RESPONSIBILITY ALLOCATION

DECOMPOSING SYSTEMS

- Where do we put the data?
- Where do we put the features?
- What should the interfaces look like?
- How do we weave everything back together?

COUPLING & COHESION

COUPLING

Type of Coupling	Effect
Runtime / operational	Consumer cannot run without the provider
Development	Code changes in producer and consumer must be coordinated
Responsibility	Two things change together because of shared responsibility or concepts

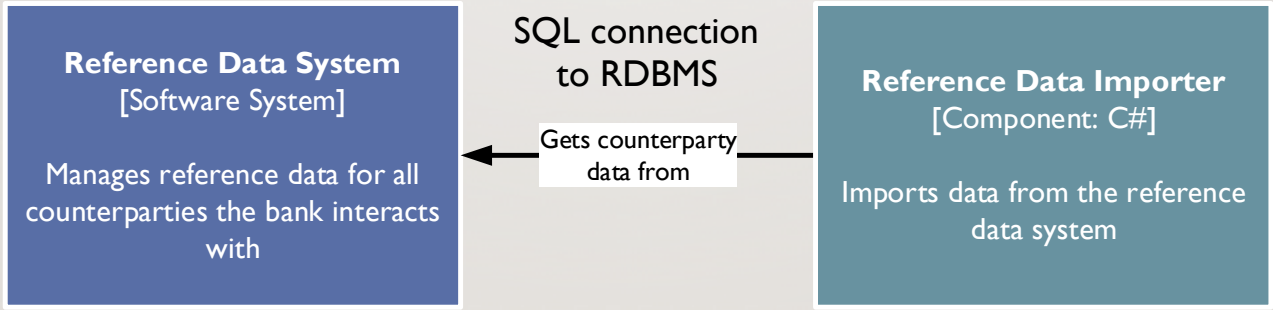
Any or all can be present at the same time

EXAMPLE OF ANALYZING COUPLING

Operational: Strong. SMTP is synchronous, connection-oriented, conversational

Development: Weak. SMTP is well-defined standard with history of interoperability

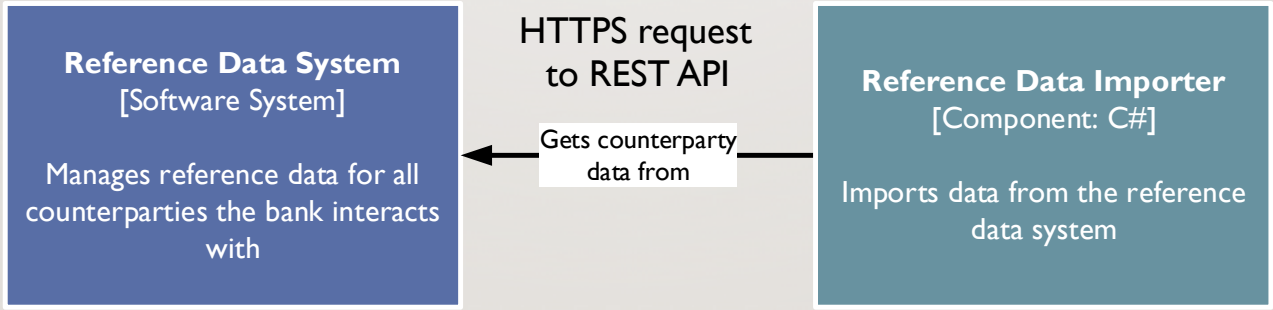
EXAMPLE OF ANALYZING COUPLING



Operational: Very strong. Dependent on availability of server. Must be aware of topology and failover strategy

Development: Very strong. Dependent on schema, server version, protocol version.

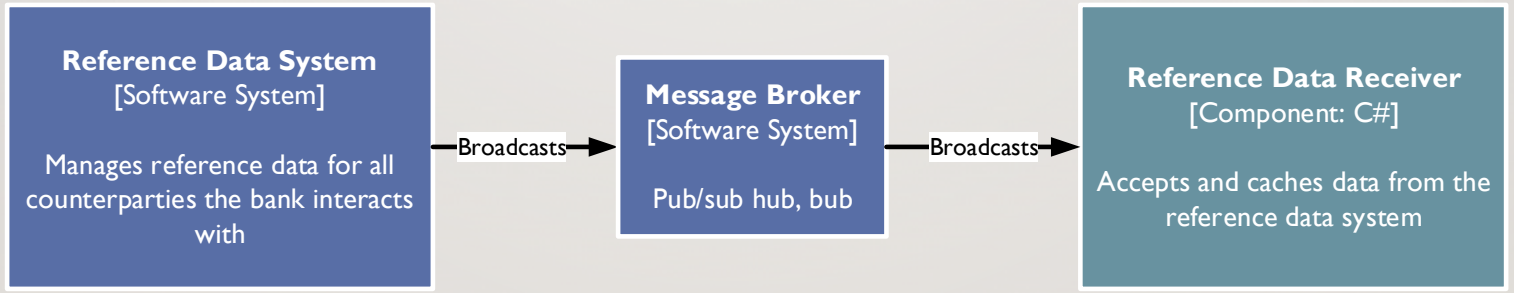
EXAMPLE OF ANALYZING COUPLING



Operational: Strong, but less than before. Dependent on availability of server.

Development: Strong, but less. Insulated from data format changes. Open encoding can further reduce coupling

EXAMPLE OF ANALYZING COUPLING

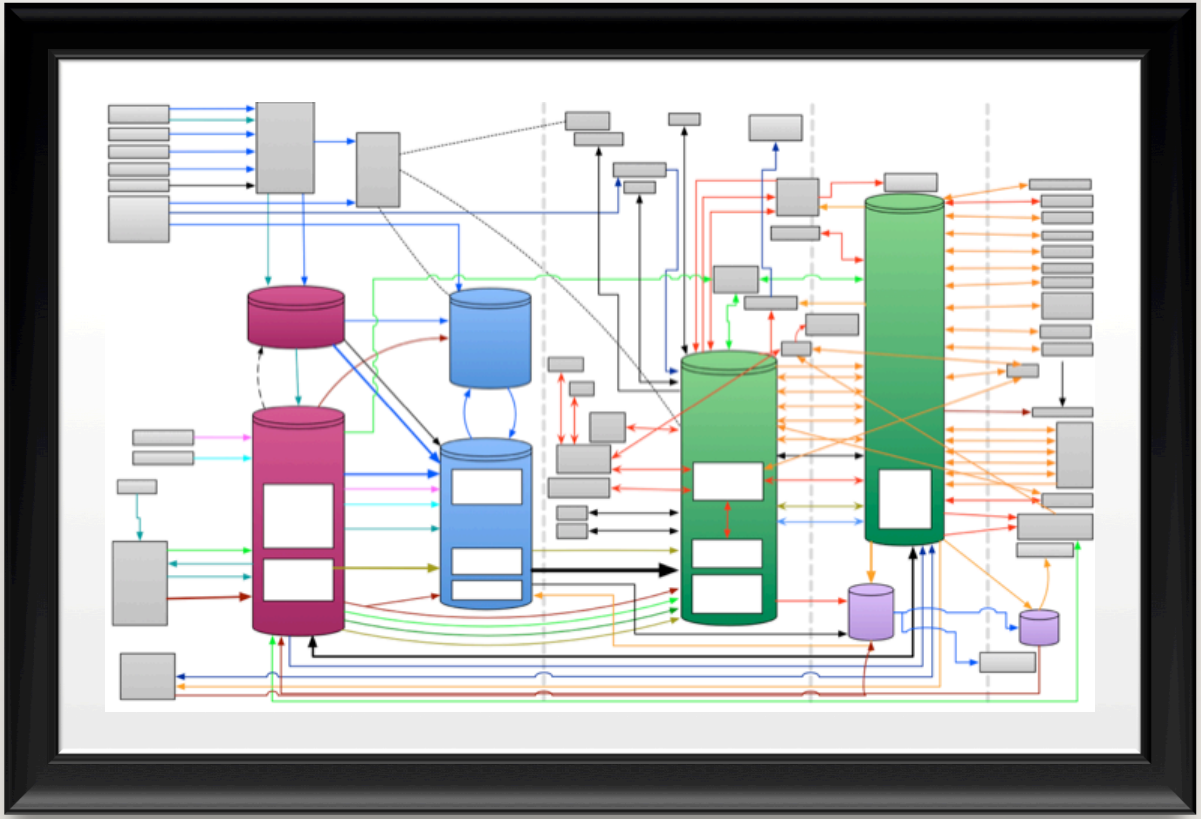


Operational: Very weak. Receiver can run with stale data when either broker or upstream are broken.

Development: Weak. Insulated from schema changes.

“LONG CHAIN” INTERFACES

A SIMPLE ARROW CAN HIDE A GREAT DEAL



EACH "INTERFACE" WAS REALLY A CHAIN

1. Extract tables to files
2. Push files across network
3. Load tables into "LZ"
4. Process into "cold" DB
5. Swap hot & cold DBs (hours later)

1. Send message to queue
2. Take message from queue, unwrap, inspect, and dispatch to 1-of-N other queues.
3. Drain queue to file
4. Batch job wakes up 2 times a day, does FTP to remote end
5. Another batch job pulls a reconciliation file, drops file into file system
6. Parser reads the file, shreds it into messages, puts them on another queue

ARCHITECTURE QUALITIES IN LONG CHAINS

Losses accumulate:

- Latency strictly worse than the slowest link in the chain.
- Availability strictly worse than the least available link.
- Throughput strictly worse than the throughput of the worst bottleneck
- Security strictly worse than the security of the weakest link

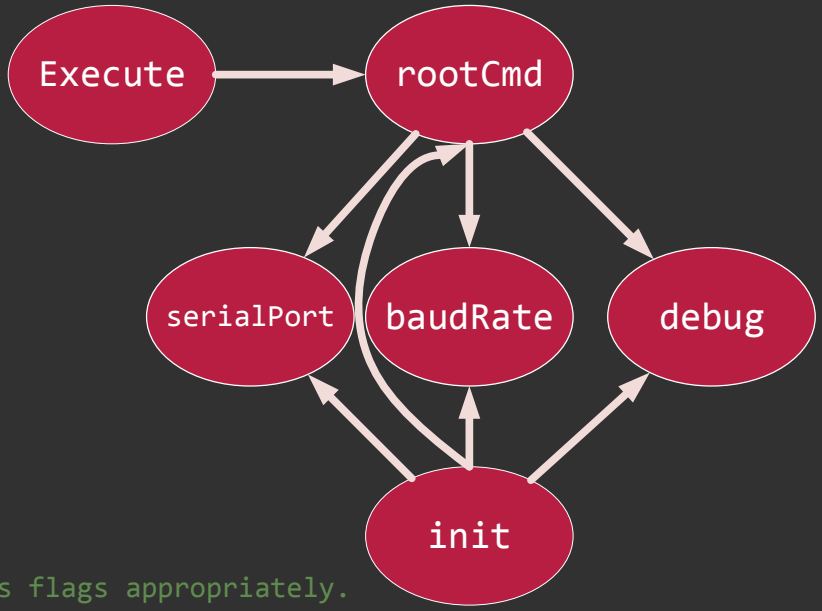
COHESION

COHESION

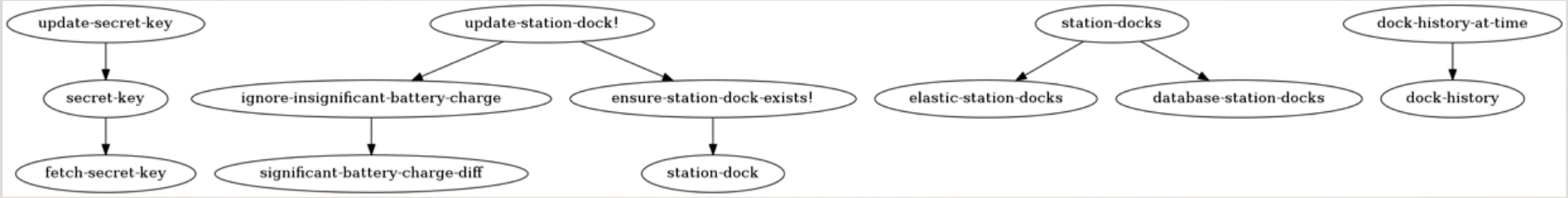
- Does the module "fit" together as a logical unit?
- Look at references between functions and variables
- Are they fully connected? Or partitioned?
- Much easier to see in the code than the early designs.
- Iterate and adjust the architecture!

```go
import (
    "fmt"
    "os"

    "github.com/spf13/cobra"
)

var (
    serialPort string
    baudRate int
    debug bool
)

var rootCmd = &cobra.Command{
    Use: "roc.simulator",
    Short: "Simulate hardware found on the Kiosk"}

// Execute adds all child commands to the root command and sets flags appropriately.
func Execute() {
    if err := rootCmd.Execute(); err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}

func init() {
    rootCmd.PersistentFlags().StringVar(&serialPort, "port", "/dev/ttyS0", "Serial port to respond on")
    rootCmd.PersistentFlags().IntVar(&baudRate, "baud", 115200, "Baud rate")
    rootCmd.PersistentFlags().BoolVar(&debug, "debug", false, "Report diagnostics on stderr")
}
```

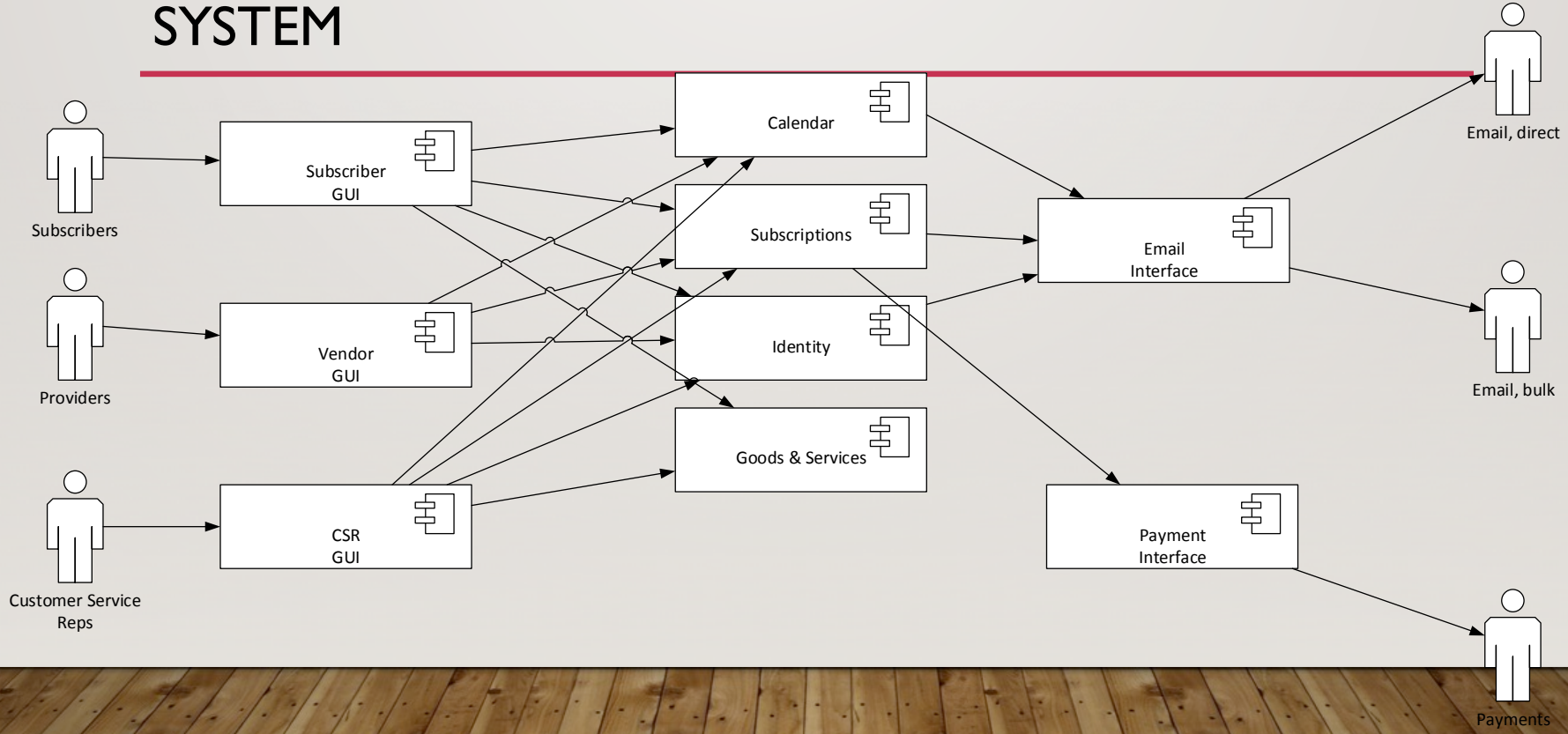
```go
import (
    "fmt"
    "os"

    "github.com/spf13/cobra"
)

var (
    serialPort string
    baudRate int
    debug bool
)

var rootCmd = &cobra.Command{
    Use: "roc.simulator",
    Short: "Simulate hardware found on the Kiosk"}

// Execute adds all child commands to the root command and sets flags appropriately.
func Execute() {
    if err := rootCmd.Execute(); err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}

func init() {
    rootCmd.PersistentFlags().StringVar(&serialPort, "port", "/dev/ttyS0", "Serial port to respond on")
    rootCmd.PersistentFlags().IntVar(&baudRate, "baud", 115200, "Baud rate")
    rootCmd.PersistentFlags().BoolVar(&debug, "debug", false, "Report diagnostics on stderr")
}
```



NOT VERY COHESIVE



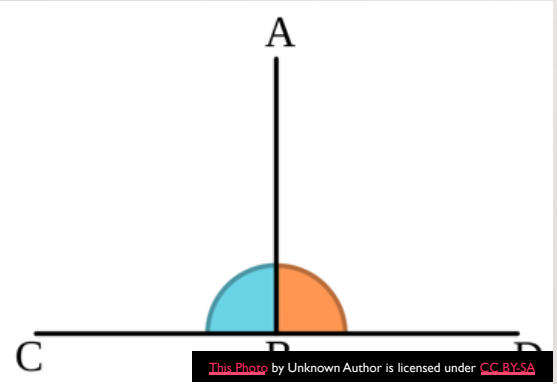
LOOKING BACK: OUR FIRST STAB AT THE SAMPLE SYSTEM



ORTHOGONAL

"You keep using that word…"

ORTHOGONAL: IN MATH



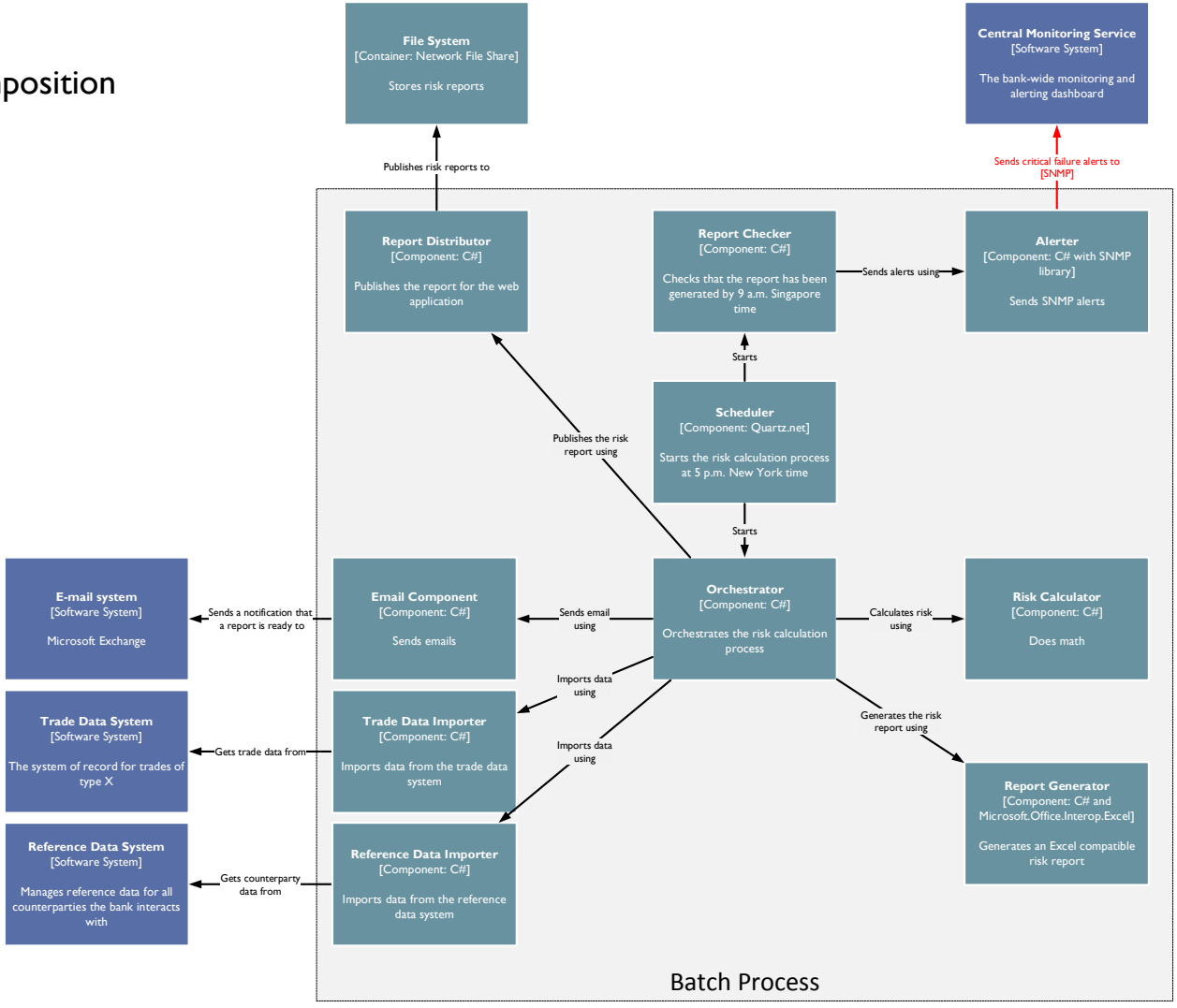
- Dot product of one vector onto the other is zero.
- Zero projection → Perpendicular
- Intersection, but no overlap

ORTHOGONAL: IN SOFTWARE

- Separation of concerns
- High cohesion within a module or component
- Low coupling between modules or components
- Little overlap in functionality between modules
- Information hiding / decision hiding

Activity: Let's Evaluate This Decomposition

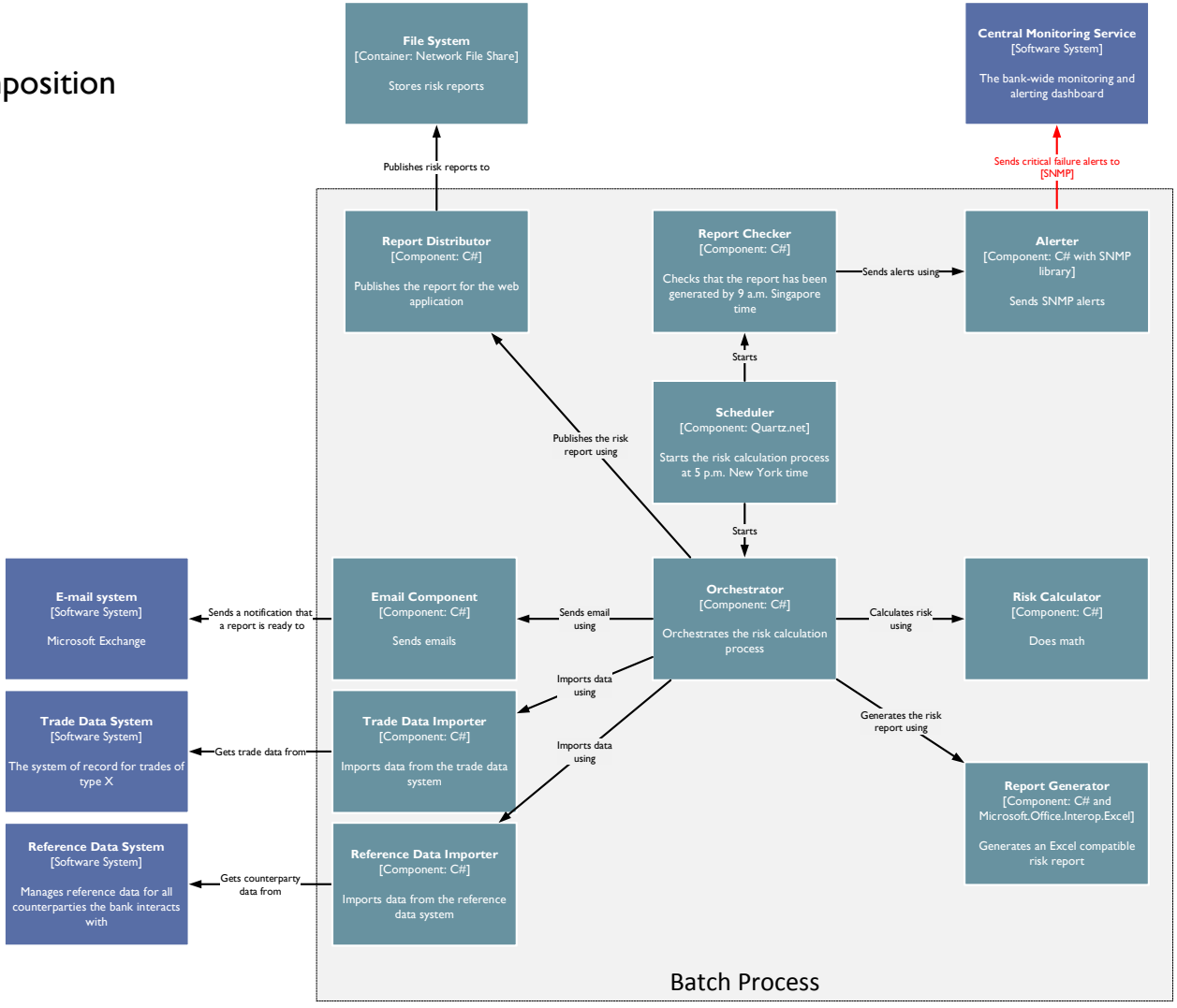
Coupling between modules?
Not bad.



Activity: Let's Evaluate This Decomposition

Coupling between modules?
Not bad.

Cohesion within components?
We can't tell from this level.

