CEN 5016: Software Engineering

Spring 2024

University of Central Florida

Dr. Kevin Moran

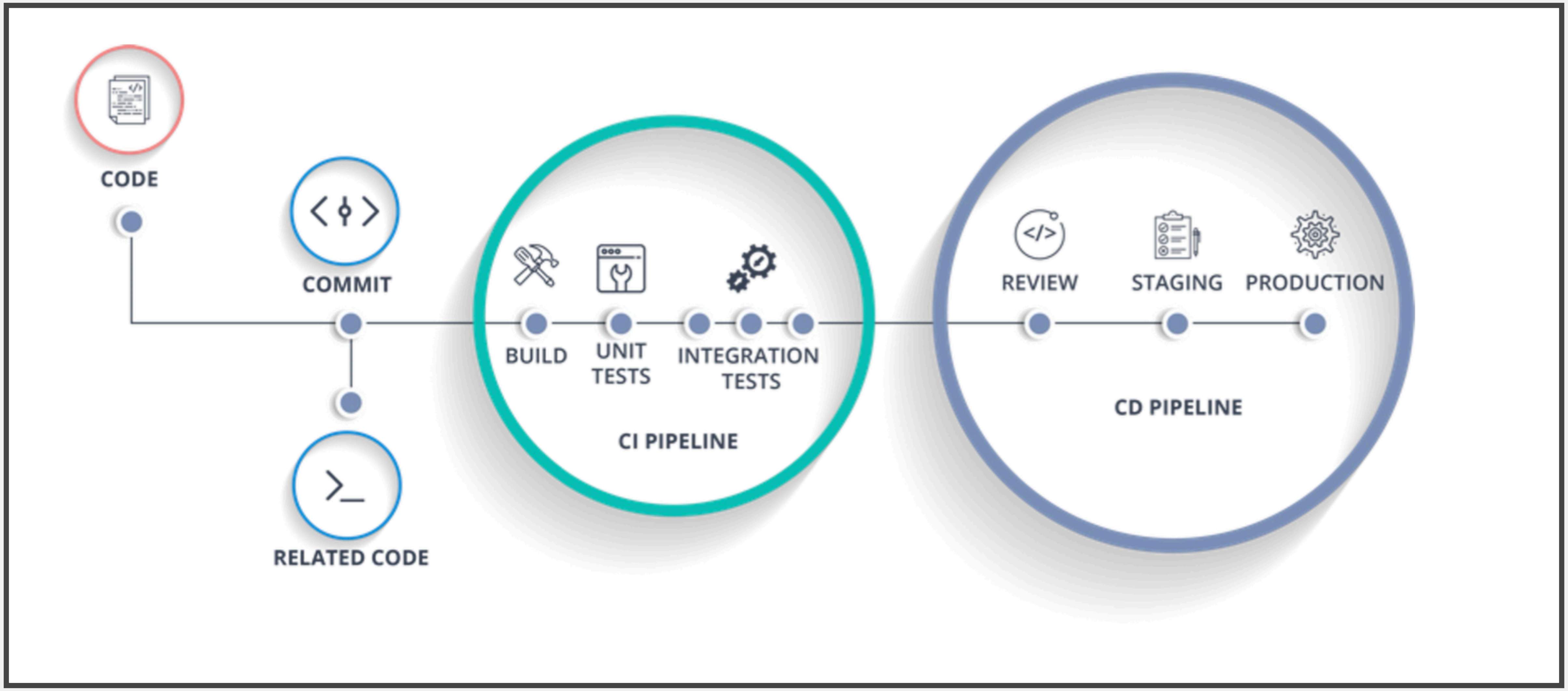
Week 4- Class 1: Introduction to Software Architecture

Administrivia

- *Assignment 2 Due Today*
- *Assignment 3 & SDE Project Part 1*
  - Will be posted this evening
  - Both will be due Tuesday February 6th
  - Plan ahead!

Software Testing

Continuous Integration & Deployment



How Good Are Our Tests?

Code Coverage

- Line coverage
  - Statement coverage
  - Branch coverage
  - Instruction coverage
  - Basic-block coverage
  - Edge coverage
  - Path coverage
  - ...

