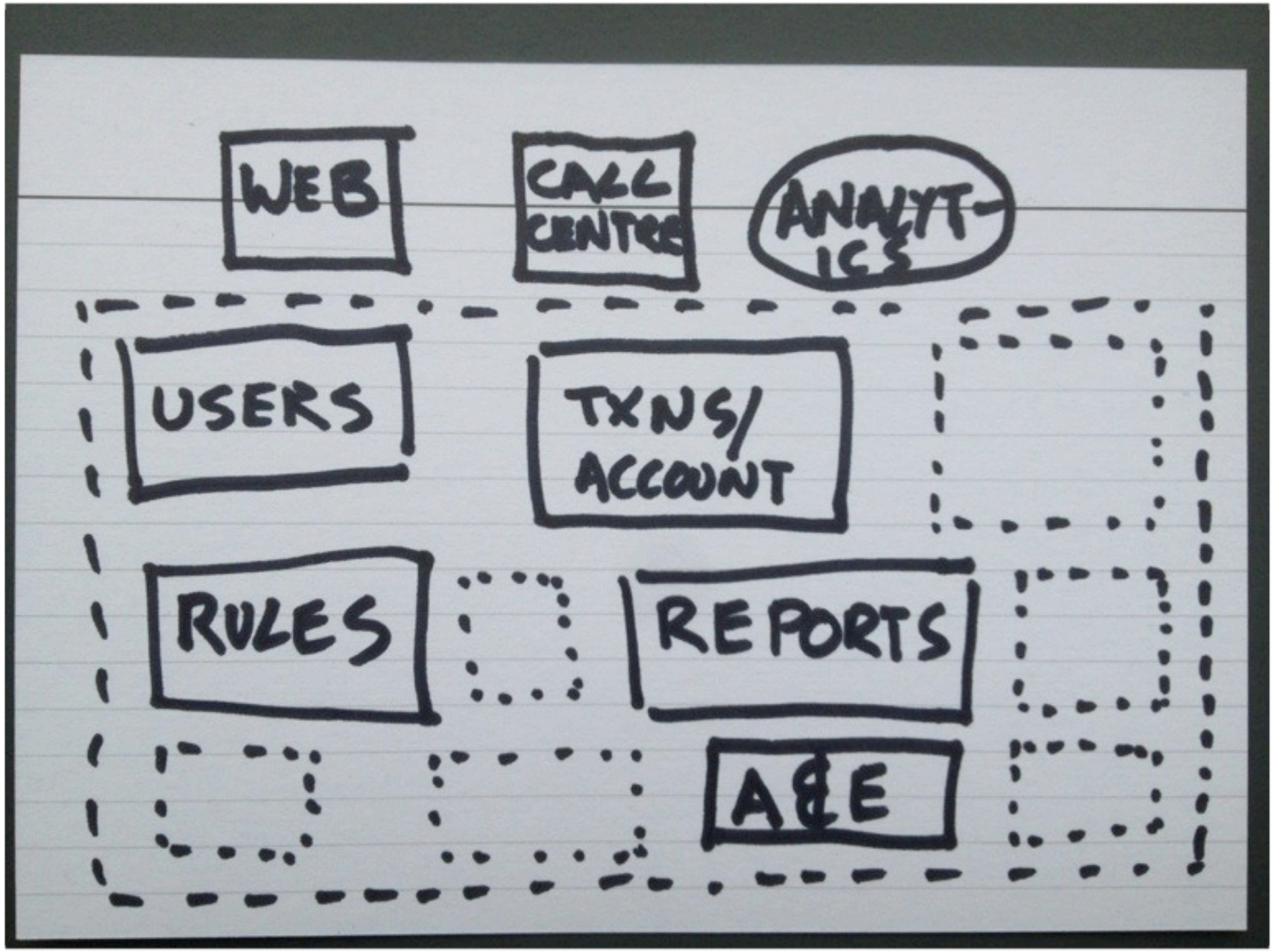
heavy industrial



commercial



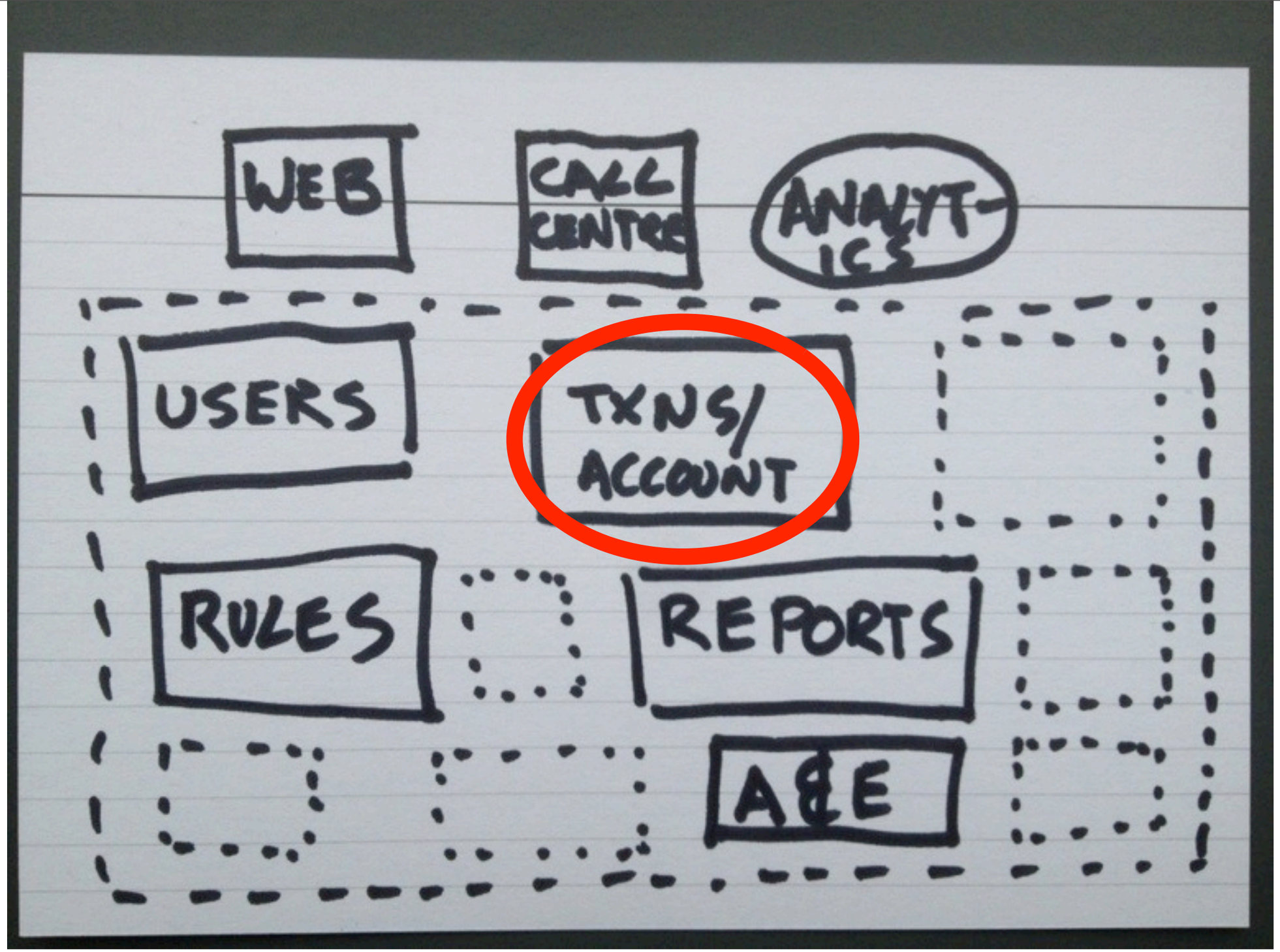

SALE
light residential



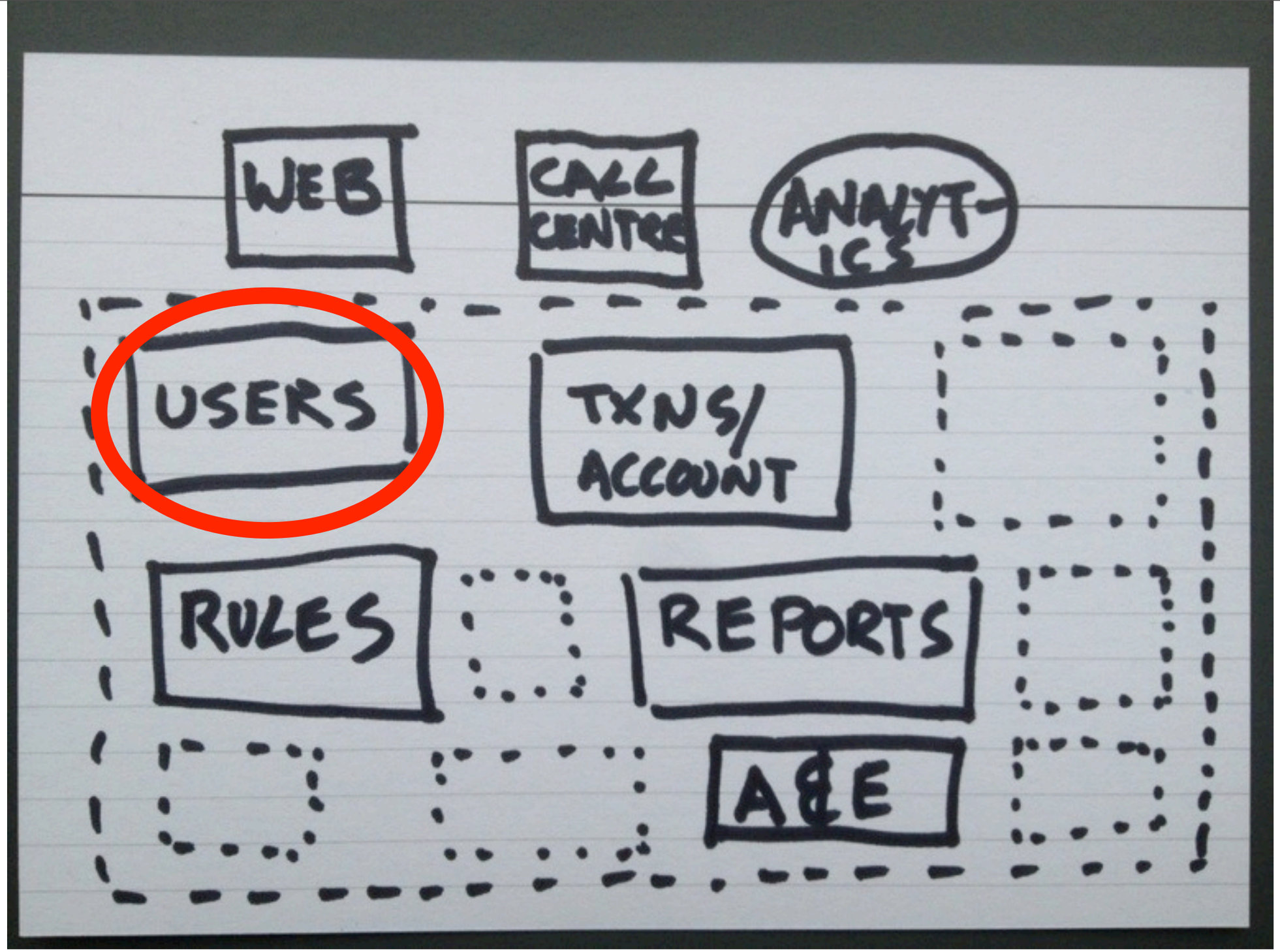
Each small box represents a capability,

composed of one or more services

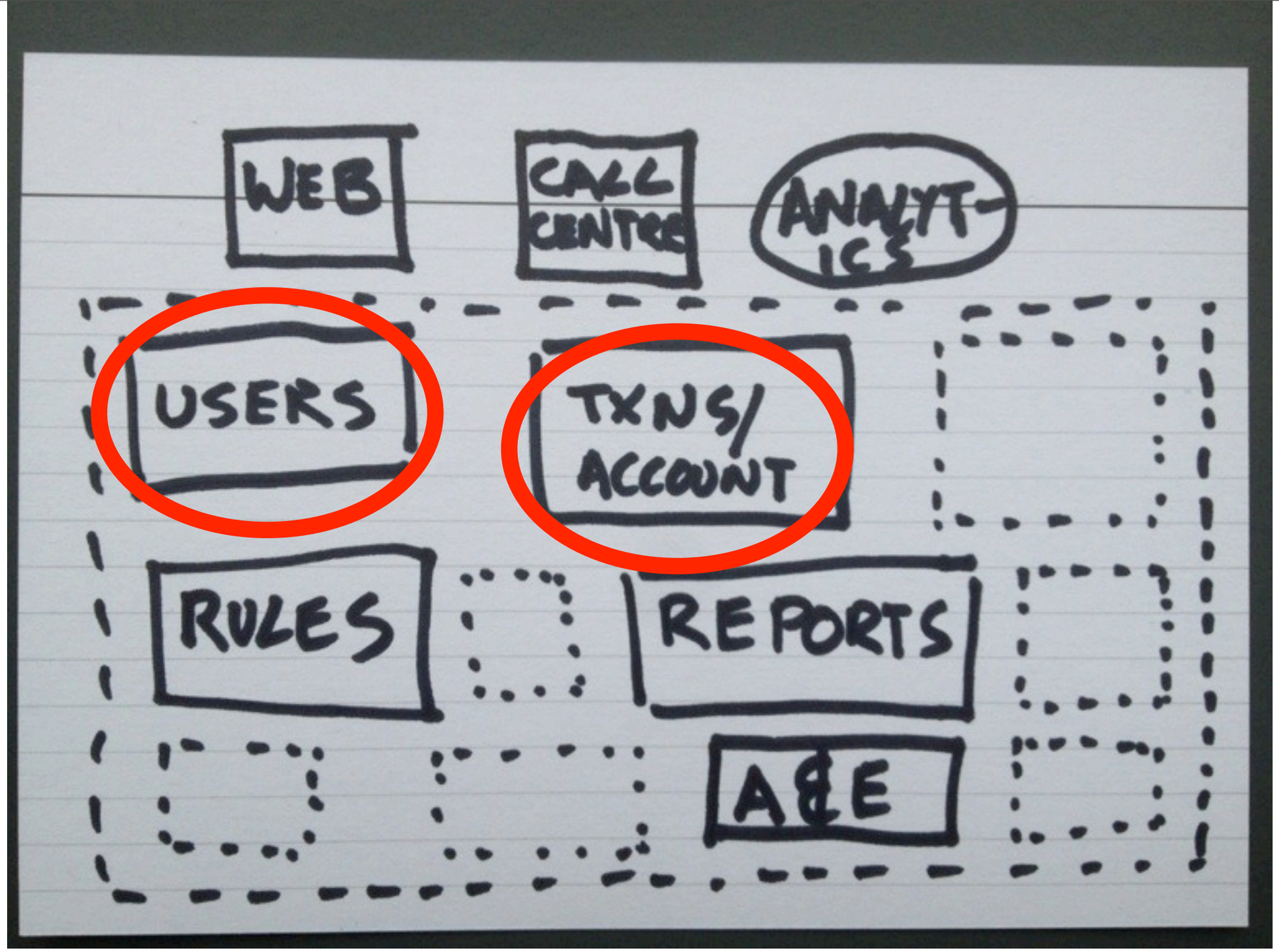
And there were some, umm, interesting non-functional requirements too



1000TPS, 99th percentile latency of < 2 seconds



Support a user base of 100 million active customers



Support bulk loads of 30 – 90 million records nightly and keep them for six months

(16,200,000,000 records)

Did I mention PCI Level 1?

