System Design

go/drx-learns-system-design

Presenters: codywilson@, dagafono@, jefftk@, nickgonzalez@, sbelov@, tianguo@, xiaoyong@

Xiaoyong to talk about goal, format, introduce instructors, expectations of cohort meetings, 3 mins

Disclaimer

We created this course from the scratch. Therefore it is

- Experimental
- Subjective
- Incomplete
- ...

Skip this slide

Session #1

April 1, 2022

Why it is important

- **Design for change**
- Code is written once, but needs to be changed and maintained for years
- Not optimizing for maintainability results in failed projects,
  projects never started due to fear of complexity,
  unstable systems, engineering pains
- Team's velocity can vary by orders of magnitude depending on software design choices.

Speaker: Stan, 5-10 mins

A typical software engineering team can spend 10x-100x more time reading, understanding, debugging, maintaining and updating the existing code than writing the first version for a given functionality. Given such a ratio, optimizing for a highly-maintainable codebase over time is a no-brainer investment – and can distinguish a successful software project from a disastrous one. Lack of modular design, of proper component decoupling is one of the primary reasons for software systems' early deaths and painful lives of their maintainers. Team's development velocity can vary greatly – sometimes orders of magnitude – depending on software design choices.

How can teams design, write code and change it over time to keep the codebase from becoming rot? What design choices can get a software project to achieve sustainably high development velocity and keep the engineering team engaged and motivated?

This course will attempt to present and discuss some design principles and patterns that are essential for building highly-maintainable, modular systems that can evolve with the real world. C++ will be the focus language, but most of the material will apply to other object-oriented languages as well.

Focus of this course

- Design levels : **class, component, API,** business model, data model, architecture, binary
- Principles and patterns that help to
  - achieve sustainably high development **velocity**
  - prevent the codebase from degrading too quickly (**healthy code**)
  - keep the engineering team engaged and motivated (**healthy team**)
- C++, but most of the material applies to other OO languages also.

Speaker: Dmitriy, 2 mins
Class-level & component-level : group of classes
Principal & patterns essential for : velocity, healthy code, healthy team

What good looks like...

Short term (months)

- Meet changing requirements
- Easy to onboard
- Easy to work in parallel
- Easy to test
- Easy to debug
- Easy to reason about
- Consistent

Long term (quarters)

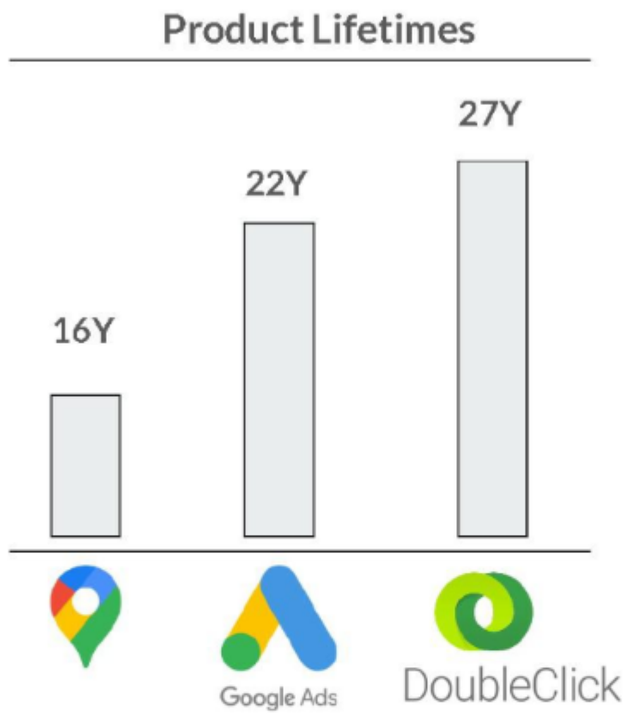
- Easily update technical infrastructure
- Easily adapt and extend for new requirements
- Easily validate correctness
- Resilient to unrelated changes

Speaker: Nick, 2-3 mins
Problems we are designing for
Definition of good and maintainable systems
What bad looks like
Principles we'd follow
Process that aid design (or tools)
Goals -> Principles -> Patterns -> tools

Things reusable from Nick's deck:
Slide 31 "reasons for bad architecture"
#45 design pattern categories
#51 design process

Reasons for bad design

- Lack of experience
- Lack of standards
- Time pressures at beginning of project
- **Underestimate product lifetime**
- Underestimate maintenance burden
- Overestimate stability of requirements
- “Fix it later” attitude



Speaker: Nick, 2-3 mins

For underestimate product lifetime, talk about underestimate the long term impact of a code change.

Principles

Simplicity
Uniformity
Flexibility
Separation of concerns
Testability

Speaker: Nick, 30 secs

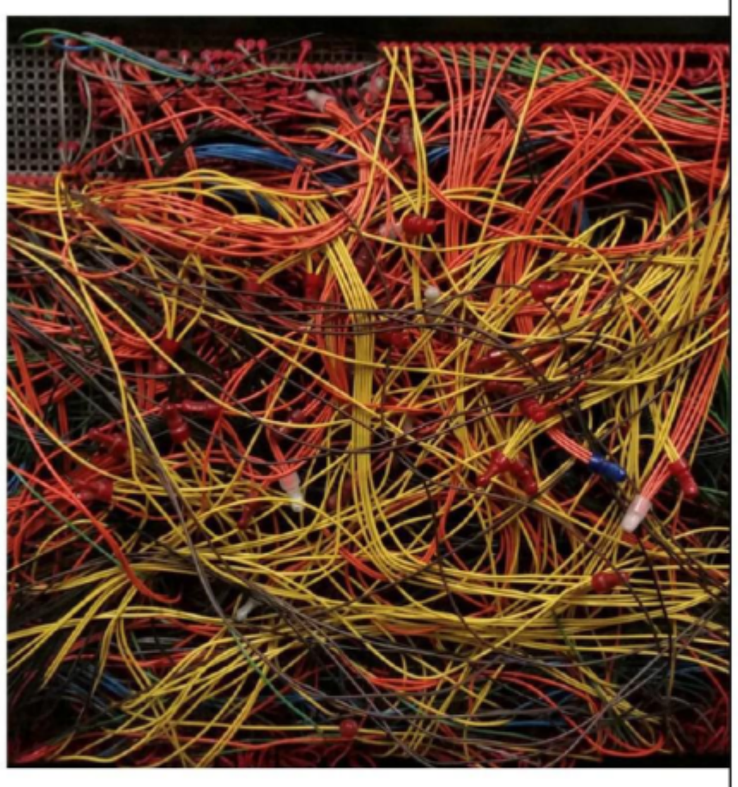
Talk about balancing act, making tradeoffs

Simplicity

sem "one" + **plek** "fold"

Limit the number of concerns per system.

- Easy to understand / reason about
- Easy to change / maintain
- Easy to debug
- Prerequisite for flexibility
- Prerequisite for reliability
- Not over-engineered
- Not the same as "no design"



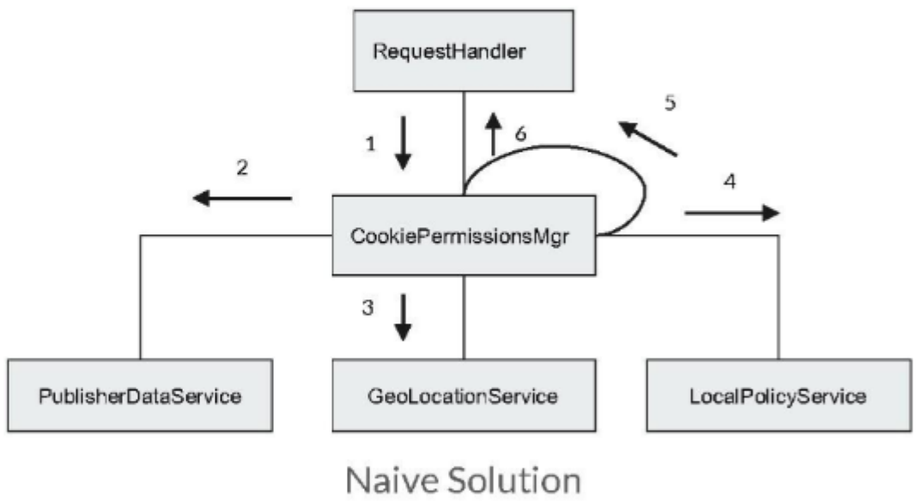
Simplicity is prerequisite for reliability - Dijsktra

Speaker: Nick, 3 mins

Simple design is not the same as no design

Simplicity: Example, Cookie Permissions

Given a web request figure out if we can write cookies.