Code Coverage

LCOV - code coverage report

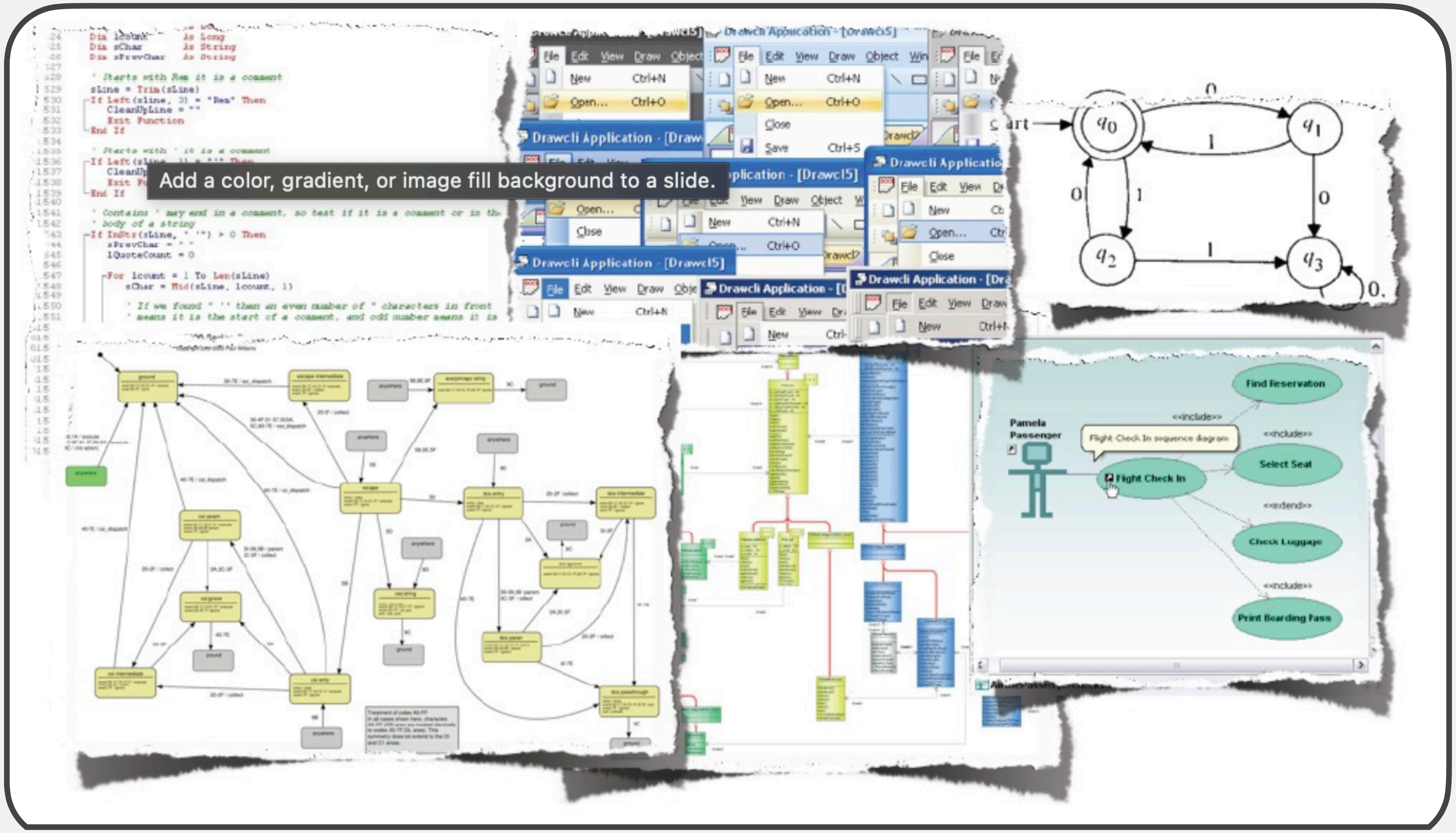
Current view: top level - test
Test: coverage.info
Date: 2018-02-07 13:06:43

	Hit	Total	Coverage
Lines:	6092	7293	83.5 %
Functions:	481	518	92.9 %

Filename	Line Coverage		Functions	
asn1_string_table_test.c	58.8 %	20 / 34	100.0 %	2 / 2
asn1_time_test.c	72.0 %	72 / 100	100.0 %	7 / 7
bad_dtls_test.c	97.6 %	163 / 167	100.0 %	9 / 9
bftest.c	65.3 %	64 / 98	87.5 %	7 / 8
bio_enc_test.c	78.7 %	74 / 94	100.0 %	9 / 9
bntest.c	97.7 %	1038 / 1062	100.0 %	45 / 45
chacha_internal_test.c	83.3 %	10 / 12	100.0 %	2 / 2
ciphername_test.c	60.4 %	32 / 53	100.0 %	2 / 2
crltest.c	100.0 %	90 / 90	100.0 %	12 / 12
ct_test.c	95.5 %	212 / 222	100.0 %	20 / 20
d2i_test.c	72.9 %	35 / 48	100.0 %	2 / 2
danetest.c	75.5 %	123 / 163	100.0 %	10 / 10
dhtest.c	84.6 %	88 / 104	100.0 %	4 / 4
drbgtest.c	69.8 %	157 / 225	92.9 %	13 / 14
dtls_mtu_test.c	86.8 %	59 / 68	100.0 %	5 / 5
dtlstest.c	97.1 %	34 / 35	100.0 %	4 / 4
dtlsv1listentest.c	94.9 %	37 / 39	100.0 %	4 / 4
ecdsatest.c	94.0 %	140 / 149	100.0 %	7 / 7
enginetest.c	92.8 %	141 / 152	100.0 %	7 / 7
evp_extra_test.c	100.0 %	112 / 112	100.0 %	10 / 10
fatalerrtest.c	89.3 %	25 / 28	100.0 %	2 / 2
handshake_helper.c	84.7 %	494 / 583	97.4 %	38 / 39
hmactest.c	100.0 %	71 / 71	100.0 %	7 / 7
ideatest.c	100.0 %	30 / 30	100.0 %	4 / 4
igetest.c	87.9 %	109 / 124	100.0 %	11 / 11
lhash_test.c	78.6 %	66 / 84	100.0 %	8 / 8
mdc2_internal_test.c	81.8 %	9 / 11	100.0 %	2 / 2
mdc2test.c	100.0 %	18 / 18	100.0 %	2 / 2
ocspapitest.c	95.5 %	64 / 67	100.0 %	4 / 4
packettest.c	100.0 %	248 / 248	100.0 %	24 / 24

```
    1  /    1:            if ((err = SSLHashMD5.final(&hashCtx, &hashOut)) != 0)
    0  /    1:                    goto fail;
           :     }
           :     else {
           :            /* DSA, ECDSA - just use the SHA1 hash */
