Design Patterns

Reasons:

1. Shared vocab.
2. Leverage past design
3. Flexibility
4. Reusability

General principles:

1. Composition > Inheritance
   "has-a" "is-a"
2. Liskov subst. princip: subclass should behave like superclass

Design patterns

- **Creational** — Flexible obj creation
  - Factory
  - Builder
  - Prototype
  - Singleton
- **Structural** — Flexible prog. struct.
  - Adapter
  - Bridge
  - Decorator
  - Facade
  - Proxy
  - Flyweight
- **Behavioral** — Communication
  - Interpreter
  - Template
  - Visitor
  - Observer

Software Arch. Intro

Design process

Ideation → Analysis → Selection → Elaboration → Iteration

Analysis: Determine Criteria (Fits purpose; Cost of prod.; Cost of operation; Future) / Apply criteria

Architecture: parts of sys. that are hard to change

Based off:

1. Functional req: what is system supposed to do.
2. Non-" : how system should be
   ↳ availability, durability

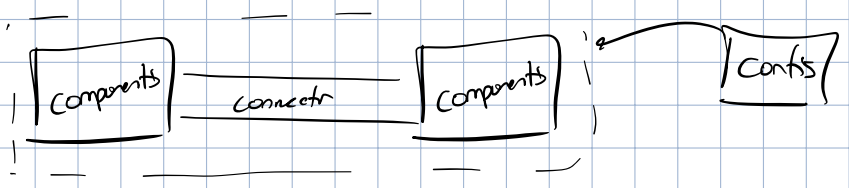
Prescriptive arch → [Impl.] → Descriptive arch (how system actually impl.)

Problems:

① Drift: important/principal decisions in descriptive not in prescriptive, not violate

② Erosion: " , violating prescriptive

Broadly:



Views

Topological goals:

① Minimize coupling b/w components

② Maximize cohesion w/in components

View: subset of arch. design decision

A: Component view

B: Sequence diagram

C: Deployment " : maps components ↔ physical devices

D: State diagram

Non-functional properties

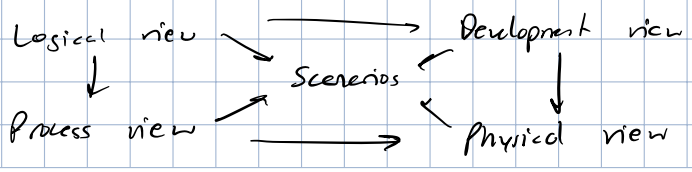
Efficiency — Evolvability

Complexity — Dependability

Scalability

ARCHITECTURAL STYLES

4+1 View model



Logical view: functionality of system (eg. UML)

Process view: dynamic aspects (eg. comms, run time behavior). Sequence diagrams.

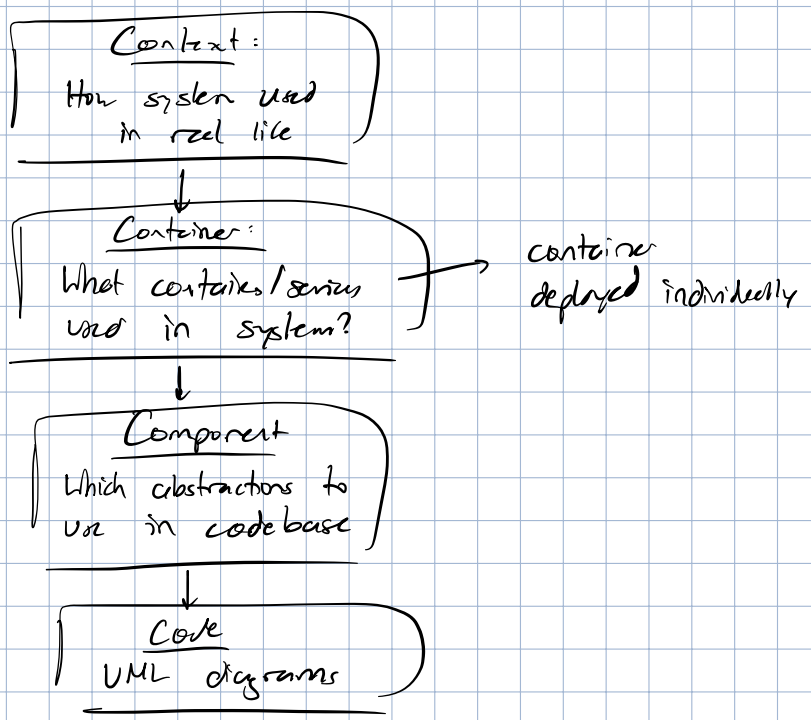
Development view: system from programmer perspective, uses UML