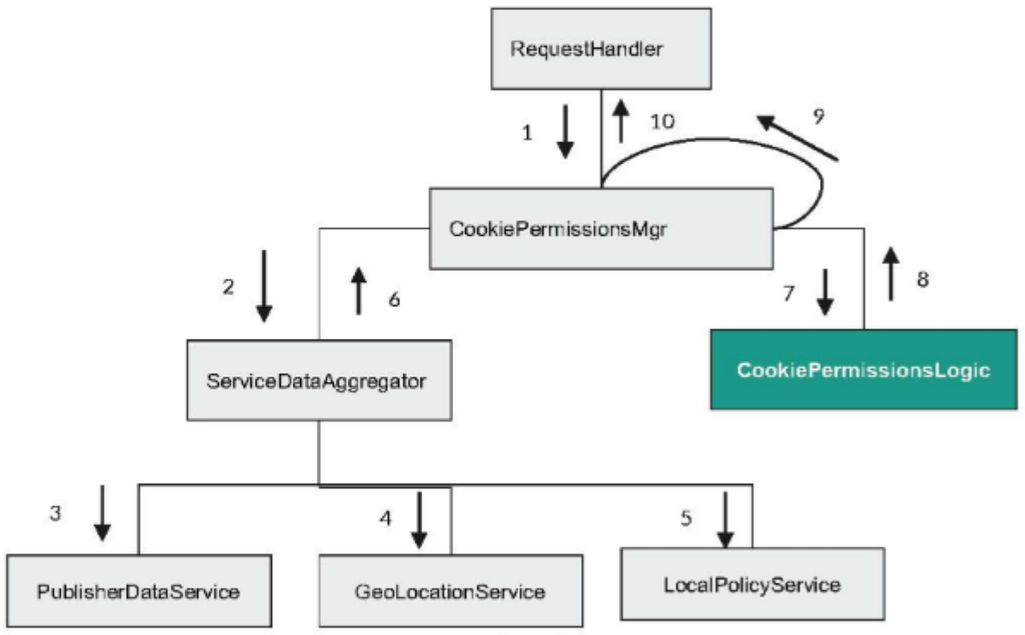
Naive Solution

CookiePermissionsMgr Concerns

- the entire http request
- the format of the response
- rpc service communication
- network failure
- service failure
- business logic for cookie permissions
- PublisherDataService
- GeoLocationService
- LocalPolicyService

Nick: 2 mins

Simplicity: Example, Cookie Permissions



Simple Solution

Concerns per system

- CookiePermissionsMgr
  - the entire http request
  - the format of the response
- ServiceDataAgrregator
  - rpc service communication
  - network failure
  - service failure
  - PublisherDataService
  - GeoLocationService
  - LocalPolicyService
- CookiePermissionsLogic
  - required inputs for business logic
  - business logic for cookie permissions

Nick: 2 mins

Simplicity: Example, Cookie Permissions

Naive solution

- CookiePermissionsMgr is a complex
- Too many concerns
- Difficult to test
- Inflexible
- Difficult to reason about
- Requires integration tests to verify business logic
- Inputs and outputs are unclear

Simple solution

- Business logic is isolated
- Service data aggregation is separate and reusable
- Easy to unit test business logic
- Easy to adapt to changes in service architecture
- Easy to adapt to changes in HTTP protocol request changes
- Easy to adapt to policy changes
- Easy to adapt to changes in rpc protocols
- More likely to produce correct results
- Easier to debug

Nick: 2 mins

Uniformity

Solve similar problems in the same way.

- Reduces the number of different solutions
- Fewer concepts & patterns to learn (simplicity)
- Fewer surprises (readability, maintainability)
- Fewer kinds of defects that can occur (reliability)
- Easier to adopt (reusability)

Speaker: Dmitriy, 2 mins Illustration : no flexibility, no simplicity either
Uniformity is about solving problems in the same way Following this principle results in : Simpler, More Predictable, No surprises Any solution has potential issues Uniformity -> Simplicity : by reducing variety of design approaches we make software simpler. Uniformity -> Maintainability : implementation is done in a predictable familiar way
Uniformity -> Reliability : Any solution has potential issues - itself or with how we use it
Uniformity -> Reusability : others following the uniformity principle is more likely to re-use your component if it follows it, because it does things in an “right” expected way

Uniformity Case Study : Function Error Handling

- Make it asynchronous: continuations, Future/promise, Task/AsyncTask, Producer
- Add arenas to the picture

Dmitriy, 2 mins
Design API for producing a value, which may fail□□The right answer is of course : “It depends”. There are valid use-cases for all of them
There are valid use-cases for all, but the last one is the default. Let’s not worry about arenas.
C++11 vs C++17 (RVO)□□StatusOr is the default□□Fu□□go/totw/76
go/totw/labs/statusor-patterns-and-antipatterns

Uniformity Case Study : RPC Error Propagation

- RPC status with canonical error space
- RPC status with custom error space
- Custom Error proto in Response
- StatusProto in Response
- A combination of the above
- No consistent propagation within binaries either
- Example 1 : Line Item PSI error propagation
- Example 2 : Rule PSI error propagation

Speaker: Dmitriy, time tbd

Uniformity via Standards

Creating and following standards is a way to achieve uniformity.

- Create
  - Outcomes, Processes, Tools, Libraries, Frameworks
  - Existence, Quality/Coverage, Up-to-dateness
- Follow
  - Scope and quality of adoption
  - Pace of adaptation to standard changes
- ... or not
  - Uniformity is not the only criteria (e.g. performance considerations)
  - Often similar problems are not similar enough (e.g. different constraints)
  - Different scopes (team, component, binary, serving stack) may have different standards

Dmitriy, 3 mins
Example : go/c-style□Coverage includes nuances/exceptions/edge conditions□SE : craft, science, art. Standards can guide, but also limit by stifling creativity and confuse□

Flexibility

- What we want from our system is always changing
  - External factors: ecosystem changes, regulations, publisher demand
  - Internal factors: new features, turndowns, optimizations
- Design to help future engineers easily make changes:
  - Loosely coupled sections (separation of concerns)
  - Elegantly handle *understood* complexity
- But not too much flexibility! Avoid building things that:
  - Solve problems we don't see clearly yet
    - Overengineering
    - "You Aint Gonna Need It" (YAGNI)
  - Are so flexible they can't be well tested
    - Be especially careful with run-time configuration

Owner: Jeff 10 mins for all flexibility

Flexibility example: Genotype

- Client-side engineers used to run experiments with, essentially:

- Not flexible enough: no concept of flags separate from experiments
- Minimal integration with Mendel: logging-only

Owner: Jeff

Flexibility example: Genotype

- Genotype introduced flags to the client

- Flags could be tested individually
- Flags could be combined
- Closer integration with Mendel
    - Define flags in code, experiment on them with Mendel

Owner: Jeff

(Over-)Flexibility example: GPT Services

- GPT was designed to have many *Services* for different things it could do
- Loading ads used the *Publisher Ads Service*
- A lot of design and code to support future engineers who would add *Services*
- But use cases for *Services* never came up
- Eventually removed
  - Except for where the publisher-facing API depends on it

Owner: Jeff
Negative example

Session #2

dory/drx-learns-system-design

April 15, 2022

“Weakest link” critical to software design

- Limits of the **human brain**
  - Can hold / operate on a limited set of concepts in short-term memory
  - The Magical Number Seven, Plus or Minus Two
- Limits of **human communications**
  - Limited bandwidth, interpretation difficulties, perception, emotions…
- How do we design for these constraints?
- >90% time/effort spent after first version of code is written
  - By many engineers (inc. original author) reading, understanding, debugging, updating
  - Subject to these human constraints

Speaker: Stan

Single responsibility principle

- **Single, focused purpose** for each module (interface, class, service)
- Solve for human design constraint
- Modules implemented independently,
  interact with each other via clear interfaces
  - Encapsulation
  - Information hiding
- Infrastructure vs business logic
- Different concepts, entities, actors of business logic
  - Map units to natural problem domain
  - Express in design, code with natural, ubiquitous language

Speaker: Stan

Separation of Concerns

- *Coupling*: independent modules/decisions/data tied together
  - Lack of encapsulation, "leaky" abstraction
  - Dependence on implementation details
- *Decoupling*: only have dependencies where necessary
  - Isolate decisions where possible
  - Modules don't depend on implementation details of each other
- Why?
  - Module Y depends on the implementation of module X
    - I want to change X to fix a "simple" bug
    - All of a sudden Y breaks

Speaker: Stan

Example: Producers and business logic

C++ Producers: framework for asynchronous programming

- All I/O calls – RPC, lookups, file access – should be non-blocking.
- Each producer a graph "node", activated upon availability of its inputs (I/O operations completing)
- Significant framework "overhead":
  - webserver::Input<Type>, webserver::Output<Type>
  - OnInputsReady()
  - State-scheduling control: ScheduleStateWithFunction, ScheduleStateMethodWithFunction, ScheduleStateWithDependencySet...

Should business logic live in producers?

Key points:
Producers could be an example - which relies on regular key-val lookup but does something more
Split infra concerns from data model concerns
Mention they are other options to implement the logic and what considerations (uniformity, inertia) that came into the decision

Example: IP denylists for RTB partners

Problem:

- Bidders want to block requests from certain IPs or CIDR ranges of IPs (11.22.33.44, 22.33.44.0/24, 33.44.55.64/28)
- Data set mapping IP/range to bidders blocking it:
  11.22.33.4 4/32 → A
  11.22.33.0/24 → B,D
  11.22.33.64/28 → C
- Given client IP, find all blocked ranges and bidders
  11.22.33.44.91→11.22.33.0/24,11.22.33.64/28→(B, D, C)

Example: IP denylists for RTB partners

Business logic

Infrastructure

- Issue an asynchronous batch KeyVal dataset lookup, schedule a completion callback
- Upon completion, execute business logic
- Pass output to the next step in the control flow

Example: IP denylists for RTB partners

Business logic contract

[redacted]

- One concern: get blocked bidders
- Simple function, inputs and outputs
- Easy to read, reason about, test, change
- Can delegate to infrastructure abstractions (raw KV lookup)

Infrastructure contract

[redacted]

- Look up values for given keys (asynchronously)
- Lower level

Example: IP denylists for RTB partners

Business logic contract

[redacted]

- One concern: get blocked bidders
- Simple function, inputs and outputs
- Easy to read, reason about, test, change
- Can delegate to infrastructure abstractions (raw KV lookup)

Infrastructure contract

[redacted]

Producer simply delegates to business logic, manages scheduling
Future result becomes producer's output

Alternative: everything in a producer

Producer's responsibilities

- Accept raw inputs from upstream (TargetingRequest, not focused)
- Business logic: find blocked CIDR ranges & blocked bidders
- Raw KV lookup
- Scheduling, providing outputs
- Not focused on a single goal, harder to test, reason about, change…

Anti-pattern: “God class”, “Orchestration module”

RTB CalloutHandler roles

- Bid request prefiltering
- QPS quota throttling
- Generating (and mutating!) bid requests
- Making callouts
- Processing results
- 2K lines
- 5K+ lines “unit” test
- 20 constructor-injected dependencies
- Single GetBids() public method effectively with 15 arguments

Q&A

dory/drx-learns-system-design

Homework for Session #2

- Reflect on a project that you are currently working on or have worked on in the past regarding:
  - Did the project design follow any of the design principles discussed here? Was it designed with potential change and long-term maintenance in mind?
  - Were there any tradeoffs made when considering different principles?
  - What was your experience with making changes to that project? Did it feel fast / slow, rewarding / painful?
- Write down a few bullet points re above and share your experience with your cohort.
- Each cohort picks one example from the cohort discussions and presents it to the group in Session #3.