    0  /    1:            dataToSign = &hashes[SSL_MD5_DIGEST_LEN];
    0  /    1:            dataToSignLen = SSL_SHA1_DIGEST_LEN;
           :     }
           :
    1  /    1:     hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
    1  /    1:      hashOut.length = SSL_SHA1_DIGEST_LEN;
    1  /    1:      if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    0  /    1:          goto fail;
           :
    1  /    1:      if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    0  /    1:          goto fail;
    1  /    1:      if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    0  /    1:          goto fail;
    1  /    1:      if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    0  /    1:          goto fail;
    1  /    1:      if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    0  /    1:          goto fail;
    1  /    1:          goto fail;
           :      if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
           :          goto fail;
           :
           :     err = sslRawVerify(ctx,
           :                        ctx->peerPubKey,
           :                        dataToSign,                      /* plaintext *,
           :                        dataToSignLen,                   /* plaintext le
           :                        signature,
           :                        signatureLen);
           :     if(err) {
           :            sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
           :                   "returned %d\n", (int)err);
           :            goto fail;
           :     }
           :
           : fail:
    1  /    1:      SSLFreeBuffer(&signedHashes);
    1  /    1:      SSLFreeBuffer(&hashCtx);
    1  /    1:      return err;
           :
    1  /    1: }
           :
```