Finally, this is a product build.

So it needed to be modular / <cough> “infinitely configurable”

The product need to to be…

- Performant
  - fairly high throughput both transactional and batch
- Fault tolerant
  - One thing about the cloud, you are designing for failure right?
- Configurable
  - On a per install or SaaS basis
- Portable
  - Fortunately not to Windows…
- Maintainable
  - over multiple versions and years
- Supporting big data sets
  - Billions of transactions available
  - Millions of customers available

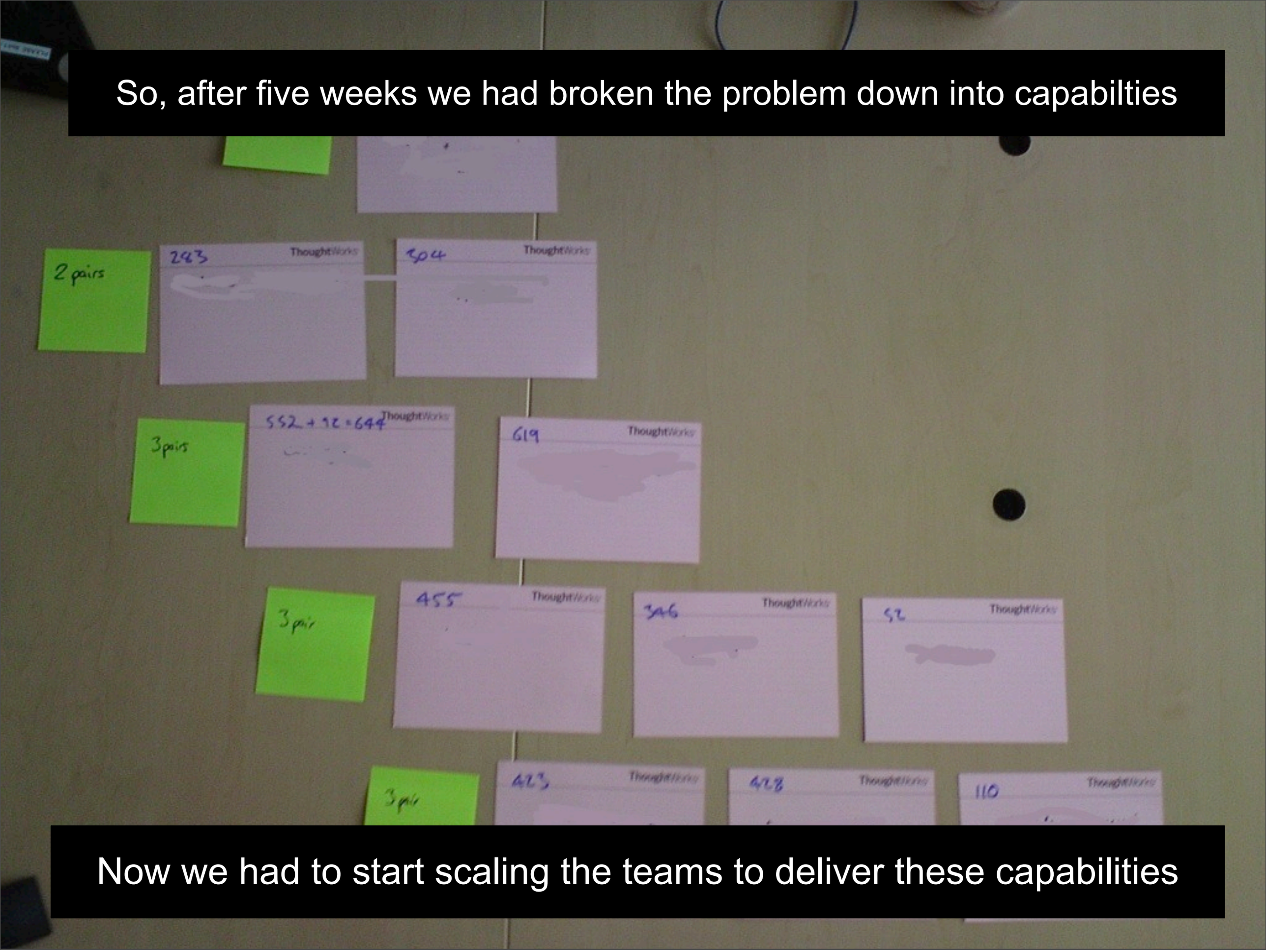
Plus ça change, plus c'est la même chose.