Physical view: physical layer of system

Scenarios: use cases

C4 Model

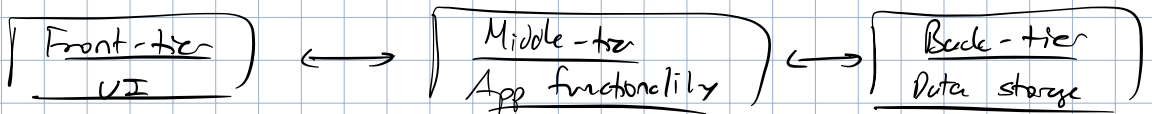
Communicate arch. at diff. abstractions



Architectural Patterns

Defn: set of design decisions applicable to recurring design problems. Can account for diff. contexts

① Three-tiered pattern



Ex:// MVC, sense-compute-control in sensor systems

Architectural Styles

Properties of good arch:

① Consistent, principled techniques

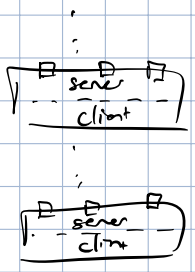
② Resilient

③ Guidance through arch

④ Reuse of knowledge

Style #1: Layered

Hierarchy, w/ each layer exposing APIs



Pros:
1. Abstraction
2. Evolvable & not disruptive to change
3. Standardized interfaces possible

Cons:
1. Not universally applicable
2. Performance

Style doesn't fully dot. arch.
↳ Defines domain discourse

Style #2: Dataflow

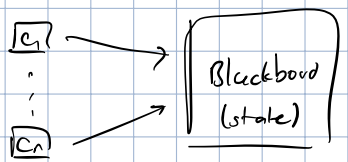
Dataflow

- Batch-sequential
  - Seq. prog. executed in order & data passed sequentially in aggregate
  - Ex: // Airflow
- Pipe-and-filter
  - Components filter data streams
  - Data incrementally produced
  - Ex: // Flink
  - Invariants:
    1. Filters are indep.
    2. Filters should not know of up/downstream filters

Style #3: Shared Memory

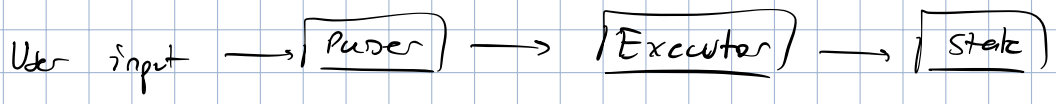
Shared Memory

- Blackboard

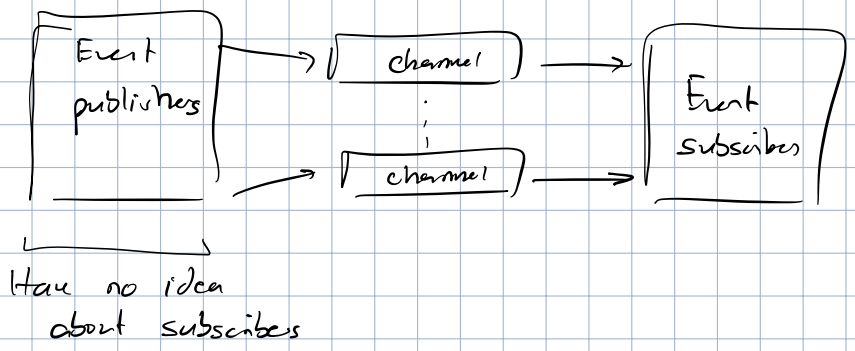


- Rule-based
  - User input → inference engine → rule / query
  - use rules to resolve