We Can Measure Coverage on Almost Anything



Be Aware of Coverage Chasing

- Recall: issues with metrics and incentives
  - Also: Numbers can be deceptive

- 100% coverage != exhaustively tested
  - "Coverage is not strongly correlated with suite effectiveness"

- Based on empirical study on GitHub projects [Inozemtseva and Holmes, ICSE'14]

- Still, it's a good low bar
  - Code that is not executed has definitely not been tested

Coverage of What?

- Distinguish code being tested and code being executed
- Library code >>>>> Application code
  - Can selectively measure coverage
- All application code >>> code being tested
  - Not always easy to do this within an application

Coverage != Outcome

- What's better, tests that always pass or tests that always fail?
- Tests should ideally be falsifiable. Boundary determines
- specification
- Ideally:
  - Correct implementations should pass all tests
  - Buggy code should fail at least one test
  - Intuition behind mutation testing (we'll revisit this next week)
- What if tests have bugs?
  - Pass on buggy code or fail on correct code
- Even worse: flaky tests
  - Pass or fail on the same test case nondeterministically
- What's the worst type of test?

Test Design Principles

- Use public APIs only
- Clearly distinguish inputs, configuration, execution, and oracle
- Be simple; avoid complex control flow such as conditionals and loops
- Tests shouldn't need to be frequently changed or refactored
  - Definitely not as frequently as the code being tested changes

Anti-Patterns

- Snoopy oracles
  - Relying on implementation state instead of observable behavior
  - E.g. Checking variables or fields instead of return values
- Brittle tests
  - Overfitting to special-case behavior instead of general principle
  - E.g. hard-coding message strings instead of behavior
- Slow tests
  - Self-explanatory(beware of heavy environments, I/O, and sleep())
- Flaky tests
  - Tests that pass or fail nondeterministically
  - Often because of reliance on random inputs, timing (e.g. sleep(1000)), availability of external services (e.g. fetching data over the network in a unit test), or dependency on order of test execution (e.g. previous test sets up global variables in certain way)

Takeaways

- Most tests that you will write will be muuuuuuch more complex than testing a sort function.
- Need to set up environment, create objects whose methods to test, create objects for test data, get all these into an interesting state, test multiple APIs with varying arguments, etc.
- Many tests will require mocks (i.e., faking a resource-intensive component).
- General principles of many of these strategies still apply:
  - Writing tests can be time consuming
  - Determining test adequacy can be hard (if not impossible)
  - Test oracles are not easy
  - Advanced test strategies have trade-offs (high costs with high returns)