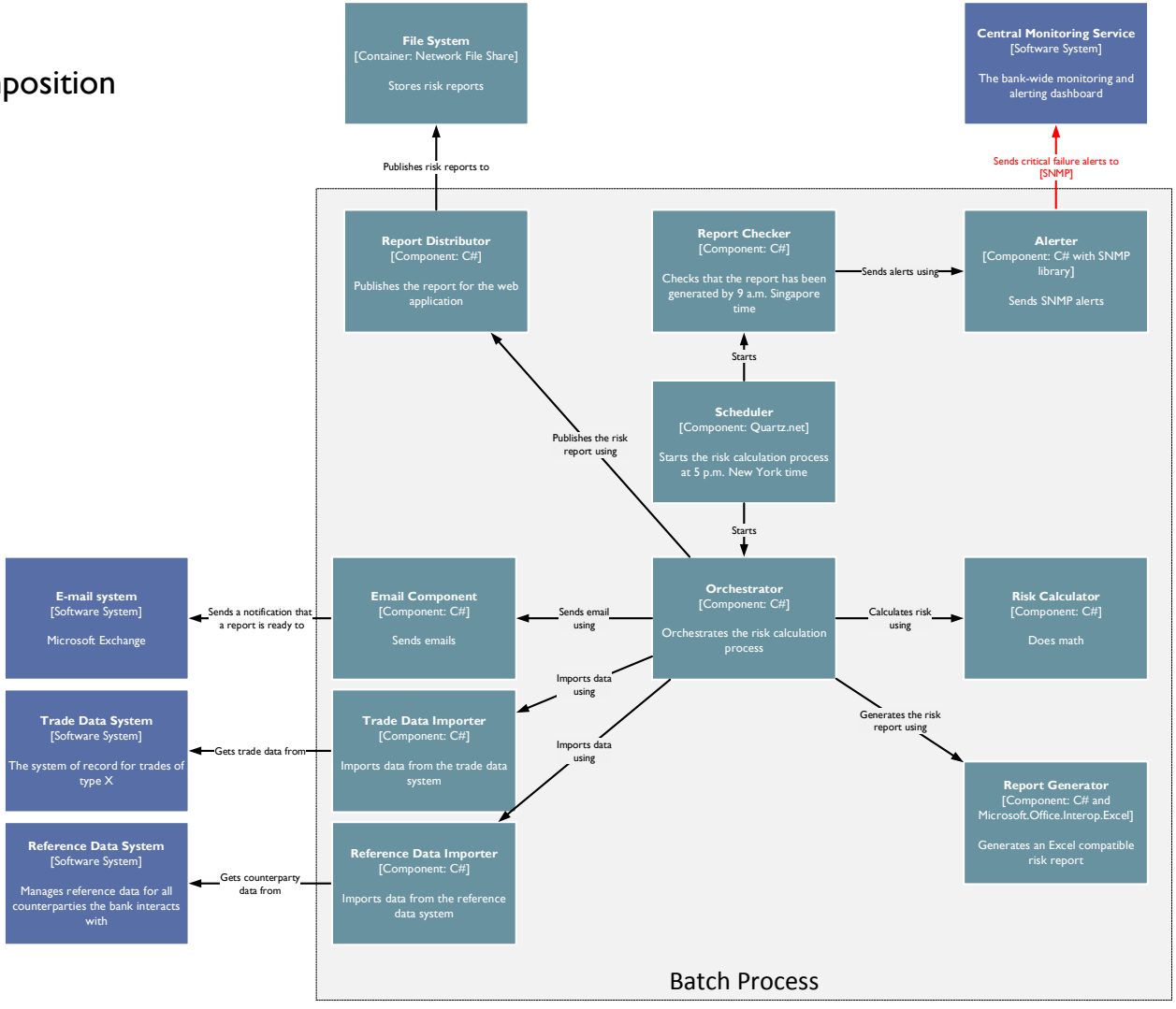


Activity: Let's Evaluate This Decomposition

Coupling between modules?
Not bad.

Cohesion within components?
We can't tell from this level.

Overlapping functionality?
Some, in the importers



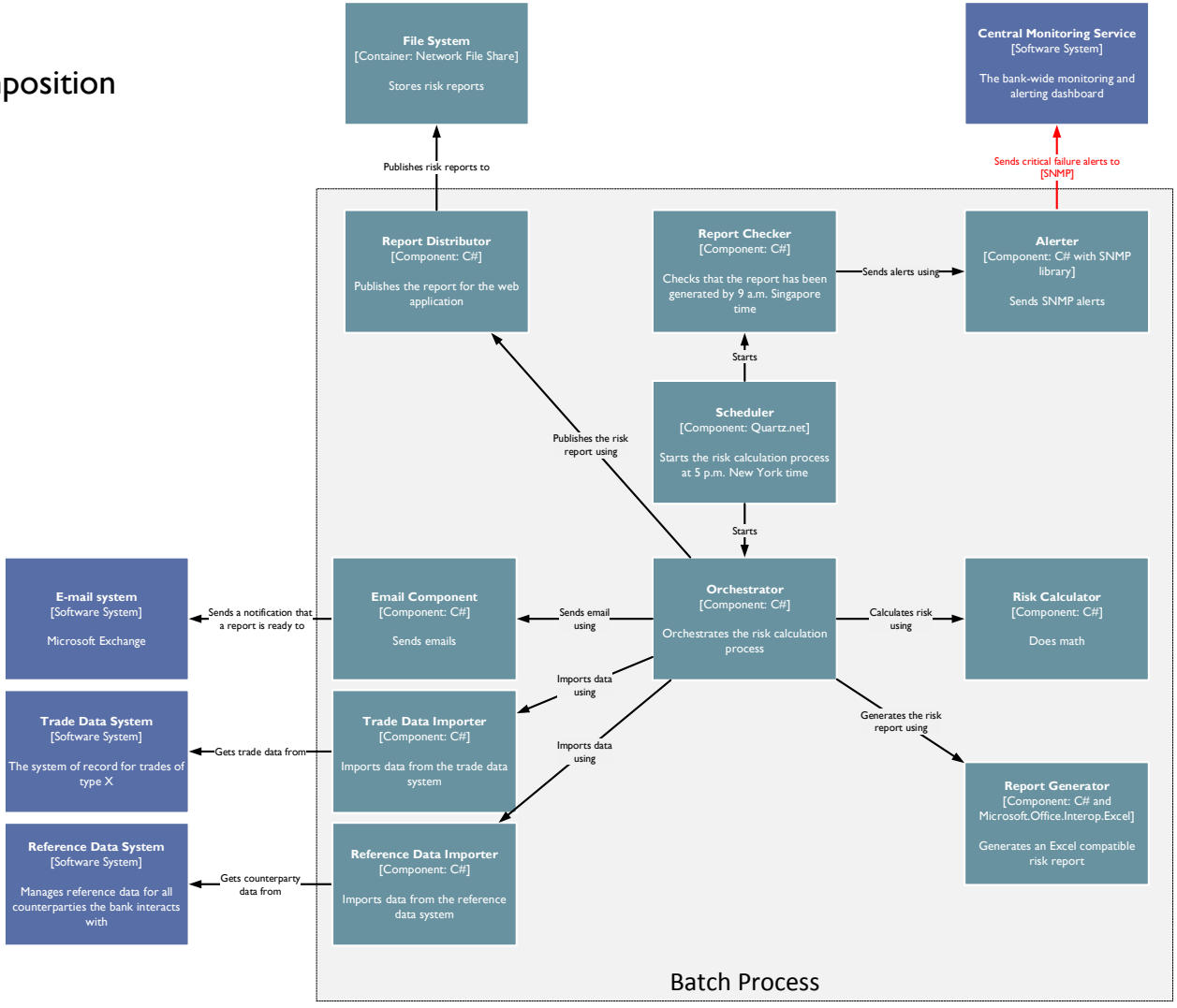
Activity: Let's Evaluate This Decomposition

Coupling between modules?
Not bad.

Cohesion within components?
We can't tell from this level.

Overlapping functionality?
Some, in the importers

As in the Parnas paper, much depends on the API design.



Activity: Let's Evaluate This Decomposition

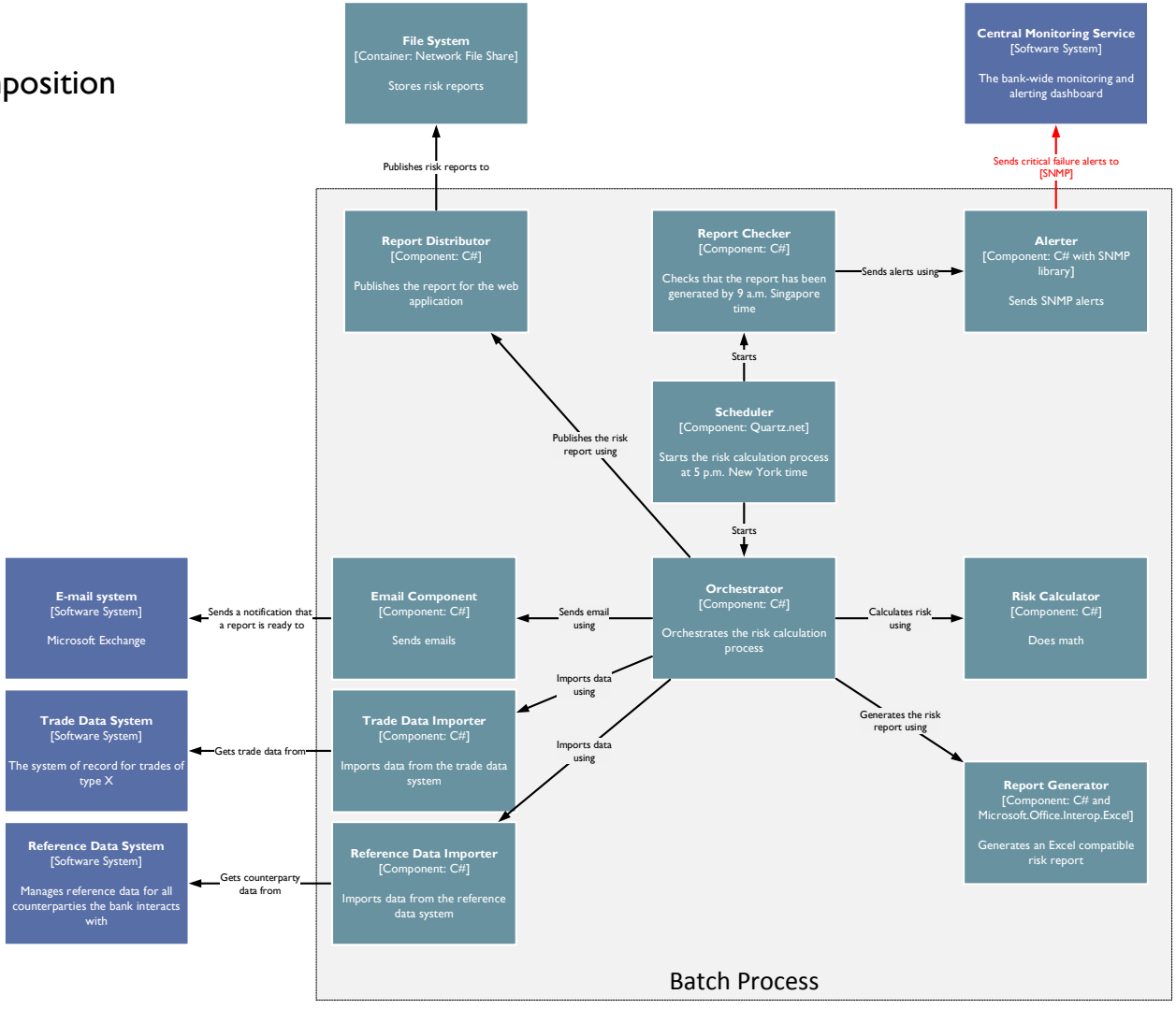
Coupling between modules?
Not bad.

Cohesion within components?
We can't tell from this level.

Overlapping functionality?
Some, in the importers

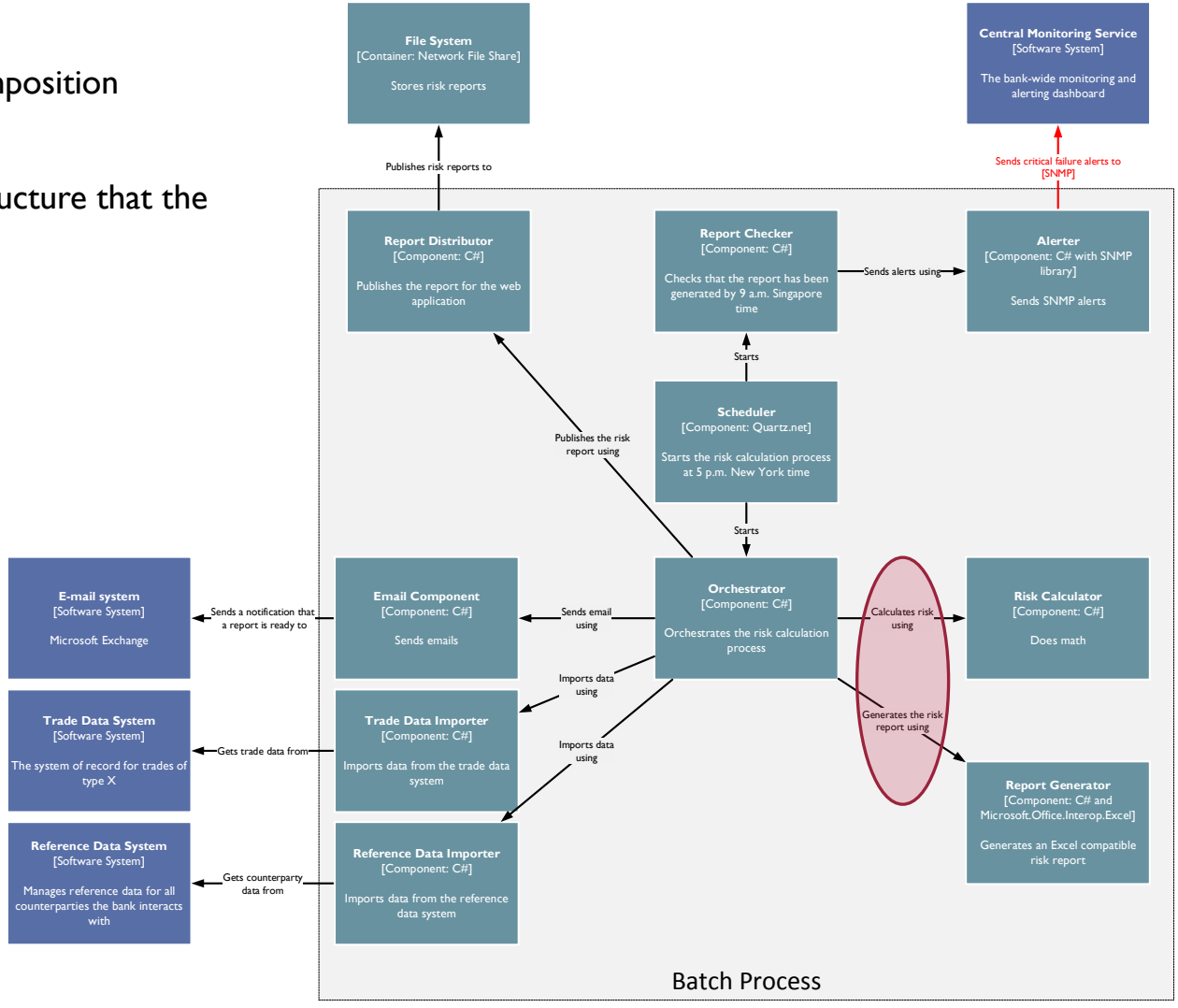
As in the Parnas paper, much depends on the API design.

Here are some places that are likely to present trouble



Activity: Let's Evaluate This Decomposition

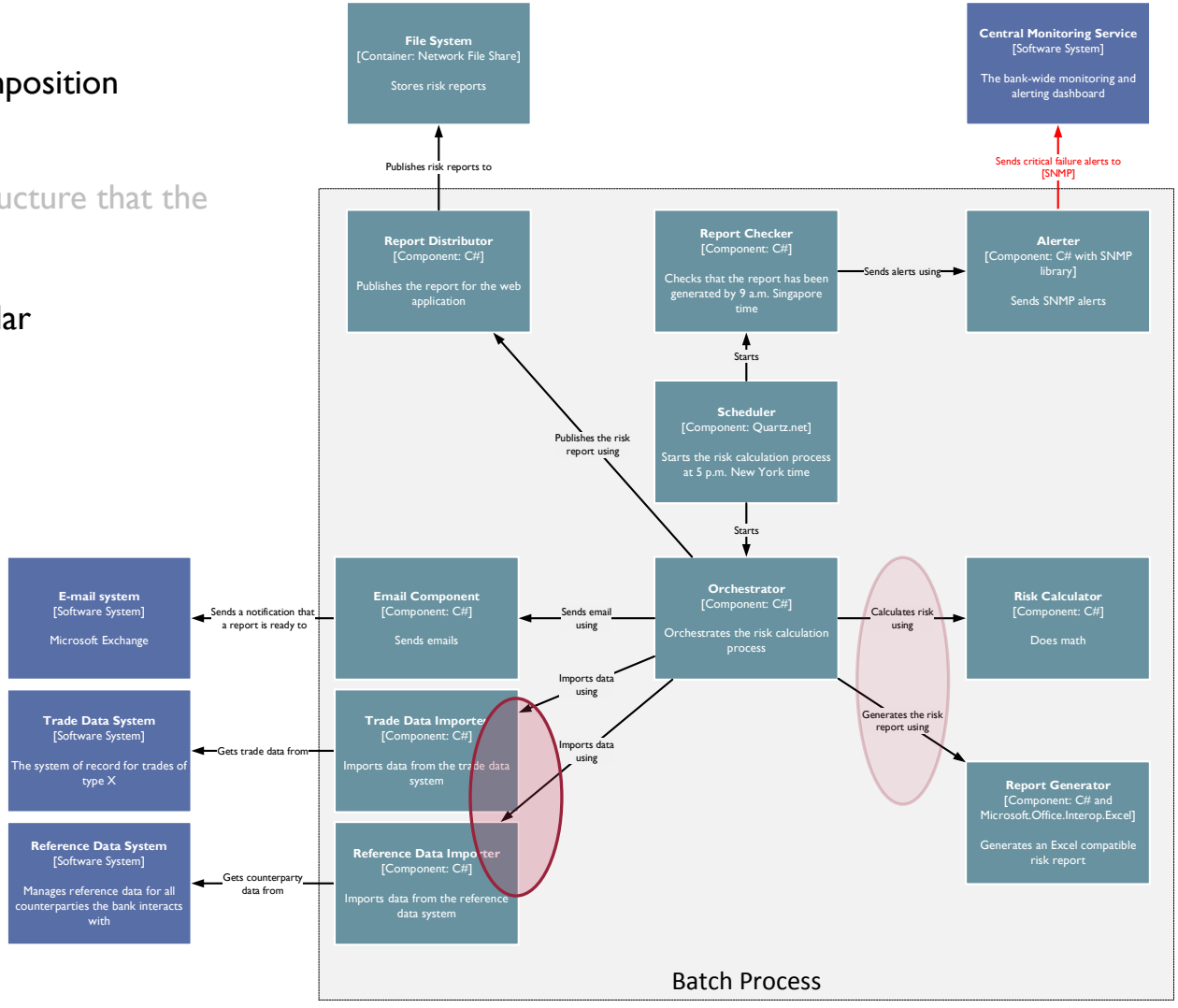
Risk calculator produces a data structure that the report generator must consume.



Activity: Let's Evaluate This Decomposition

Risk calculator produces a data structure that the report generator must consume.

Data importers probably have similar implementation needs

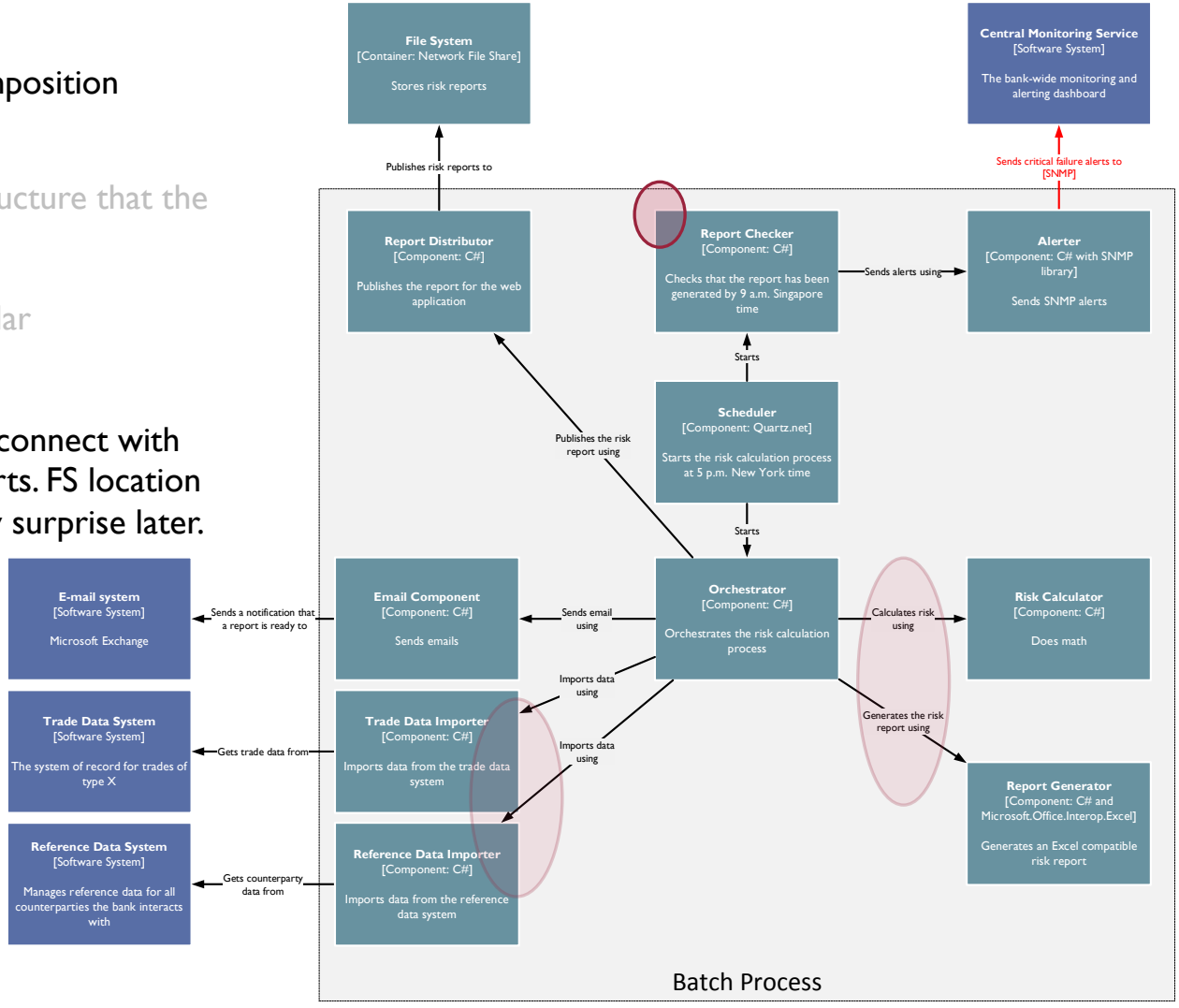


Activity: Let's Evaluate This Decomposition

Risk calculator produces a data structure that the report generator must consume.

Data importers probably have similar implementation needs

Report checker doesn't appear to connect with the file system that holds the reports. FS location is *latent coupling* that will be a nasty surprise later.



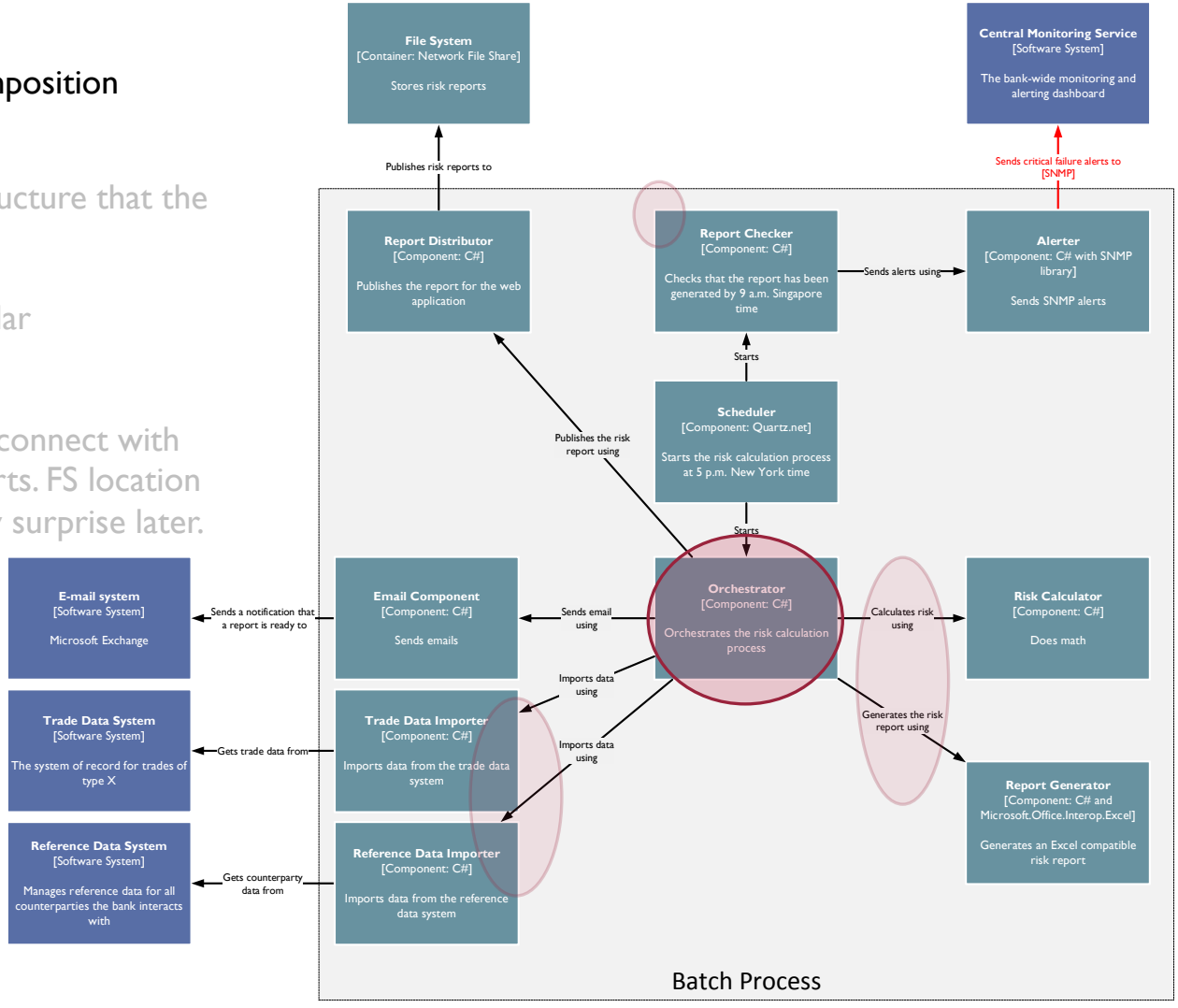
Activity: Let's Evaluate This Decomposition

Risk calculator produces a data structure that the report generator must consume.

Data importers probably have similar implementation needs

Report checker doesn't appear to connect with the file system that holds the reports. FS location is *latent coupling* that will be a nasty surprise later.

Orchestrator might end need to do lots of data transformation to bridge interfaces.