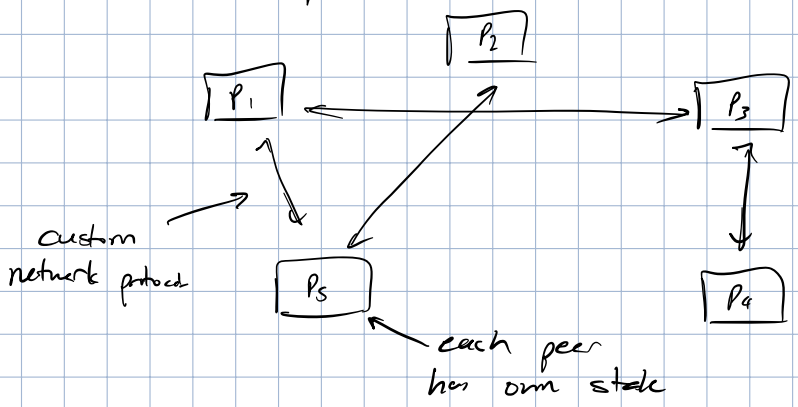
Style # 4: Interpreter



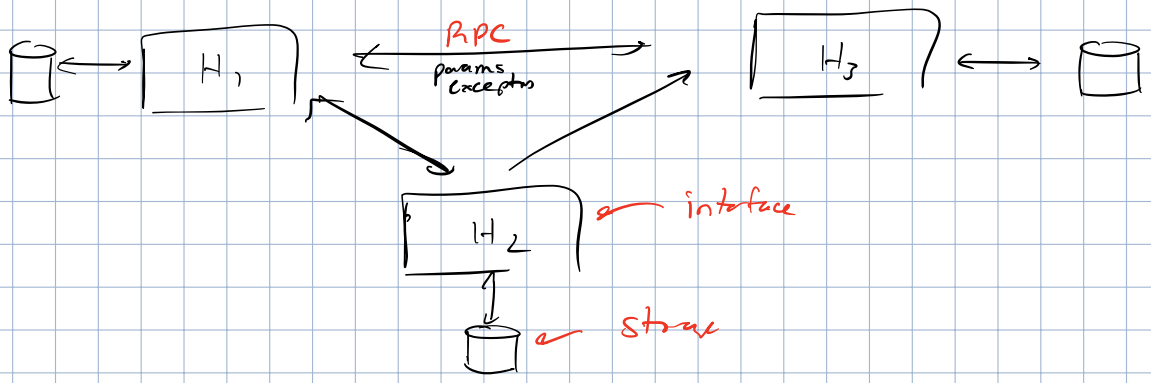
Style # 5: Implicit Invocation



Style # 6: Peer-to-peer



Q: How to work w/ distributed objects:



Design recovery

Problem: codebase exists but arch. is not documented

Obj: determine topology, components, connectors.

Method ①: Syntactic clustering

Look @ static, code-level relationships

Method ②: Semantic clustering

Look @ dynamic input & entire sys. domain

ARCHITECTURAL MODELING

Defn: artifact that captures design decisions used in arch.

What to model:

- Components
- Connectors
- Interfaces
- Config

} Rationale & constraints

- Dynamic aspects
- Functional & non-functional

Models should be unambiguous, accurate & precise

Approach: make subset model (views) to model everything

↳ Should be consistent across all views

↳ Inconsistency types:

1) Direct inconsistency (just conflict plainly)
2) Refinement " : high-level conflict w/ low level
3) Static vs. Dynamic
4) Dynamic vs. dynamic: descr. of dynamic aspects conflict
5) Functional vs. non-functional

Eg: // UML

Pros: ubiquitous, extensive docs & support

Cons: difficult to evaluate consistency

Can even have architecture descr. languages

SECURITY

Principles

Confidentiality
Unauthorized parties shouldn't have access to data

Integrity
Only authorized parties can change data

Availability
Resources available to authorized parties at all times

Design principles:

1. Least privilege: give each component as little privilege as required
2. Fail-safe default: deny access if explicit perm. absent
3. Economy of mechanism: adopt simple security mechanisms
4. Complete mediation: ensure all access is permitted
5. Open design: secrecy should not be necessary for security
6. Separation of privilege: introduce multiple parts to avoid exploitation
7. Least common mechanism: limit critical resource sharing to few mechanisms
8. Psychological acceptability: make security mechanisms usable
9. Defense in depth: have multiple layers of countermeasures