Speaker: Xiaoyong

Session #3

dory/drx-learns-system-design

April 29, 2022

Cross-serving learning-focused OKR

Objective:

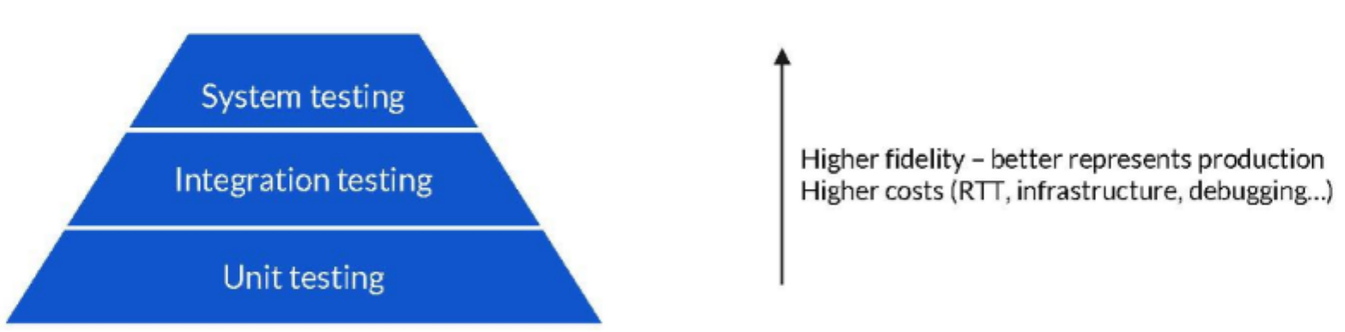
- Build a culture of learning and growth to heighten individual and team impact

Key Results:

- Complete the pilot program go/drx-learns-pilot
  - 4-5 instructor-led sessions, combined with cohort-based learning.
  - Attendance: 70+% of L3 and L4 SWEs participating in 2+ sessions.
- Create plan for the full program (based on learnings from the pilot)
  - Create and align on the set of baseline expectations for what L3/4 eng should know regarding system design and project impact
  - Create materials for program
  - Satisfaction of program as assessed via regular surveys
  - A supporting structure to encourage this kind of program, e.g. recognizing/rewarding teaching and volunteering

Testability

Levels of testing



Speaker: Stan

Designing for testability: unit tests

- Test only public API, interface contract
  - Avoid testing implementation details – anything that isn't important for the contract
- Rely on dependency injection
  - Inject dependencies as mocks or test implementations
  - Allows to focus on testing a single unit
- Testability + separation of concerns = synergy
  - Virtuous loop: SoC benefits testability; designing for testability benefits SoC.

Speaker: Stan

Designing for testability: unit tests

- Testing side effects (I/O, calls to mutator methods): more complex
  - Avoid side effects in design unless absolutely necessary
  - Define only essential side effects – explicitly as part of the contract
  - Test those – using mock or test dependencies
- TDD: write tests before implementation – against the interface contract
  - Then implement in the simplest possible way to make tests pass

Speaker: Stan

More on test-driven development

- Leads to simplicity: only minimal implementation that satisfies unit tests
- Benefits API design
  - Focus on the interface first, write and feel how the contract will be used...
  - Before implementing: avoids "tail wagging the dog" (implementation dictating API design)
- Incentivizes designing and building for change
  - Write minimal set of test cases that matter → minimal implementation that matters → faster test and implementation changes in future
- Iterative *flow*: add a test case that fails, then make it pass via implementation
- Ensures close to 100% test coverage
  - No code written that's not needed to satisfy test cases
- Results in higher quality, lower defect ratio → higher velocity of delivery, lower debugging / troubleshooting costs

Speaker: Stan

Example

IP denylist lookup unit test

Speaker: Stan

Cohort Presentations

Cohort 5: Passthrough BOW on DFP ⇨ DFP Inreds

Before

After

go/dfp-inred-passthrough

[redacted]

Use cases, Apple News. DFP pubs want to serve ads directly into the Apple News iOS app. Publisher Partners (e.g. Turner, WaPo, Conde) want to book campaigns directly into Apple News inventory using their own DFP network. Apple sends a server-side request to its special fee-exempt DFP account, which then forwards to the publisher's DFP.
impression around 357,671,835
total DRX impression: 123,390,020,240
~0.3% of DRX traffic
At certain point, we wanted to deprecated this traffic. However, some ongoing projects are leveraging this traffic, e.g. go/dfp-video-partner-inventory-sharing and go/am-to-am-inventory-sharing. These new use cases indicate that even though the traffic is low currently, cross-publishers usage will increase in the future.

[redacted]

The initial design was proposed over 10 years ago.

It's more of a hack rather than a well thought system design. It was a pain in the system.

it blurs the lines between internal and external APIs, and have historically required poking custom holes through public APIs with special handling, which is error prone.
not flexible. If you want to add a signal, you need to add it to the redirected_url. It's difficult to figure out what happened, as it's a long way for outer SM to inner SM. Inner Bow processing needs to be understood, to make sure the signal is added correctly.
Unnecessary re-processing of the request (e.g. Geo resolution) for the redirected request, which increases the latency. For the signals that are the same for outer and inner request, we don't need to process again. mention Disney
rendering is implicit and difficult to understand, which is also error prone. There were different ways to concatenate the clickstring & viewstring

concatenated through click macro, destination_url
not concatenated for certain creative types.

To resolve these issues and optimize the traffic, we propose to remove the HTTP callback through DFP Bow, and further to implement DFP->DFP inred ad retrieval as a first class concept of DFP stack.

We've completed the milestone of this project, which is to make the Bow passthrough.

Cohort 5: Passthrough BOW on DFP ⇨ DFP Inreds

Before

Simplicity:
- − Double parsing, rendering
- − 8 Inner headers for Apple News

Uniformity:
- − One-off hacky solution
- − Different ways to concatenate clickstring and viewstring

Flexibility:
- − Difficult to add new features

Decoupling:
- − HTTPOverRPC API
- − Blurs the internal vs external line

After

Simplicity:
- + Single parsing, rendering
- + No more inner headers

Uniformity:
- + Consistent ad retrieval (The ART Project)
- + Consistent way to concatenate clickstring and viewstring

Flexibility:
- + Flexible for adding features

Decoupling:
- + No more HTTPOverRPC API
- + Simple library call

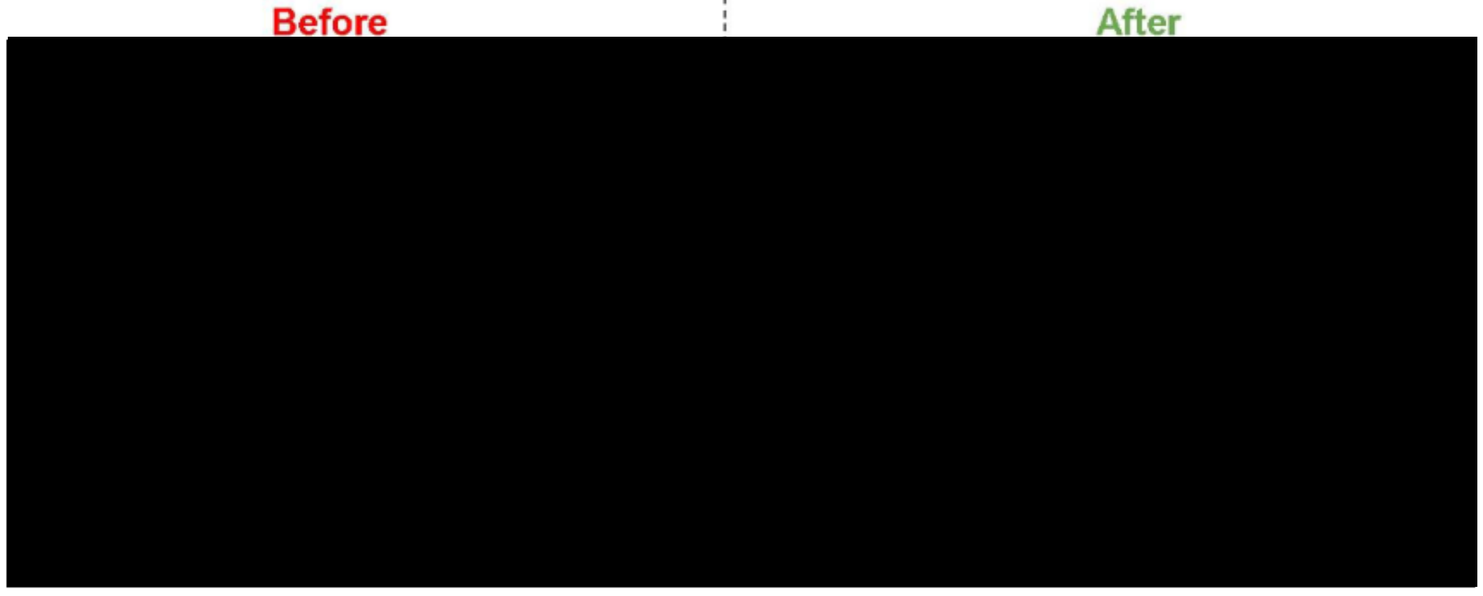
go/dfp-inred-passthrough

Make DFP Bow Passthrough on DFP->DFP Inreds (go/dfp-inred-passthrough)
Did the project design follow any of the design principles discussed here?
Simplicity
Easy to understand / reason about
current phase:
send SupermixerRequest instead of inred url
single rendering instead of double rendering
Easy to change / maintain
easy to add signal
Uniformity
Solve similar problems in the same way.
align with ART project. Consistent ad retrieval
Flexibility
easy to add feature
Decoupling: only have dependencies where necessary
no longer depends on HTTPOverRPC API
it blurs the lines between internal and external APIs, and have historically required poking custom holes through public APIs with special handling, which is error prone.
Was it designed with potential change and long-term maintenance in mind?
potential change: migrate video playlist traffic and video inventory sharing
long-term maintenance: deprecate HttpOverRpc
Were there any tradeoffs made when considering different principles?
cannot pass nested ContentAdResponse back to Bow
What was your experience with making changes to that project? Did it feel fast / slow, rewarding / painful?
slow:
too many fragile assumptions
How the clickstring/viewstring should be concatenated
the JSON wrapper should use inner/outer creative
whether client_enviroment should be consistent
bugs:
page_correlater
certain creative types/ viewstring not concatenated
no one knew how it works; required digging & exploring
testing
experiments
DFP AB test