(The more things change the more they stay the same)

- The only way we could hit anything like the timescales required was to scale the programme quickly

- And that meant multiple teams in multiple workstreams

So, after five weeks we had broken the problem down into capabilties



Now we had to start scaling the teams to deliver these capabilities

Tip 2

Use Conway's Law to structure teams

"…organizations which design systems … are constrained to produce designs which are copies of the communication structure of those organizations"

Melvin Conway, 1968

The first business capability - Users

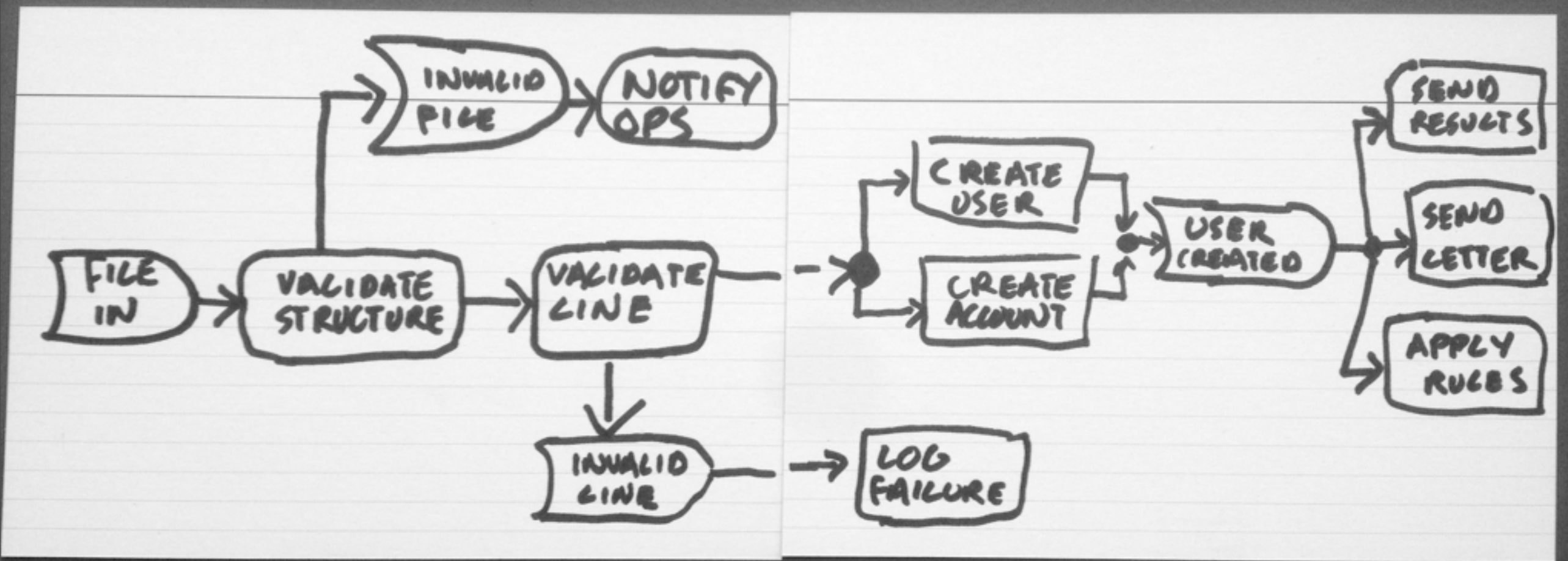
- Responsible for creation and maintenance of users in the system
  - Up to 100 million of them per instance of the product
- Used by many clients with many usage patterns
  - Call centre and website – CRUD
  - Inbound batch files – CRUD x hundreds of thousands per night
- Many downstream consumers of the data
  - Fulfilment systems for example

Tip 3

The Last Responsible Moment

Don’t decide everything at the point you know least

We started with a business process…



and noticed something funny…

events...