For each, can either have:

1. Discretionary access control: look @ request identity → decide
2. Mandatory " : requestors must have policy

DESIGN INTRO

Analysis vs. design:

Analysis addresses function, but design models implementation

Use recurring design for:

1. Abstraction
2. Flexibility
3. Modularity
4. Elegance

STUPID principles: Singleton, tight coupling, untestable, premature opt., indescriptive nams, duplication

SOLID principles:
- Single responsibility: classes should only have 1 task
- Open/closed: classes should be open to extension (subclasses), closed for modification
- Liskov substitution: subtypes should behave like supertype
- Interface segregation: only key methods in interface
- Dependency inversion: minimize coupling & prevent high-level modules depending on lower levels

Lower-level principles:
- Encapsulate variability
- Program to abstractions, not implementations
- Composition > inheritence
- Loose coupling

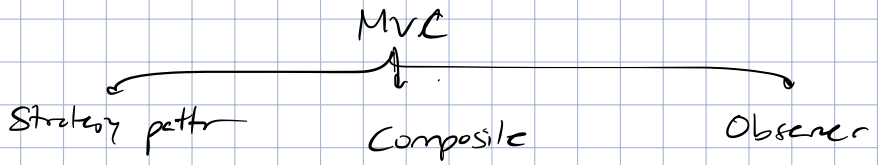
Make it simple! DRY & KISS

Code smells:
- Large classes/methods
- Control struct. nesting
- Duplicate code
- Static state
- No encapsulation/info hiding
- Many parameters
- Implementation classes > abstractions

Refactoring is necessary → good test coverage is useful

MVC

Know MVC &:



MVP (model-view-presenter) has presenter instead of controller
↳ Business logic done in presenter rather than model

MVVM (model-view-viewmodel) works exactly like MVC but model direct notifs. to ViewModel

Dependency injection

Dependency inversion principle: program to interfaces, not to implementations

We don't want higher-level modules to be dependent on:

1. Implementation details
2. Lower-level modules

Dependency injection involves:

1. Take components → add config data
2. Components & config given to injection framework
3. Injector bootstraps fake object w/ data & class

} Ex: // Guice framework

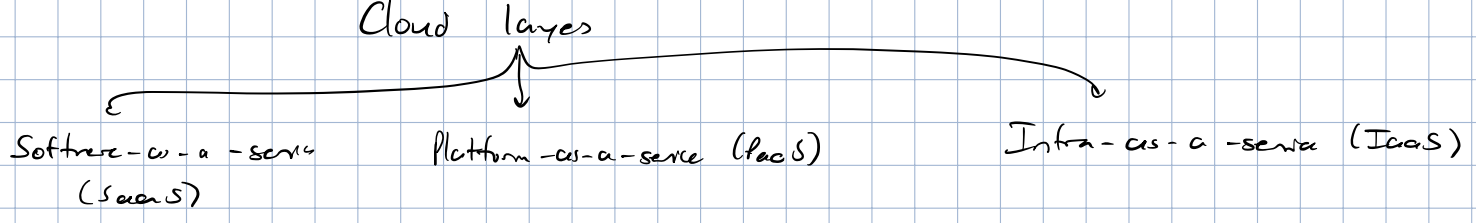
Cloud / REST Architectures

Characteristics of cloud services:

1. On-demand self-service
2. Broad network access
3. Resource pooling
4. Rapid elasticity
5. Measured service

Benefits of cloud:

1. Agility
2. Scalability
3. Cost
4. Reliability
5. Security



Serverless compute: cloud provider acts as server, dynamically managing resource allo.