rewarding

Cohort 2: Arbitrary Mediation Chain Rendering

Before | After



go/arbitrary_chain_rendering

Cohort 2: Arbitrary Mediation Chain Rendering

Before

Simplicity:
- − Partition and merging

Uniformity:
- − Different representations of 3p networks and google networks

Flexibility:
- − Hard coded assumption on the ordering of networks

Separation of Concerns:
- − Chain partitioning and resembling

After

Simplicity:
- + No partition and merging

Uniformity:
- + Uniform representation of 3p networks and google networks

Flexibility:
- + Able to handle more flexible chains, i.e, interleaving 3p & 1p

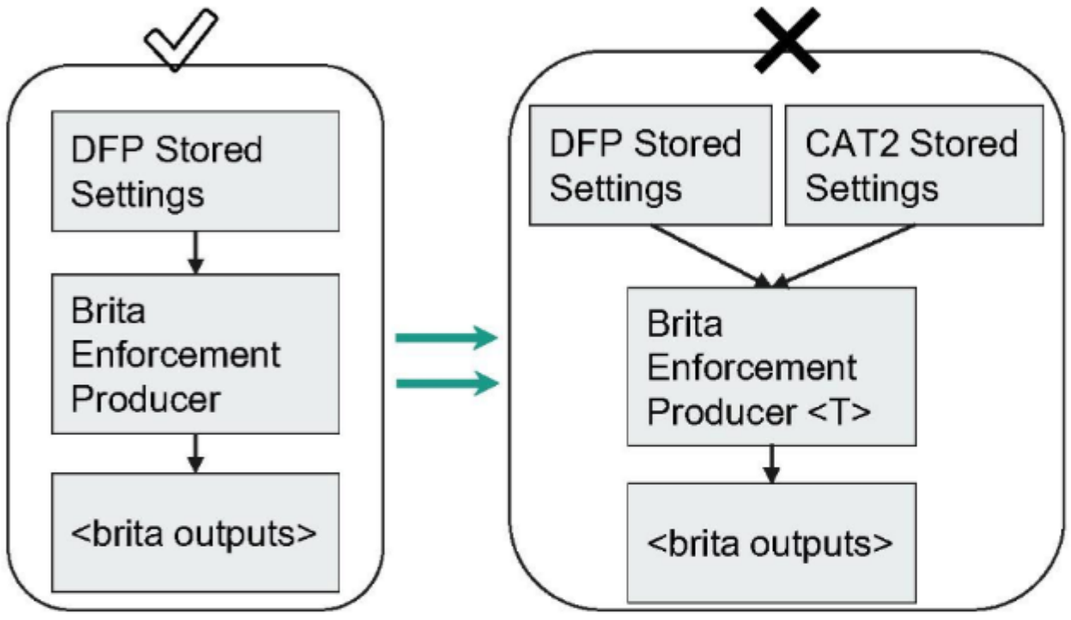
Separation of Concerns:
- + focus on rendering

Testability:
- + Rendering A/B testing, BOW regression test

go/arbitrary_chain_rendering

Cohort 7: BritaEnforcementProducer

- Producer initially designed & built to be used exclusively in DFP Supermixer, later needed CAT2 support
- New design: Template out the producer to accept either Stored Settings, or anything with `network_id()`
- Better solution: Migrate all logic to new library that handles business logic, have producer take in network_id instead
- Overly flexible: "Maybe we'll need to know what type of stored settings we're using!"
  - Spoiler alert: We don't
- Not simple: Templated code in header file, not intuitive to read code if unfamiliar
- Easy to add new extraction results, but just as easy to do this with the better design



Cohort 8: Minimum Bid to Win for Header Bidding

go/mbtw-design

Cohort 8: Minimum Bid to Win for Header Bidding

- **Uniformity:** Design was influenced heavily by prior art of server-to-server callouts. PostbackProxy and Harpoon were used to execute the callouts.
- **Flexibility:** Some effort was made to make it easy to use the system outside header bidding contexts (in the future). Where a new abstraction was needed, it was made generic. If an existing abstraction was available, it was used directly, even if it was header-bidding specific.
- **Single Responsibility Principle:** Responsibility was considered at a system architecture level. For example, we decided Supermixer should not be responsible for executing the callouts directly - instead that responsibility is delegated to PostbackProxy. This introduces complexity, but was a worthwhile tradeoff.
- **Decoupling:** While all the new Supermixer logic was run within a Producer, the actual Producer code was just coordinating inputs/outputs. Actual processing (i.e. forming the URL) was done by library code, which made things more readable and easier to test.

Cohort 4&6: Deprecating adx-{repeated}-query-fields

Before

After

Credits: go/auction-logging

Cohort 4&6Since 2016 - SSQ initiative - SSQ Infra - @henndiyHard to manage - pipelines owned across serving
Simplifying protos - while no-op

Cohort 4&6: go/deprecating-adx-repeated-query-fields

Before

Simplicity:
- − Aggregation at query level doesn't establish separation of concerns / single responsibility principle. No granular control.

Uniformity:
- − Not uniform across products - Video pods, single winner etc.

Flexibility:
- − Priorities changed over time causing the proto to be a melting pot for multiple use-cases.
- − Hard no-op change

Decoupling:
- − Long time maintenance and tech-debt is a concern. Testability is a major concern across pipelines since the protos seem overloaded in usage. Same protos tested differently across codebase.

After

Simplicity:
- + More granular control, but single responsibility principle might still not be achieved.

Uniformity:
- + Not uniform across products, but provides granular control in aggregation.

Flexibility:
+ Same as above, as it improves testability and uniformity in application of proto.

Decoupling:
+ Buyside-sellside separation is a major win. (go/j4w-rtb-buyer-identity)

Thoughts -
- + As a noogler it is slow but rewarding, as one gets to understand a lot about the serving stack

Session #4

dory/drx-learns-system-design

May 13, 2022

Cohort 1: Interscroller and Fullscreen Inline

- We had initially planned on implementing a new rendering for Interscroller ads
  - We chose to pivot when we realized that it was not revenue positive to have the new rendering
- **What went well**
  - **Uniformity:** We set the different responsive ad formats all the same way.
- **What was difficult:**
  - **Flexibility:** We were sort of able to pivot (and ultimately did), but it did take a bit longer than I would have liked.
  - **Decoupling:** All size decisions are incredibly complicated and intertwined. This means that figuring out what is due to one reason or another is difficult
- **Other Considerations:**
  - **Separation of Concerns:** All sizing decisions done in one place, but the specific decisions are intertwined.

Cohort 3: Publisher Provided Signals Evolution

PPS Alpha ⟶ PPS Beta

- "Temporary, proof of concept, E2E test" -> "Permanent, product feature, commercialization"
- **Uniformity:** Use Pipes & Tables, Dynamic Files, User Data Accessor, User Data Entity
- **Flexibility:** Allow for new signal sources and data lookups
- **Separation of Concerns:** High-level functionality exists in the appropriate part of the stack
  - Data lookup in Supermixer request processing; Privacy decisions in UDA; UDE generation in IBA
- **Separation of Concerns/Uniformity:** Coupling between Supermixer and IBA Server + special exceptions due to privacy policy propagation
  - UDE created in Supermixer; "Special" column accessor in IBA to bypass privacy restrictions
- **Simplicity/Separation of Concerns:** Low priority at low-level design AKA everything in a producer

Design Patterns


? ?
Inter faces

Interfaces

- Separation of concerns -> Modularity -> Interfaces
- Boundaries between components
- C : function declaration
- Java : interface
- C++ : pure abstract class
- Building blocks for most design patterns

Dmitriy

Interfaces : design principles

Cohesive	Collection of related functions
Opaque	Encapsulates / hides the implementation details
Independent	Independent of other systems, infrastructure and future product direction
Narrow	Does one thing well
Stable	Not likely to change

Interfaces define the language of the system.

Nick

Interfaces Example: Debug server event logging

Log all interesting events that occur during startup and request processing for a complex backend RPC server.