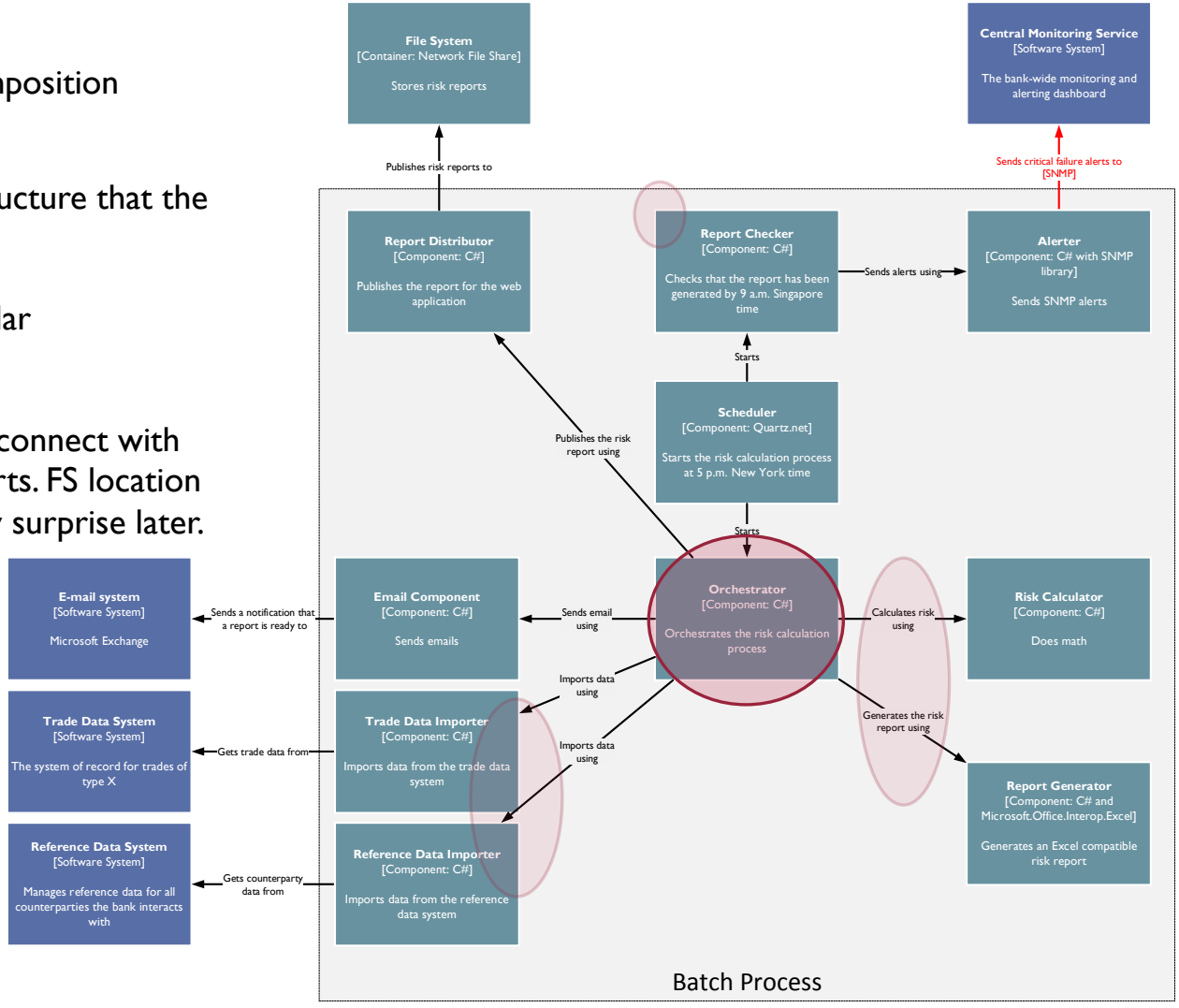
Activity: Let's Evaluate This Decomposition

Risk calculator produces a data structure that the report generator must consume.

Data importers probably have similar implementation needs

Report checker doesn't appear to connect with the file system that holds the reports. FS location is *latent coupling* that will be a nasty surprise later.

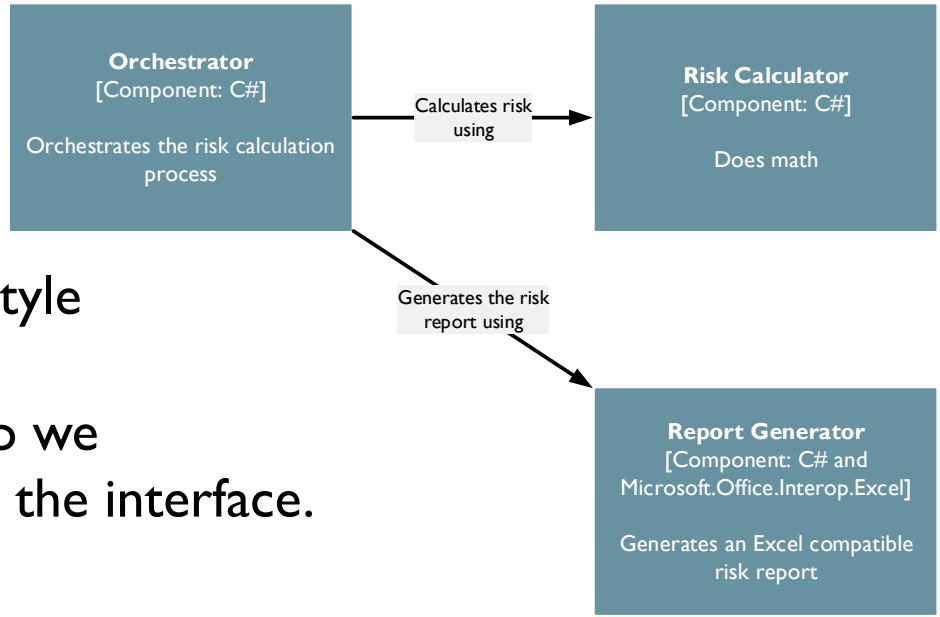
Orchestrator might end need to do lots of data transformation to bridge interfaces.

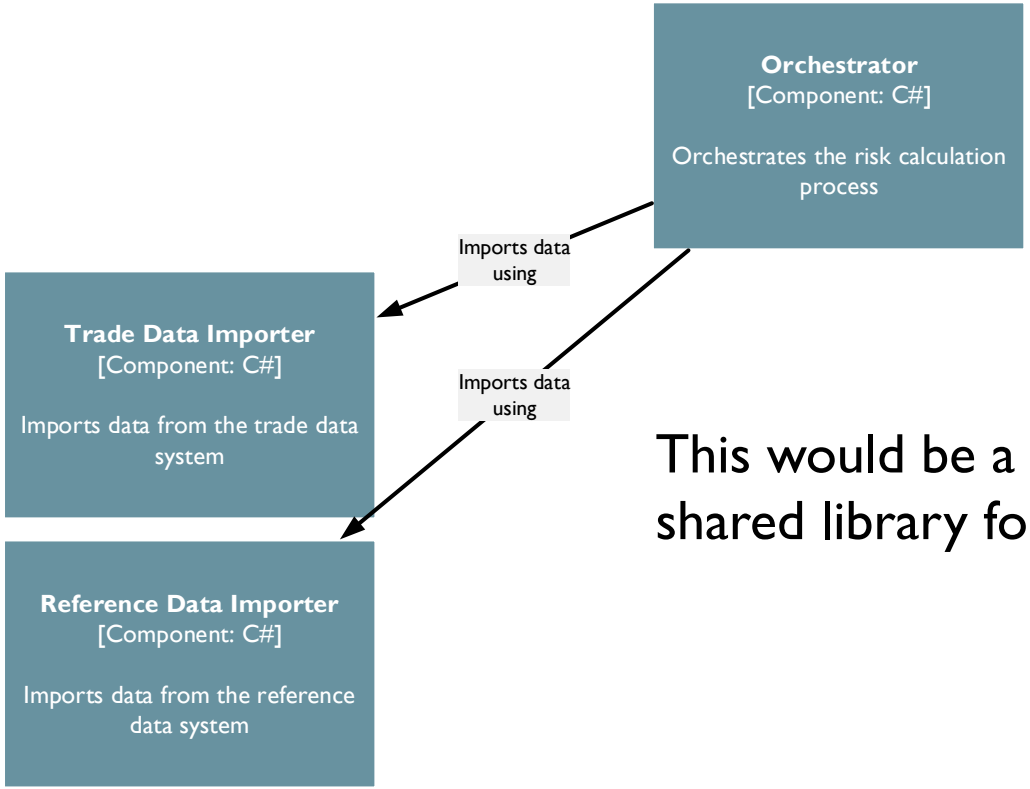


Problem: Risk calculator produces a data structure that the report generator must consume.

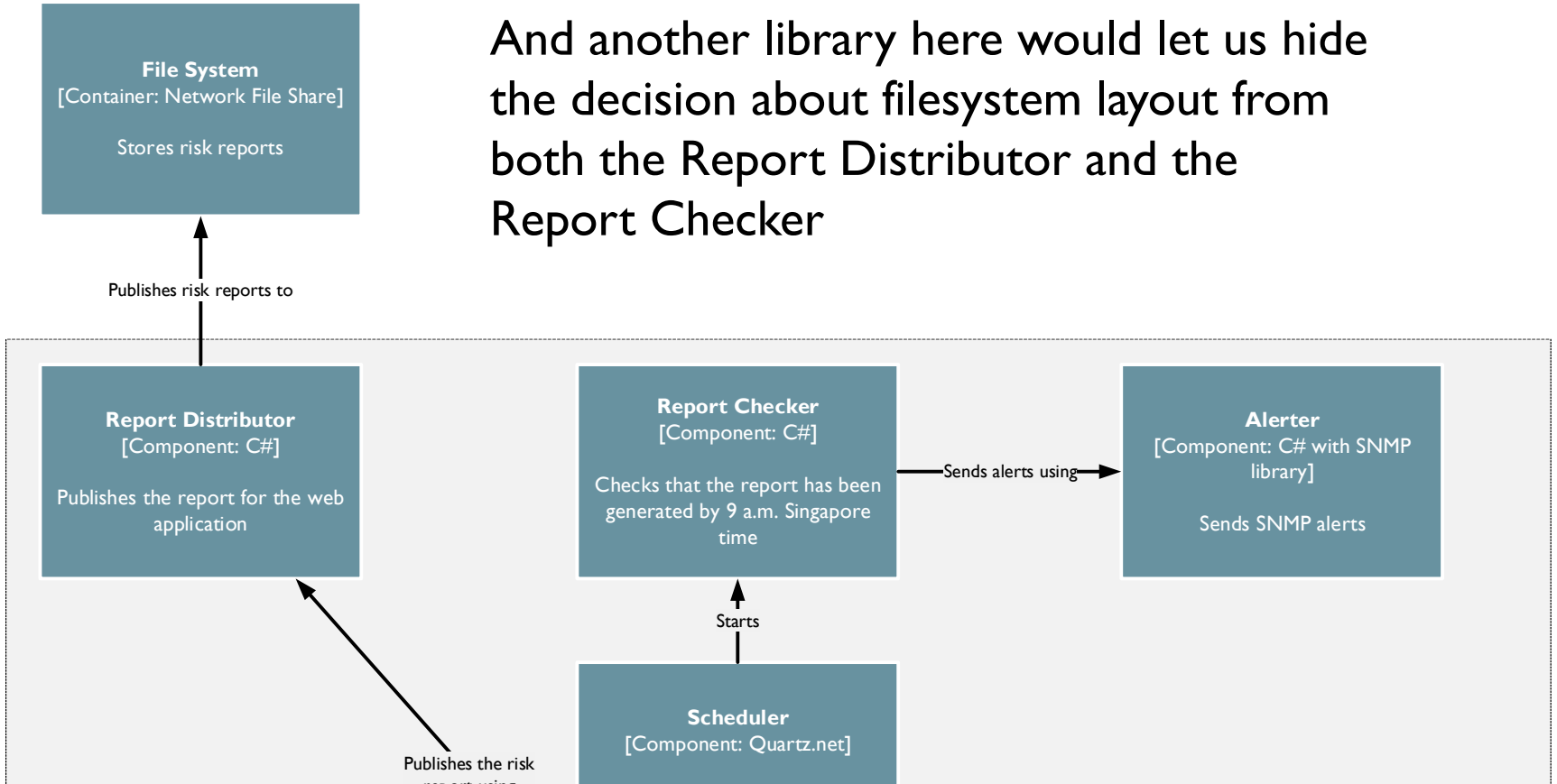
Solutions depend on architectural style

Here we're in a Windows service so we might use a shared library to define the interface.





This would be a good place to use a shared library for common implementation.



And another library here would let us hide the decision about filesystem layout from both the Report Distributor and the Report Checker

USE ALL YOUR TOOLS

1. Module structure – layout of your code and libraries
2. Component structure – interactions between runtime components
3. Abstraction – Emphasize similar interfaces & data formats

Find solutions by rotating your perspective

When looking at components, think about modules

When looking at modules, think about components

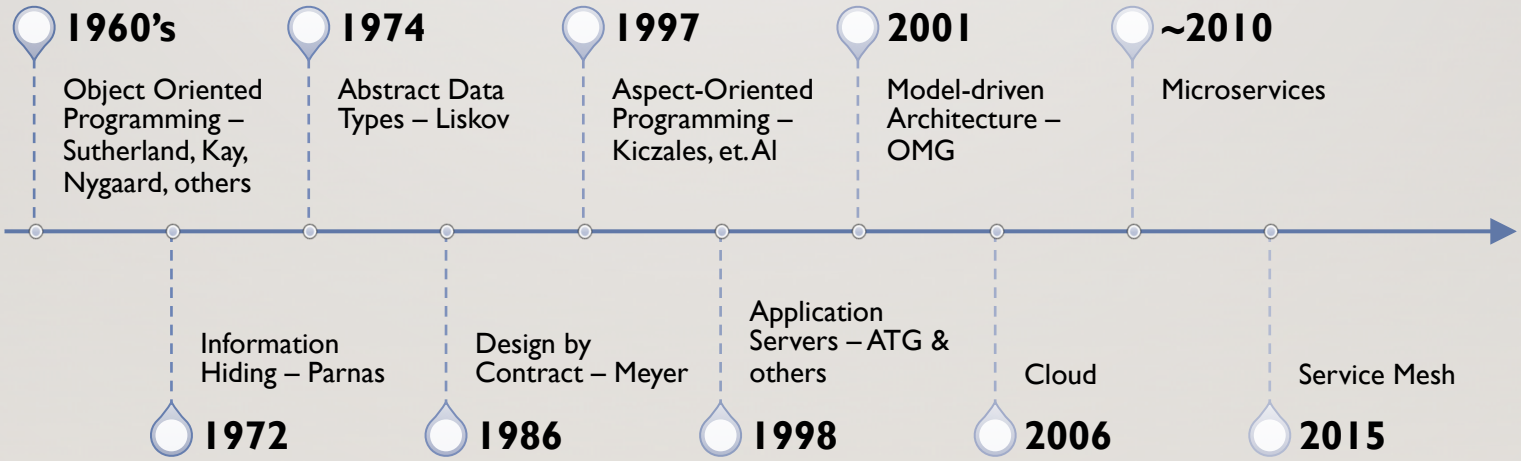
When looking at data, think about code

When looking at code, think about data

Clearing the Question Queue

SEPARATION OF CONCERNS

SoC: A PERENNIAL STRUGGLE



COMMON TO EVERY SYSTEM

- Input/Output channels
- Initialization
- Configuration, credentials
- Configuration, performance
- Storage
- Query
- Consistency
- Encryption, authn, authz
- Deployment
- Failure and recovery

DOMAIN SPECIFIC — SUBSCRIPTIONS

- Bank interface
- Payment handling
- Customer service
- Refunds
- Fraud detection/mitigation

IDEAL SEPARATION

- One mechanism per concern (maybe even less than one per concern!)
- All perfectly orthogonal & composable

PRAGMATICALLY: PICK YOUR BATTLES

- Look at your architectural priorities, constraints, and ASRs.
- Solve for those first

DIMENSIONS TO WORK WITH

- Modules (e.g., Library)
- Components
- Processes
- Hosts
- Services
- Geographies

Beware target fixation

EXAMPLE: CREATION CENTER

LIFETOUCH PHOTO STUDIOS

- Embedded in other stores
- Multiple brands
- (At the time) not reliably connected
- No on-site support staff
- High turnover of associates w/seasonal hiring
- Centralized printing facility