* Dan North coined the term Enterprise Night Bus…

ESB*

I know what you are thinking…

Or you could use the web

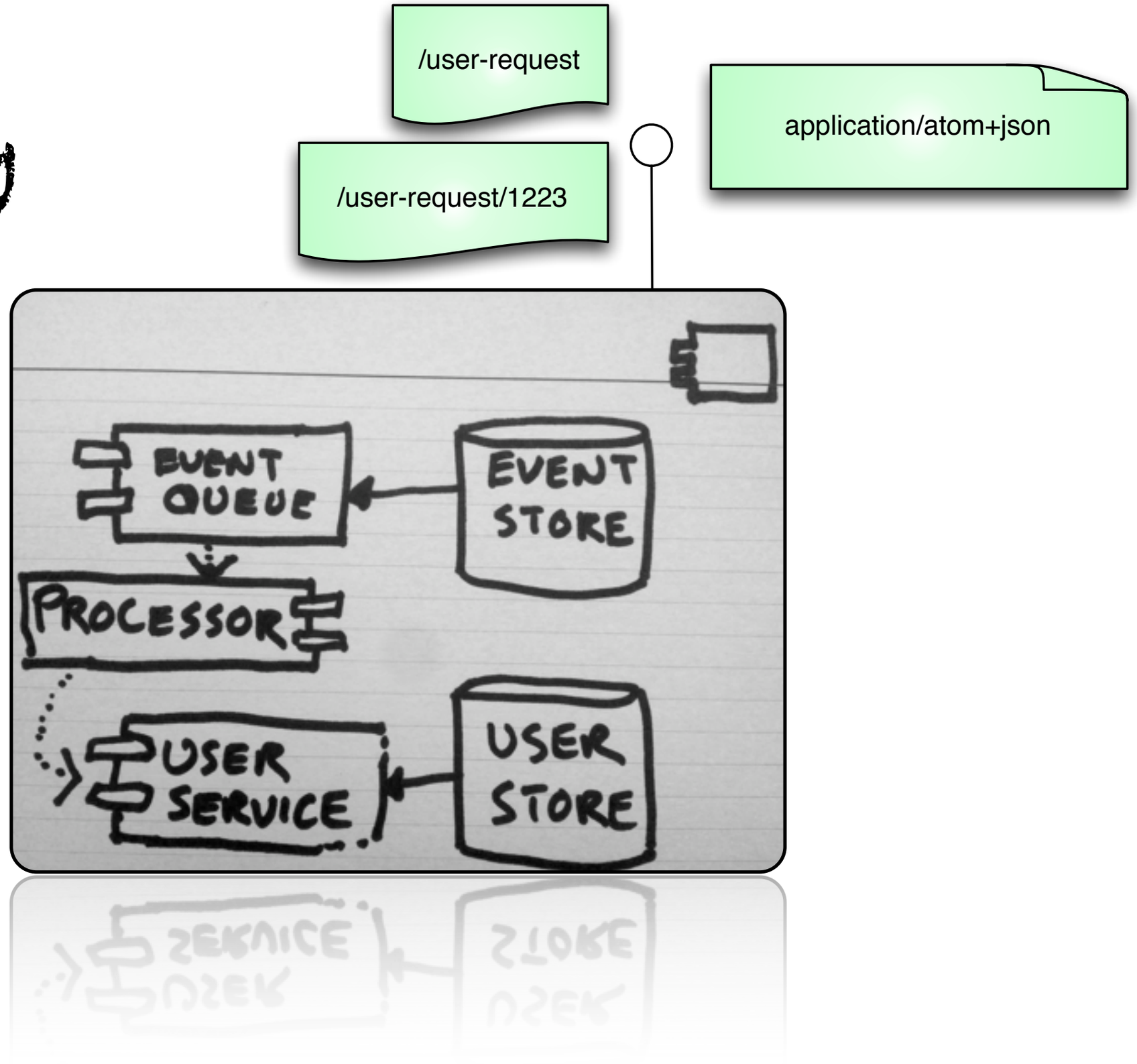
REST in Practice

Tip 4

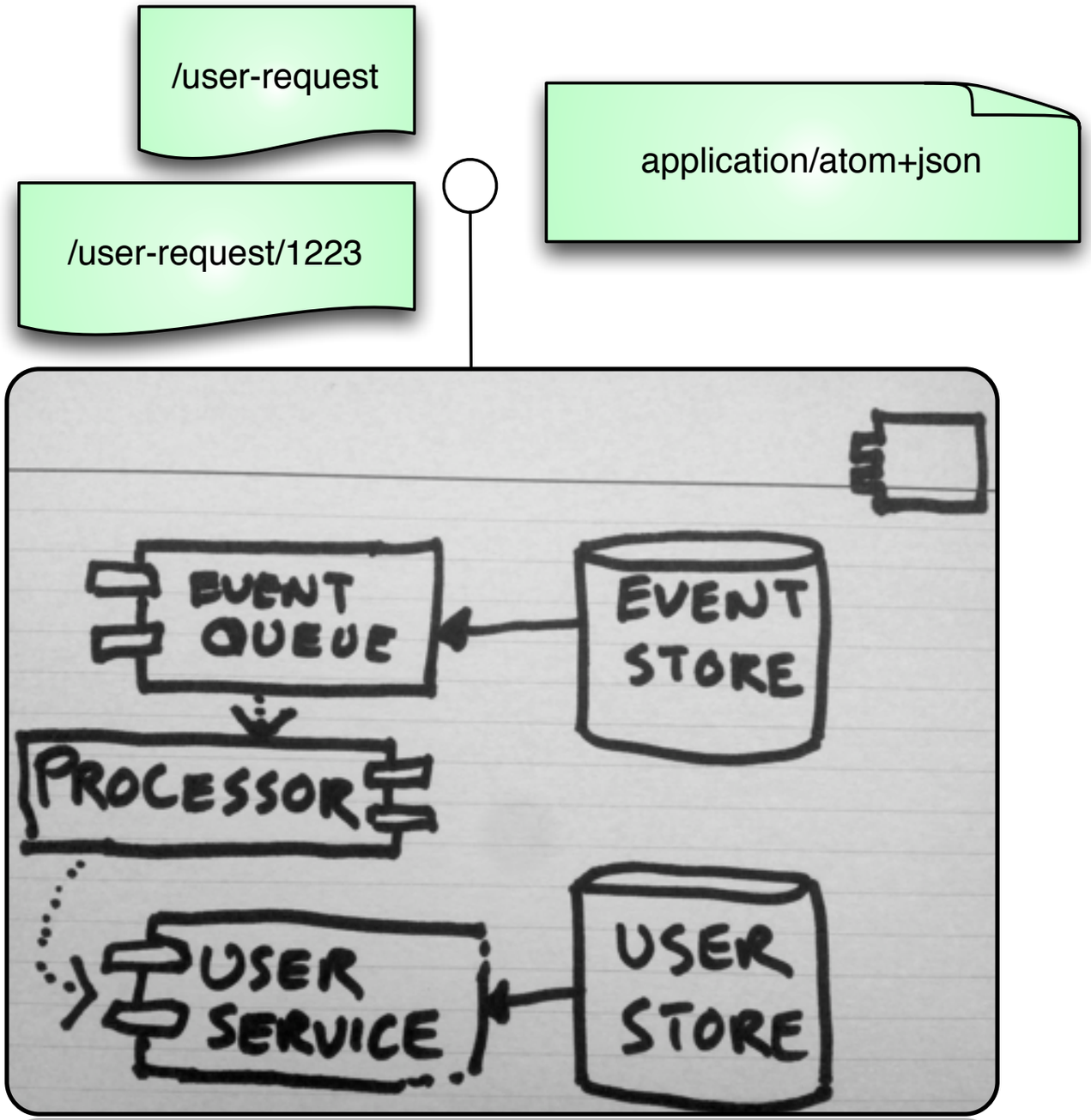
Be of the web, not behind the web

RFC 5023 to be precise

Our Users Capability

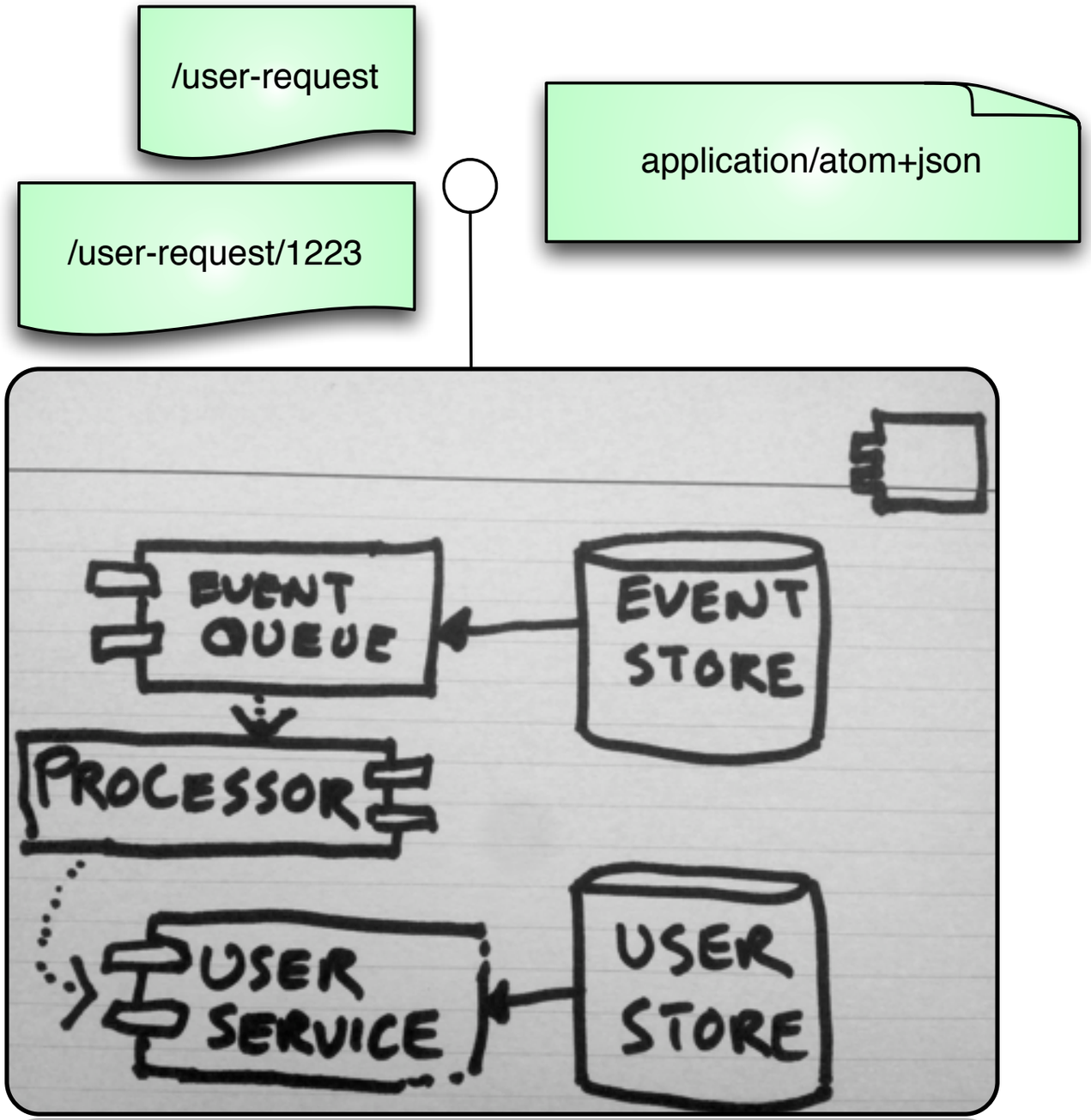


Our Users Capability




```
Standard resource representations using well known
web standards – atom+json
```

Our Users Capability




```
Reified the request to create a user. Clients POST
a request to create a user as an entry to an atom
collection.
```