Simple Solution

- *Add LOG(INFO) calls whenever something interesting happens.*

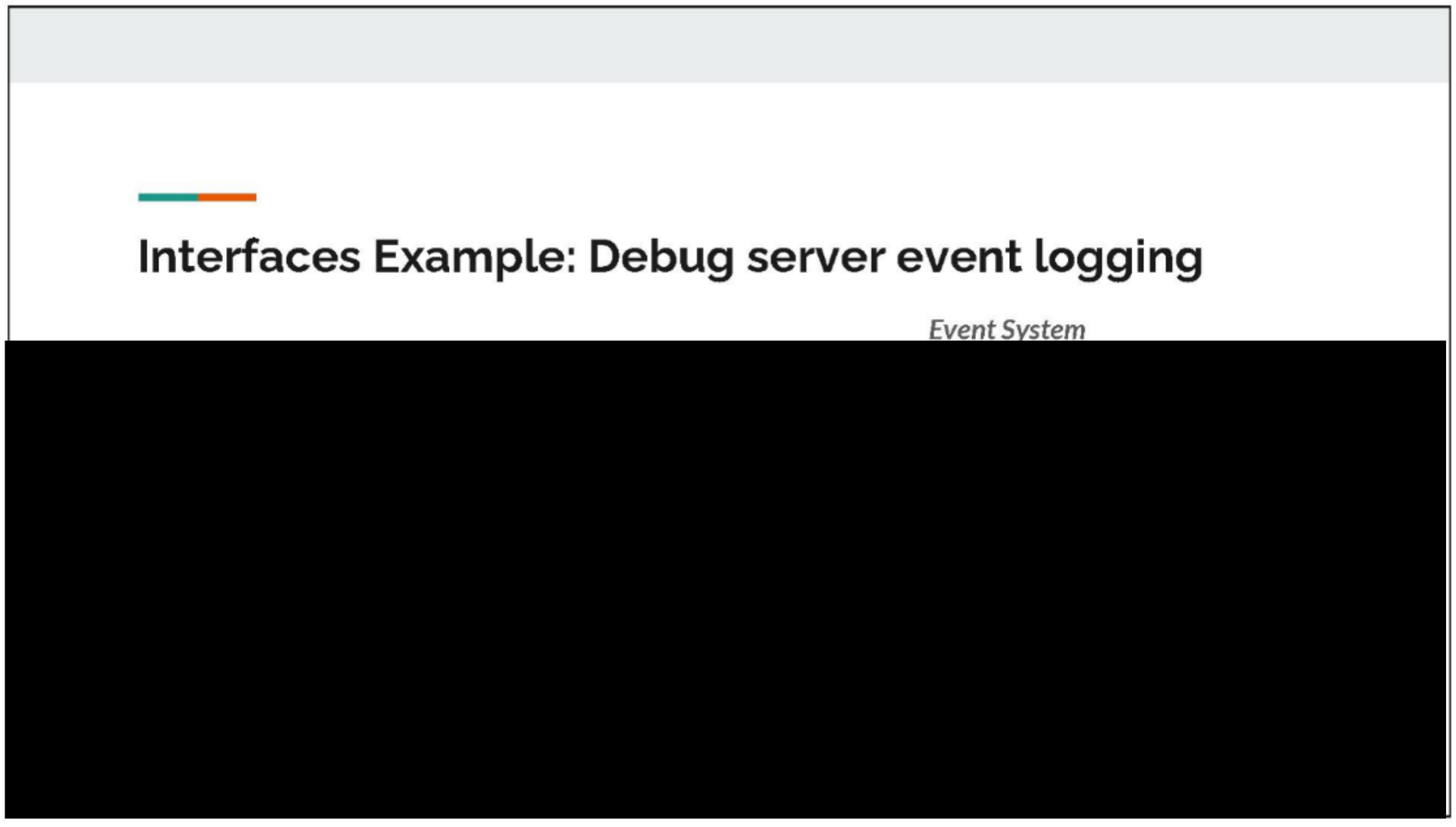
Likely scenarios

- *Now let's trace a specific request through the system*
- *It's too noisy, only log events for specific requests.*
- *It's too hard to access, just display the log in a web page via a debug end point.*
- *I think there might be problem, let's track "X" in real time.*
- *I can't automate anything, can you also write a structured log?*

Nick

Interfaces Example: Debug server event logging

Nick



Nick

A great alternative example is Superfilter:

Interfaces Example: Debug server event logging

Log all interesting events that occur during startup and request processing for a complex backend RPC server.

Simple Solution

- *Inflexible*
- *Exposed implementation details*
- *Much more difficult to maintain*
- *Not cohesive*

Event System

- *Simple*
- *Easy to test*
- *Establishes an implementation independent language (Event, Notify, Handle)*
- *Encapsulates implementation details*
- *Flexible*

Nick

Homework

- Design a set of C++ interfaces for one of the options below
- Draw a UML diagram (optional)
  - Reference 1, Reference 2 (Wikipedia)
- Improve as we progress through the course

Options:

- Dory Backend
  - List entries, add entry, vote, leave comment
- Moma Profile Backend
  - Search profiles, get profile, get reporting chain and teams

Each cohort to prepare a slide for discussion in the next session on 5/27/2022.

Dmitriy

Discussion : Models before Modules

- Are 10 modules/interfaces better than 1?
- Poorly designed interfaces can be counterproductive
- Models, then modules
- Business and infrastructure concerns/abstractions and their interactions
- Interface : Lightweight formal documentation of domain concepts

Dmitriy, then discussion

Session #5

dory/drx-learns-system-design

May 27, 2022

Survey

Please fill out go/drx-learns-survey-2

Today we'll discuss homework on Interfaces

- Design a set of C++ interfaces for one of the options below
- Draw a UML diagram (optional)
  - Reference 1, Reference 2 (Wikipedia)
- Improve as we progress through the course

Options:

- Dory Backend
  - List entries, add entry, vote, leave comment
- Moma Profile Backend
  - Search profiles, get profile, get reporting chain and teams

Each cohort to prepare a slide for discussion in the next session on 5/27/2022.

Dmitriy

Cohort 2 - Backend

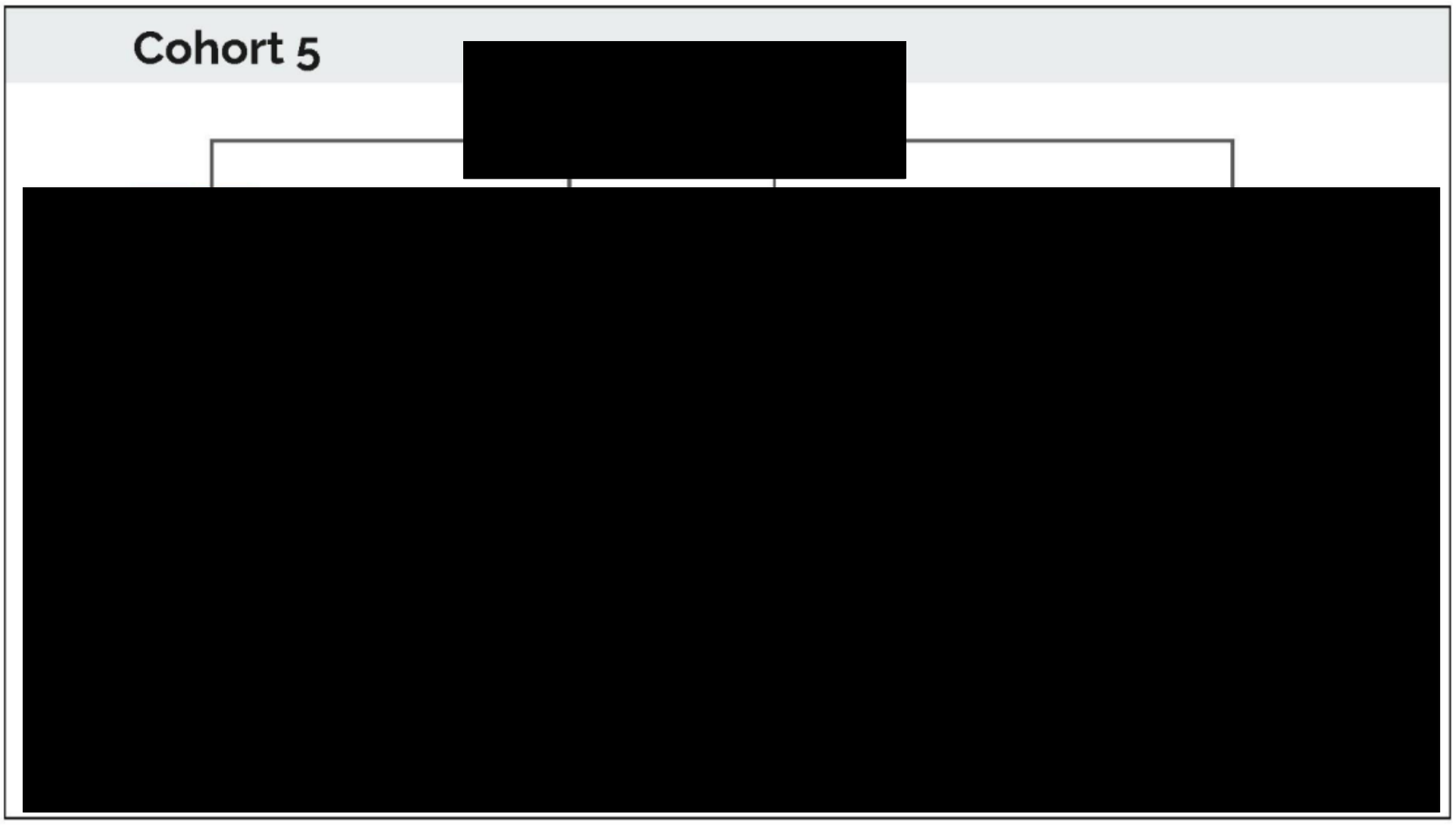
[redacted]