Products are regional and seasonal

Customers expect correct products

Production is centralized

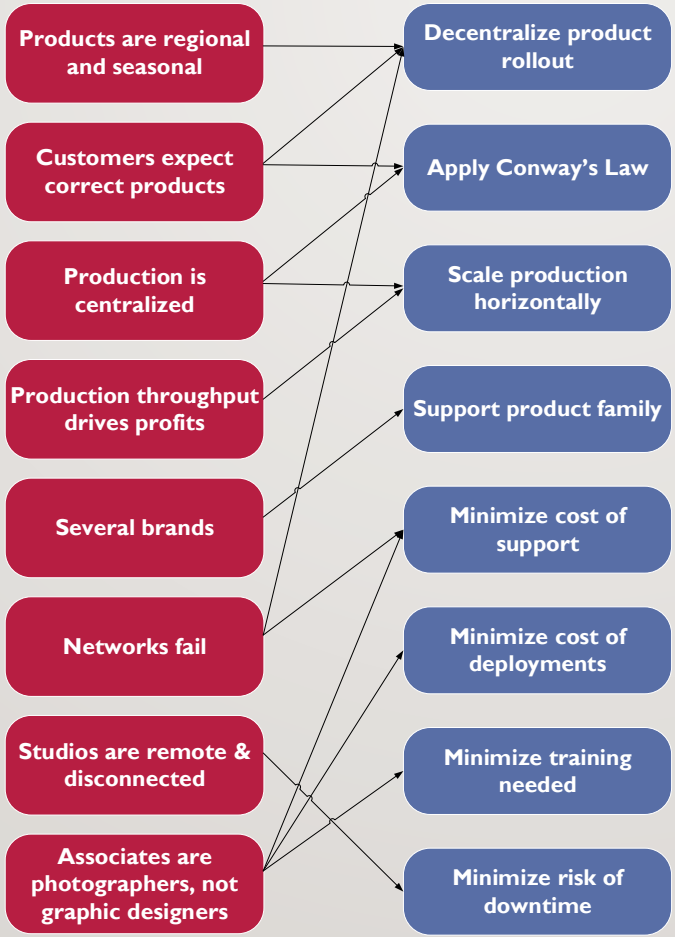
Production throughput drives profits

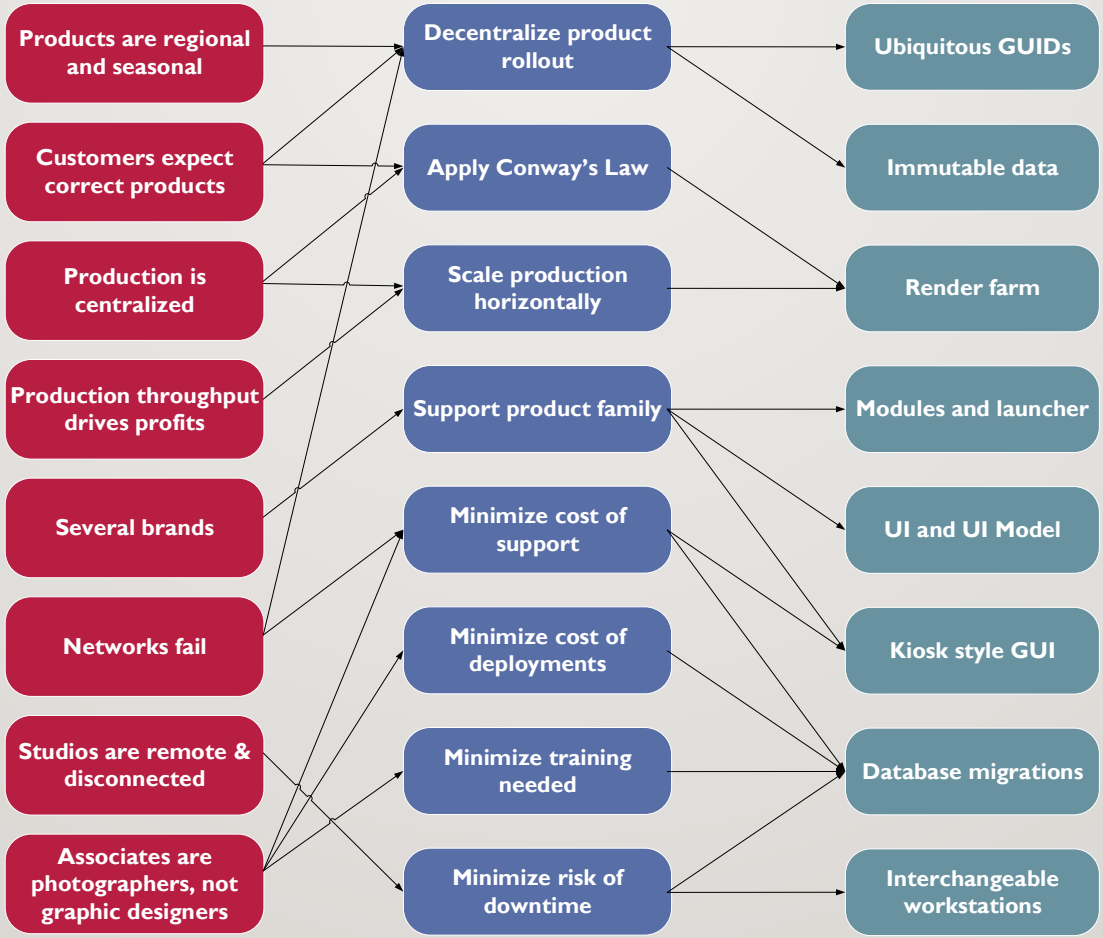
Several brands

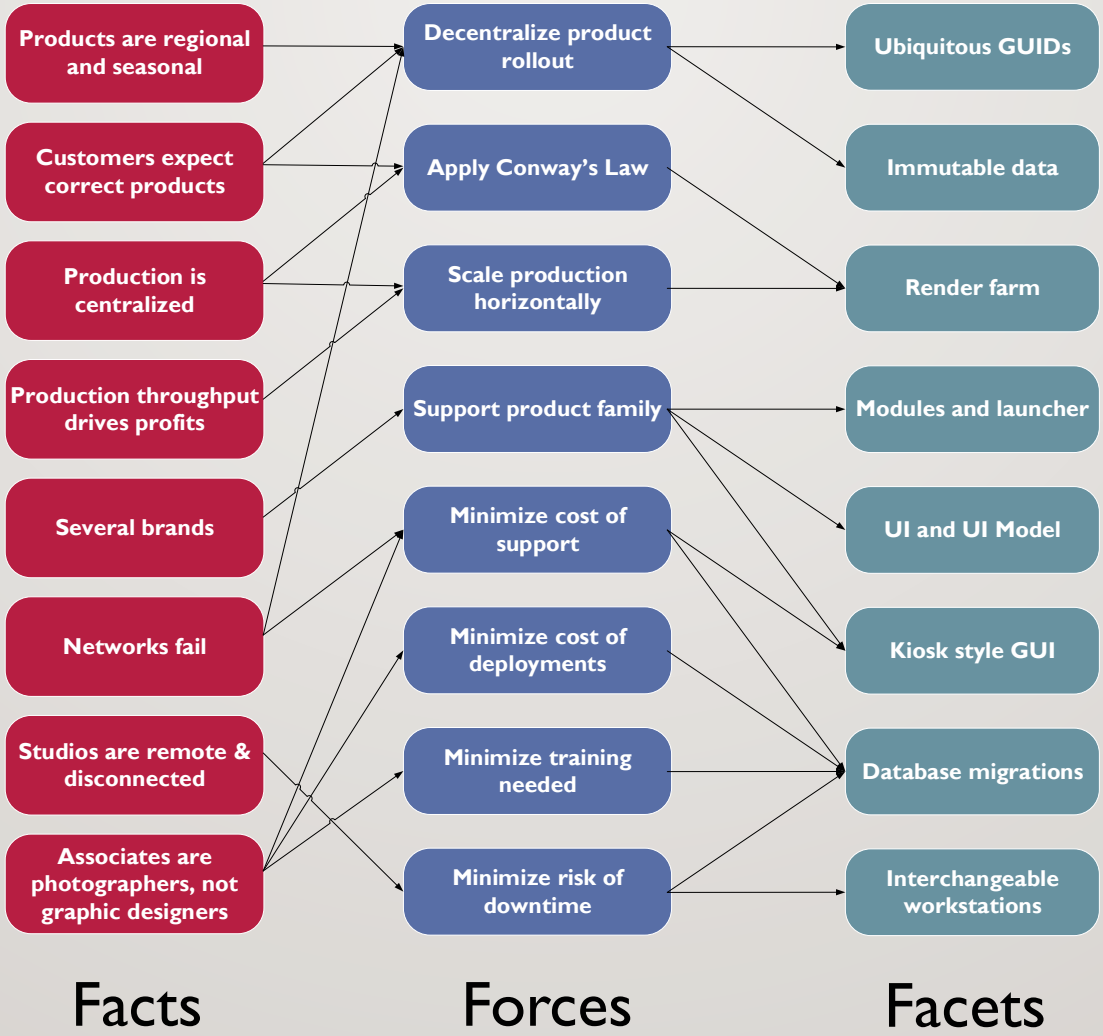
Networks fail

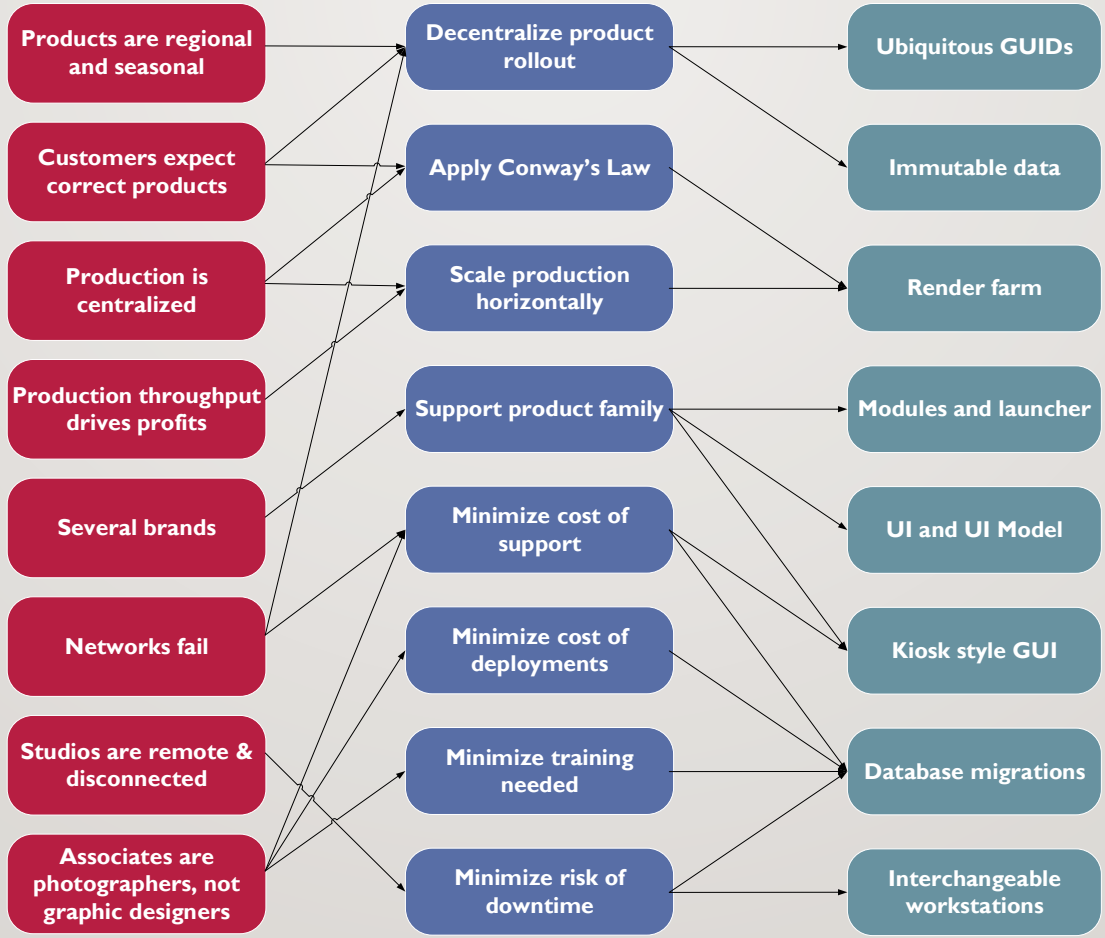
Studios are remote & disconnected

Associates are photographers, not graphic designers

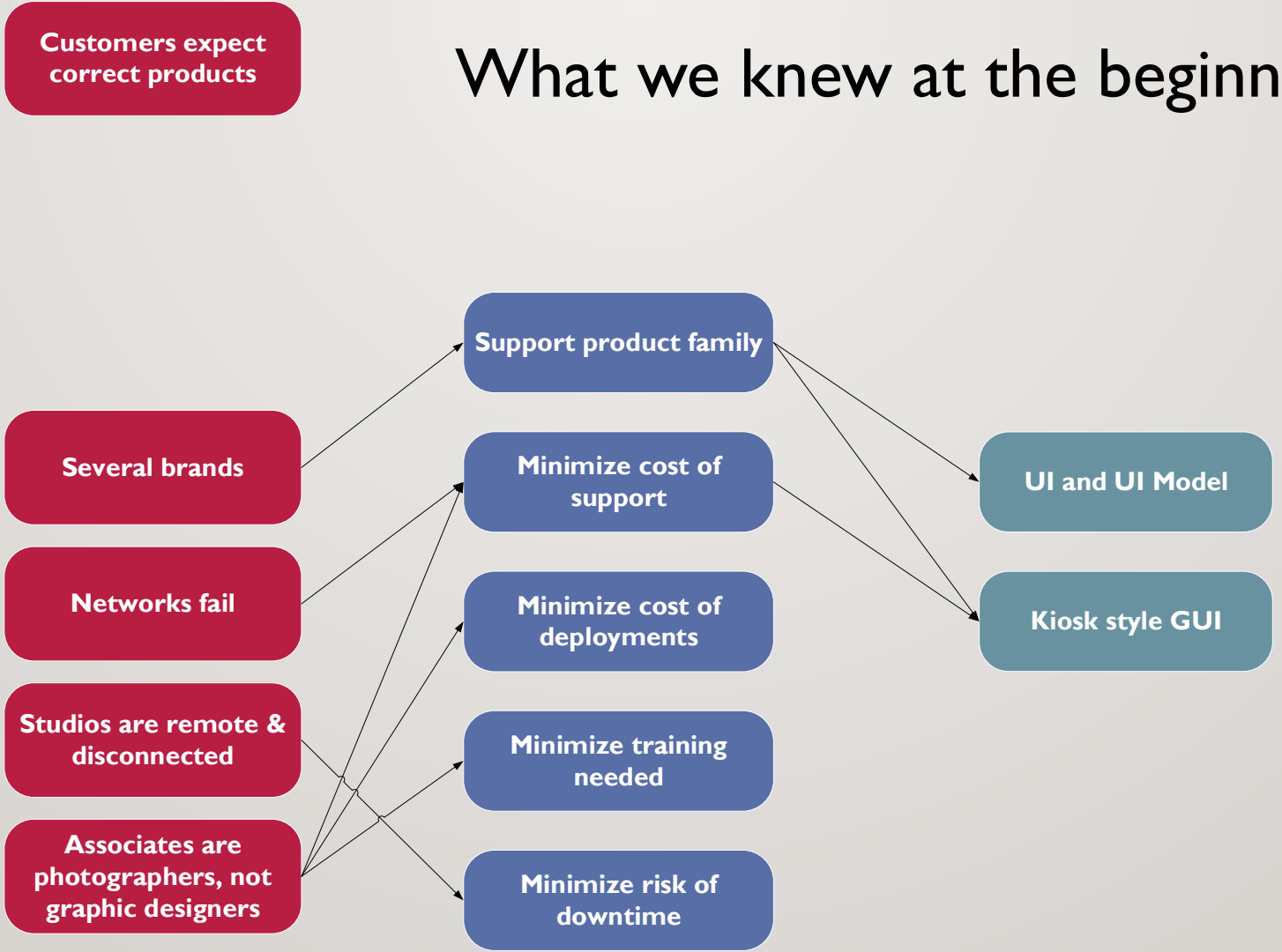

Products are regional and seasonal
Customers expect correct products
Production is centralized
Production throughput drives profits
Several brands
Networks fail
Studios are remote & disconnected
Associates are photographers, not graphic designers
Decentralize product rollout
Apply Conway's Law
Scale production horizontally
Support product family
Minimize cost of support
Minimize cost of deployments
Minimize training needed
Minimize risk of downtime


Products are regional and seasonal
Customers expect correct products
Production is centralized
Production throughput drives profits
Several brands
Networks fail
Studios are remote & disconnected
Associates are photographers, not graphic designers
Decentralize product rollout
Apply Conway's Law
Scale production horizontally
Support product family
Minimize cost of support
Minimize cost of deployments
Minimize training needed
Minimize risk of downtime
Ubiquitous GUIDs
Immutable data
Render farm
Modules and launcher
UI and UI Model
Kiosk style GUI
Database migrations
Interchangeable workstations


Products are regional and seasonal
Customers expect correct products
Production is centralized
Production throughput drives profits
Several brands
Networks fail
Studios are remote & disconnected
Associates are photographers, not graphic designers
Decentralize product rollout
Apply Conway's Law
Scale production horizontally
Support product family
Minimize cost of support
Minimize cost of deployments
Minimize training needed
Minimize risk of downtime
Ubiquitous GUIDs
Immutable data
Render farm
Modules and launcher
UI and UI Model
Kiosk style GUI
Database migrations
Interchangeable workstations
Facts
Forces
Facets

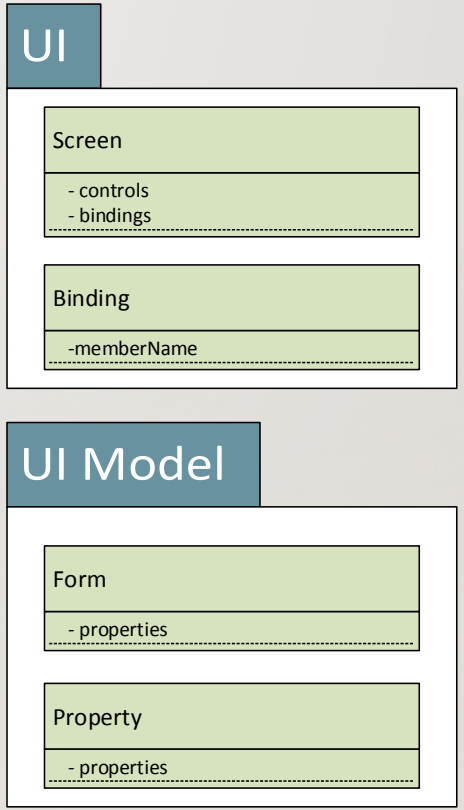

Products are regional and seasonal
Customers expect correct products
Production is centralized
Production throughput drives profits
Several brands
Networks fail
Studios are remote & disconnected
Associates are photographers, not graphic designers
Decentralize product rollout
Apply Conway's Law
Scale production horizontally
Support product family
Minimize cost of support
Minimize cost of deployments
Minimize training needed
Minimize risk of downtime
Ubiquitous GUIDs
Immutable data
Render farm
Modules and launcher
UI and UI Model
Kiosk style GUI
Database migrations
Interchangeable workstations
Constraints
ASRs
Concerns

What we knew at the beginning.



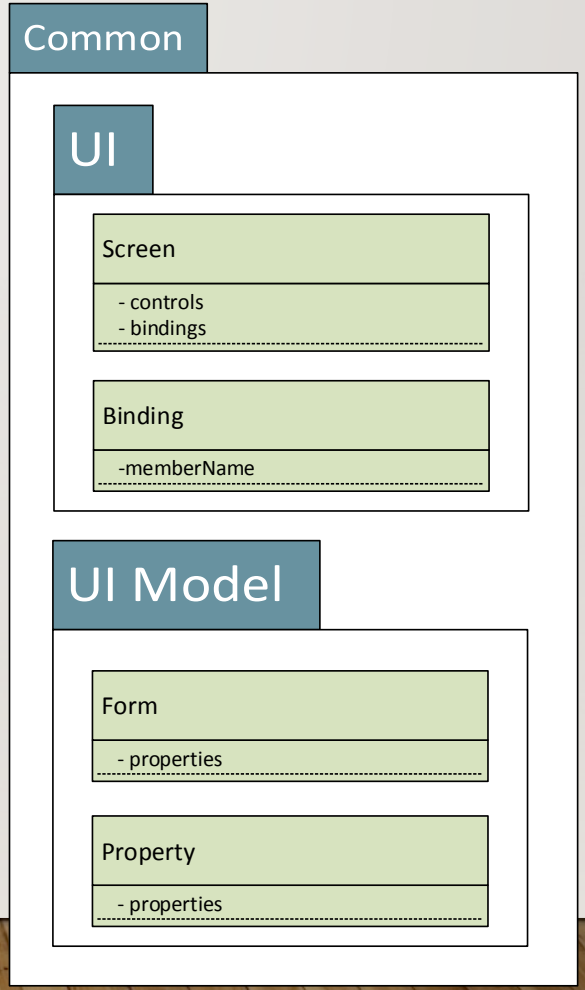
COMPOSITIONS THAT WORKED WELL

- Screen – visual. Populated with controls.
- Form – logical. Offers properties & coordinates their changes.
- Binding – mediator. Connects a property to one or more aspects of a control.



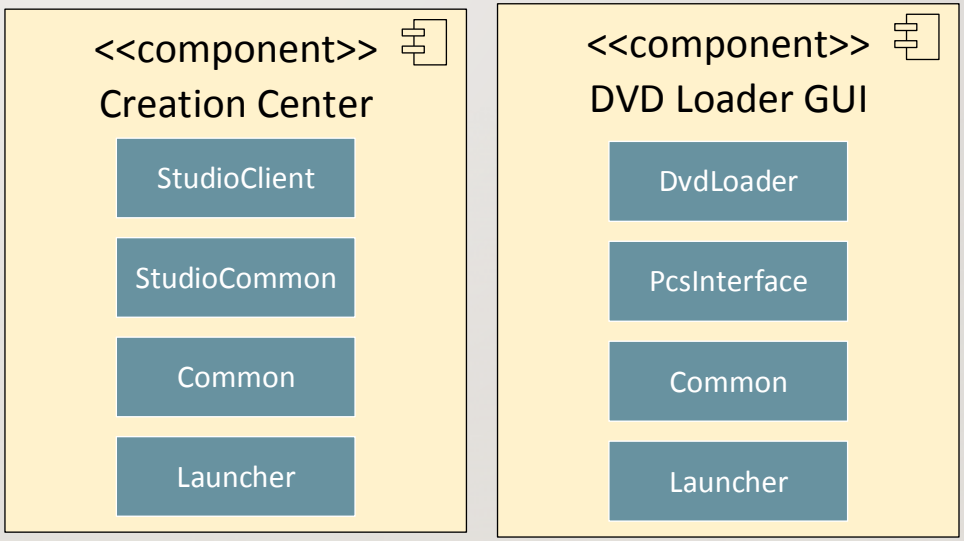
COMPOSITIONS THAT WORKED WELL

- Classes were packaged in the Common module.



COMPOSITIONS THAT WORKED WELL

- Modules bundled, run together
- `StudioClient`, `DvdLoader` and other GUI modules depend on `Common`
- DI files there create Form classes, but only instances of Property, Screen, Control, & Binding objects.



FROM MODULES TO COMPONENTS

We could combine modules into components. They didn't care what was in the component.

The UI machinery didn't care how it was packaged.

Deciding which GUI modules to use didn't impose any constraints on packaging.

Deciding on packaging didn't impose any constraint on the GUI.

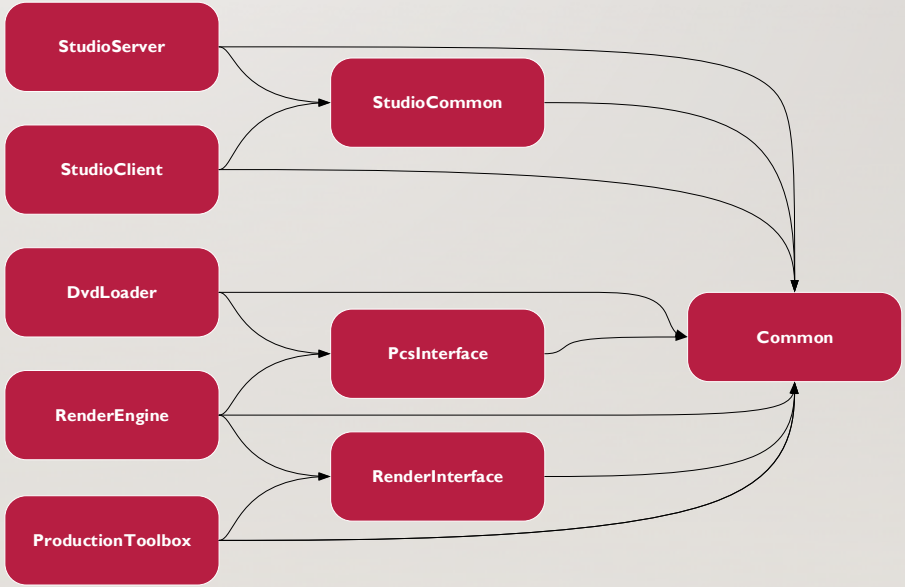
That's orthogonality.

THIS WAS ALSO INCREMENTAL ARCHITECTURE

1. Initial concept of layers
2. Property-binding architecture
3. Studio server vs Studio client
4. Use Spring for modules
5. Launcher builds classpath & configpath
6. Database migrations
7. Build setup.exe from CI, with test installation
8. Production interface, render farm, toolbox
9. Product creation GUI

BUT SOME CHALLENGES

- Common was the remains of our original monolithic project.
- *Everything* coupled to Common.



BUT SOME CHALLENGES

- Later, stores got connected.
- But the idea of a DVD was baked in hard
- That's what happens when ASRs and fundamental constraints change!

Clearing the Question Queue

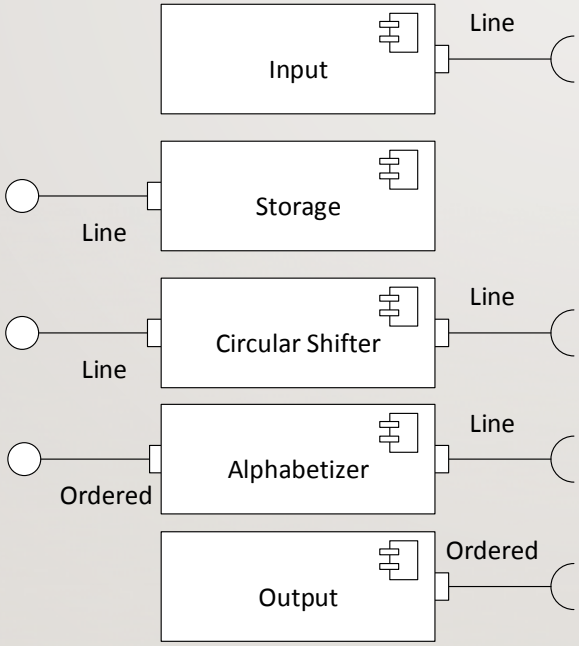
LOCALITY

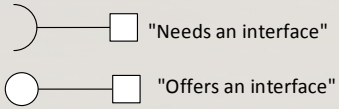
LOCALIZE DECISIONS; DON'T RYI
(Reveal Your Implementation)

- Don't let entity types proliferate through systems. Keep them local.
- Use common interfaces to avoid RYI
- Use common representations/media types/data formats to avoid RYI

RECALL THE KWIC INDEX

1. **Line Storage**
   Offers functional interface: SETCH, GETCH, GETW, DELW, DELLINE
2. **Input**
   Reads EBCDIC chars, calls line storage to put them into lines.
3. **Circular Shifter**
   Offers same interface as line storage. Makes it appear to have all shifts of all lines.
4. **Alphabetizer**
   Offers sort function INIT, and access function iTH that gets a line.
5. **Output**
   Repeatedly call iTH on alphabetizer, printing the line.
6. **Control**
   Similar to first approach, call each module in sequence.

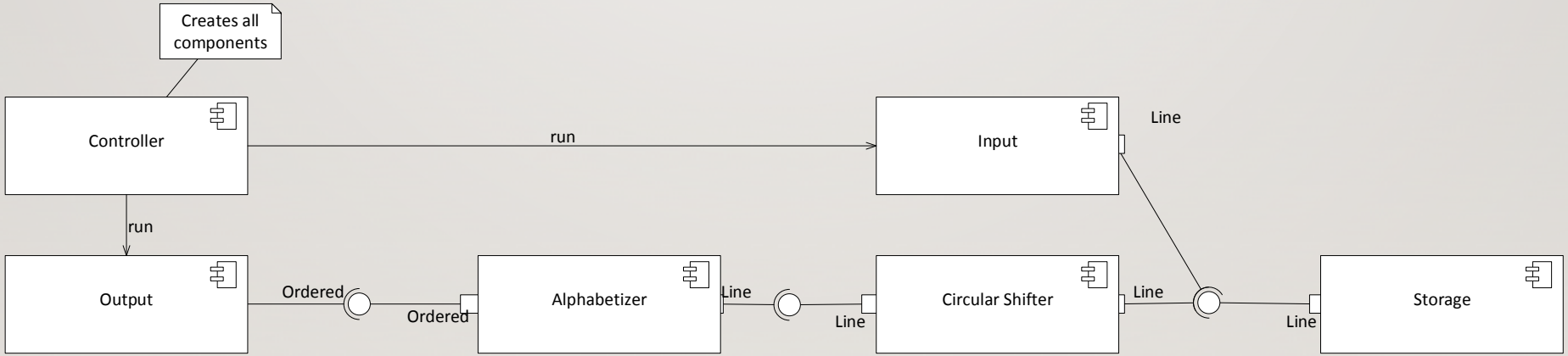




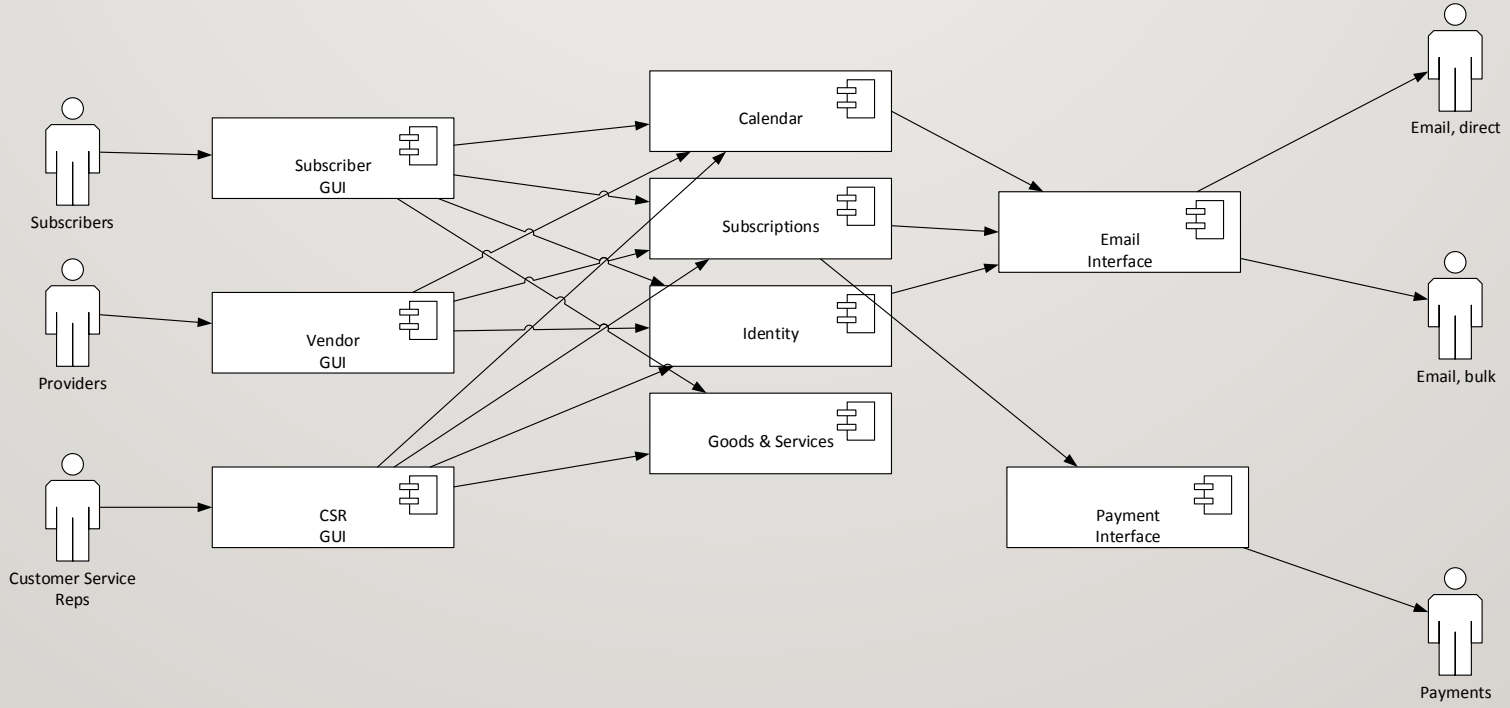
WHY DID THE SECOND MODULARIZATION SURVIVE CHANGE BETTER?

- Very few data types
- Small number of well defined interfaces
- Highly composable
- Limited RYI

WHY DID THE SECOND MODULARIZATION SURVIVE CHANGE BETTER?

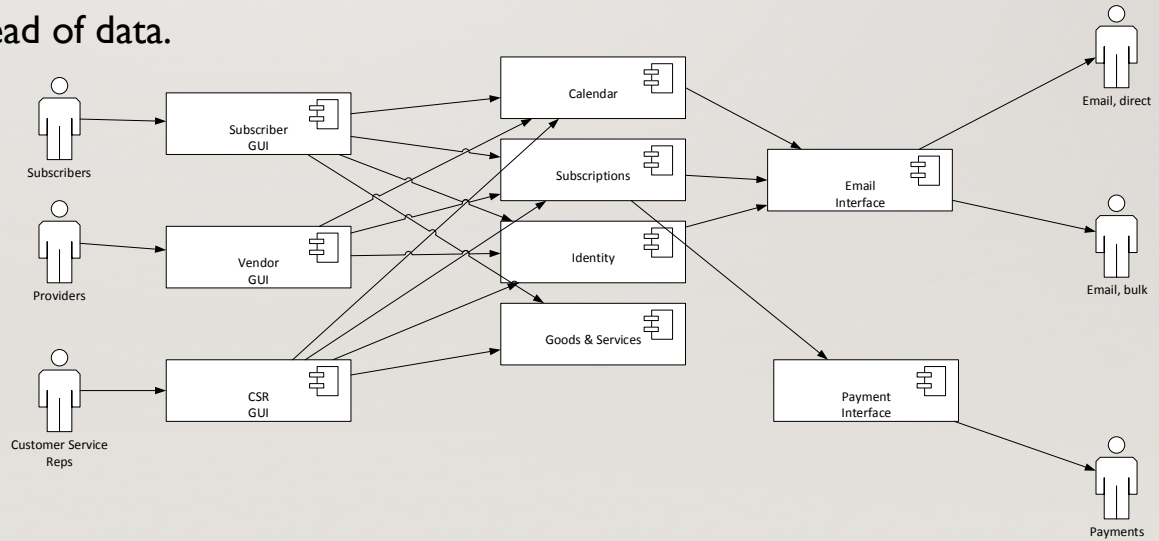


BACK TO OUR SAMPLE SYSTEM



ACTIVITY: UNCOVER A COMMON INTERFACE

- What kinds of things are done by many uses?
- Think about behavior instead of data.
- Ideas in chat, please.



ONE IDEA: RECURRING ACTIONS

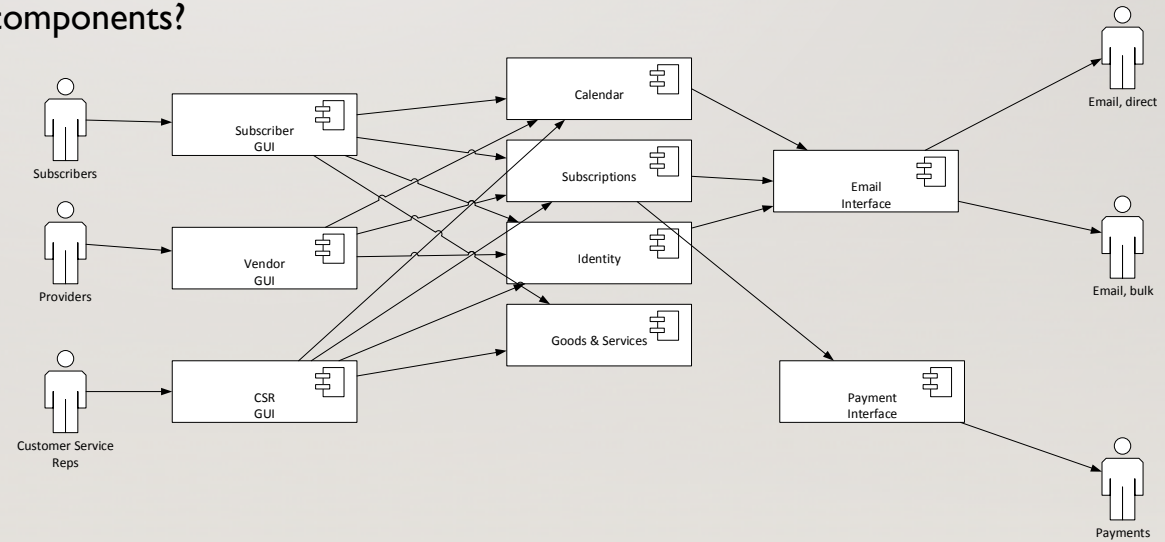
- Subscription: Charge a credit card periodically. Do something when it succeeds.
- Subscription: Signal a vendor to deliver services periodically.
- Payment methods: Send email before credit card expires.

A component that computes dates into the future and schedules actions?

Or an interface for a "time based event" that multiple components can implement?