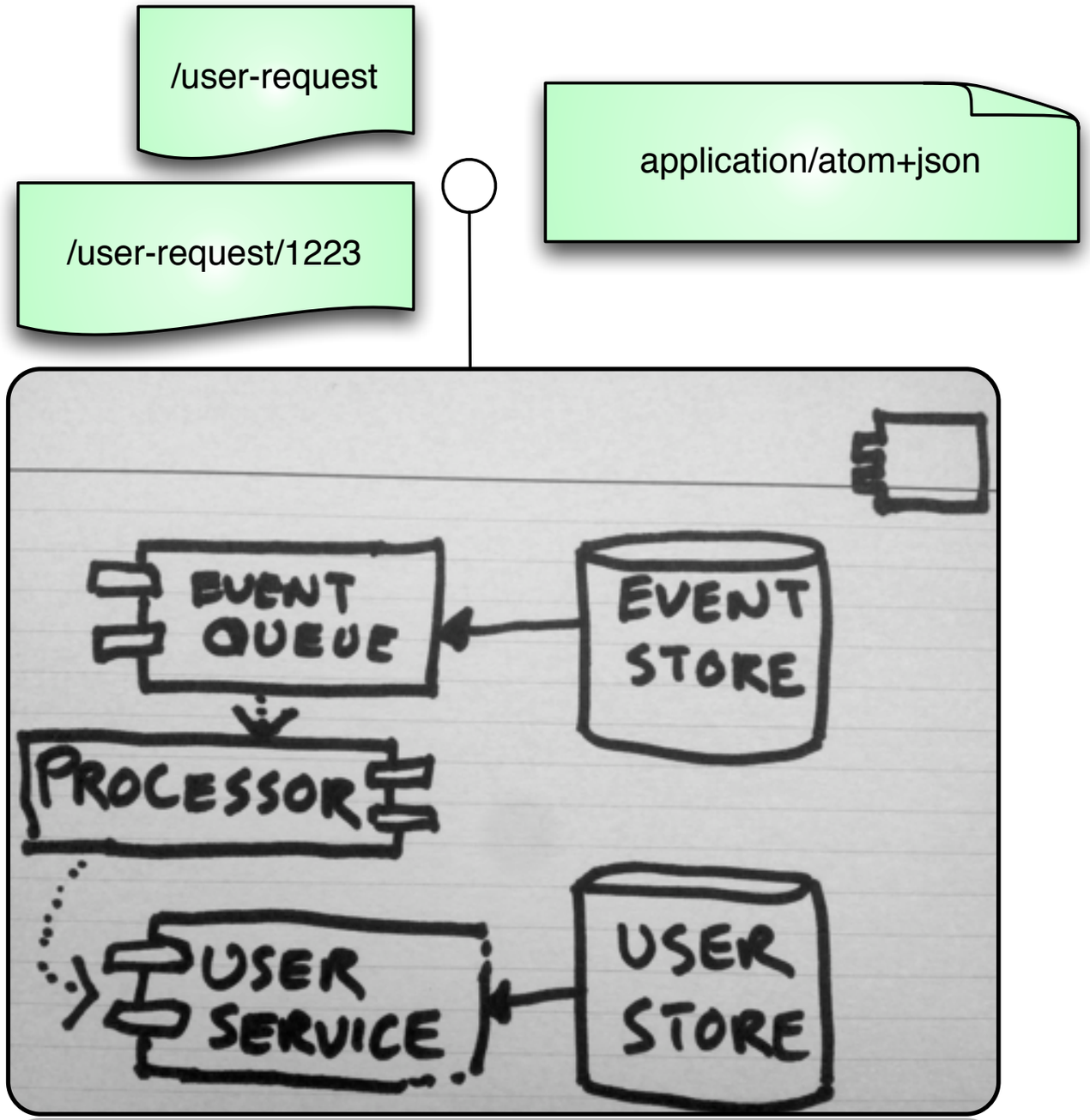
Tip 5

If something is important, make it an explicit part of your design

Reify

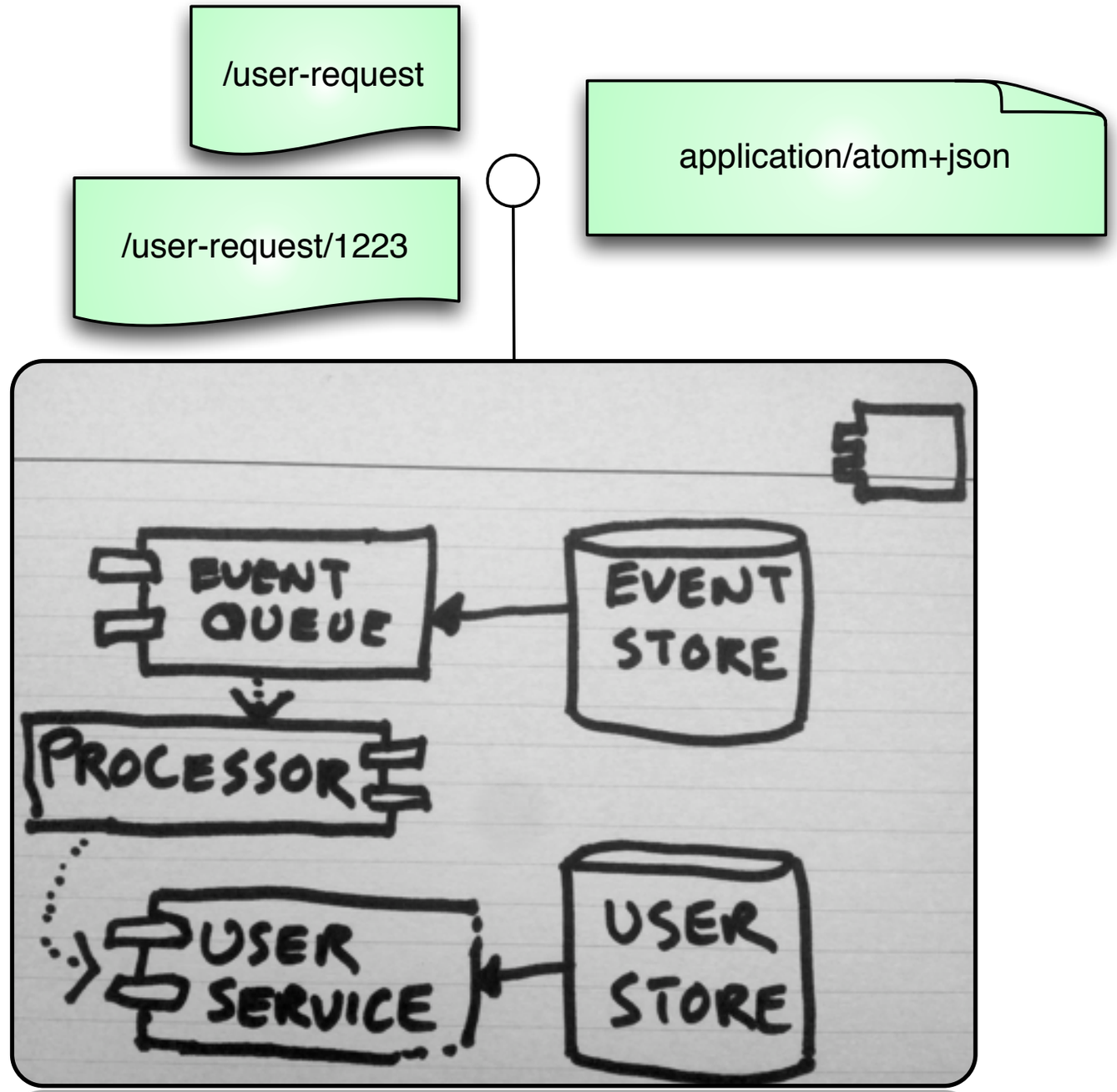
to convert into or regard as a concrete thing: to reify a concept.

Our Users Capability



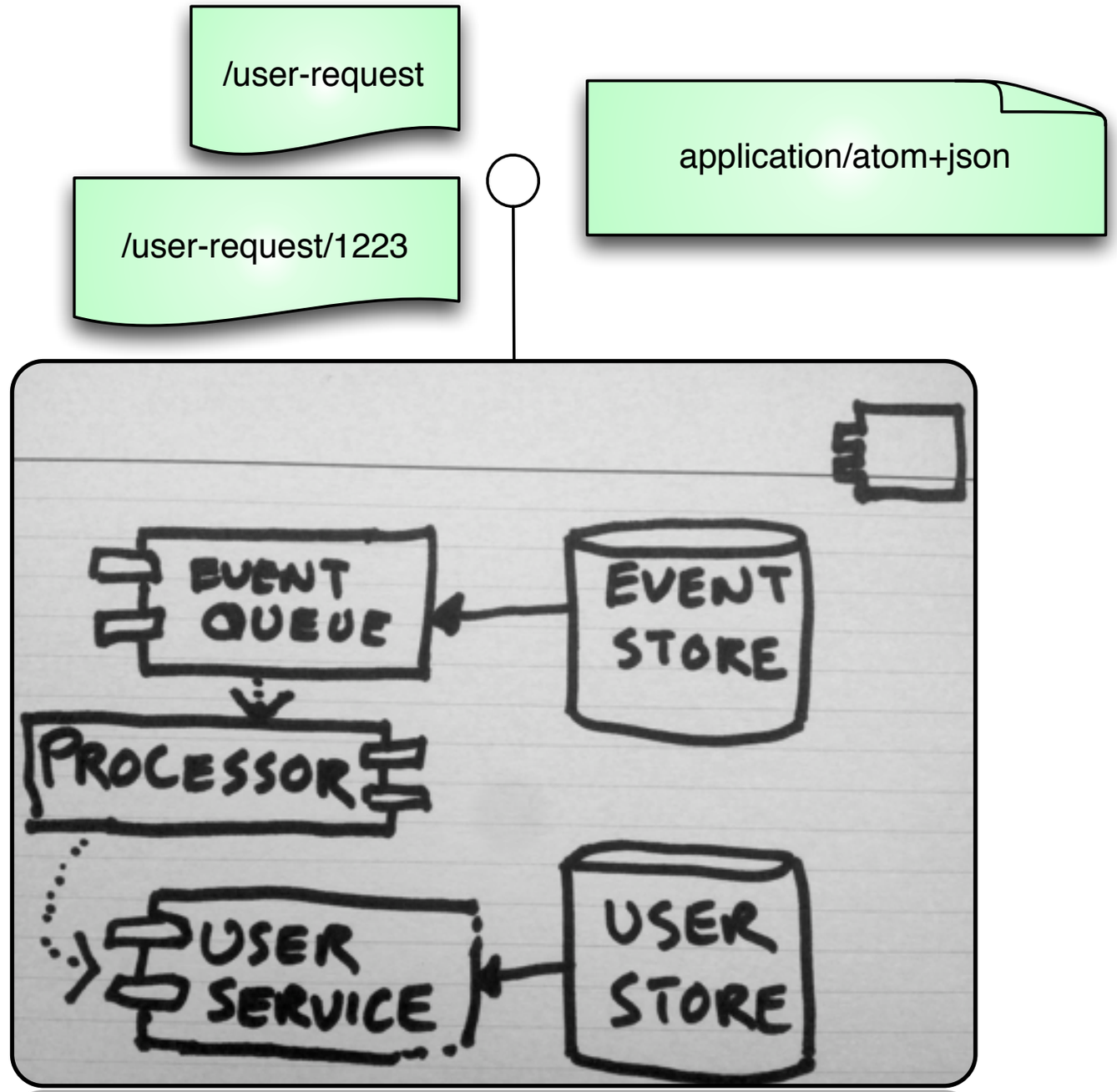
Event queue has the single responsibility of managing state transitions for the request to create a user