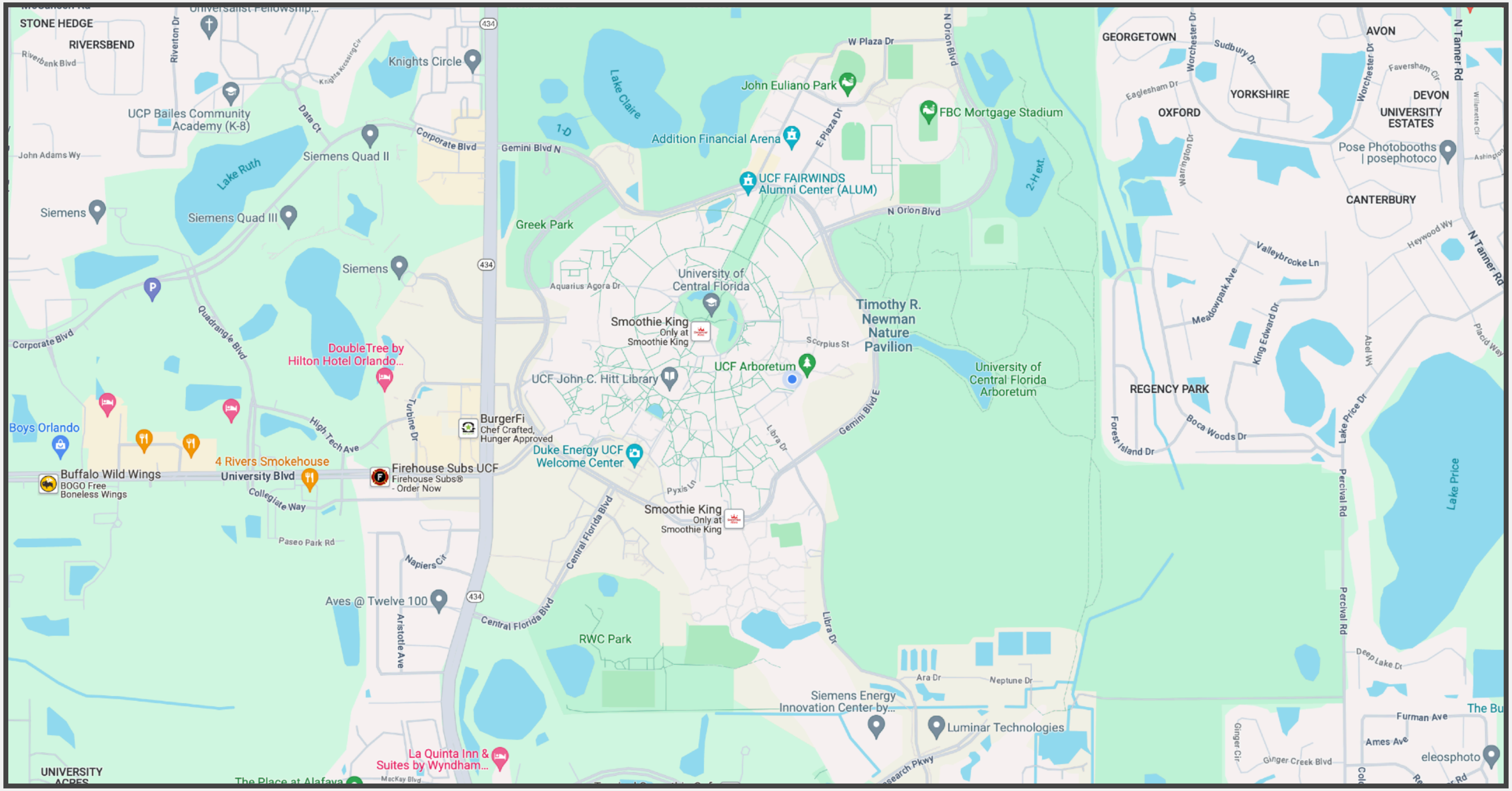
Intro to Software Architecture

Learning Goals

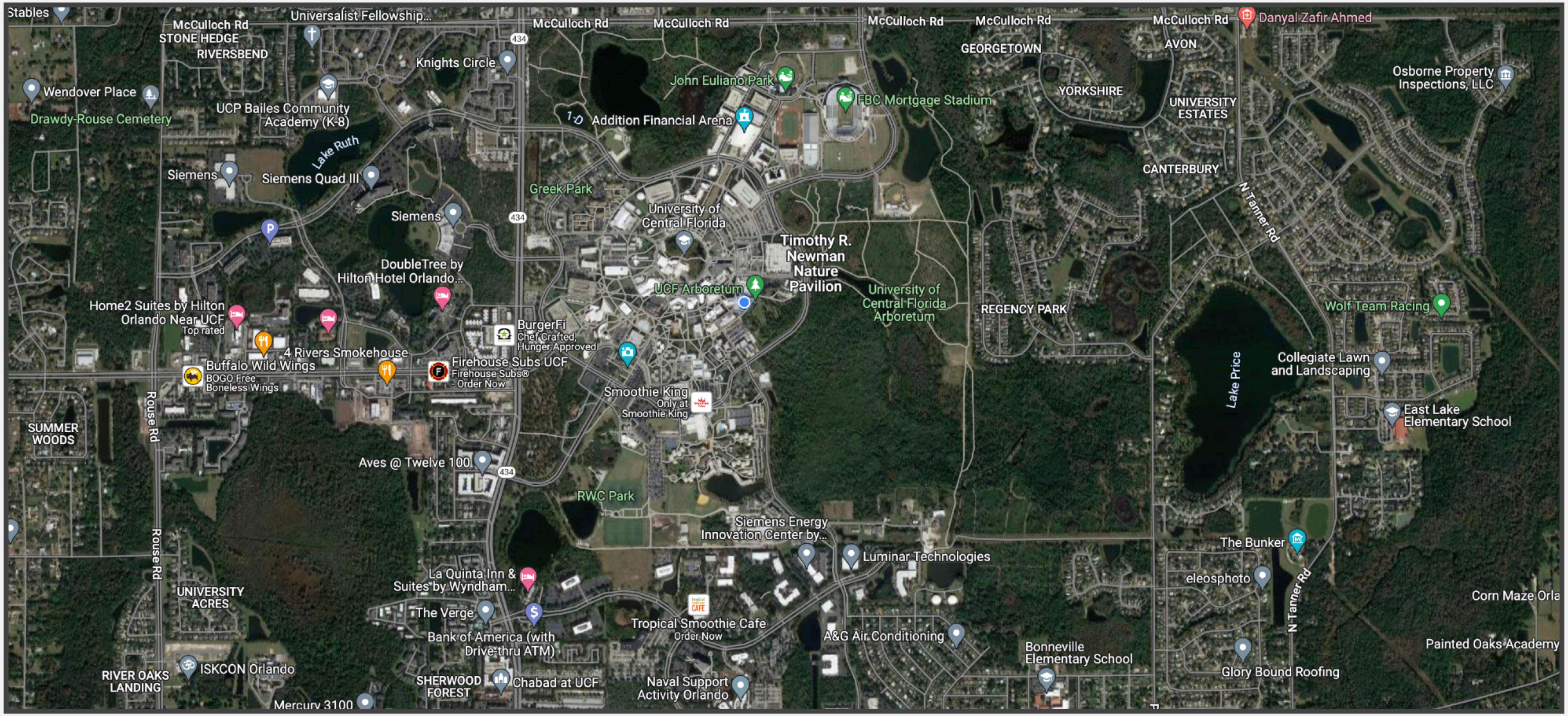
- Understand the abstraction level of architectural reasoning
- Appreciate how software systems can be viewed at different abstraction levels
- Distinguish software architecture from (object-oriented) software design
- Use notation and views to describe the architecture suitable to the purpose
- Document architectures clearly, without ambiguity