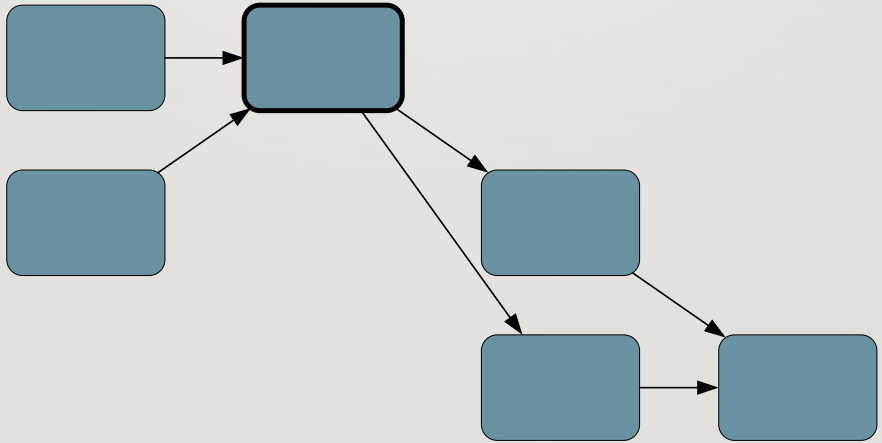
ACTIVITY: THINK OF A COMMON MECHANISM

- How did the components get created?
- What modules go into the components?
- Are there places we can benefit from sharing a module?
- Can we turn any components into shared modules?
- Ideas in chat, please.

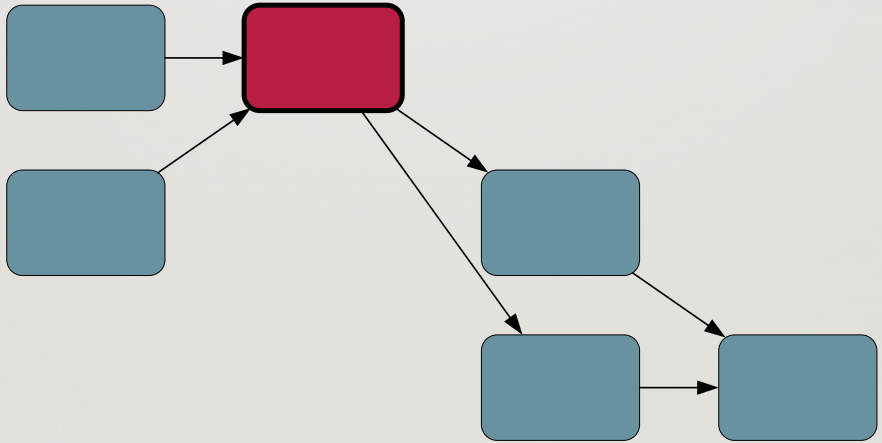


UPSTREAM AND DOWNSTREAM

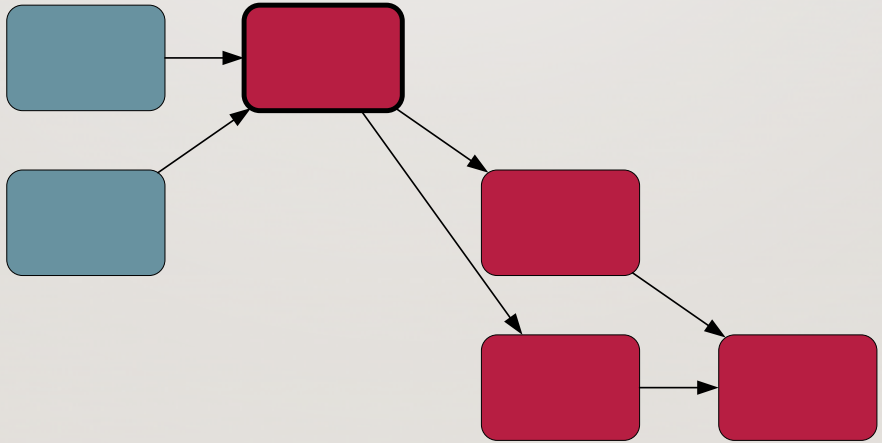
WE WORK ON ONE OR TWO COMPONENTS AT A TIME



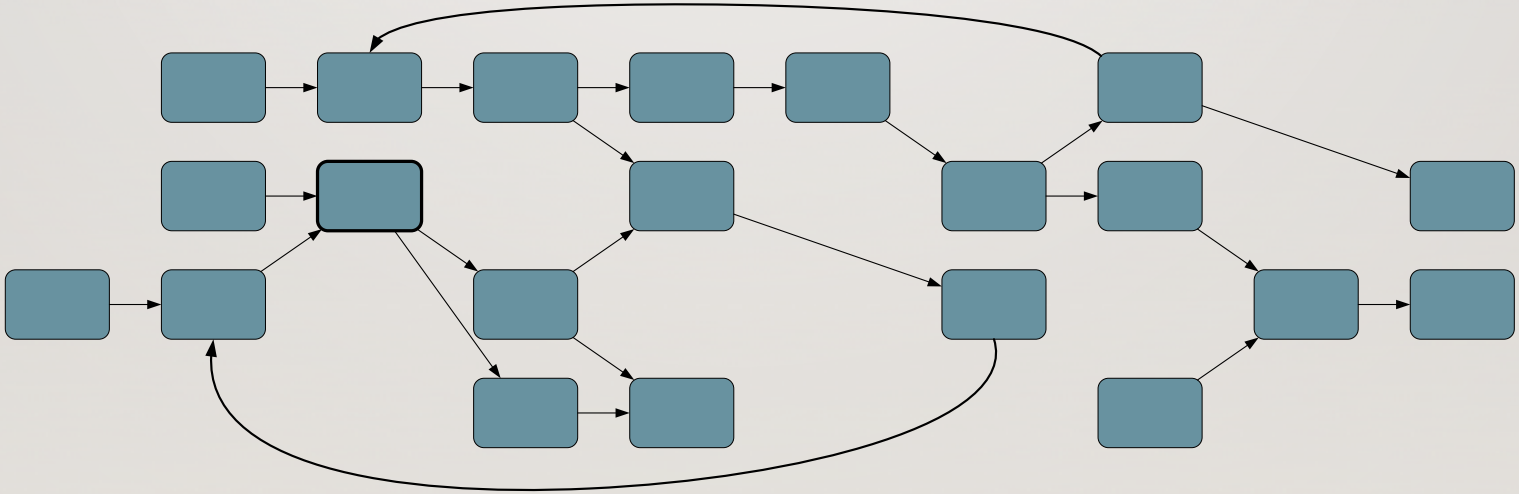
SO WE MAKE A CHANGE



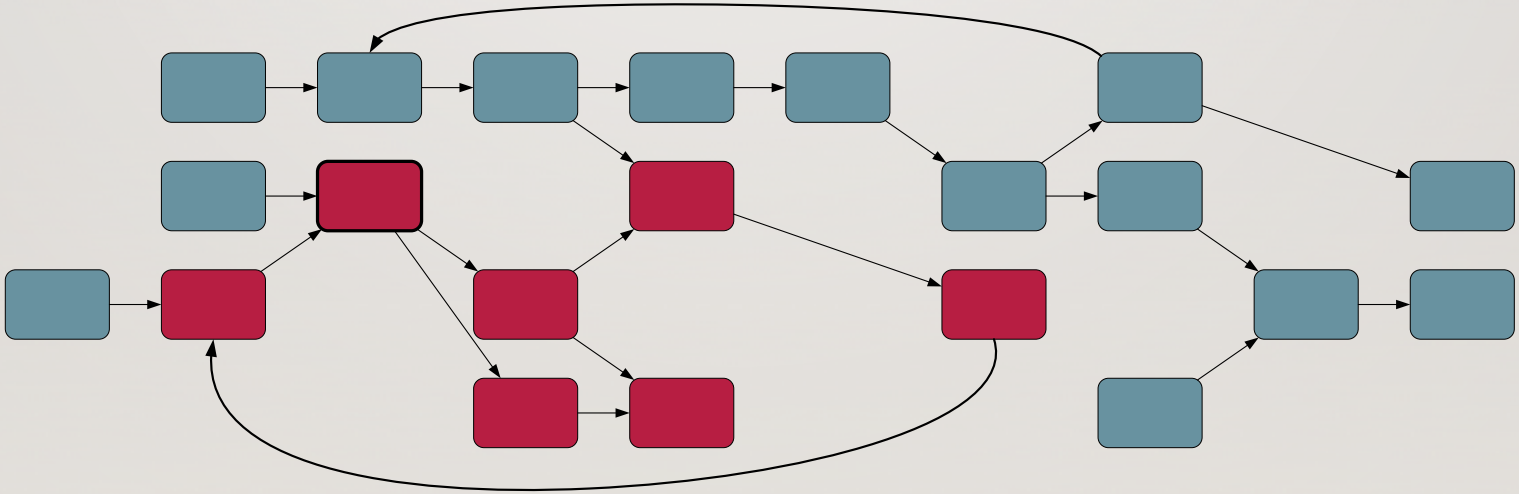
THAT HAS A RIPPLE EFFECT



BUT THE ENTERPRISE REALLY LOOKS LIKE THIS



AND OUR CHANGE HAS A BIG "SURFACE AREA"



REDUCING THE SURFACE AREA OF CHANGE

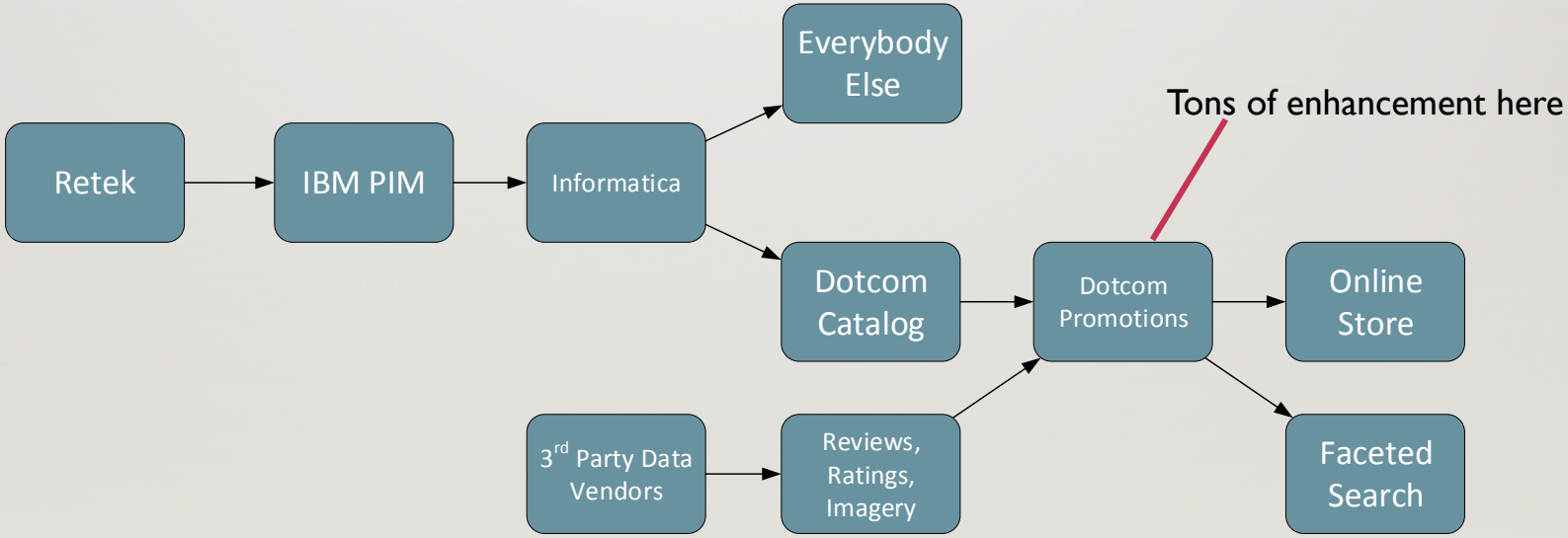
1. Augment Upstream
2. Contextualize Downstream

AUGMENT UPSTREAM

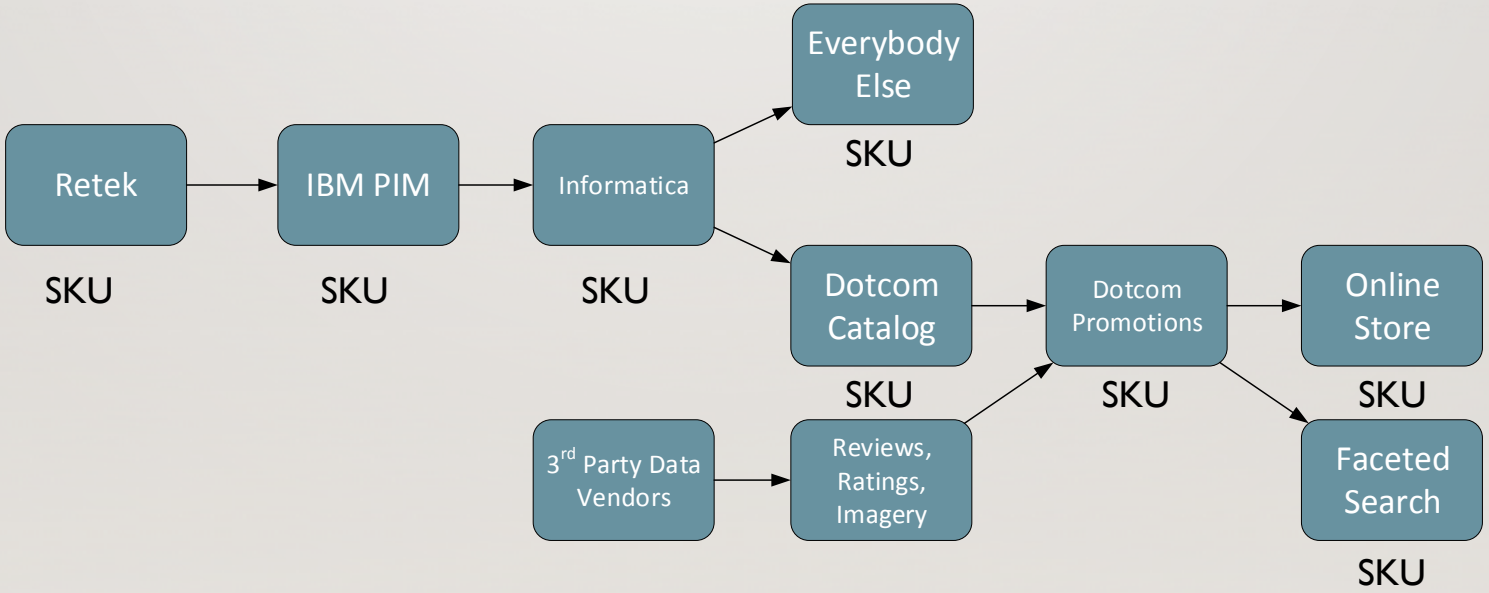
AUGMENTING

- Add to data as "early" as possible
  - Combine sources
  - Add human judgement
  - Apply ML models
- Avoid creating privileged downstreams
- Everybody wants the best data available

COUNTEREXAMPLE



ALSO AN EXAMPLE OF SEMANTIC COUPLING



SKU WAS A COMPOSITE

- Many types of attributes carried together
- Historically, these were **always** a unit
- People thought of "SKU" as a real thing, forgot that it's just a label for a collection of attributes that sometimes describe the same thing.

COGS
Distribution
Stocking
Presentation
Pricing
Delivery
Inventory

- More to that story later…

KINDS OF AUGMENTATION

- Adding attributes
- Connecting entities from different sources
- Adjusting cardinalities
- Making aggregates
- Adding derived or discovered attributes

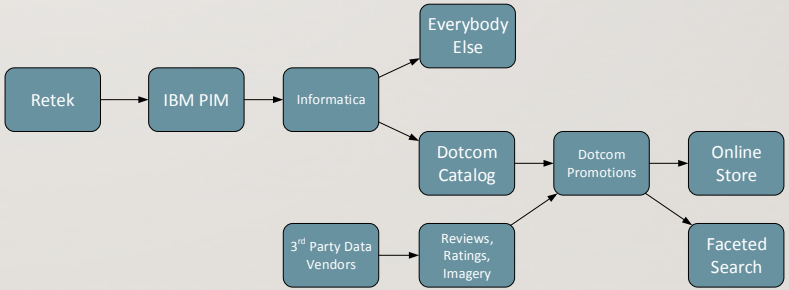
CONTEXTUALIZE DOWNSTREAM

CONTEXTUALIZING

- Applying policies and restrictions
- “isValid”
- Limiting the extent of an entity (i.e., restricting which instances to offer)
- Limiting the breadth of an entity (restricting which attributes to offer)

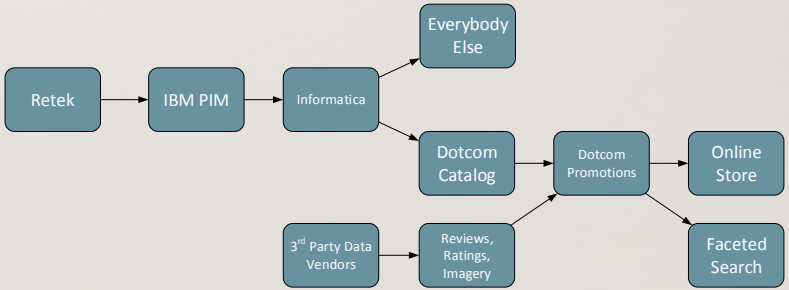
EXAMPLE: STREET DATE

- "Street date" – released for sale
- SKUs not passed from PIM until after street date
- Decision about display to end customer also impacted users of internal systems
- Cannot prepare for online display
- Cannot take pre-orders!



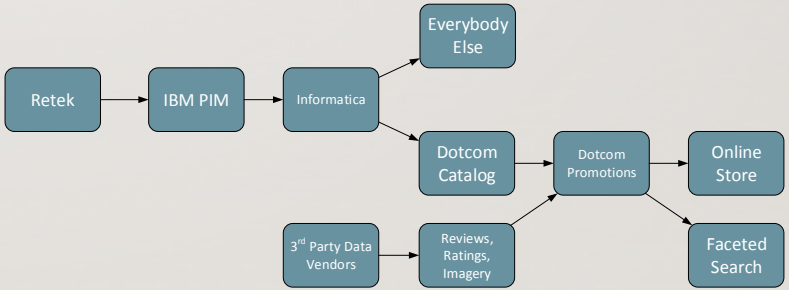
EXAMPLE: STREET DATE

- Augment upstream:
  Add attribute "street date in past?"
- Contextualize downstream:
  Send the SKUs,
  GUIs decide whether to show
  APIs decide whether to show



QUESTION: WHY "STREET DATE IN PAST?"

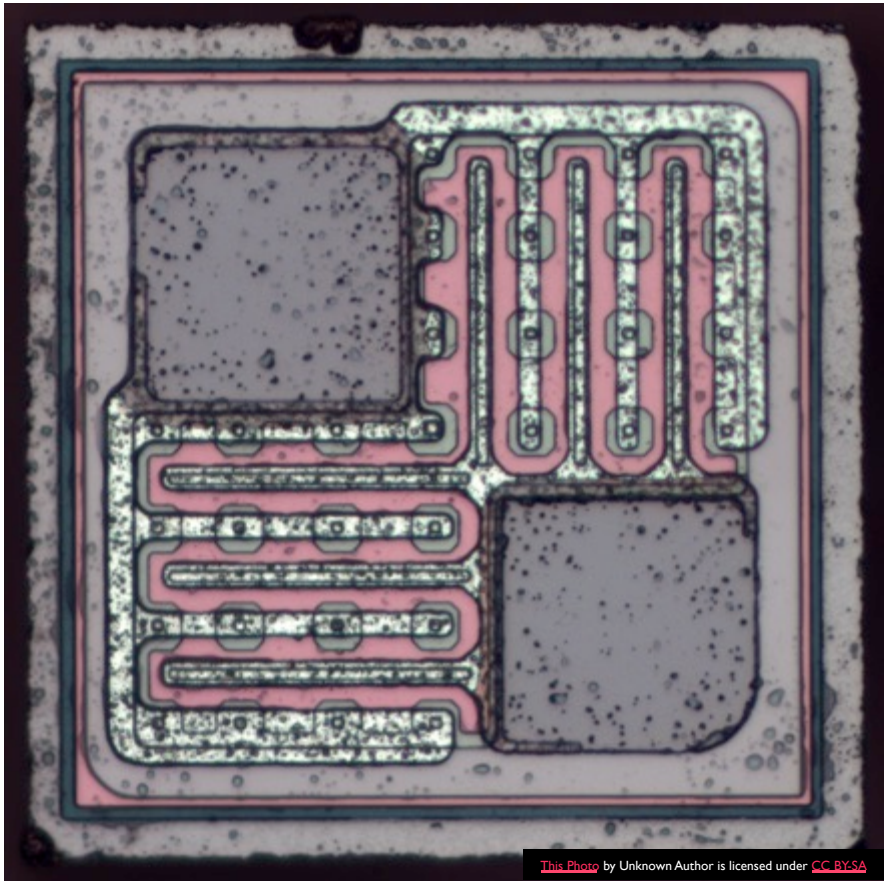
1. What would happen if we added an "allowDisplay" boolean to the SKU?
2. What would happen if we added "displayFlag = CUSTOMER" "displayFlag = INTERNAL"?

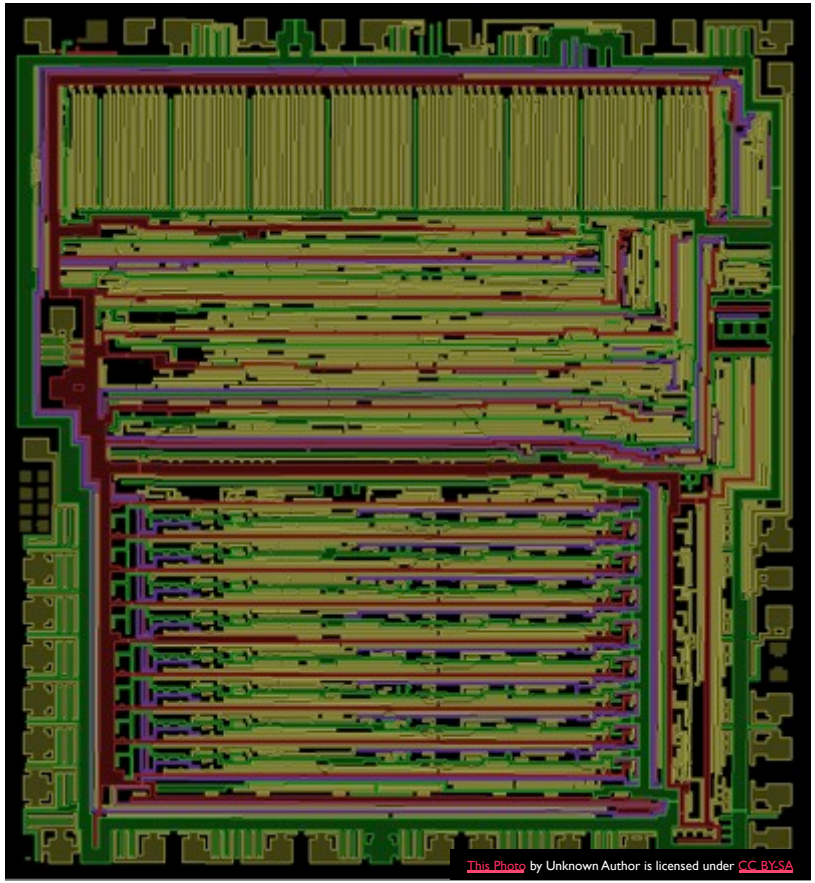


Your thoughts in chat, please.

Clearing the Question Queue

COMPLEX SYSTEMS





COMPLEXITY ARISES FROM INTERACTIONS

The behavior of the ensemble cannot be predicted from the behavior of the parts.

IMPLICATIONS

There is no steady state

Just overlapping waves of change

"New" or "old" information depends on where you sit

You are not the only source of change.

LOCAL CONTEXT

ACT LOCALLY; THINK GLOBALLY

1

Don't push work into the gaps between units

2

Avoid optimizing your work at the expense of your upstream

3

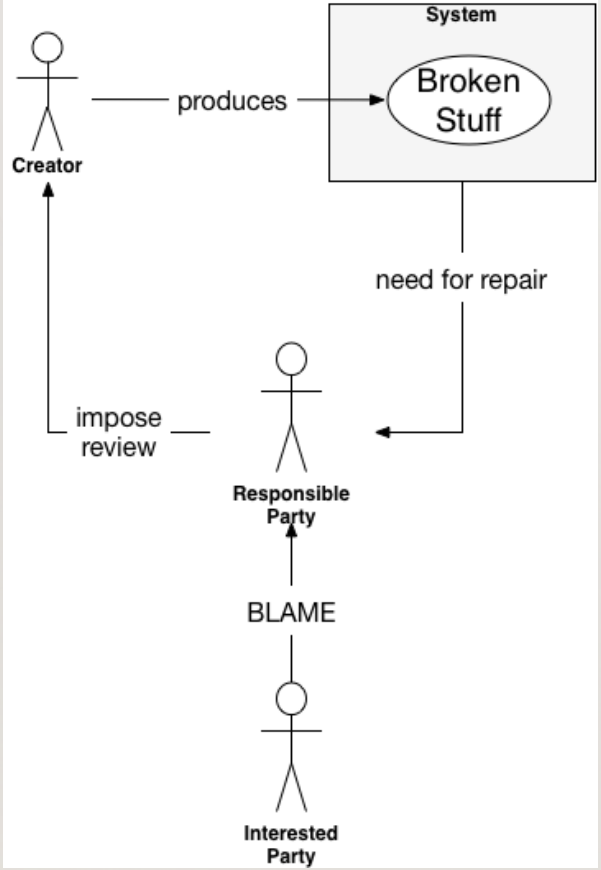
Design your processes from back-to-front

COUNTEREXAMPLE: PUSHING WORK INTO GAPS

- Content management system
- Too expensive to build "publish on" feature
- Required midnight work by merchants
- One knew VBA, automated content publishing through spreadsheet
- Multi-million $ content workflow ignored by users
- Spreadsheet survived two CMS replacements, became "mission critical"

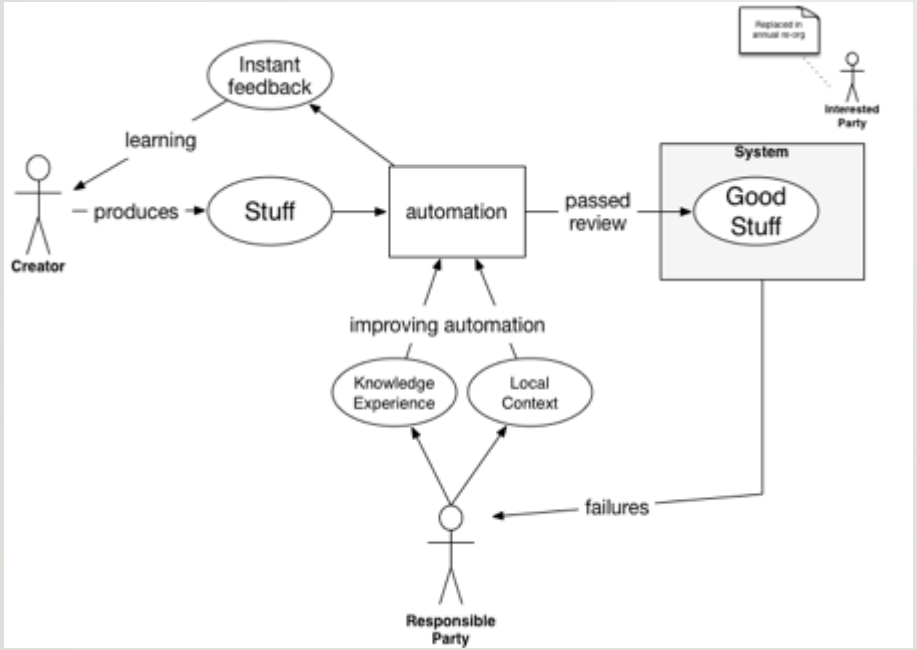
OPTIMIZING YOUR WORK AT THE EXPENSE OF UPSTREAM

- “Deployment Documents”
- Most forms of change review, architecture review



BAD EXAMPLE FROM OPS

BETTER EXAMPLE FROM OPS



PROCESS DESIGN

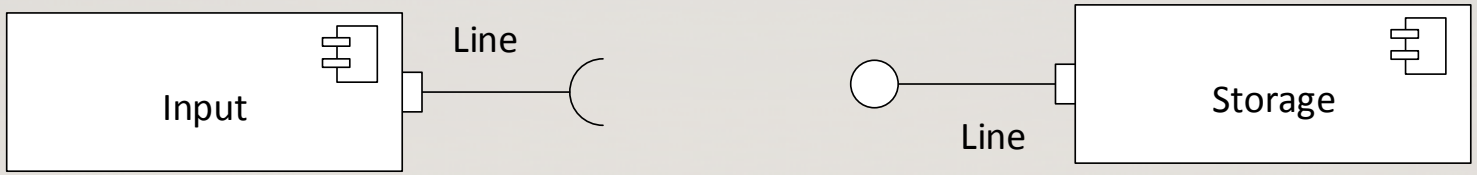
- Don’t start with “what do we produce”
- Start with “what does my customer need”

1. How do I know my output meets their specifications?
2. How do they tell me when to deliver my output?
3. How do I deliver that output?
4. How do I produce the output?
5. What inputs, skills, and tools do I need to produce the output?
6. Who do I get them from?
7. How do I measure my inputs to see if they meet my specifications?
8. How do I tell my sources to deliver their inputs?