Our Users Capability



Queue Processing Engine implemented the Competing Consumer pattern using Conditional GET, PUT and Etags against the atom collection exposed by the event queue

Our Users Capability



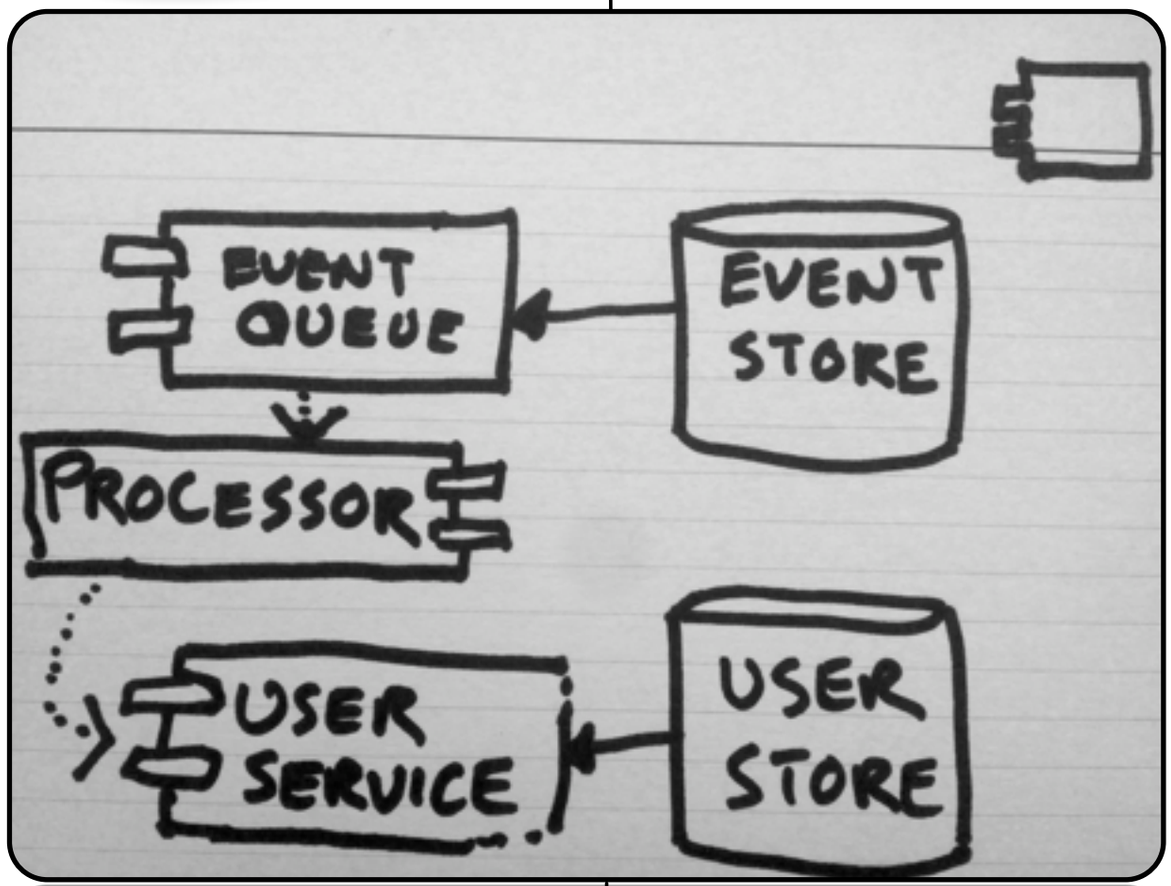
User Service and store is the system of record for users

Our Users Capability

After creation, representations of Users are available at canonical locations in well defined formats and creation events added to another atom collection

/user-request/1223

application/atom+json



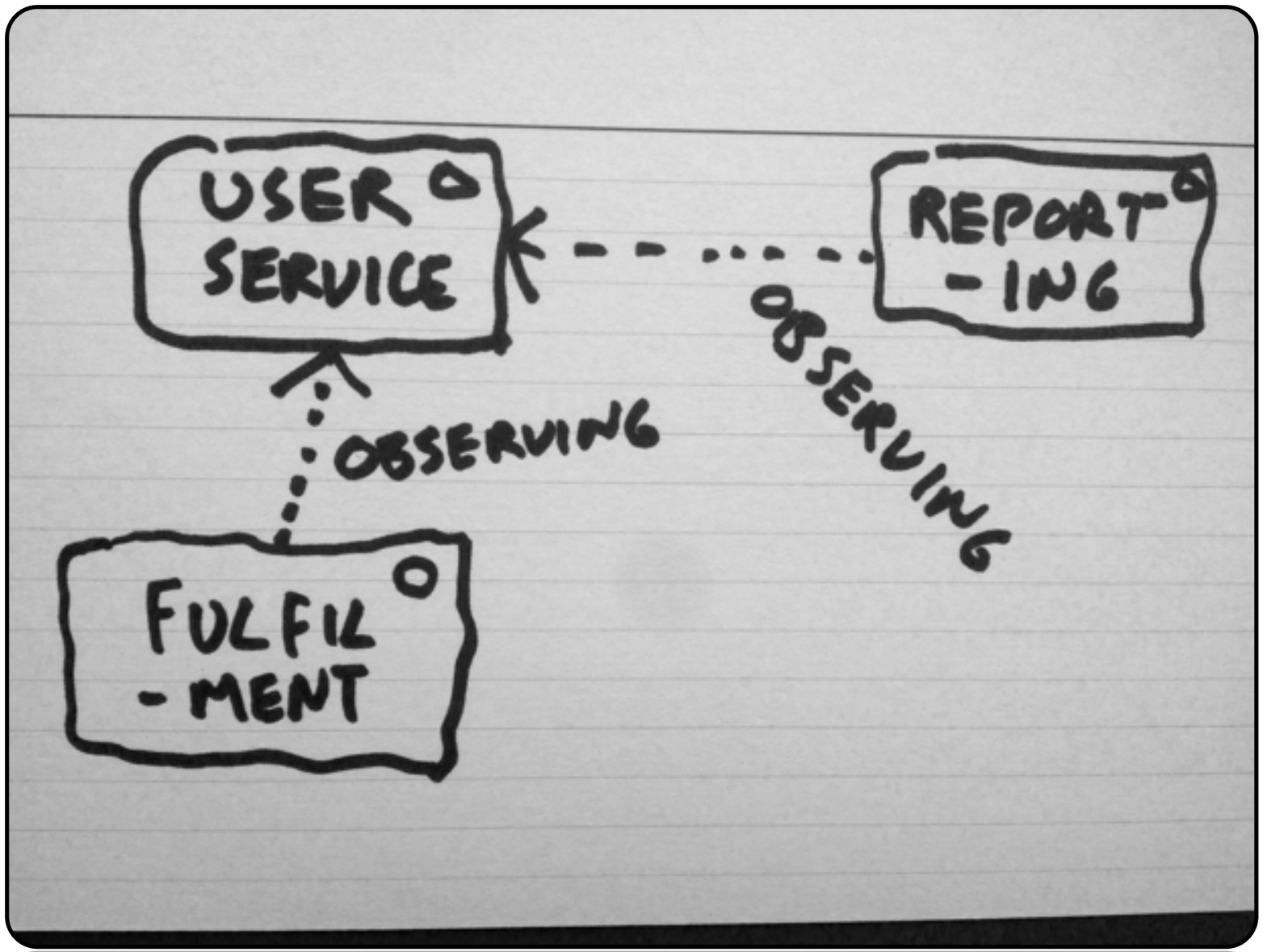
/users/142

application/vnd.user+JSON

/users

Where they are available for consumption by other downstream systems

Reporting capability polls for new user events



Fulfilment capability polls for new user events

Our micro-services

- User Request Queue
  - Forms the transactional boundary of the system
- Request Queue Processor
  - Competing Consumer processes events on the queue and POSTs them to
- User Service
  - System of record for Users in the system
  - Responsible for all state changes of those users
  - Exposes events on those users to other systems

CHARACTERISTICS OF MICRO-SERVICES

Small with a single responsibility

- Each application only does one thing

- Small enough to fit in your head
  - James' heuristic
  - "If a class is bigger than my head then it is too big"

- Small enough that you can throw them away
  - Rewrite over Maintain

Containerless and installed as well behaved Unix services

- Embedded web container
  - Jetty / SimpleWeb
  - This has a lot of benefits for testing (inproctester for example) and eases deployment

- Packaged as a single executable jar
  - Along with their configuration
  - And unix standard rc.d scripts

- Installed in the same way you would install httpd or any other application
  - Why recreate the wheel? Daemons seem to work ok for everything else. Unless you are *special*?