Views and Abstraction

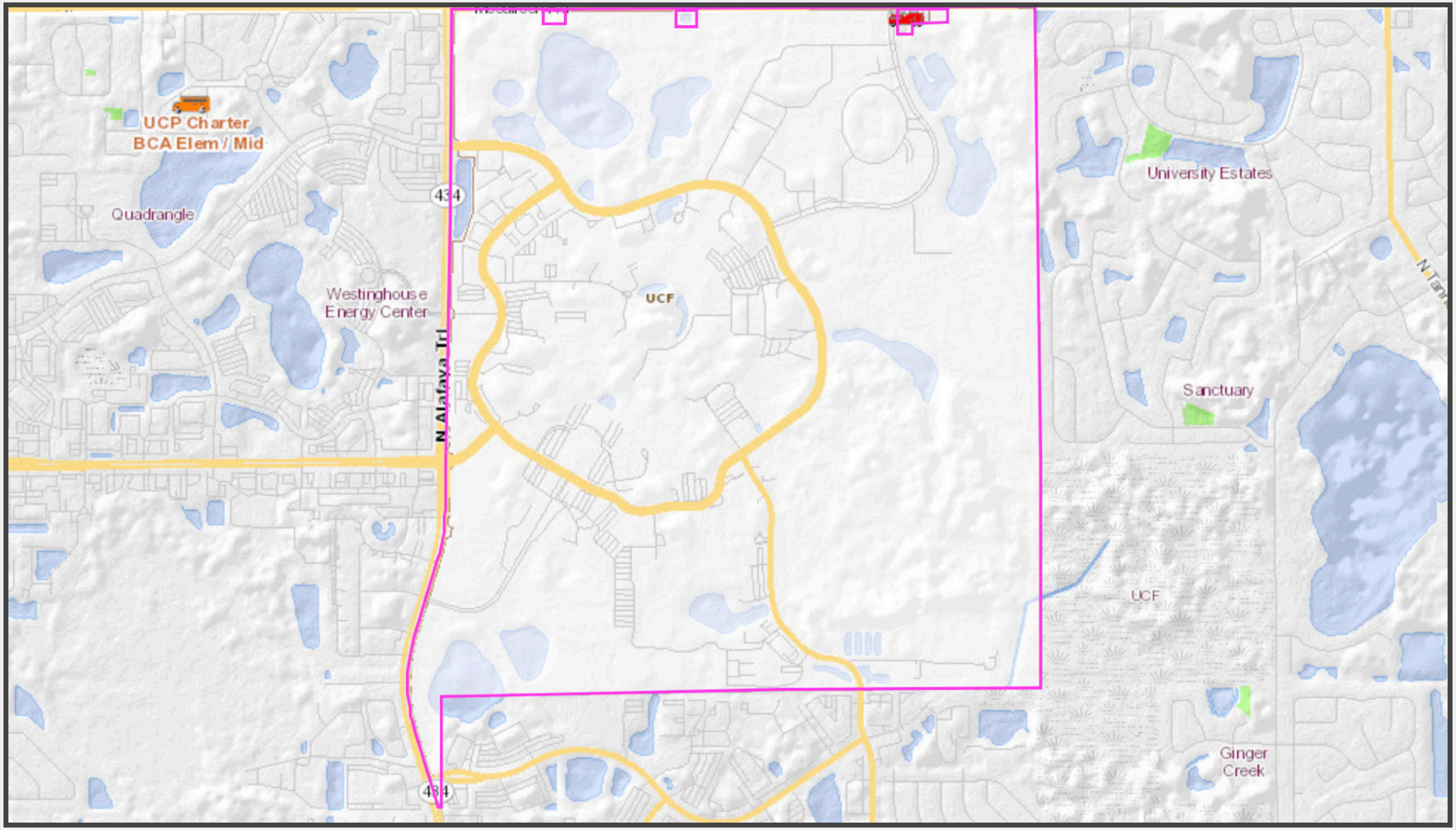
Views & Abstraction



Views & Abstraction



Views & Abstraction



Abstracted Views Focus on Conveying Information

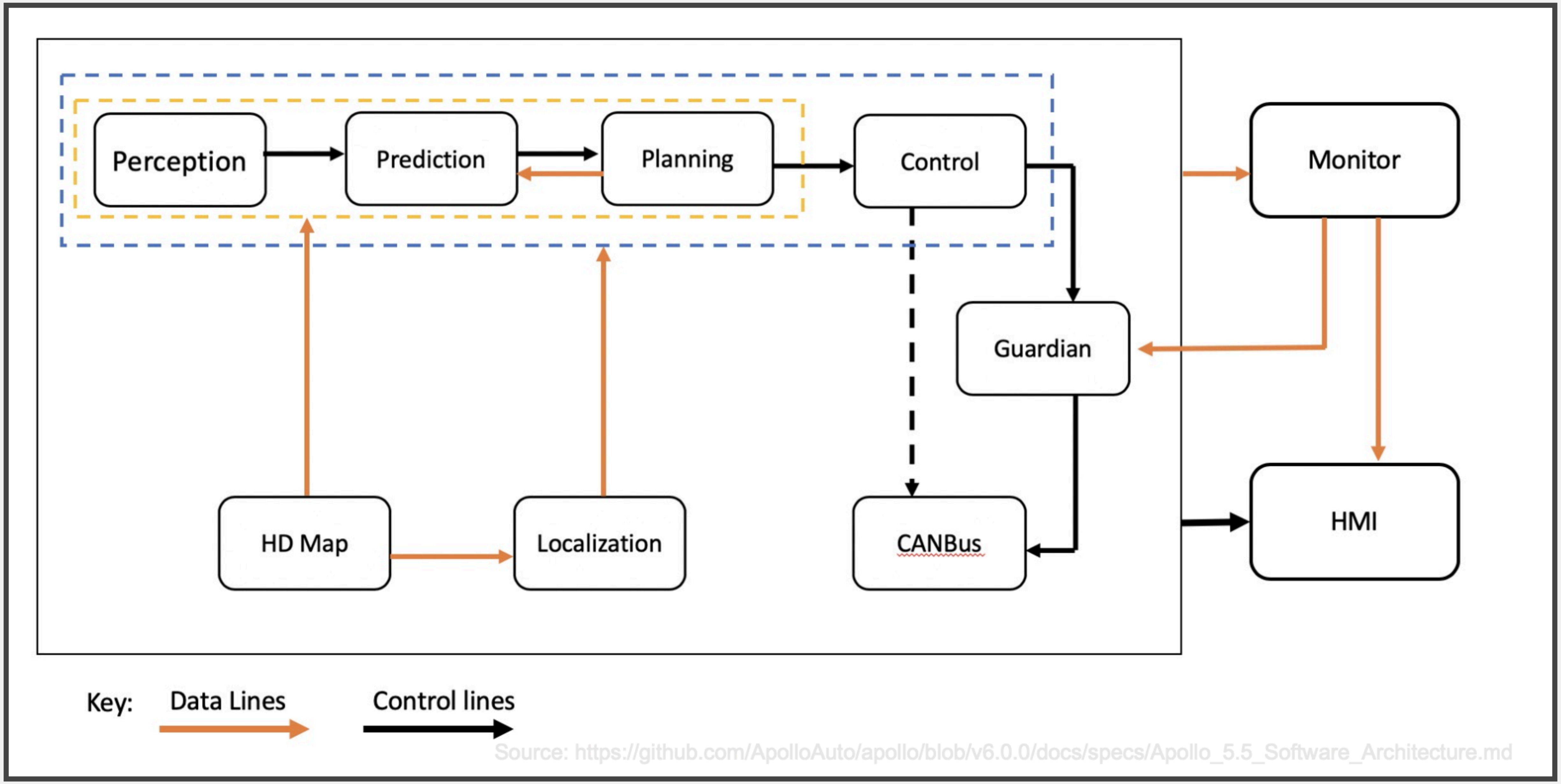
- They have a well-defined purpose
- Show only necessary information
- Abstract away unnecessary details
- Use legends/annotations to remove ambiguity
- Multiple views of the same object tell a larger story