Presenter: trentunderwood@

Cohort 2 - FrontEnd Widgets

Cohort 1

Presenter: Alberto

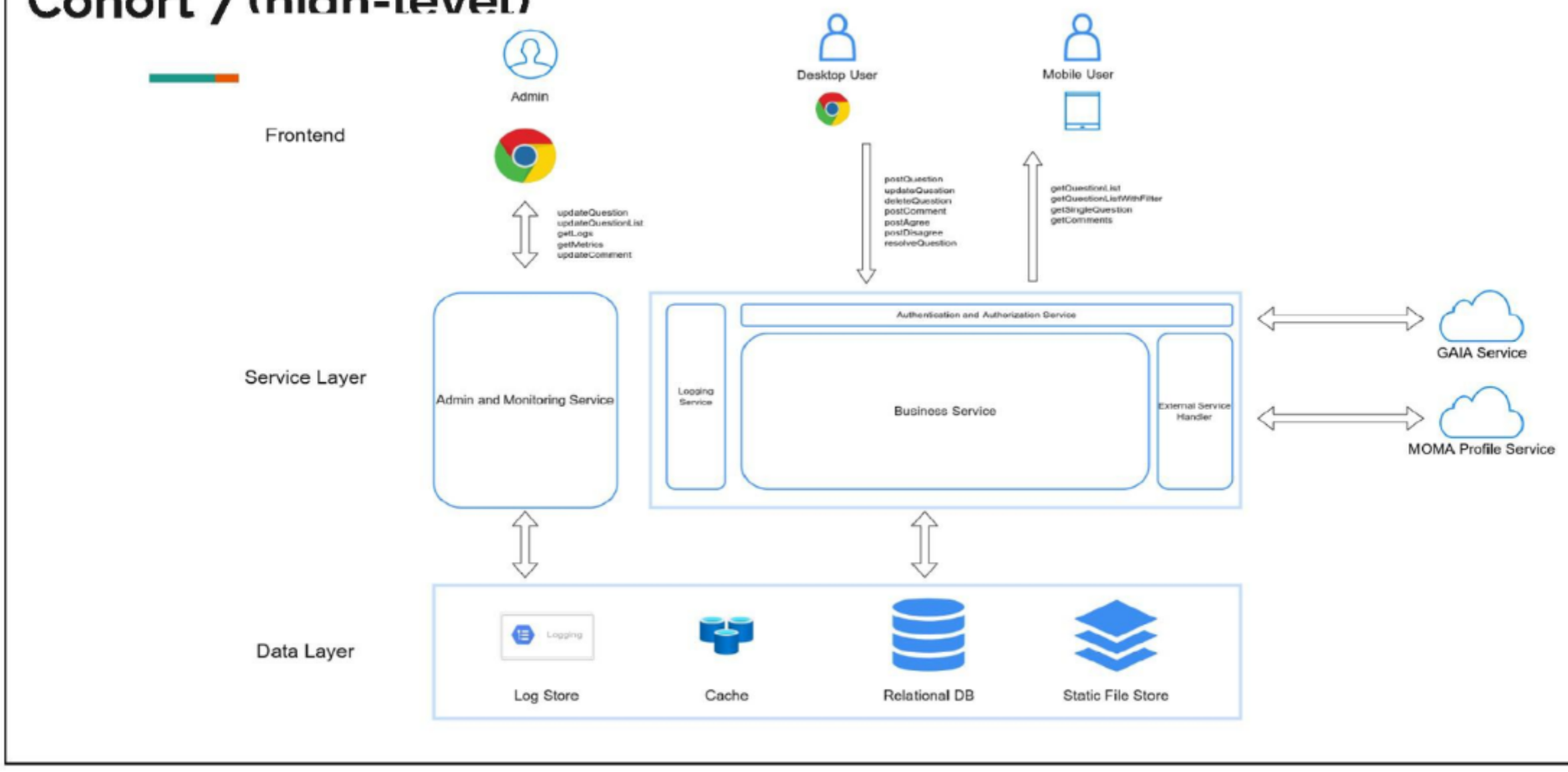


Presenter: Jackson
Dory backend

Cohort 7 (backend)

Presenter: Zayd

Cohort 7 (high-level)



Cohort 8

Presenter: Tian

Cohort 3

[redacted]

Presenter: Mark Liang

Cohort 3

[redacted]

Cohorts 4 & 6

Presenter: Ben

Session #6

dory/drx-learns-system-design

June 13, 2022

Encapsulation

Principle: **hide what might change**

- E.g., data and data structures
  - internal field not critical for the API consumer
  - state that should affect the behavior of the object, e.g. via polymorphism AuctionCandidate::is_adwords(), AuctionCandidate::is_managed_tag()
- E.g., algorithm
  - ADT for list: array vs linked;
  - bubble sort vs. quicksort
  - selecting an auction winner
- E.g., technology
  - dataset access, logging, RPC calls
  - concurrency/threading mechanisms

Credit to ghf@ for slides on encapsulation

Schedule

6/3 - Encapsulation + dependency injection + HW

6/27 - Composition vs. Inheritance + Role Interfaces + Adapter

7/11 - Domain-driven design

Survey responses (24)

Encapsulation is widely misunderstood

These are often misunderstood as “encapsulation”:

- Separate interface and implementation
  - Good idea, but alone it doesn’t avoid ripples
- Hide the implementation
  - Changes to hidden code can still hurt you
  - Can’t do this in google3 anyway
- Write getters and/or setters for data fields
  - getBalance(), setBalance()

Insight: Some interfaces are better than others

Notional "levels" of modularization

- Level 0: No interfaces
- Level 1: Functional interfaces
  - Procedural programming
  - Classes and OOP
- Level 2: Encapsulation / info hiding

What happens

- Big ball of Mud pattern
- Changes ripple through the system
- Changes are localized

82

go/design-catalog/big-ball-of-mud

Benefits of encapsulation

- Change is inevitable
  - ... but changes that **ripple** are painful. What happens if we minimize ripples?
  - Callers can work on their own roles, avoid unnecessary churn
  - Implementers get freedom to optimize & refactor
- Shorter development time
  - Enabling parallel work streams (same as any interface)
  - Minimizing ripples; less rework after refactoring (unique for info hiding)
- Flexibility
  - When inevitable change happens, want to rearrange chunks, not start from scratch
- Comprehensibility
  - APIs become easier to understand
  - Minimizes the knowledge needed to proceed (need not look into the box)

Encapsulation and Hyrum's Law

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.

Examples

- Order of sizes, deals, billing IDs in RTB bid requests impacting bidding behavior
- "Criteo currency outage" (2019): inferring currency from geo, O($MM) impact

Encapsulation: hide what might change

- "Hide" means "avoid dependencies on"
- Hyrum's Law says that, at scale, you can't eliminate dependencies
- So, are encapsulation and contracts useless?

84

This is ghf@'s viewpoint

Analogy: red lights can't stop all cars from hitting you in an intersection -- but they still help a lot and we shouldn't abandon stoplights

Longer version:
Starting around 1970, mathematicians thought programming should be done their way
They had some early success
They said the future was "proving code correct"
Correct with respect to what, exactly? The specification.
We have largely abandoned the idea of large-scale program proofs (except in the small)
Specification is helpful for other reasons, including encapsulation and DBC
Hyrum's law describes the upper limit of how much specs / contracts / encapsulation can help

Encapsulation example: RTB AdSourceInterface

- Responsible for ... retrieving and processing the matching ads, and sending the candidate ads to the auction.
- Which methods support encapsulation?
- Which methods don't?

Why SWEs may fail to encapsulate

Parnas identified these obstacles:

- “flow-chart instinct” yields ugly interfaces
- “seems like too much planning”
- “bad models” (cultural obstacles)
- “extending bad software”
- “too bad we changed it” - design not communicated

Dependency Injection

- How can a class be independent from the creation of the objects it depends on?
- How can an application, and the objects it uses support different configurations?
- How can the behavior of a piece of code be changed without editing it directly?

- We'll focus on Constructor Injection.
- Dependencies are interfaces

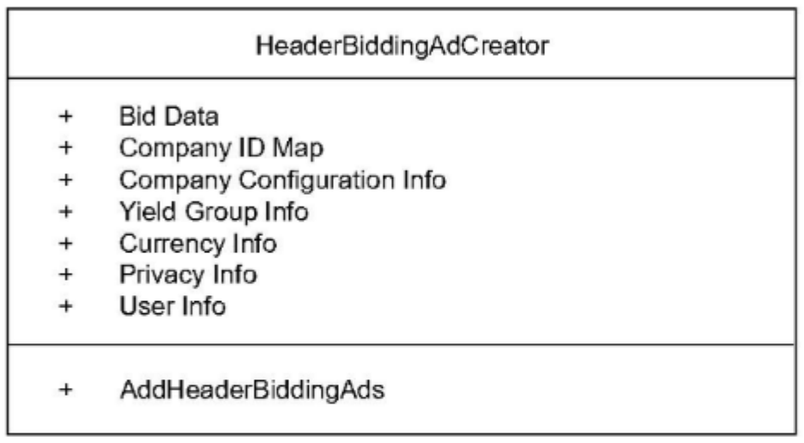
Cody's prep: https://docs.google.com/presentation/d/17oFr0g3ypSx6k-7tHlN1X8R72ZhAjG2XZccOtv4586Q/edit?resourcekey=0-ihGBsvsfIasEHmKX8ep3Nw#slide=id.p

Dependency Injection: Header Bidding

- We have a set of bids coming from the client
- These bids may or may not be considered eligible as ads
  - No buyer presence in a yield group for the query
  - No consent from the user for that buyer to serve
- Naive design:
  - Add relevant parameters to the function creating the ads, and drop them as needed
- Dependency Injection design:
  - Put each decision behind an interface
  - Pass interfaces to the function
  - The function then just asks “is this allowed”

Naive Interface

- Very bulky, nine complex parameters
- Bag of data without meaning
- *why* are these the inputs isn't clear
- Testing the code requires testing many input combos



Better - Add some Modularity

- Break one decision with multiple concerns into parts.
  - YieldGroupCompanyMapping: Does the pub allow this bidder on the query?
  - HeaderBiddingVendorConsent: Does the user allow this bidder on the query?
- Owning class coordinates construction.
  - All the inputs to these classes are still input to HeaderBiddingAdCreator.
- Still need to test creation and interaction o subclasses.
- Can't mock subclasses for tests.