Located in different VCS roots

- Each application is completely separate
- Software developers see similarities and abstractions
  - And before you know it you have One Domain To Rule Them All
- Domain Driven Design / Conways Law
  - Domains in different bounded contexts should be distinct – and its ok to have duplication
  - Use physical separation to enforce this
- There will be common code, but it should be ***library and infrastructure*** code
  - Treat it as you would any other open source library
  - Stick it in a nexus repo somewhere and treat it as a binary dependency

Provisioned automatically

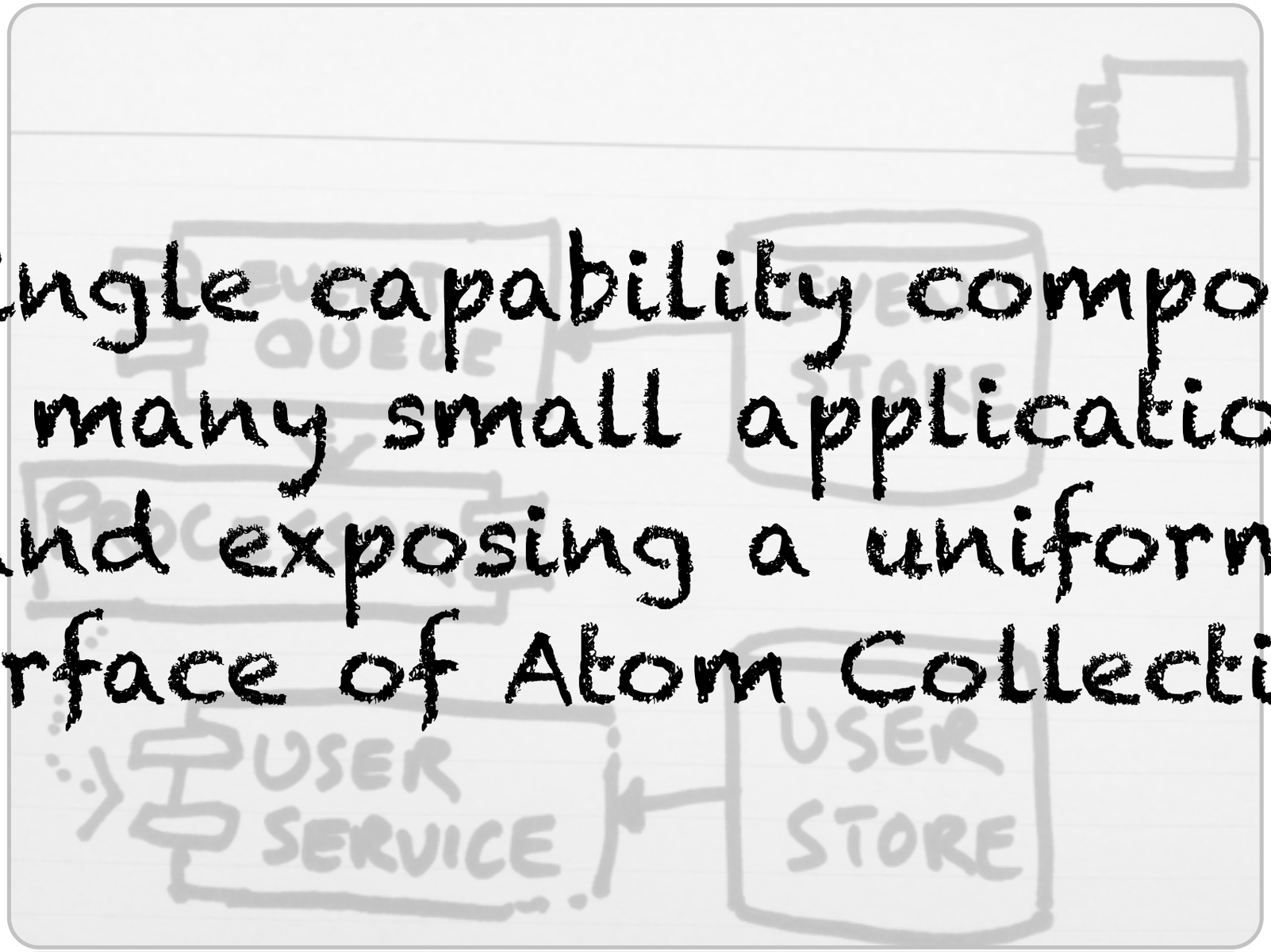
- The way to manage the complexity of many small applications is declarative provisioning
  - –UAT:
    - 2 * service A, Load Balanced, Auto-Scaled
    - 2 * service B, Load Balanced, Auto-Scaled
    - 1 * database cluster

Status aware and auto-scaling

- What good is competing consumer if you only have one consumer?
  - We don't want to wake Laura up at three in the morning any more to start a new process

- Use watchdog processes to monitor in-app status pages
  - Each app exposes metrics about itself
  - In our case, queue-depth for example
  - This allows others services to auto-scale to meet throughput requirements

A single capability composed of many small applications and exposing a uniform interface of Atom Collections



How the capabilities form a product

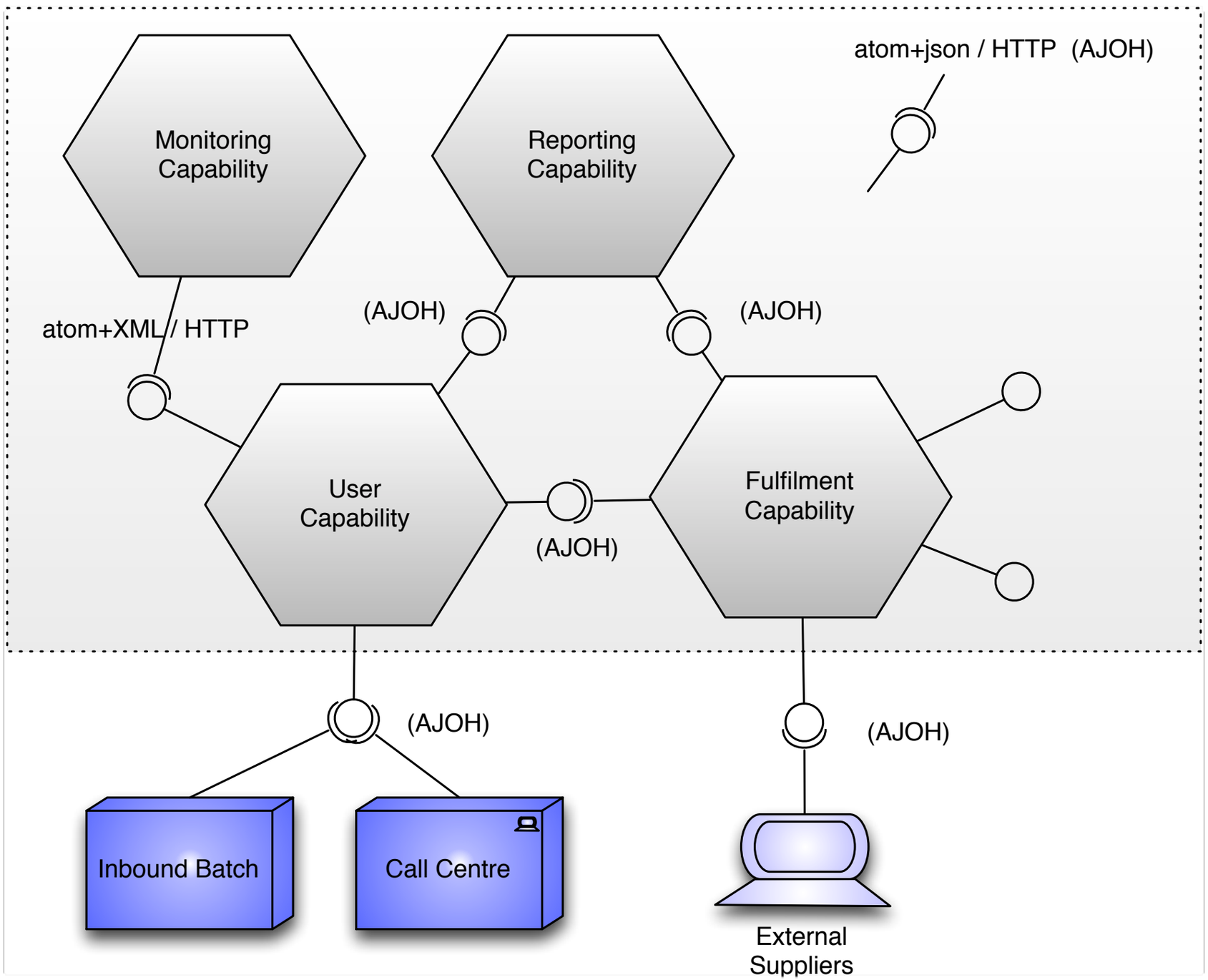
They interact via the web's uniform interface

- HTTP
  - Don't fight the battles already won
  - Use no-brainer force multipliers like reverse proxies

- HATEOAS
  - Link relations drive state changes
  - Its an anti-corruption layer that allows the capability to evolve independently of its clients

- Standard media types
  - Can be used by many different clients
  - You can monitor it using a feed reader if you want
  - and it makes your QA's lives a *lot* easier