Software Architecture Case Study: Autonomous Vehicles

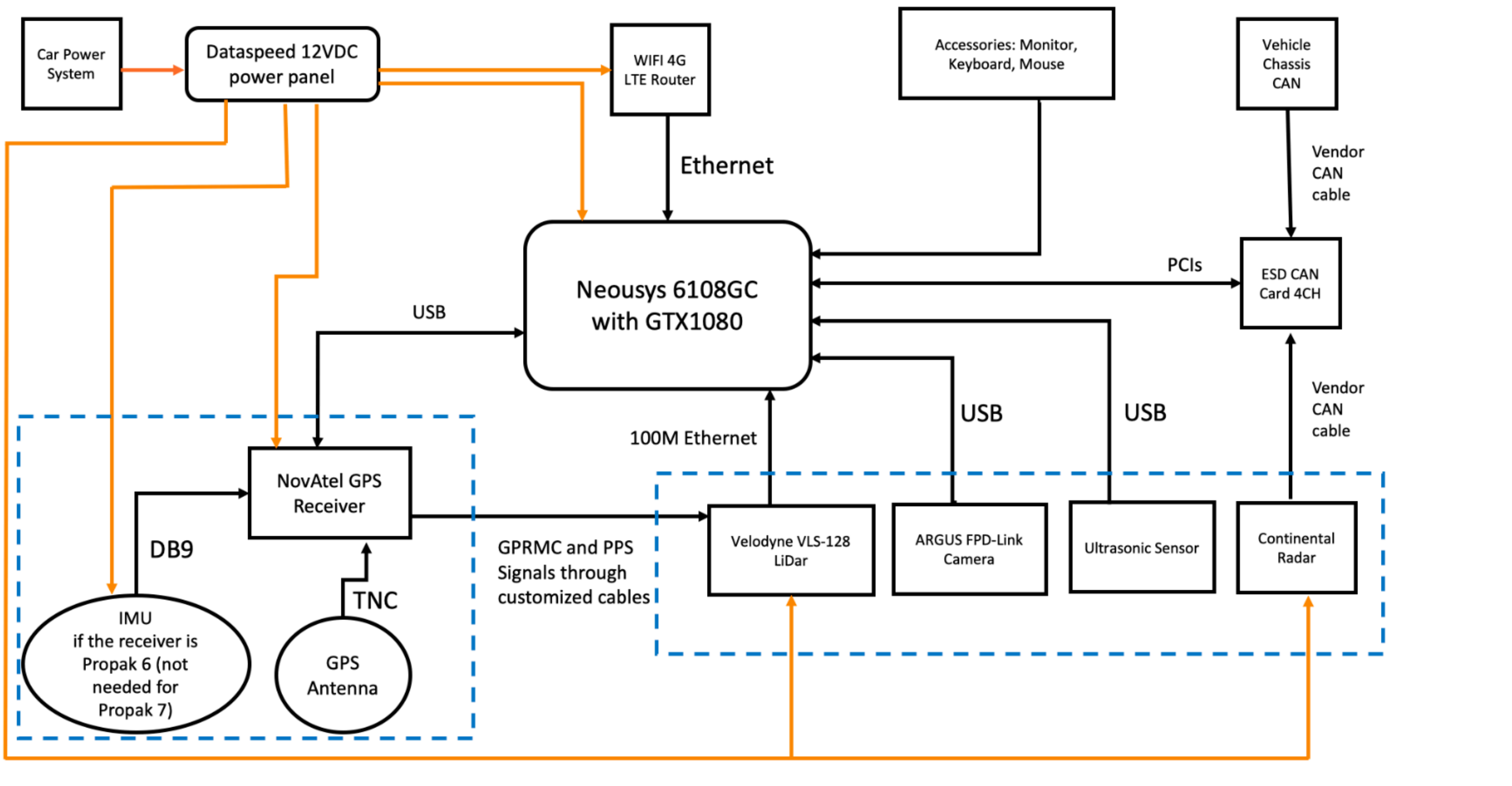
Case Study: Apollo

- Check out the "side pass" feature from the video:
  - http://tinyurl.com/cen24-vid
- **Source:** https://github.com/ApolloAuto/apollo
- **Doxygen:** https://hidetoshi-furukawa.github.io/apollo/doxygen/index.html