Best - Inject APIs into the Ad Creator

- Dependency Injection in the constructor
  - Pass the "decision" classes into the HeaderBiddingAdCreator constructor.
  - Instead of creating them as part of construction.
- Inputs have meaning, instead of being bags of data/primitives
- Can use mocks for testing
  - Only have to test "decision" class API interactions, not the classes themselves.
- Can test API boundaries, instead of a monolith

Dependency Injection : Filtering

- For a given ad, do request, publisher, and policy constraints allow it to serve?
  - If not, filter it
- A common API for different kinds of candidates
- Each filter operates on that same API per-candidate
- But each filter may have different non-candidate state
  - For lots of different kinds of filters, how can we create a framework to add and change them frequently?

Dependency Injection - Filtering

- A pure virtual filter interface defines a simple API for each filter.
- Each filter may take different dependencies to help with their decisioning.
- One filter can even take *other* filters as injected dependencies.

Dependency Injection - Filtering

Dependency Injection - Filtering

Dependency Injection - Filtering

- Each filter can be tested separately.
- Can use mocks to test the filter manager.
- Separates each filtering decision into its own module

Dependency Injection : SelectAndResolveController

- Very wide APIs
- Tight coupling
- Hard to test
- Hard to modify
- Hard to extend
- 2K lines of “glue code”
- A lot of mutable state

Disclaimers
The current state isn’t anyone’s fault
The description of the current state may not be fully accurate
This is not an actual refactoring proposal, the diagrams are for illustrative purposes

Dependency Injection : SARC

[redacted]

Dependency Injection : SARC

[redacted]

Dependency Injection : SARC

Data dependencies get injected directly into their consumers vs being plumbed through multiple layers of “parent” components.

Dependency Injection : SARC

- Modular
- No coupling
- Narrower APIs
- Easier to read, modify, extend and test

Disclaimers

- *The current state isn't anyone's fault*
- *The description of the current state may not be fully accurate*
- *This is not an actual refactoring proposal, the diagrams are for illustrative purposes*

Encapsulation: discussion

- Can you think of an example of poor encapsulation in your project?
- How could that example benefit from encapsulation?

102

This is ghf@'s viewpoint

Analogy: red lights can't stop all cars from hitting you in an intersection -- but they still help a lot and we shouldn't abandon stoplights

Longer version:
Starting around 1970, mathematicians thought programming should be done their way
They had some early success
They said the future was "proving code correct"
Correct with respect to what, exactly? The specification.
We have largely abandoned the idea of large-scale program proofs (except in the small)
Specification is helpful for other reasons, including encapsulation and DBC
Hyrum's law describes the upper limit of how much specs / contracts / encapsulation can help

Homework

- Identify ways to improve encapsulation in your codebase?
- Find examples of Dependency Injection in your codebase
- Find opportunities for using Dependency Injection in your codebase

Session #7

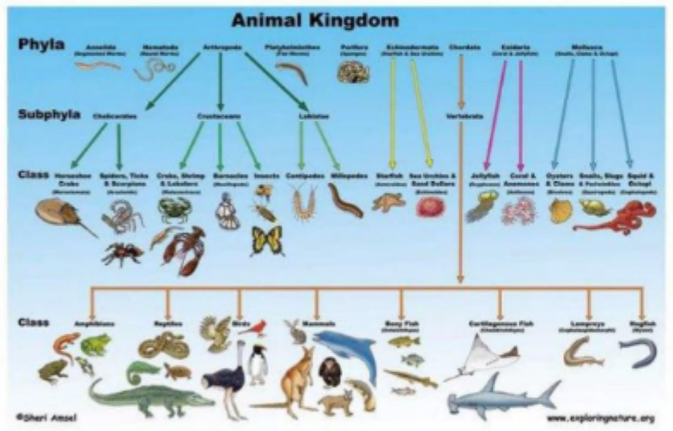
dory/drx-learns-system-design

July 18, 2022

Composition vs. Inheritance

Composition

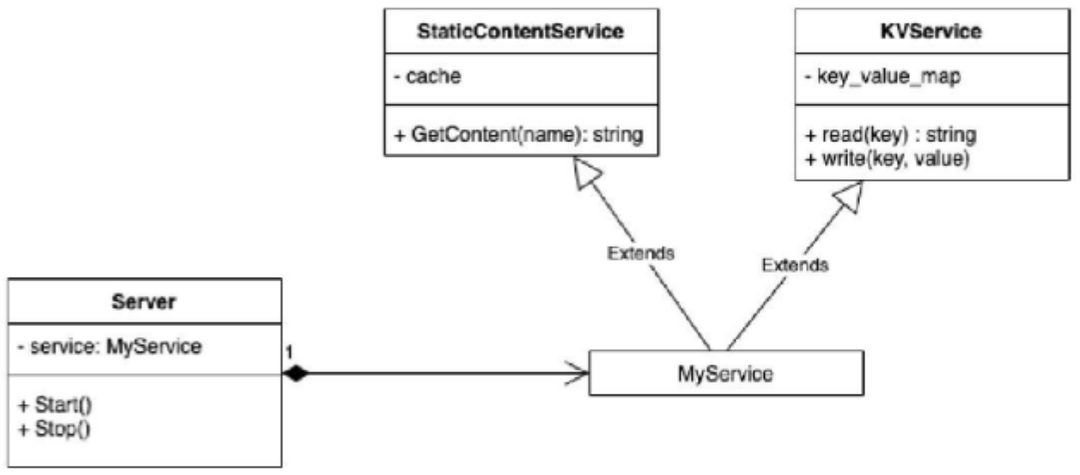
Inheritance



Composition vs. Inheritance

Example:

Implement a key value storage service that also retrieves static content.

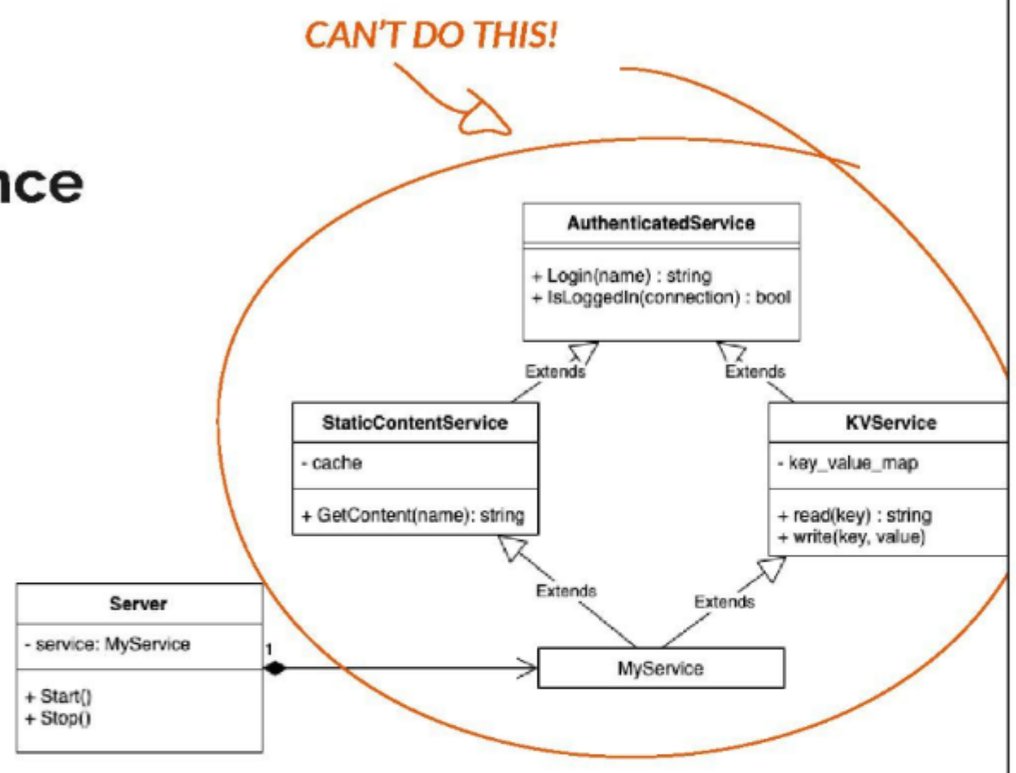


*We can use **inheritance** to reuse existing implementations, but…*

Composition vs. Inheritance

*... sharing **implementation** through **inheritance** has limitations*

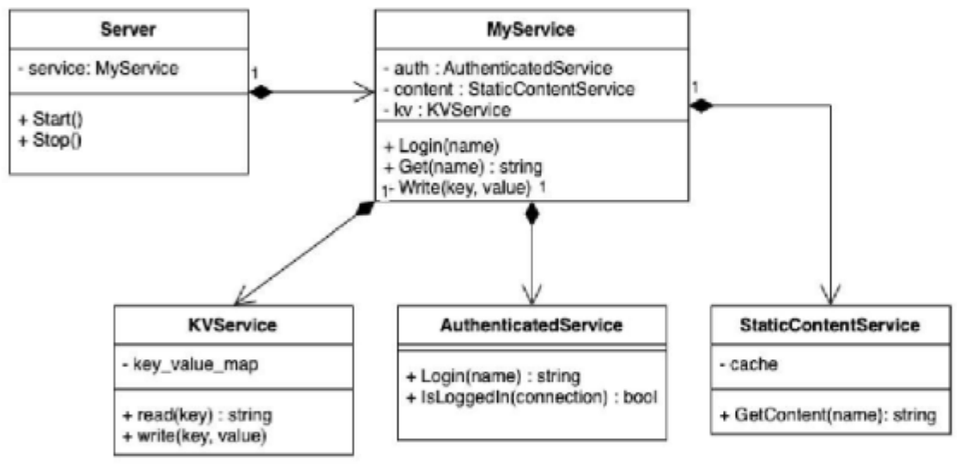
- tightly coupled
- “diamond of death”
- difficult to maintain
- limited flexibility
- compile time only



Composition vs. Inheritance

Prefer composition over implementation inheritance.