Quality/feedback	1. How do I know my output meets their specifications?
Signaling	2. How do they tell me when to deliver my output?
Logistics	3. How do I deliver that output?
Methods	4. How do I produce the output?
Requirements	5. What inputs, skills, and tools do I need to produce the output?
Sources	6. Who do I get them from?
Quality	7. How do I measure my inputs to see if they meet my specifications?
Signaling	8. How do I tell my sources to deliver their inputs?

PROCESS DESIGN

- Creates an integrated value stream
- Connects all the way to the customer
- Avoids the deadwood of hiring "gap fillers" between groups

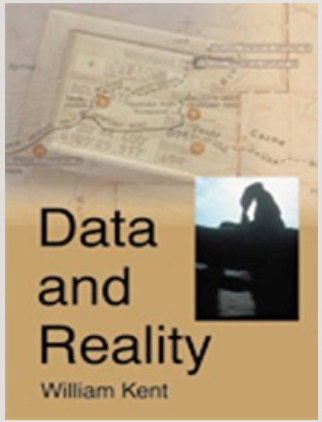


INFORMATION ARCHITECTURE

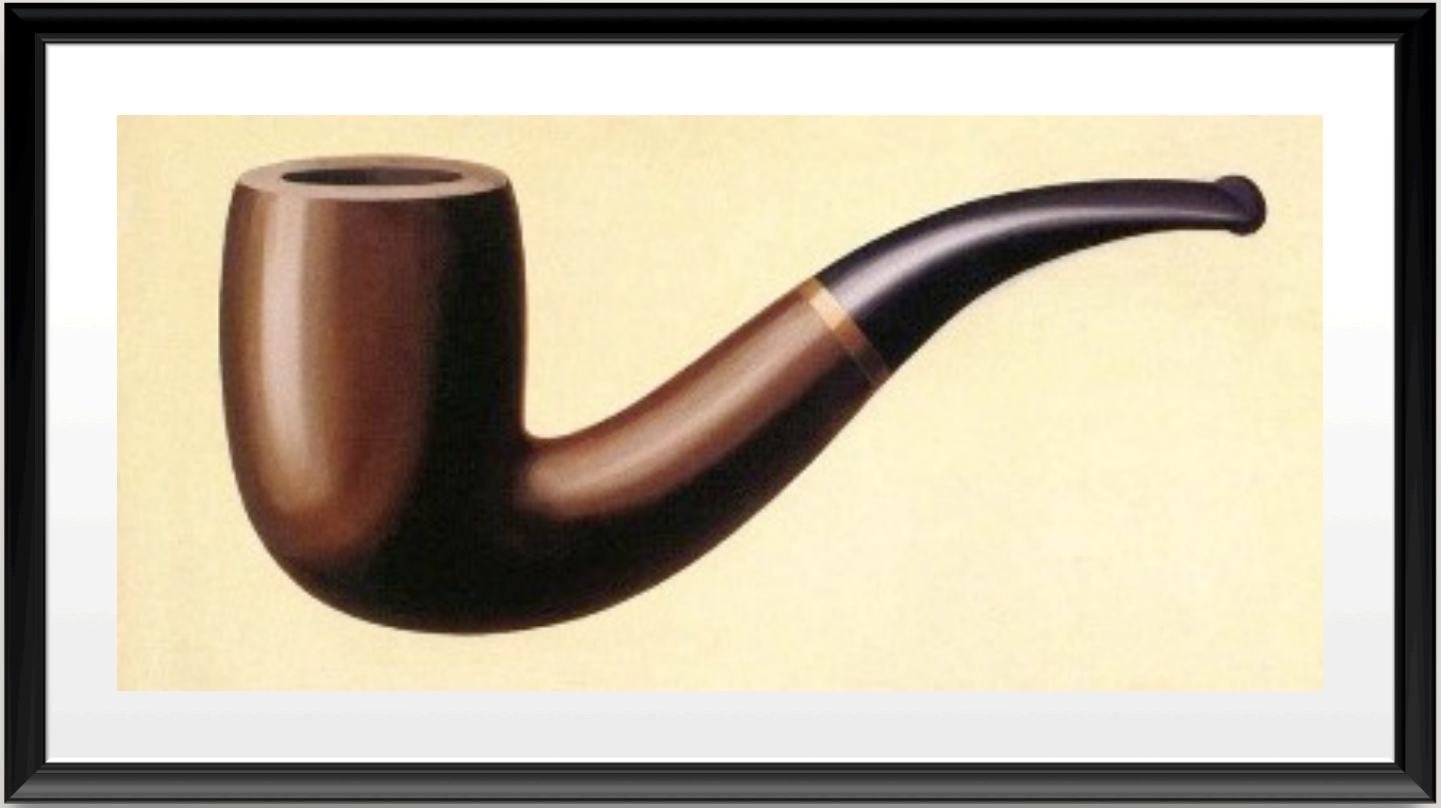
DATA AND REALITY

DATA AND REALITY

SECOND EDITION



THIRD EDITION



"La Trahison des Images" ("The Treachery of Images")
By René Magritte, 1898-1967. The work is now owned by and exhibited at LACMA.

Ce n'est pas un

THE SYMBOL IS NOT THE THING

- We don't put people in our databases.
- We put representations of people in our databases.

TYPE:

1. Set of allowed values
2. Operations on those values

TYPE:

1. ~~Set of allowed values~~
   Decision process to determine if a value is a member
2. Operations on those values

ENUMERATED TYPE

We can list all the allowed values

Decision process just checks for membership in the set.

E.g., Boolean or rows of a table

ENCODED TYPE

- Some types are hard to enumerate.
- Very big sets: E.g., all rational numbers.
- Sets subject to redefinition: E.g., Addresses, Names, other social constructs

- We encode these in primitive values

COMMON ENCODINGS

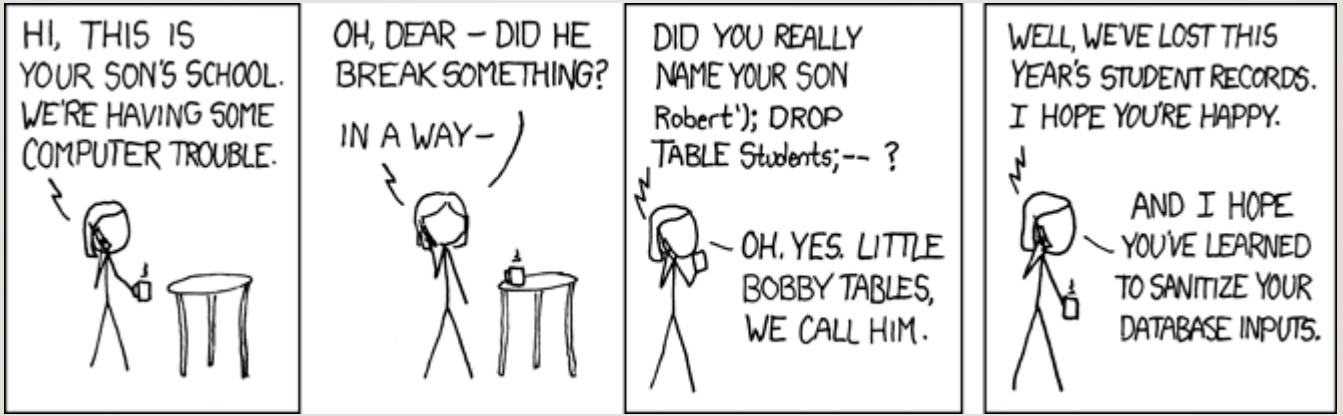
Data type	Representation
Name	String
Height	16-bit 2's complement
Salary	16-bit 2's complement

THE TROUBLE WITH ENCODING

- The decision procedure becomes checking syntax
- "Does this value meet the syntax for encoding?"
- Rather than "Is this value a member of this type?"

- Opens door to incorrect acceptance or rejection of values

THIS IS A TYPE ERROR



https://www.xkcd.com/327/

ATTRIBUTE TYPES

- Height **isa** Integer?
  - Lacks unit
  - Allows too many operations
  - Allows too many values… what does a negative height mean?

ANOTHER BROKEN ATTRIBUTE TYPE

- Money **isa** double?
  - Imprecise operations, will lose or manufacture money
  - Imprecise representation
  - Lacks currency, so allows illegal conversions and combinations
  - Allows too many operations.
  - What does $\$5 \; mod \; 3$ mean?

- Please don't ever do this.

YET ANOTHER BROKEN ATTRIBUTE TYPE

- Datetime **isa** long?
  - Allows too many operations.
  - Allows incorrect closure under subtraction. $Time - Time \rightarrow Duration$
  - Is it wall clock (goes backward, jumps forward)?
  - Is it system time?
  - What time zone?

PITFALLS OF SYNTAX

Data type	Representation	Sample value	Pitfall
Name	String	♣♦♥♠↑↓↑↓ ←→←→™	Canonicalization. Duplicates.
Height	16-bit 2's complement	190	Inches? Meters? Pixels?
Salary	16-bit 2's complement	2.5	'Height' x 'Salary' == ???

MY RECURRING THEME

- Focus on the behavior.
- What are valid operations on the attributes?
- Can you make a useful type equation about the attributes?

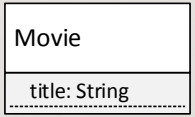
E.g., does yield anything meaningful?

What about ?

ATTRIBUTE VERSUS RELATIONSHIP

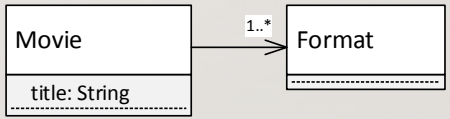
THE STAR TREK PROBLEM

- "Star Trek II: The Wrath of Khan is a movie"



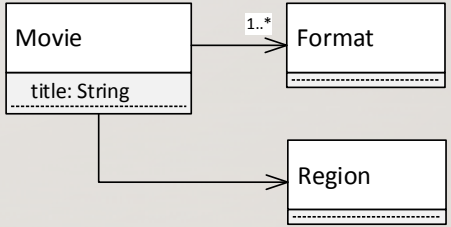
THE STAR TREK PROBLEM

- "Star Trek II: The Wrath of Khan is a movie"
- It is available on Blu-ray and DVD



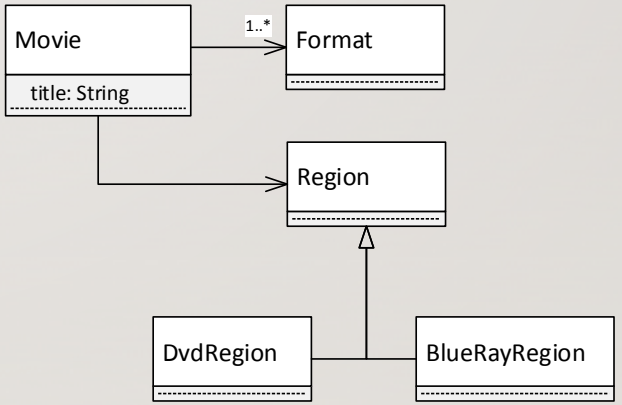
THE STAR TREK PROBLEM

- "Star Trek II: The Wrath of Khan is a movie"
- It is available on Blu-ray and DVD
- DVDs are region-coded



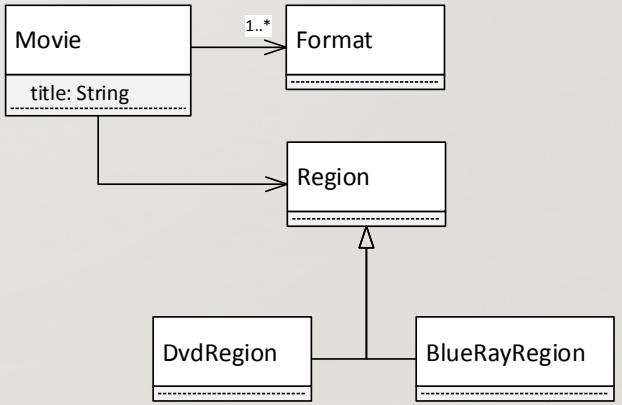
THE STAR TREK PROBLEM

- "Star Trek II: The Wrath of Khan is a movie"
- It is available on Blu-ray and DVD
- DVDs are region-coded
- Blu-rays are region-coded with different regions



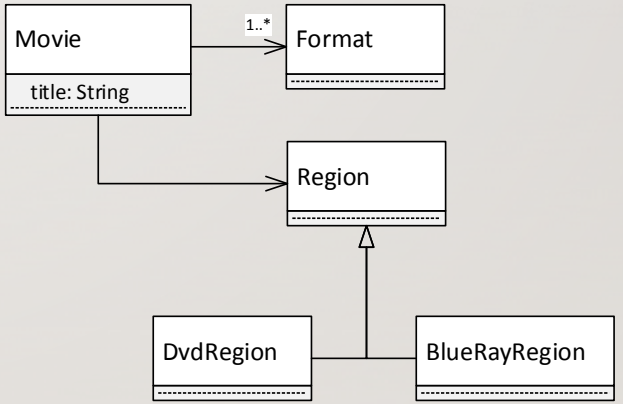
THE STAR TREK PROBLEM

- "Star Trek II: The Wrath of Khan is a movie"
- It is available on Blu-ray and DVD
- DVDs are region-coded
- Blu-rays are region-coded with different regions



THE STAR TREK PROBLEM

- "Star Trek II: The Wrath of Khan is a movie"
- It is available on Blu-ray and DVD
- DVDs are region-coded
- Blu-rays are region-coded with different regions



- "I want all Star Trek movies with the original cast."
- "I want all Star Trek movies in the JJ Abrams timeline."
- Now track inventory of each format of disc in each store location (entity == type of disc)
- Now track digital licenses issues for online viewing (entity == individual playback)

THE STAR TREK PROBLEM

- Precise, normalized models exist to exclude invalid values.
- Hard to use when previously unknown or invalid dimensions come into play.
- Grouping or collecting in new ways requires new attributes,
  which means more precision in the model.

TAKEAWAYS

- Data is not reality
- We represent parts of reality in our systems
- Models exist to represent data
- But they also exclude whatever can't be represented

Clearing the Question Queue

SOME DIFFICULT IDEAS

SOME DIFFICULT IDEAS

- Data duplication is OK. Just make sure you can reconcile the duplicates.

SOME DIFFICULT IDEAS

- Data duplication is OK. Just make sure you can reconcile the duplicates.
- There is no "natural" data model

SOME DIFFICULT IDEAS

- Data duplication is OK. Just make sure you can reconcile the duplicates.
- There is no "natural" data model
- Relational, KVS, hierarchic, network, document, graph… they're all just ways to group attributes into entities.

SOME DIFFICULT IDEAS

- Data duplication is OK. Just make sure you can reconcile the duplicates.
- There is no "natural" data model
- Relational, KVS, hierarchic, network, document, graph… they're all just ways to group attributes into entities.
- Each one is optimized for certain kinds of transaction and query patterns

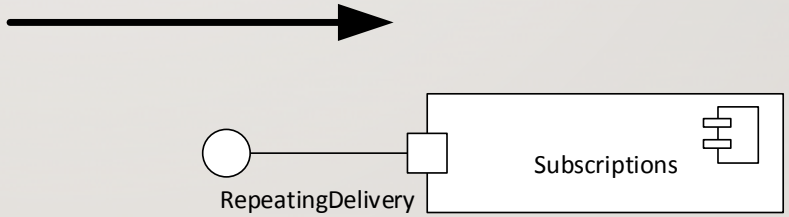
SOME DIFFICULT IDEAS

- Data duplication is OK. Just make sure you can reconcile the duplicates.
- There is no "natural" data model
- Relational, KVS, hierarchic, network, document, graph… they're all just ways to group attributes into entities.
- Each one is optimized for certain kinds of transaction and query patterns
- Too much normalization impedes future adaptability

IDENTIFIERS AND OWNERSHIP

STUDY IN IDENTIFIERS

```
{
  “item”: “123123123”,
  “party”: “99349394”,
  “scheduleType”: “1”
}
```



Suppose we need details about “123123123”
How do we know what system to call?

“NAKED” IDENTIFIERS HAVE IMPLICIT CONTEXT

Must have code that knows how to get details:

- Host
- Port
- Protocol
- Type of query
- Format of results

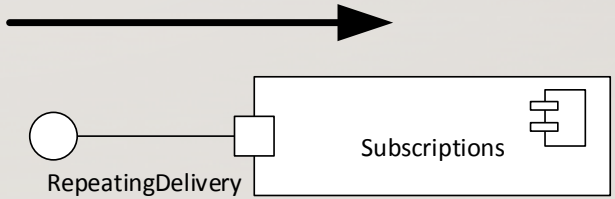
Also assumes just one authority for details

USE EXPLICIT CONTEXT

- Make context explicit to allow multiple authorities: URL or URN
- Enable many sources
- Standardize the media type or representation
- Use logical names, not application or service names
- Proxies and rewrite rules can keep the URLs functioning

BETTER

```
{
  “item”: “https://example.com/skus/123123123”,
  “party”: “https://example.com/people/
99349394”,
  “schedule”: “schedule:weekly”
}
```



REQUIRED FIELDS

REQUIRED FOR WHAT?

- Recall: data model determines what to exclude
- Changing requirements harder to support with more restrictions on data

EXAMPLE: PROX CARD TABLE FOR BIKE SHARING

Field	Type	Constraint
User_id	String	Not null
Card_type	String	Not null, enumerated
Status	String	Not null, enumerated
Name	String	Not null

Activity: For each "not null," think of a valid use (or future change) that it disallows. Thoughts in chat, please.

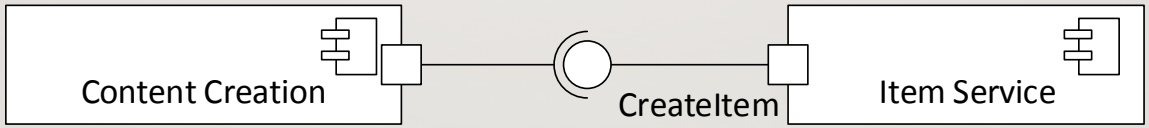
"IS VALID" METHODS

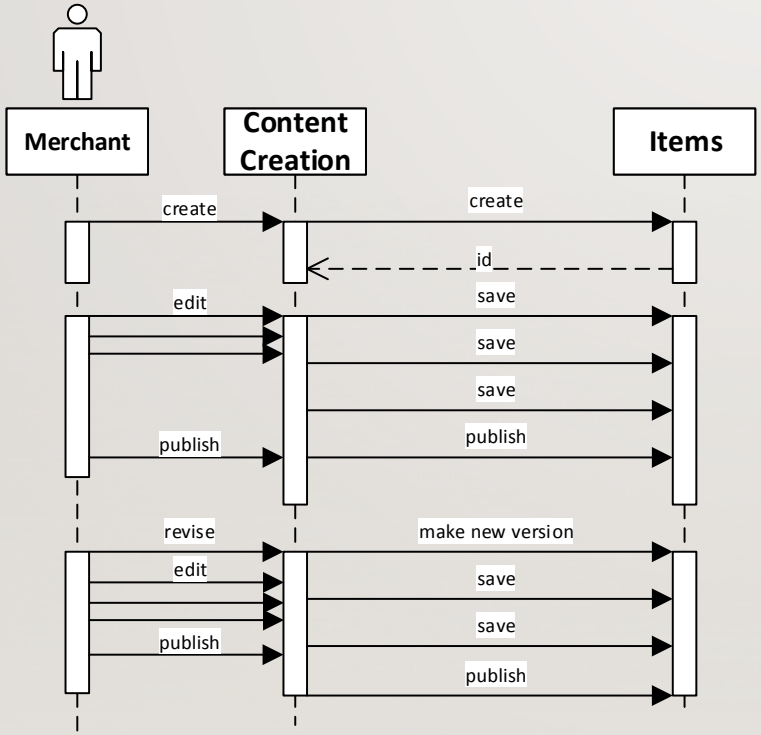
- Again: "is valid" for what operation?
- "Valid" implies policy
- Recall: Contextualize downstream

EXAMPLE: INSIDE A COMMERCE SYSTEM

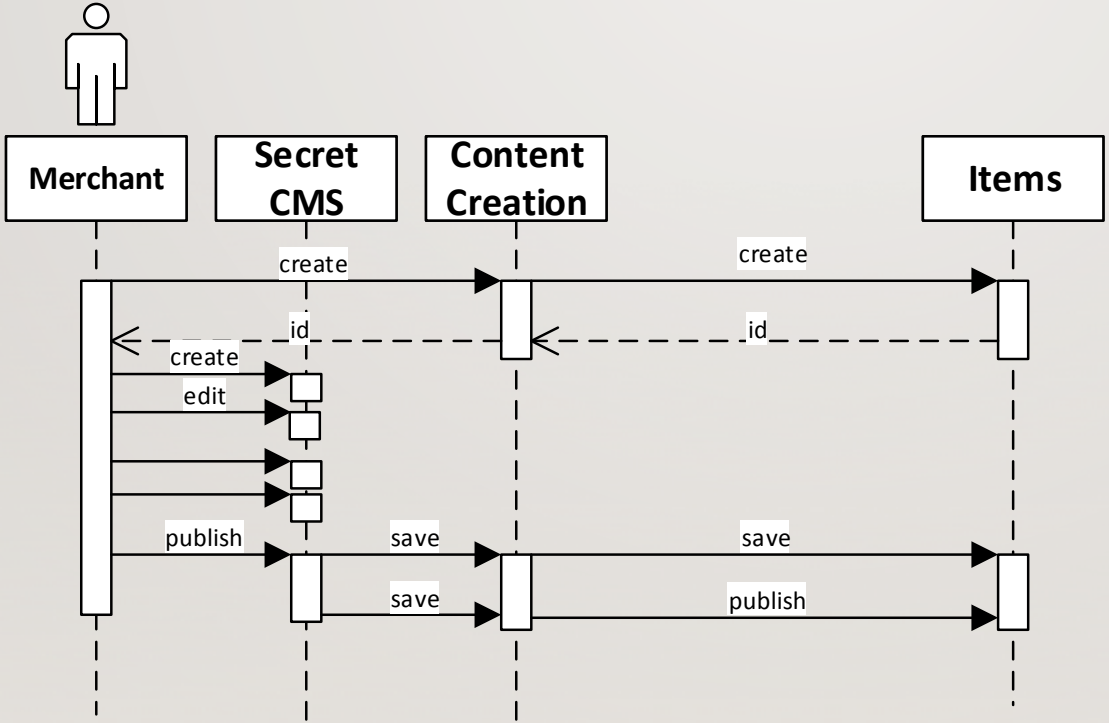
```
public interface Item {
  String getName(int version);
  void setName(int version, String name);

  String getDescription(int version);
  void setShortDescription(int version, String shortDescription);

  boolean isValid(int version);

  void publishItem(int version);
  int getLatestVersion();
  int[] getAllVersions();
  …
}
```

EXAMPLE: INSIDE A COMMERCE SYSTEM





EXAMPLE: EXPECTED INTERACTION



WHAT REALLY HAPPENED

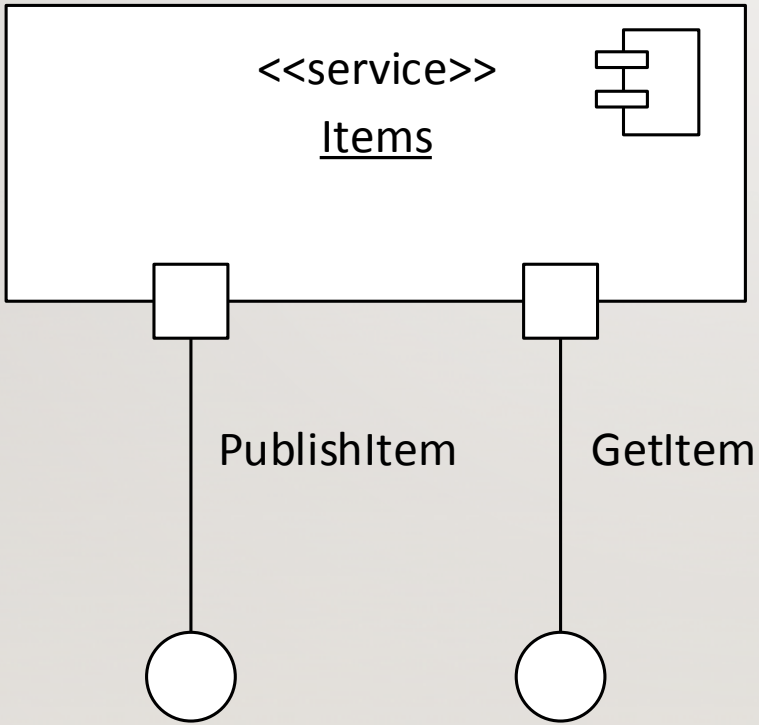
INTERFACE SEGREGATION

```
public interface Item {
  String getName(int version);
  void setName(int version, String name);

  String getDescription(int version);
  void setShortDescription(int version, String shortDescription);

  boolean isValid(int version);

  void publishItem(int version);
  int getLatestVersion();
  int[] getAllVersions();
  …
}
```