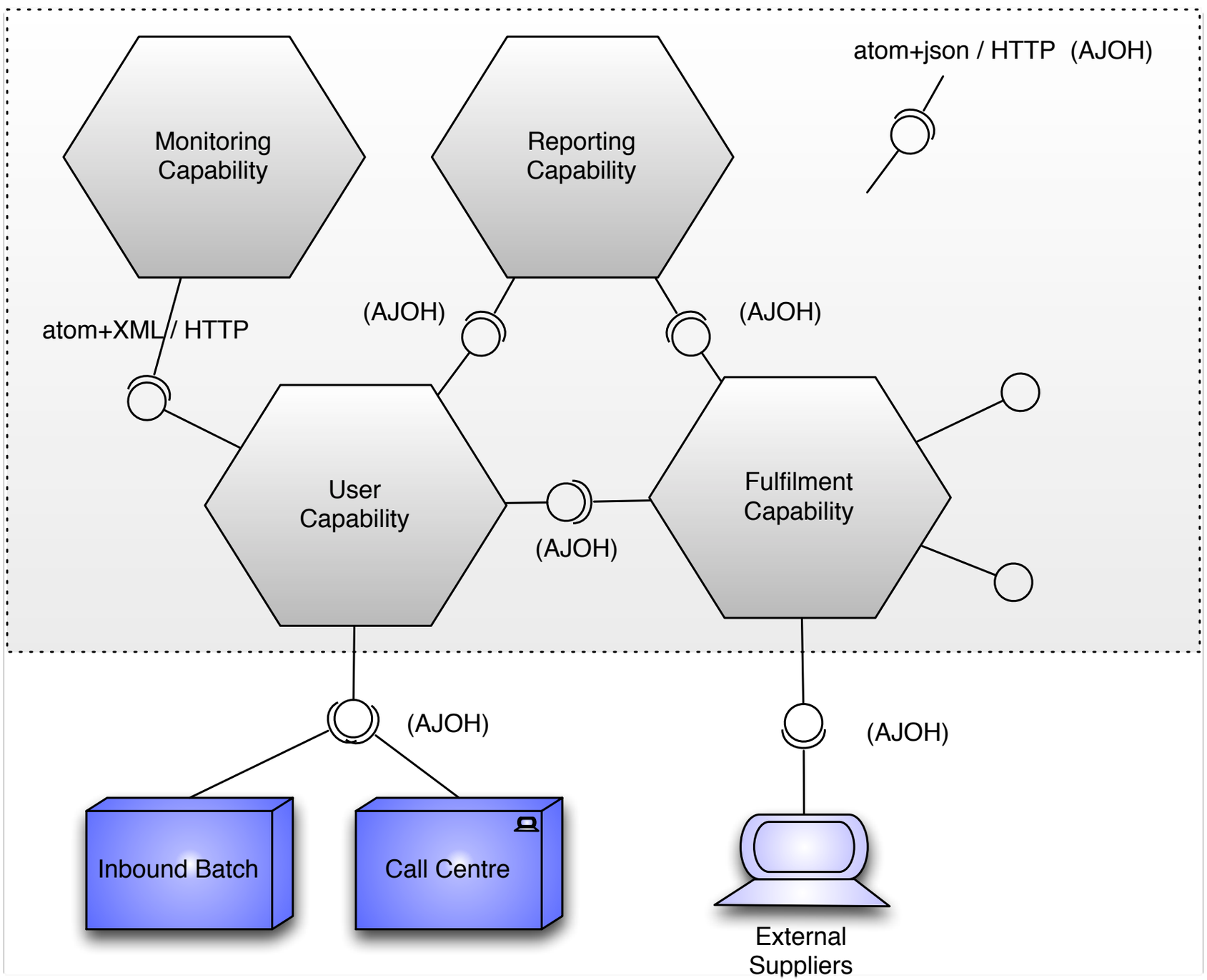
Capabilities poll each other for events, forming an eventually consistent system of systems



Tip 6

Favour service choreography over orchestration

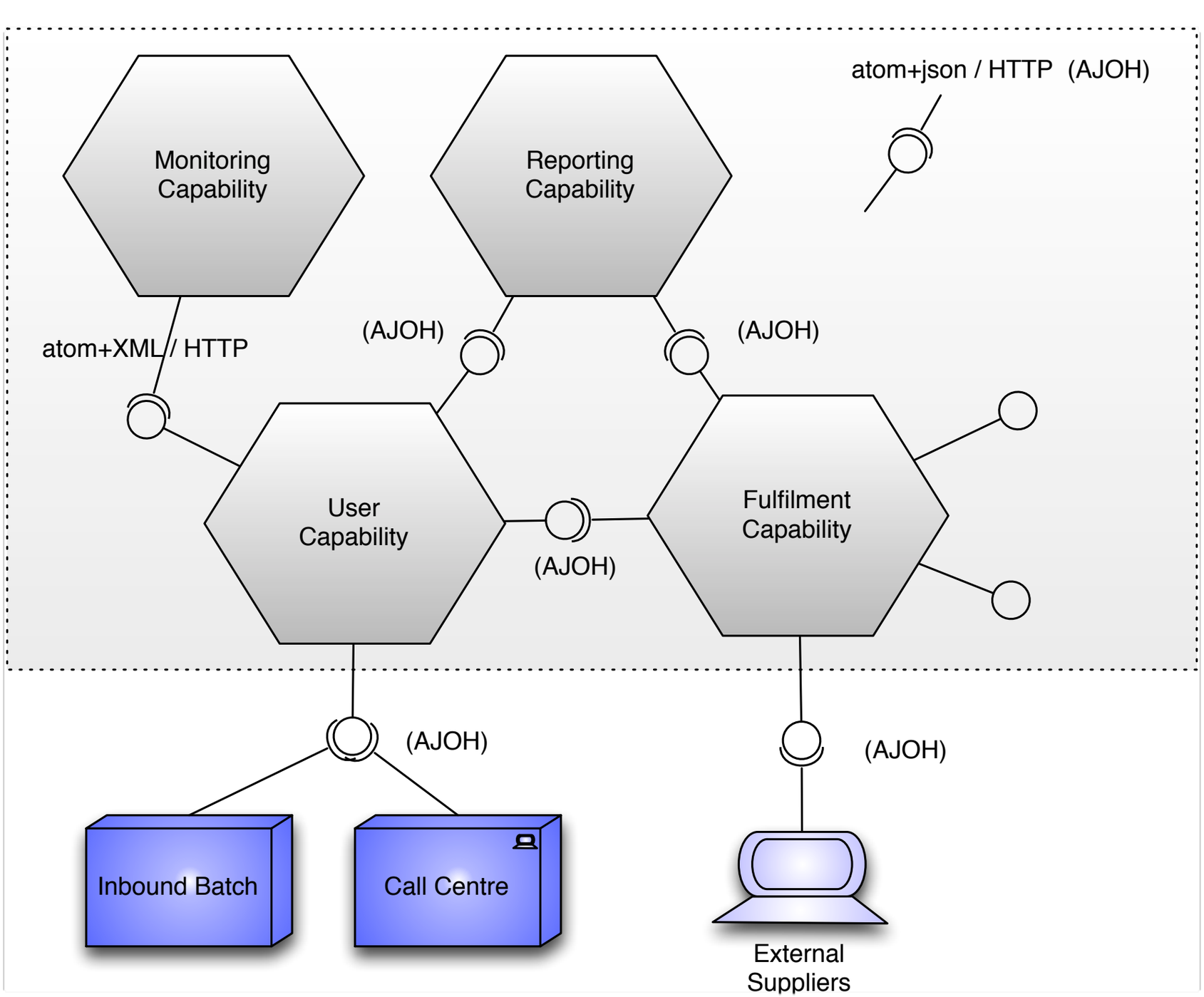
Each is decoupled from it's clients. scalable, testable and deployable individually



Tip 7

Use hypermedia controls to decouple services

Each developed by a separate team, using whatever tech they choose



Our stack

- Embedded Jetty (current project uses SimpleWeb)
- PicoContainer for DI
- Hibernate (but wrote our own SQL)
- Abdera for Atom
- Smoothie charts
- Metrics @codahale
- Graphite

Tip 8

Make it easy to do the right thing and hard to do the wrong thing

use tooling and architectural tests to make complex tasks simple

Infrastructure automation stack

- Fabric with boto
- AWS, but deployable to anything with SSH
- Maven (boo)
- Git
- Puppet for provisioning

NO SILVER BULLETS

this was hard to do well

- We haven't even talked about
  - Versioning
  - Integration
  - Testing
  - Deployment

- Eventual Consistency can be tricky for people to get there head around

- Developers like using ***enterprisy software***
  - No one got fired for choosing an ESB
  - Convincing people to use the web is hard

SUMMARY

but "invented a slightly better one. That finally got changed once more to what we have today. He put pipes into Unix." Thompson also had to change most of the programs, because up until that time, they couldn't take standard input. There wasn't really a need; they all had file arguments. "GREP had a file argument, CAT had a file argument."

The next morning, "we had this orgy of `one liners.' Everybody had a one liner. Look at this, look at that. ...Everybody started putting forth the UNIX philosophy. Write programs that do one thing and do it well. Write programs to work together. Write programs that handle text streams, because that is a universal interface." Those ideas which add up to the tool approach, were there in some unformed way before pipes, but they really came together afterwards. Pipes became the catalyst for this UNIX philosophy. "The tool thing has turned out to be actually successful. With pipes, many programs could work together, and they could work together at a distance."

The Unix Philosophy :s/pipes/http/

Lions commentary on Unix 2nd edition

Consistent and reinforcing practices

Hexagonal Business capabilities composed of:

Micro Services that you can

Rewrite rather than maintain and which form

A Distributed Bounded Context.

Deployed as containerless OS services

With standardised application protocols and message semantics

Which are auto-scaling and designed for failure

Tip 9

Always have ten tips

ThoughtWorks®

Thanks!

@boicy

http://bovon.org

jalewis@thoughtworks.com