- loosely coupled
- can support runtime reconfiguration
- naturally supports encapsulation



Composition vs. Inheritance

Composition	*Inheritance*
- models a“has a” relationship - objects communicate through interfaces - can / should be combined with inheritance for interfaces - can be difficult to get right	- models an “is a” relationship - well suited for interfaces - inheriting for implementation has limitations - can be easier at first, but quickly becomes unmaintainable

Adapter: CAT2AdAdapter

- Buyside has a representation of an ad
  - To bid
- Sellside has a representation of an ad
  - To decide a winner
  - To generate a response to Bow
- Buyside doesn't want pollute their interface with sellside details
- Sellside doesn't want to maintain buyside data structures
- How do we decouple? Enter, adapters

Adapter: CAT2AdAdapter

[redacted]

Wrappers and Adapters

- Use case for multiple implementations of a single interface
- Reuse a class that doesn't implement an interface client code needs
- Provide an alternative interface for an existing class
  - (Hopefully) don't have to change the interface of the existing class
- "Adapting" class A to implement the interface of class B
  - Create a new class AtoBAdapter implementing class B with methods of class A

Role Views

- Variation of the Adapter/Wrapper pattern
- A use-case specific interface on top of a large proto/struct/class
- Good data model != Good APIs

Role Views : Examples

- API Narrowing :

[redacted]	Which fields are being read?
	Which fields are being written?
	[redacted]

- API Customizations :
  - return optional<int> for a proto field - a better API than has_ checks.
  - return MonetaryValue instead of MonetaryValueProto

Role Views : CAT2 Auction Video Request Inputs

Role Views

- Pros
  - API Narrowing
  - API Customization
  - Better encapsulation
  - Better testability
  - Fuller API ownership
  - Refactoring Enablement / Insulation
- Cons
  - More work (short-term)

Homework (pick one option)

Option 1 - Composition vs Inheritance

Discuss with your cohort situations where implementation inheritance might be preferred over composition. Be prepared to answer the following questions:

- What were some of the situations that were discussed?
- Were there any situations discussed where the group ultimately decided that inheritance was the better solution?
  - Why?
  - Why not?

Feel free to use any of the examples below:

- Testing frameworks
- Integration / unit test data sharing
- Execution frameworks like producers
- Logging frameworks

Option 2 - Adapters / Role Views

- Find example(s) of Adapters/Role Views in your codebase
- FInd potential use-cases for Adapters/Role Views in your codebase

Session #8

dory/drx-learns-system-design

Aug 15, 2022

What is domain-driven design?

"Domain-driven design is both a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains."

– Eric Evans

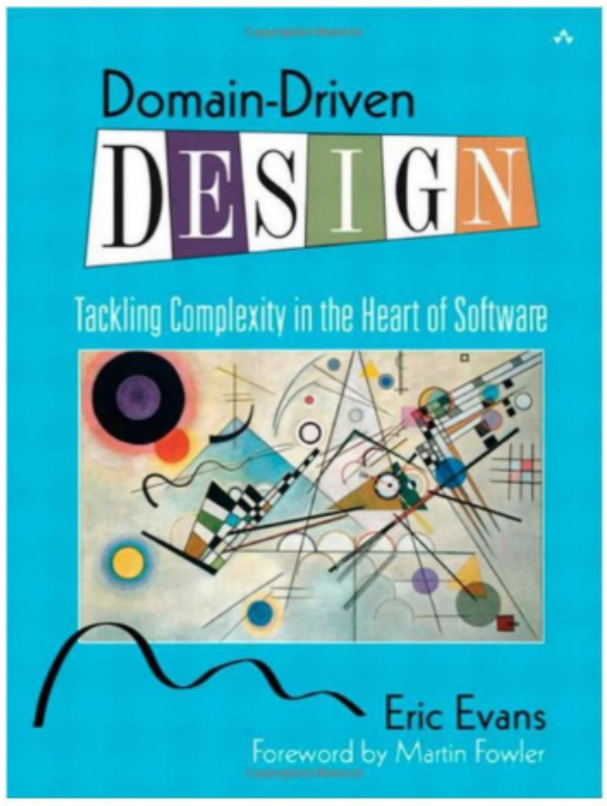
https://www.dddcommunity.org/uncategorized/preface/

What is domain-driven design?

- Philosophy, a way of thinking about delivering software
- A way to communicate about a software project, the underlying domain and the code
- An approach to software development
- **Software can be a model representative of the real world domain**
  - Concepts in domain model directly tied to code
- **Focus on important:** the core domain and domain logic
  - Hide the rest (infrastructure, implementation details)

Eric Evans classic book

https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215



What is a model?

- Real-world problems are messy
- Model is **NOT** a perfect copy or description of the real world
  - Messiness of the real world might impede efficient software development
- Representation of some subset of a real world problem...
- ...that is *useful*, *convenient* and *efficient* for designing and developing software
- Can include concepts (objects), their lifecycle and relationships
- Can systematize, organize, simplify or give aliases to the real-world objects

Effective modeling ingredients

- Bind model & implementation
- Cultivate a *language* based on model
- Develop a *knowledge-rich* model
  - Model is not "*just data*" – captures rich interactions for solving complex, real problems
- Distill the model
  - Remove unneeded concepts
- Brainstorm & experiment

Knowledge crunching & Continuous learning

- Iteratively try different models & identify most successful one
- Lots of discussions between engineers & domain experts
- Continuously refine the model based on better understanding of domain
- Implementation from early prototype to richer model via refactoring

What does your team do?

- Has your team built a meaningful model(s) for your project(s)?
- Are these models universally known to / agreed upon by team members?
- Does your team practice knowledge crunching?
- Does your team refactor when the model changes?

Knowledge crunching techniques

- Drawing mind maps / diagrams
- Domain expert interview
- Review of existing domain-related artifacts (documents, diagrams, ...)
- Prototyping in code / tests
- Experimentation / Trial and error
- Refactoring

Translation slows down communication

Does your code use the same terminology / language as business people and users?

Do you have to *translate* from language in a PRD to concepts in your codebase / talking to peer eng?

Core philosophy of DDD is utilize the same language from PRD to unit tests - within clear boundaries

Ubiquitous language

- Everyone, technical and nontechnical, must speak the same ubiquitous language:
  - use the same **terms**, and
  - give the same **names** to
  - the same **concepts**,
  - within specific **boundaries**.
- Used **in code**, among engineers, domain experts, product / business stakeholders
  - Remove the "translation layer" between engineer and business talk

Core philosophy of DDD is utilize the same language from PRD to unit tests - within clear boundaries

Example: What is a Buyer? - Business Perspective

- These tend to be real world entities
- Tied to companies or people
- Example entities
  - Agency - A company representing advertisers, booking campaigns
  - Advertiser - A company that wants to show advertisements
  - External tech companies bidding on behalf of the above

Example: What is a Buyer? - Product Perspective

- Tend to think in terms of User Journeys
- Examples:
  - Someone spending dollars
  - Someone negotiating a deal
  - An entity a publisher wants to block
  - A “Bidder” (we could unpack this as well)

Example: What is a Buyer? - Engineering Perspective

- Tend to think in terms of pre-defined ID spaces
- Examples:
  - Buyer Networks - An ill-defined "seat" concept in practice
  - Customers - literally a row in the customer table
  - DSPs - the technical endpoints, not the companies
  - DV3 Partner - someone with a DV3 account

Expressing model in software

- **Entities**: identity and lifecycle
  - Publisher, buyer...
- **Value objects**: no conceptual identity
  - Slot geometry
- **Services**: operations over entities and value objects...
  - Do not fit cleanly under either an entity or a value object
- **Modules**: logical grouping of highly-cohesive domain elements

Domain vs. infrastructure-driven design styles

- Infrastructure-driven – focus on:
  - Binaries
  - RPCs
  - Data "piping"
  - Producers
- Domain-driven – focus on:
  - Modeling real-world concepts
  - Entities and relationships
  - Actions and services
  - Knowledge-rich model with continuous refinement
  - Encapsulation
- Which one do you use more frequently?
- Which one does your team use more frequently?
- Why?

Example: get bids from RTB bidders

- Each bidder may have multiple URLs to send bid requests to
- Certain optimization function governs which URL to send a request to
  - E.g., based on lower latency
- Each URL has a QPS limit and a protocol associated with it
- Once sent, a bid request may succeed or fail
- A bid response may include zero or more bids

Entity
Service
Value object

Example: getting bid responses from bidders

Example: getting bid responses from bidders

- Bidding endpoint: allows to send bid requests and receive bid responses
  - "Transport layer"
  - Associated with a specific bid request protocol, network URL
- Routing strategy: selects the best matching bidding endpoint to send a given candidate bid request for a given bidder to
  - Based on bidder configurations, other optimization constraints (latency, availability...)
  - May not always succeed (no valid endpoints with available capacity)
- Bid request factory: creates outgoing bid requests in a specific wire protocol
  - Based on request-level and bidder-level inputs

Example: getting bid responses from bidders

Why domain-driven design?

- Forcing function for choosing **clear concepts & behaviors** of the system being built
  - Reduce ambiguity
  - Seek precision in requirements & their expression in the modeling language, software
  - Look for patterns, simplifications
- **Increase velocity, lower costs** of development and maintenance
  - Thanks to using ubiquitous language in communication
    - Across the engineering team, domain experts, business, legal, gTech...
  - Thanks to clear manifestation of the model in code
  - Thanks to rich knowledge of the model on the team
- Learning: team constantly
  refines their understanding of the real-world problem at hand
  - Points to better opportunities for solving our users' problems

Next steps

- Consider how DDD could be applied to your project & what benefits it could bring
- Consider what are the impediments of applying DDD on your team
- Pick a read on domain driven design (an article, a book, ...)
- Try DDD on one of the future or current projects