INTERFACE SEGREGATION – IN CODE

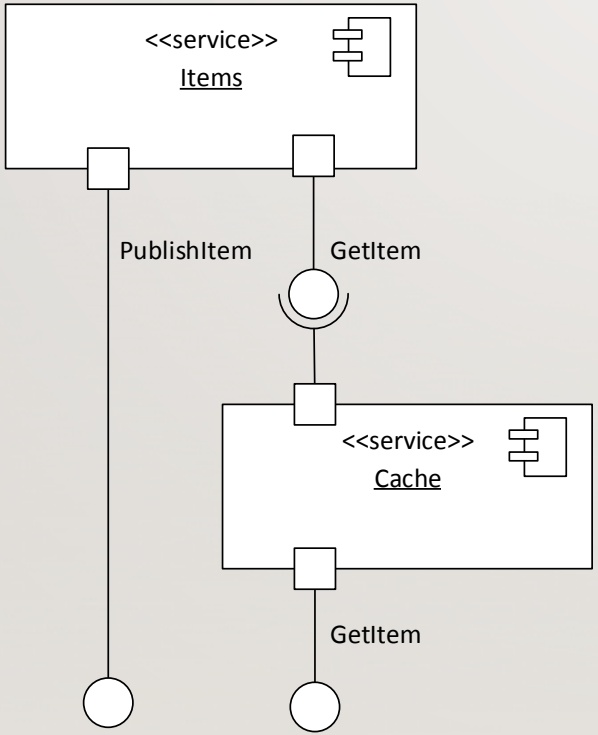
```
public interface Versions<T> {
  <T> getLatestVersion();
  List<T> getAllVersions();
  void publish(T details);
  ...
}
```

```
public interface Item {
  String getName();
  String getShortDescription();
  ...
}
```



INTERFACE SEGREGATION – AT SERVICE LEVEL

- Caller uses the interface it needs
- Caller isn't aware both are from same component



INTERFACE SEGREGATION – AT SERVICE LEVEL

- Caller uses the interface it needs
- Caller isn't aware both are from same component
- Allows intermediation or substitution

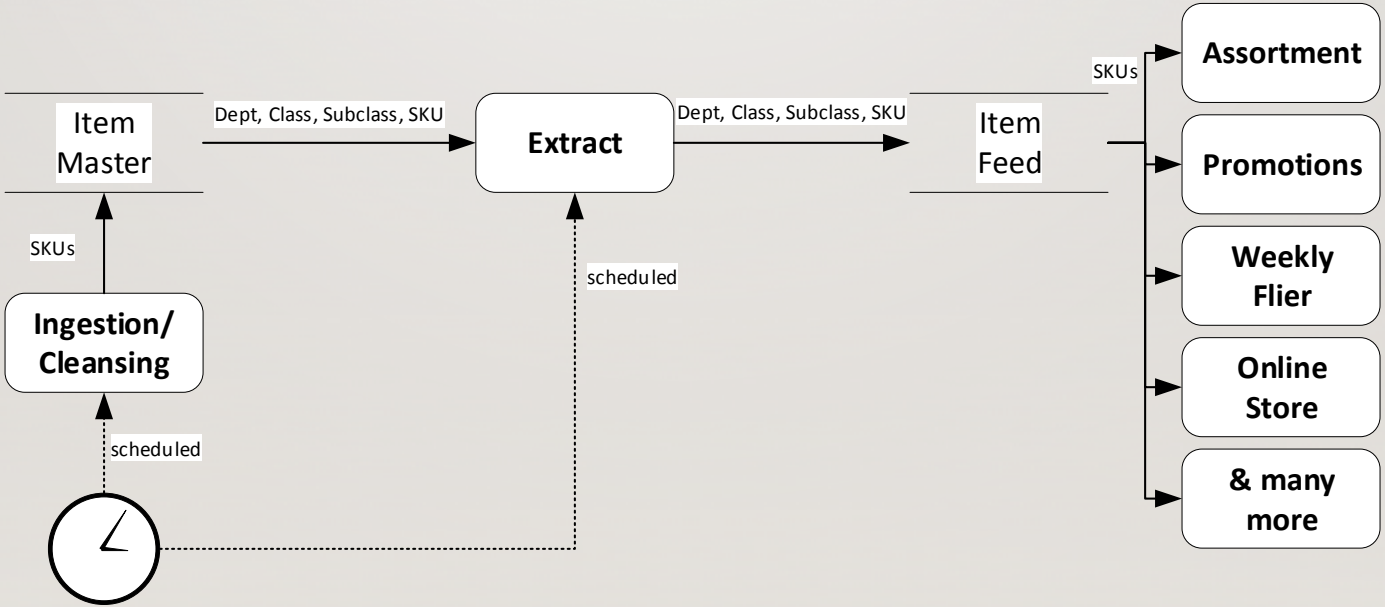
SPLITTING NOUNS

DON'T GET FOOLED AGAIN

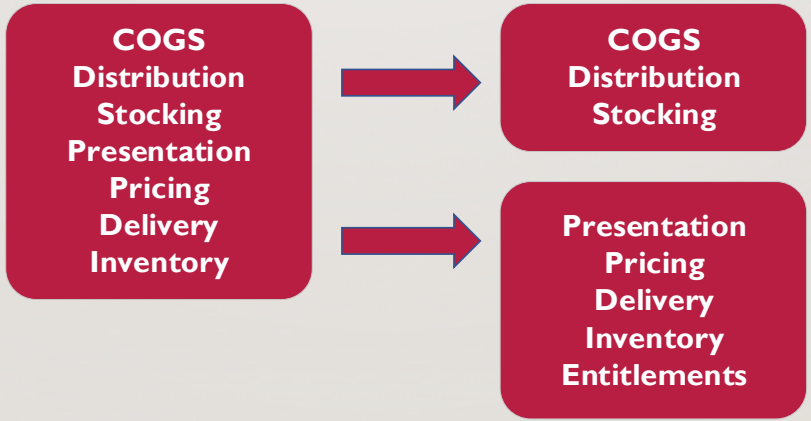
- Recall the SKU
- People thought of "SKU" as a real thing, forgot that it's an abstraction
- Many attributes needed for diverse purposes.

COGS
Distribution
Stocking
Presentation
Pricing
Delivery
Inventory

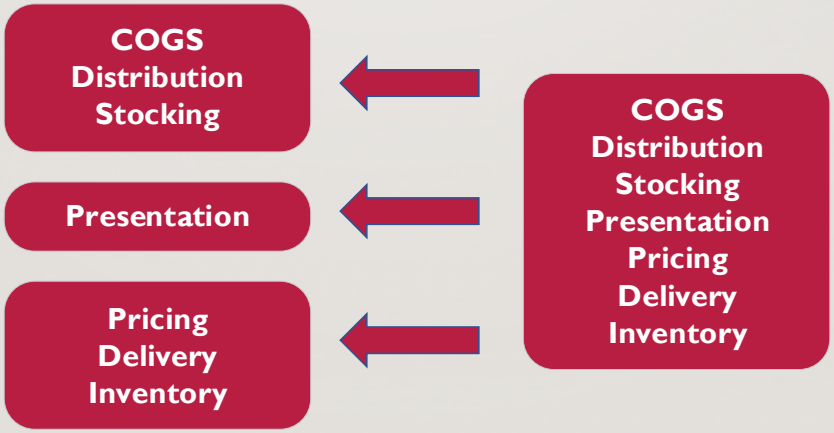
“ITEM MASTER”


Item Master
Dept, Class, Subclass, SKU
Extract
Dept, Class, Subclass, SKU
Item Feed
SKUs
Assortment
Promotions
Weekly Flier
Online Store
& many more
SKUs
Ingestion/ Cleansing
scheduled
scheduled

FOR SELLING DIGITAL GOODS



FOR MARKETPLACE



BETTER

- Use SKU *strictly* as an identifier (preferably with a URL)
- Offer interfaces to exchange a SKU for display attributes, shipping attributes, etc.
- Allow pre-packaged aggregates: sku-delivery-shipping, sku-pricing, etc.
- Decouple systems from the parts of SKU they don't need.
  - E.g., warehouse doesn't care about 27 different images of the product.

CHALLENGE FOR MICROSERVICES

- Instinct: Create "Noun" based services
- Leads to the "Entity Service" antipattern

http://www.michaelnygard.com/blog/2017/12/the-entity-service-antipattern/
http://www.michaelnygard.com/blog/2018/01/services-by-lifecycle/

ACTIVITY:
MODEL THE STUFF SUBSCRIBERS CAN GET

- Might be something delivered to their home or business
- Might be a service that someone performs for them (e.g., dog walking)
- Requires recurring payment
- Will be delivered periodically, maybe on different schedule than payment
- Should be attractive and help convert browsers to subscribers

ACTIVITY: MODEL THE STUFF SUBSCRIBERS CAN GET

1. Start listing all the attributes you can think of.
2. Now think about common use cases:
   What does a subscriber need to see when browsing?
   What does a vendor need to set up?
   What about a CSR handling an upset customer call?
3. Group your attributes by use case.
4. Now, finally, think of good names for those groups.

API DESIGN

OBJECTIVES

- Make it easy for consumers to do the right thing
- Make it possible for the provider to keep evolving

HYRUM'S LAW

With a sufficient number of users of an API,
it does not matter what you promise in the contract:
all observable behaviors of your system
will be depended on by somebody.

http://
www.hyrumslaw.com/

COROLLARY: THE LAW OF IMPLICIT INTERFACES

Given enough use, there is no such thing as a private implementation.

That is, if an interface has enough consumers, they will collectively depend on every aspect of the implementation, intentionally or not.

http://www.hyrumslaw.com/

FIRST PRINCIPLES

- Design API from the perspective of the caller
- Offer what the caller wants to achieve
- Avoid:
  - Internal state names
  - Coded fields
  - Composite fields
  - Conversations
- Be suspicious of Booleans in arguments

BREAKING APIs

BREAKING APIS

API BREAKS WHEN

- Rejects a request it would have accepted before.
- Returns less than it would have before

BREAKING APIS

API BREAKS WHEN

- Rejects a request it would have accepted before.
- Returns less than it would have before

NOT BROKEN

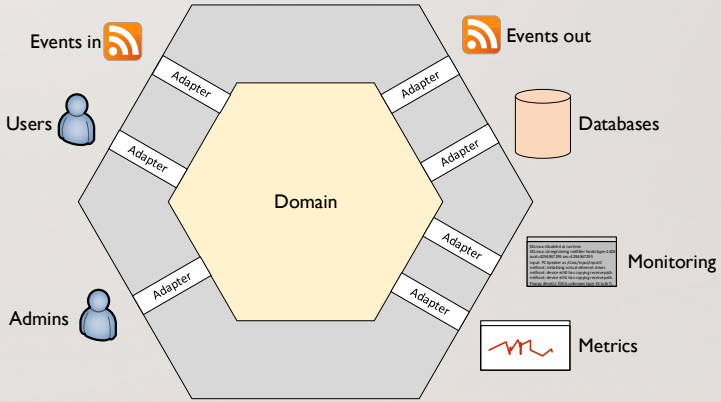
- Accepts more data in a request
- Accepts additional kinds of requests
- Rejects old requests on a new interface
- Returns more than it returned before

AVOID BREAKAGE

- Prefer adding a new interface
- Prefer adding new request types that are more restrictive
- This is easy **unless** you're use annotation based mapping on your domain classes.

REMEMBER THE HEXAGONS

- The API is **not** sticking a wire protocol on your domain entities.
- Each is an adapter from "API land" to "domain land"
- Avoid breakage by adding new adapters or changing the "outside" end of an adapter.



OPEN VS CLOSED WORLD

OPEN VS CLOSED WORLD

CLOSED WORLD

- We know all the members of a set
- We can fully partition the space of values

OPEN WORLD

- We know some of the members of a set
- We may discover that some members are actually the same thing
- We may discover that our partitioning is incomplete
- We may discover some members occupy more than one partition

RELATE BACK TO IDENTIFIERS

CLOSED WORLD IDENTIFIERS

123123123

ab2798

STATE_UNDOCKED

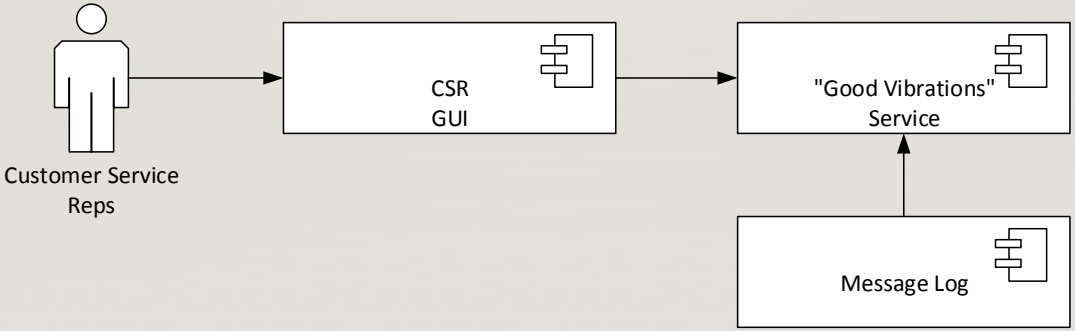
OPEN WORLD IDENTIFIERS

doi:10.1007/s10111-011-0211-6

https://example.com/people/laura

ACTIVITY: DESIGN API

- Customer service GUI needs to see all subscribers, subscriptions, vendors, payments, emails, email attempts, previous calls, … pretty much everything.
- We could couple the CSR app to every other system, but that seems brittle.
- Instead, we're going to harness the power of events.



EVENTS ARE NORMALIZED

Vendor	Customer	Event type	Event detail	Timestamp

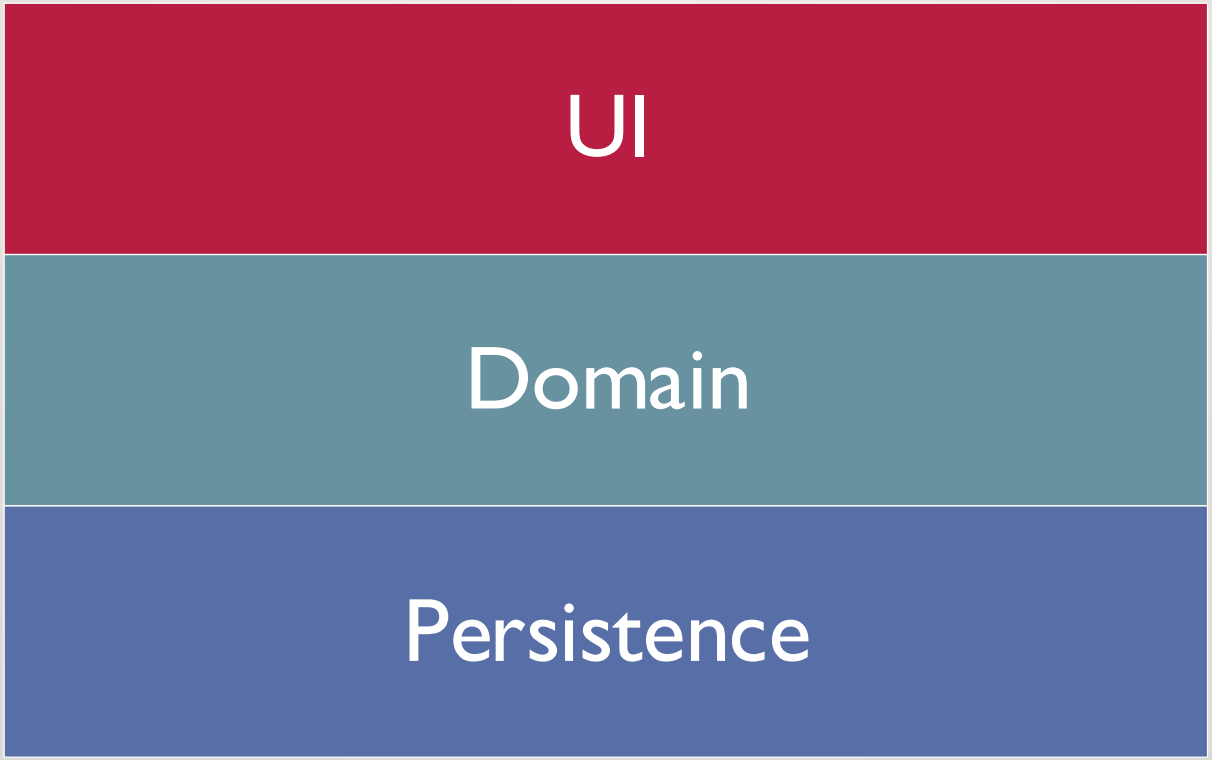
Can view record of events for a customer when a customer calls.
Can view record of events for a vendor when a vendor calls.
Can view events from systems that just got deployed today. No changes necessary to the CSR GUI.

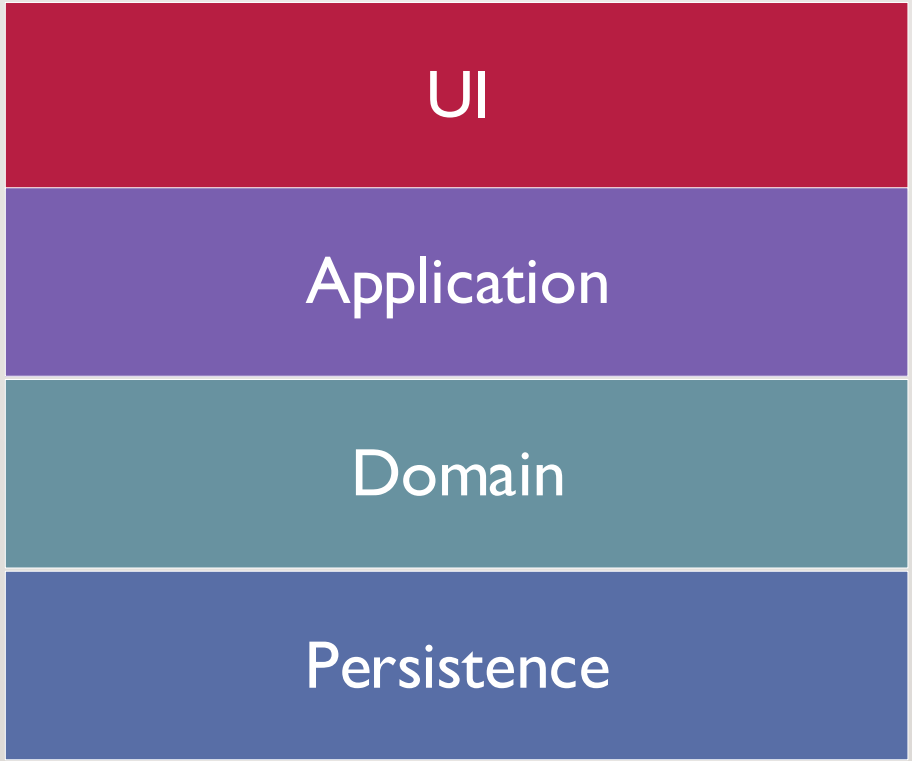
Your task:

1. Define an HTTP interface to the "Good Vibrations" service for the CSR GUI to use.
2. Define queries it can accept and the response formats it will generate.
3. Think about what needs the "open world" treatment and what can use "closed world".
4. Design from the caller's perspective. What does the CSR GUI need?

APPLICATION ARCHITECTURE

LAYERS, THE GOOD AND THE BAD


UI
Domain
Persistence


UI
Application
Domain
Persistence

Layers

Context

A large system that requires decomposition.

Mix of high-level and low-level concerns

Several operations are at the same level of abstraction

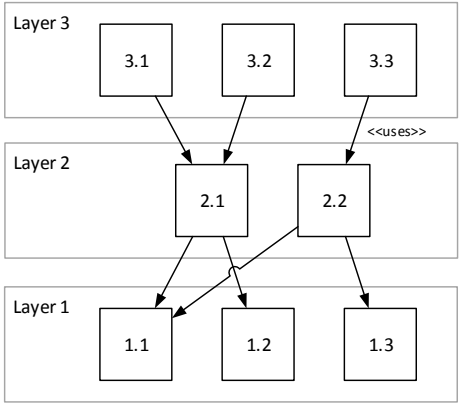
Solution

Therefore, structure your system into an appropriate number of layers and place them “on top” of each other.— Buschmann, et. al., Pattern Oriented Software Architecture, Vol 1.

Each layer should have a narrow, well-specified interface. It should be possible to substitute a different implementation of the layer without change to higher layers in the stack.

Forces

- Late changes should have limited ripple effect
- Interfaces should be stable, may be standardized
- Parts should be interchangeable
- Other systems may reuse lower layers
- Responsibilities should be grouped for comprehension and maintainability
- Further decomposition is needed for team structure and design.



Example: TCP stack Failed example: Ruby on Rails

Question: What makes a better or worse implementation of layers?

Layers

Context

A large system that requires decomposition.

Mix of high-level and low-level concerns

Several operations are at the same level of abstraction

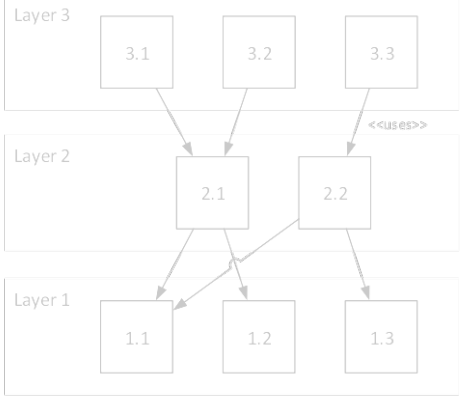
Solution

Therefore, structure your system into an appropriate number of layers and place them "on top" of each other.— Buschmann, et. al., Pattern Oriented Software Architecture, Vol 1.

Each layer should have a narrow, well-specified interface. It should be possible to substitute a different implementation of the layer without change to higher layers in the stack.

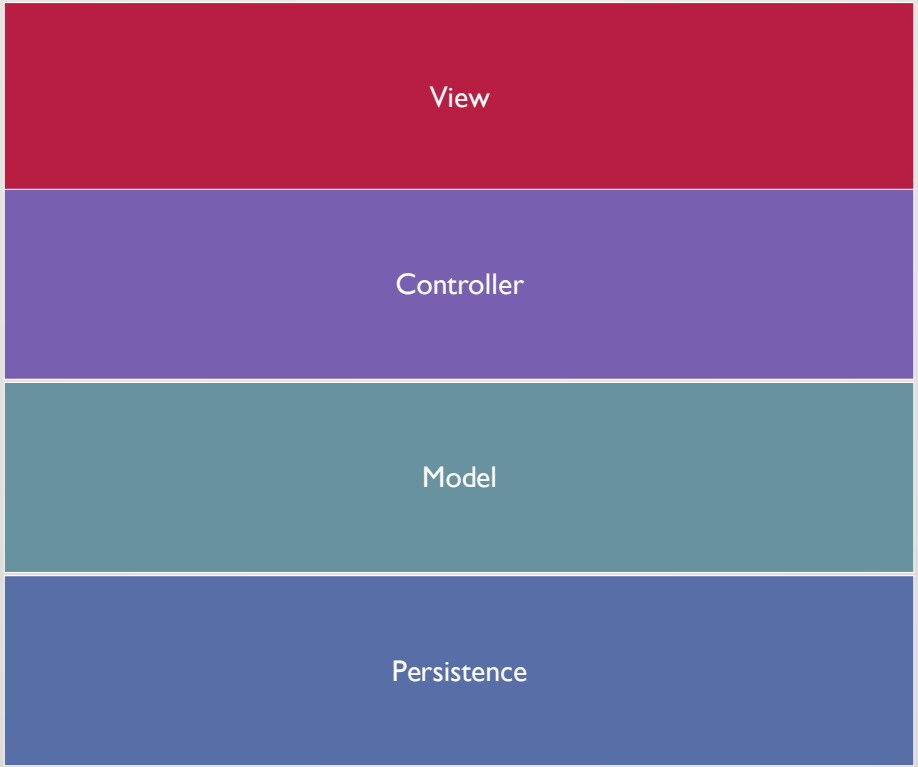
Forces

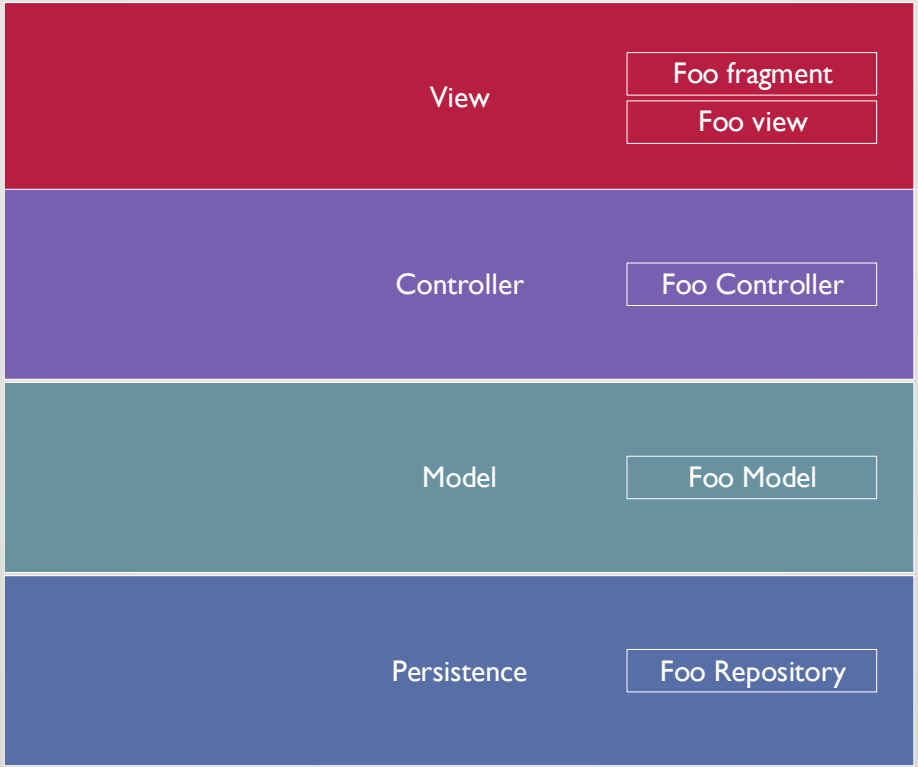
- Late changes should have limited ripple effect
- Interfaces should be stable, may be standardized
- Parts should be interchangeable
- Other systems may reuse lower layers
- Responsibilities should be grouped for comprehension and maintainability
- Further decomposition is needed for team structure and design.