↳ Closely related to function-as-a-service

REST: representational state transfer

↳ Ops: GET, POST, PUT, DELETE

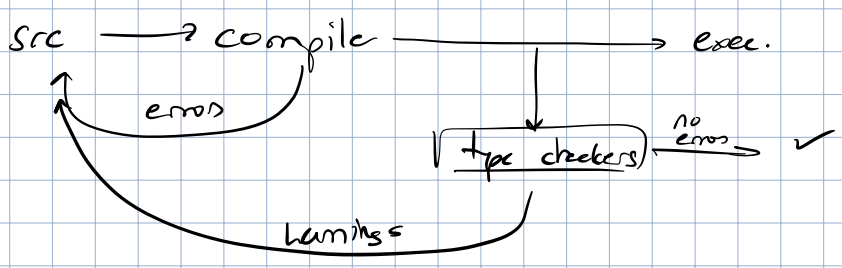
Nullness & Optionality

Plusgable type checking:

① Design type system

② Write type qualifiers or use type inferencer

③ Type checker warns about violations



Pros of type systems:

① Reduce bugs

② Improve documentation by improving code structure

③ Aid compilers + optimizers

Cons:

① Extra effort

② False +ves possible

Example: Checker in Java (@NonNull (default), @Nullable)

- Can change types depending on ctrl. flow

```
@Nullable obj x;
...
if (x != null) {
    ... ~ @NonNull
}
```

- Side effect annotations

@SideEffectFree	void foo() {...}	⇒ no side effects
@Deterministic	"	⇒ same args, same result
@Pure	"	⇒ No side effects & deterministic
@MonotonicNonNull	"	⇒ Could return null & non-null, but will not change non-null → null

- Preconditions & post-conditions
- Polymorphism (same fnc → compiles to null version & non-null version)

Type checker guarantees no bugs of certain variety & no many annotations

↳ Caveat: only for annotated code & checker could have bugs

Defining custom type system requires:

① Type hierarchy (subtypes)

② Type rules

③ Type introduction

④ Data flow

CODE QUALITY & PLs

Recall SOLID design principles

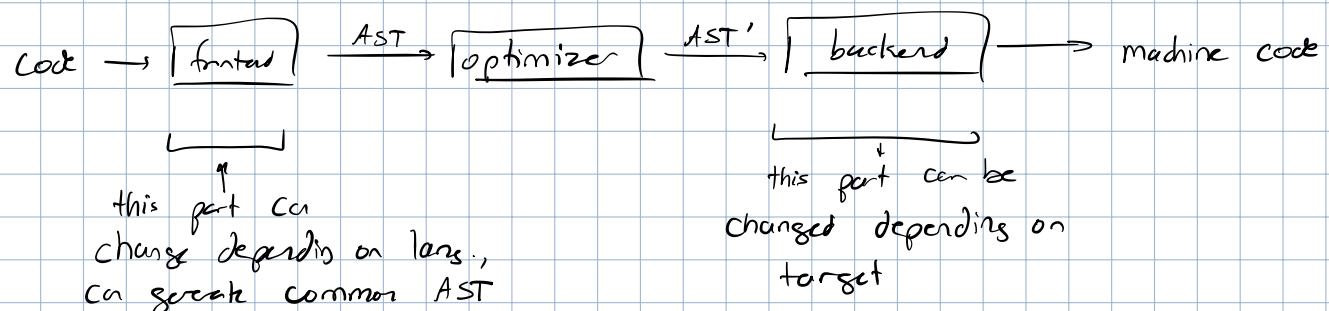
Can use bug finders

Different programming languages for diff. purposes

LLVM

LLVM has an architecture that emphasizes modularity & reusability, not common in OSS at the time

Most compilers structured like:



Advantages:

① Plug & play for diff langs & targets

② Serve broader set of programmers b/c lang. / platform agnostic

③ Separate skills to work on diff. parts of arch.

Couldn't fully realize this architecture except:

① Java/.NET compilers: forced Java paradigm for lang. tho (eg. GC, object model)

② Intrepreter patten: mdu lang interpat to C, then compile. Not efficient

③ GCC

All of theser wer designed in a monolith, so hard to break down

LLVM represents code in intermedick rep. IR which can be piper into optimizer

Frontend for LLVM teles code → IR. This is a standard interface (novel)

↳ Fronter devs only need to know IR, no complex free logic

LLVM is collection of libraries → great for re-use & reducing unnecessary code

Code generator is also coll. of libraries → mix + match

Cool capabilities:

① LLVM can be serialized → do parts of compilation + save for late (linking/installing)

② Testing is far easier