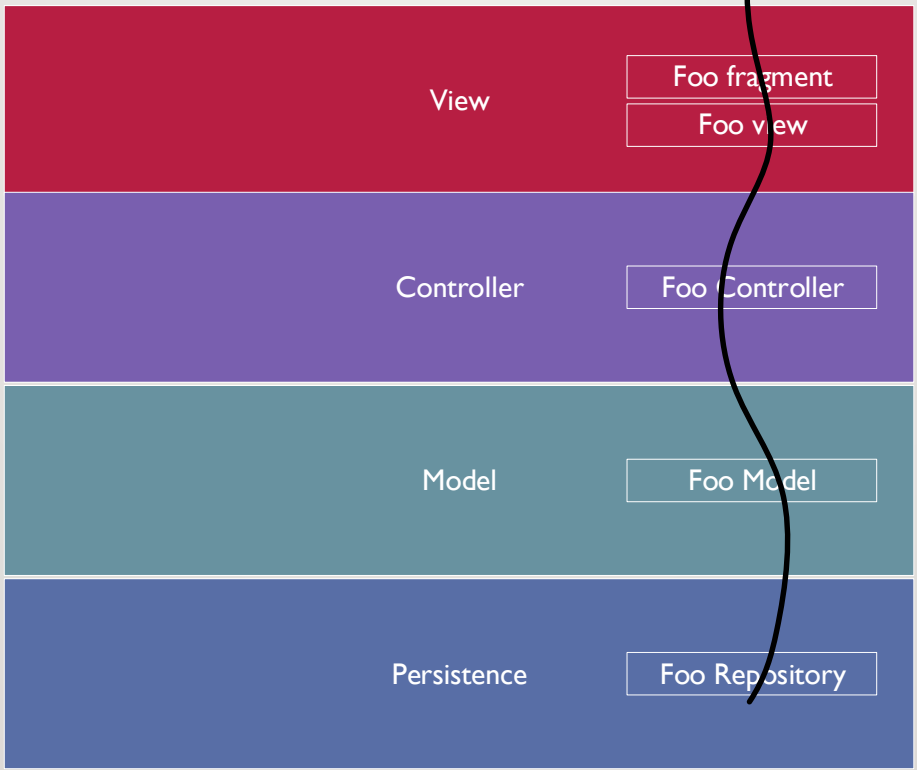
Example: TCP stack Failed example: Ruby on Rails

Question: What makes a better or worse implementation of layers?

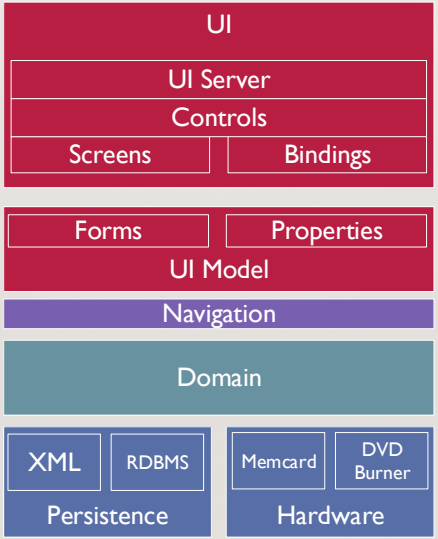

View
Controller
Model
Persistence


View
Foo fragment
Foo view
Controller
Foo Controller
Model
Foo Model
Persistence
Foo Repository

This is not how layers are meant to work.



EXAMPLE: CREATION CENTER



PORTS & ADAPTERS

DATA OUTLIVES APPLICATIONS

Example: Mailing list company, founded in 1972

- ISAM
- VSAM
- Network
- Hierarchic
- Relational
- Graph
- KVS
- Document

APPLICATIONS OUTLIVE INTEGRATIONS

Applications get replaced, but the boundaries remain. Integration tech changes constantly.

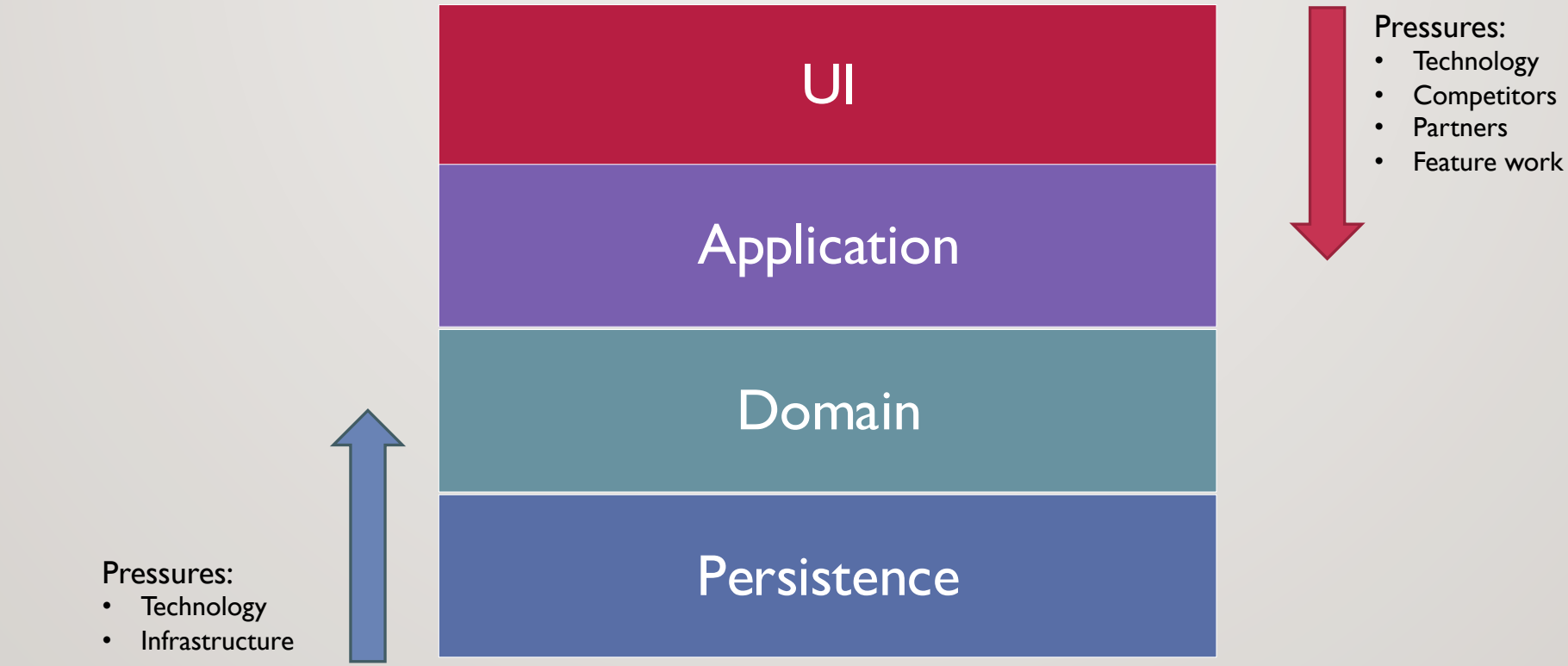

THE RIVER
THAMES
Part of Southwark


CLERKENWELL
SOAS
iversity
London
Charles Dickens Museum
Old St
A1209
A201
Great Ormond
Street Hospital
fabric
Barbican Centre
Dennis Severs' House
Brick Ln
eum
Guildhall
London
School of
Economics
and Political...
St. Paul's Cathedral
WHITECH
CITY OF
LONDON
COVENT GARDEN
Fenchurch St
Grange Tower
Bridge Hotel
Sophisticated lodging...
ry
ondon
A3211
River Thames
BFI IMAX
A3200
The Shard
Tower Bridge
a-Cola London Eye

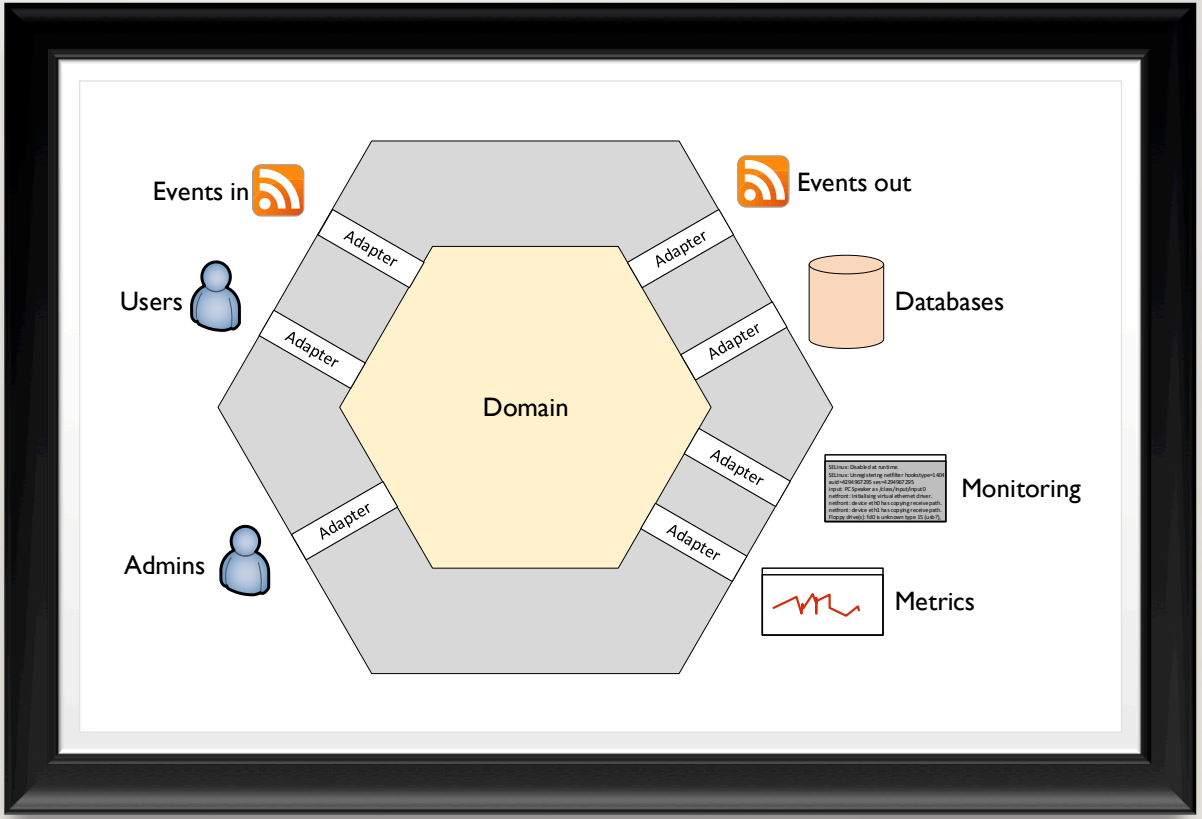
APPLICATIONS OUTLIVE INTEGRATIONS

Applications get replaced, but the boundaries remain. Integration tech changes constantly.

- CICS
- FTP
- RPC
- Sockets
- RPC
- CD-ROM
- XML-HTTP
- RPC
- ESB


UI
Application
Domain
Persistence
Pressures:
• Technology
• Competitors
• Partners
• Feature work
Pressures:
• Technology
• Infrastructure

HEXAGONAL ARCHITECTURE



ADAPTERS ARE SMALL STACKS OF LAYERS

AT THE EXTERIOR

- Technology concepts in class and function names
- Prevalent use of libraries for integration.

NEAR THE DOMAIN

- Domain concepts in class and function names
- Listener/observer/Rx to generate outbound events on the driven interfaces

ARCHITECTURAL PATTERNS

Big Ball of Mud

Context

- Big, ugly systems emerge from throwaway code
- Well-defined architectures subject to structural erosion

Discussion

Shantytowns are squalid, sprawling slums. Everyone seems to agree they are a bad idea, but forces conspire to promote their emergence anyway. What is it that they are doing right?

Shantytowns are usually built from common, inexpensive materials and simple tools. Shantytowns can be built using relatively unskilled labor. Even though the labor force is "unskilled" in the customary sense, the construction and maintenance of this sort of housing can be quite labor intensive. There is little specialization. Each housing unit is constructed and maintained primarily by its inhabitants, and each inhabitant must be a jack of all the necessary trades.

— Brian Foote and Joseph Yoder, Pattern Languages of Program Design 4

Forces

- Time – insufficient
- Cost - pressure to minimize
- Experience & Skill - may be insufficient
- Visibility - problems can grow unseen
- Accidental complexity
- Frequent change without refactoring

You need to deliver quality software on time, and under budget.

Therefore, focus first on features and functionality, then focus on architecture and performance.

See also: "Worse is Better", Richard Gabriel
Aliases: Shantytown, Spaghetti Code

Discussion prompt: a question or idea to consider.

Event Stream

Context

Scalable distributed systems with many writers

Solution

Components emit events as their primary output.

Events carry data with them.

An event may be seen by many consumers, including monitoring and observation tools.

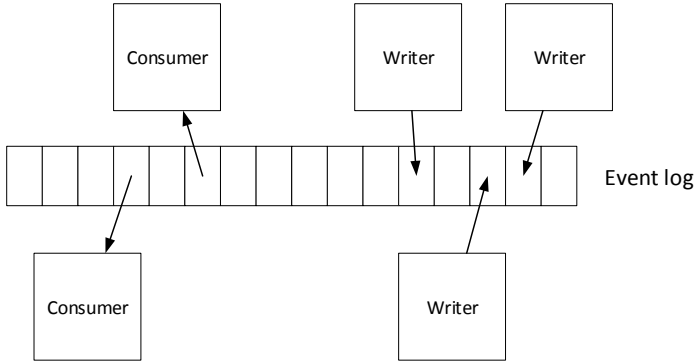
Event writers are unaware of consumers and must not require a reply.

If a normalized view of “current state” is needed, a consumer can flatten (project) some aspect of the event stream into a database.

https://msdn.microsoft.com/en-us/library/dn568103.aspx

Forces

- Queries and updates usually operate on the same entities in the same datastore.
- Relational DBMSes exhibit locking and variable performance in these cases.
- Read scaling and write scaling are rivalrous
- Access control can be difficult if frameworks expose objects with both read and write capabilities



Commonly used together with Command-Query Responsibility Segregation

Command-Query Responsibility Segregation

Context

Scalable distributed systems with many writers, particularly with a relational database involved.

Solution

Segregate operations that read data from operations that update data, by using separate interfaces.

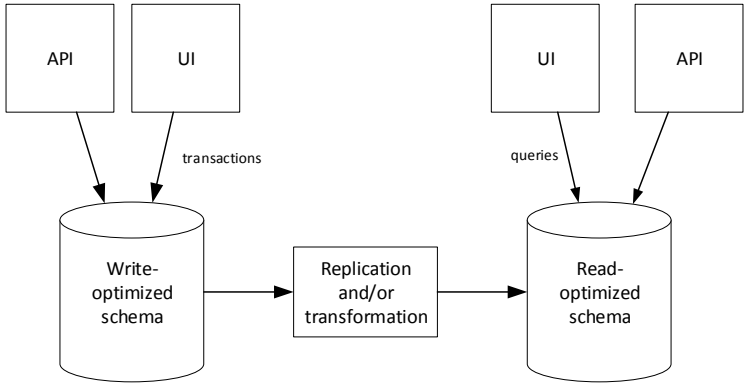
Reads and writes may operate on databases with different schemas, in which case some ETL process is needed.

When used with Event Log, the *log itself* is the write-optimized schema.

tps://msdn.microsoft.com/en-us/library/dn568103.aspx

Forces

- Queries and updates usually operate on the same entities in the same datastore.
- A good relational schema for write traffic is often slow for reads.
- Many applications have more reads than writes.
- Some applications can tolerate latency from write to read.
- Relational DBMSes exhibit locking and variable performance in these cases.
- Read scaling and write scaling are rivalrous



Commonly used together with Event Log

Dispatcher/Worker

Context

Partitioning work into semantically identical subtasks.

Discussion

The dispatcher may have the same API as the workers, to allow substitution or recursive task breakdown.

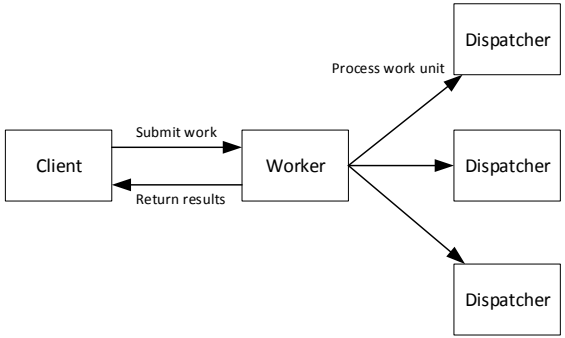
If running in many processes, fault tolerance is a concern. Dispatcher may need to rerun failed or delayed work units.

Some tasks parallelize easily, others require additional processing to combine the results. (E.g., map/reduce)

Workers may be permanent, ephemeral, or drawn from a pool.

Forces

- Clients should not be aware that the whole body of work was subdivided.
- Sub-tasks may need to be coordinated or sequenced, depending on the algorithm.
- The means of partitioning work and the task granularity should not affect the client or the processors.



Examples: Fork-join (Java), Map/reduce, GPGPU & CUDA
Aliases: Master/slave, Manager/worker

Model-View-Controller

Context

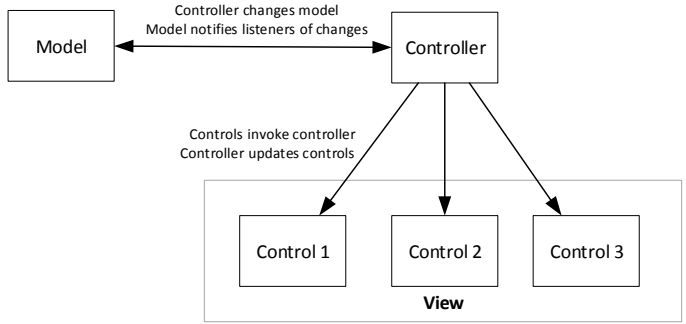
Interactive applications with a flexible human-computer interface.

Solution

- The model contains the logic about the application. It exposes attributes or properties that may be interdependent.
- The view consists of multiple user interface controls. Controls are isolated from each other and contain no logic. The view is not aware of the model.
- The controller bridges between the view and the model. It receives user interface events and pushes changes into the model. It receives model updates and pushes them into the controls.
- The view should express user-interface concepts, not domain concepts.
- The model should express domain concepts, not user-interface concepts.

Forces

- The same information appears in multiple controls.
- Controls should reflect changes immediately.
- Changes to the structure of the interface should be easy and may event be done dynamically.
- Different 'look-and-feel' standards should not affect the logic of the application.



Example: "Fahrenheit to Celcius" app.
User input to either the C or F fields should cause the other to change too.

Pipes and Filters

Context

Processing data streams

Solution

Divide task into separate processing stages.

Connect them sequentially: the output of one stage is the input to the next.

A *filter* consumes and delivers data incrementally.

A *data source* supplies the initial input, while a *data sink* consumes the final output.

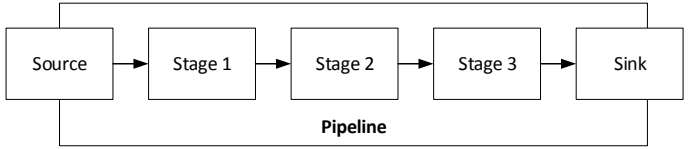
The data source, filters, and data sink are connected via pipes.

The assembly is a processing pipeline.

Some external entity constructs the pipeline.

Forces

- Data stream processing which naturally subdivides into stages
- May want to recombine stages for different pipelines
- Non-adjacent stages don't share information
- May desire different stages to be on different CPUs or hosts



Example: Unix pipes, Apache Spark

Question: When would you create a "push" pipeline versus a "pull" pipeline?

Broker

Context

Distributed, heterogeneous systems with independent components.

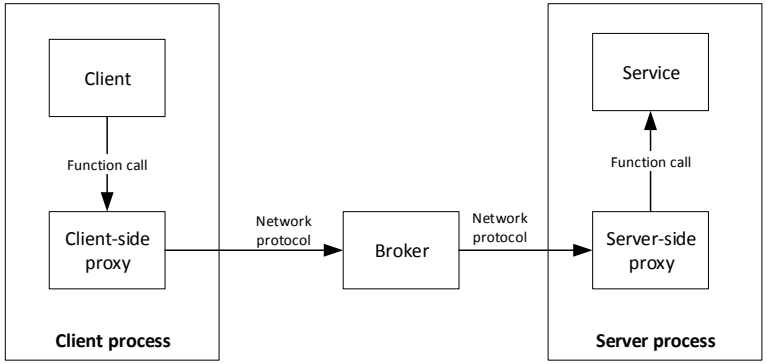
Discussion

- Introduce a broker between clients and servers.
- Services register themselves with the broker.
- Proxies and bridges may help cross network types.
- Client proxy presents a function-call interface over remote calls

See also: Peter Deutsch's "Fallacies of Distributed Computing"

Forces

- Need to communicate across process boundaries.
- Possibly heterogeneous network or transport.
- Desire for transparent remote access.
- Run-time replacement, exchange, or substitution of components.
- Architecture should hide implementation details from users.



Discussion question: When is this a pattern and when is it an anti-pattern?

Blackboard

Context

An immature domain in which no closed approach to a solution is known or feasible.

Discussion

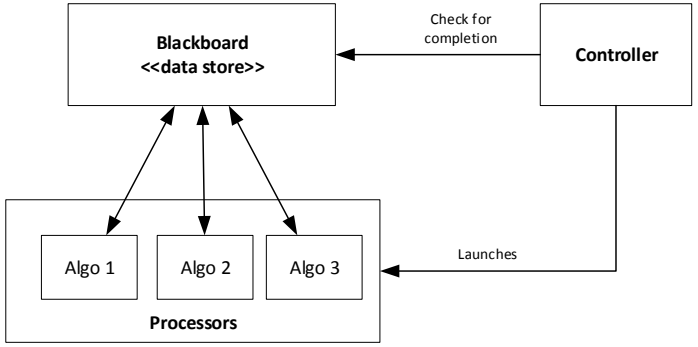
Independent programs cooperate indirectly by taking data from or placing data into a shared repository, called the *blackboard*.

A program can start running once the input it needs appears on the blackboard. When it completes, it places its result onto the blackboard.

The processors work on data at different (usually increasing) levels of abstraction.

Forces

- Complete search of solution space is not feasible
- May need to experiment with different algorithms
- Different algorithms partially solve problems.
- Input, intermediate, and final results have different representations.
- An algorithm may use (but not consume) results of other algorithms.
- There exists potential for parallelism.



Example: Speech recognition. Processors work at levels: samples, phonemes, morphemes, sentences.

Discussion question: In what other contexts might you apply the Blackboard pattern?

Microkernel

Context

Developing several applications that use similar APIs on top of the same core functionality.

Solution

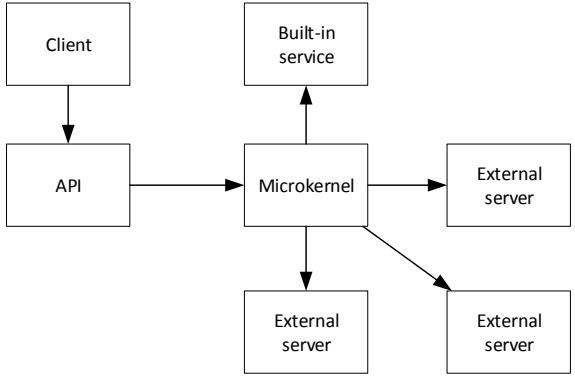
- Encapsulate fundamental platform services in a microkernel:
- Mediates communication between services (often via messages)
- Manages systemwide resources
- Offers APIs to access its functionality
- Servers provide core functions, and run outside the process space of the microkernel

Example: Mach microkernel in early versions of OS X.

Counter-example: Linux kernel

Forces

- Application must cope with continuous hardware and software evolution.
- Applications must support different but similar application platforms.
- Platform should be portable, extensible, and adaptable to allow integration of new technology.
- Application must survive failure of components.



Question: What does this look like at data center scale? What plays the role of microkernel?

FAULTS, ERRORS, AND FAILURES

DEFINITIONS

Fault	Incorrect internal state. Introduced by a defect.
Error	Observably incorrect operation.
Failure	Loss of availability. System cannot perform its mission.

THE LAW OF LARGE SYSTEMS

- Large systems operate in a state of continuous partial failure
- “Everything good” is an abnormal state
- Plan for faults
- Design error handling as a first-class concern

CIRCUIT BREAKER

```
int remainingAttempts = MAX_RETRIES;

while(--remainingAttempts >= 0) {
  try {
    doSomethingDangerous();
    return true;
  } catch(RemoteCallFailedException e) {
    log(e);
  }
}
return false;
```

CIRCUIT BREAKER

```
int remainingAttempts = MAX_RETRIES;

while(--remainingAttempts >= 0) {
  try {
    doSomethingDangerous();
    return true;
  } catch(RemoteCallFailedException e) {
    log(e);
  }
}
return false;
```

Is the problem really going to be gone 10 nanoseconds later?

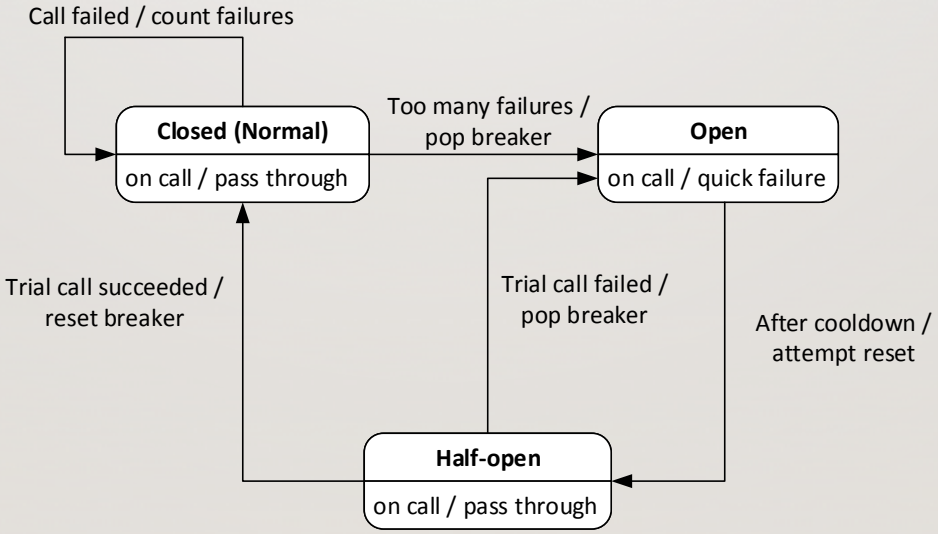
CIRCUIT BREAKER

```
int remainingAttempts = MAX_RETRIES;

while(--remainingAttempts >= 0) {
  try {
    doSomethingDangerous();
    return true;
  } catch(RemoteCallFailedException e) {
    log(e);
  }
}
return false;
```

Is the problem really going to be gone 10 nanoseconds later?

Why does every thread have to make the same discovery?

CIRCUIT BREAKER



CIRCUIT BREAKER

- Sever part of the system to save the rest

GOVERNOR

- Limit the rate of dangerous actions
- Especially if done via automation
- E.g., when autoscaling add servers quickly, shut them down slowly

DESIGN SHOCK ABSORBERS

- Fail fast
- Bulkheads
- Circuit breakers
- Load shedding
- Caches

RUN DRILLS & SIMULATIONS

- “Game day” exercises
- Practice launch
- Hurricane drills
- Zombie apocalypse

CHAOS ENGINEERING

- Techniques to surface systemic problems
- Use randomness
- Apply stresses to your system
- Create faults
- Discover the unknown-unknowns

HIGH RELIABILITY ORGANIZATIONS

- Safety is the primary objective
- Decentralized decision-making
- Culture of reliability, peer reinforcement, management emphasis
- Continuous training and simulations
- Learn from successes and near-misses
- "Blameless" post-mortems
- Avoid "Operator Error" or "Loss of Situational Awareness" as explanations.