Apollo Software Architecture

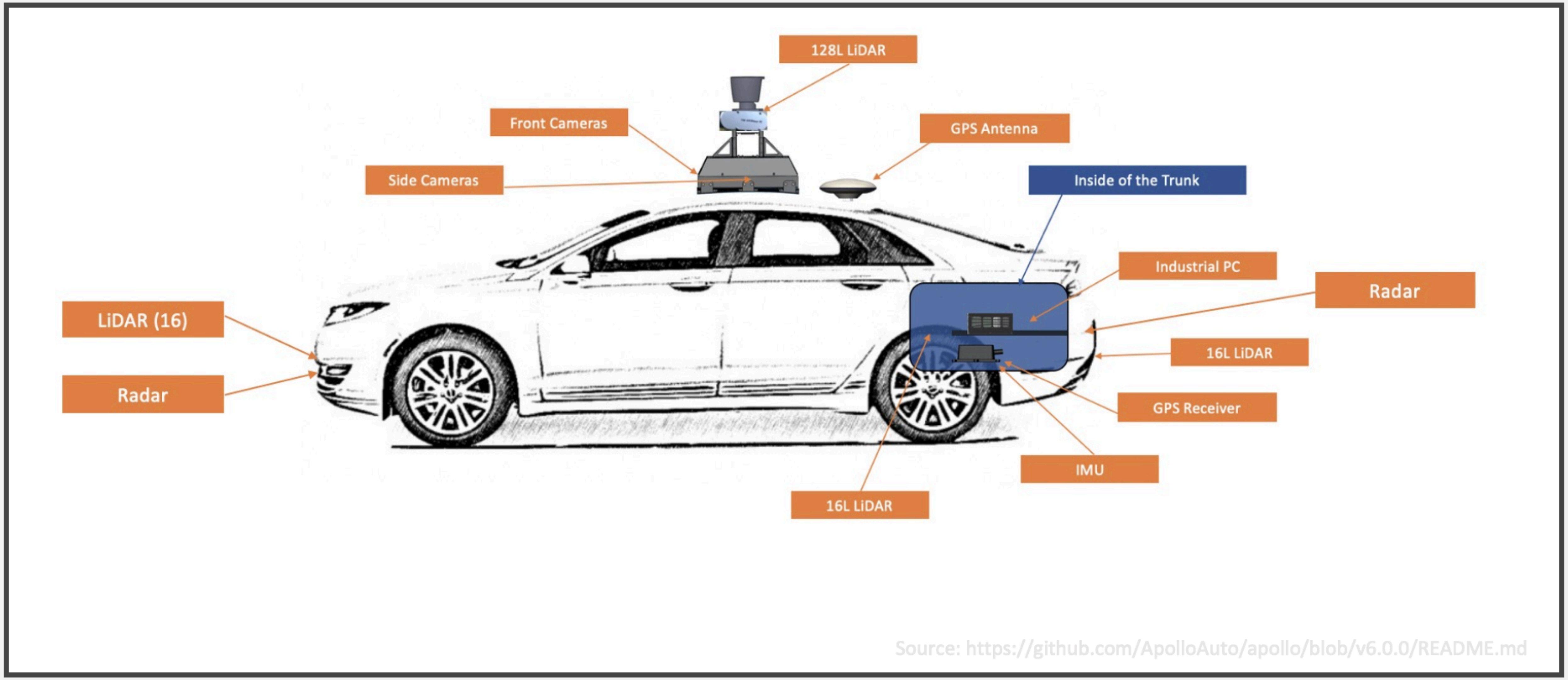


Apollo Hardware Architecture

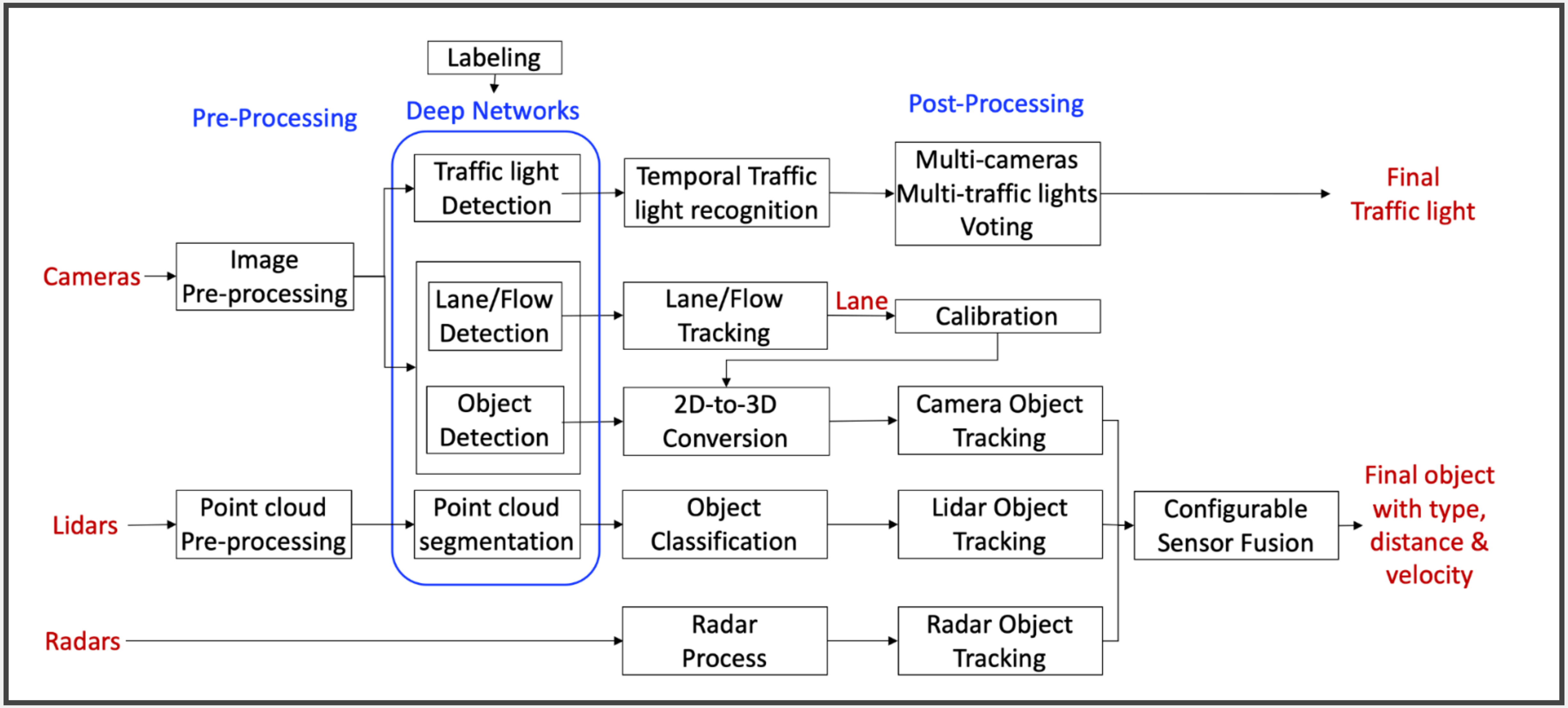


Source: https://github.com/ApolloAuto/apollo/blob/v6.0.0/README.md

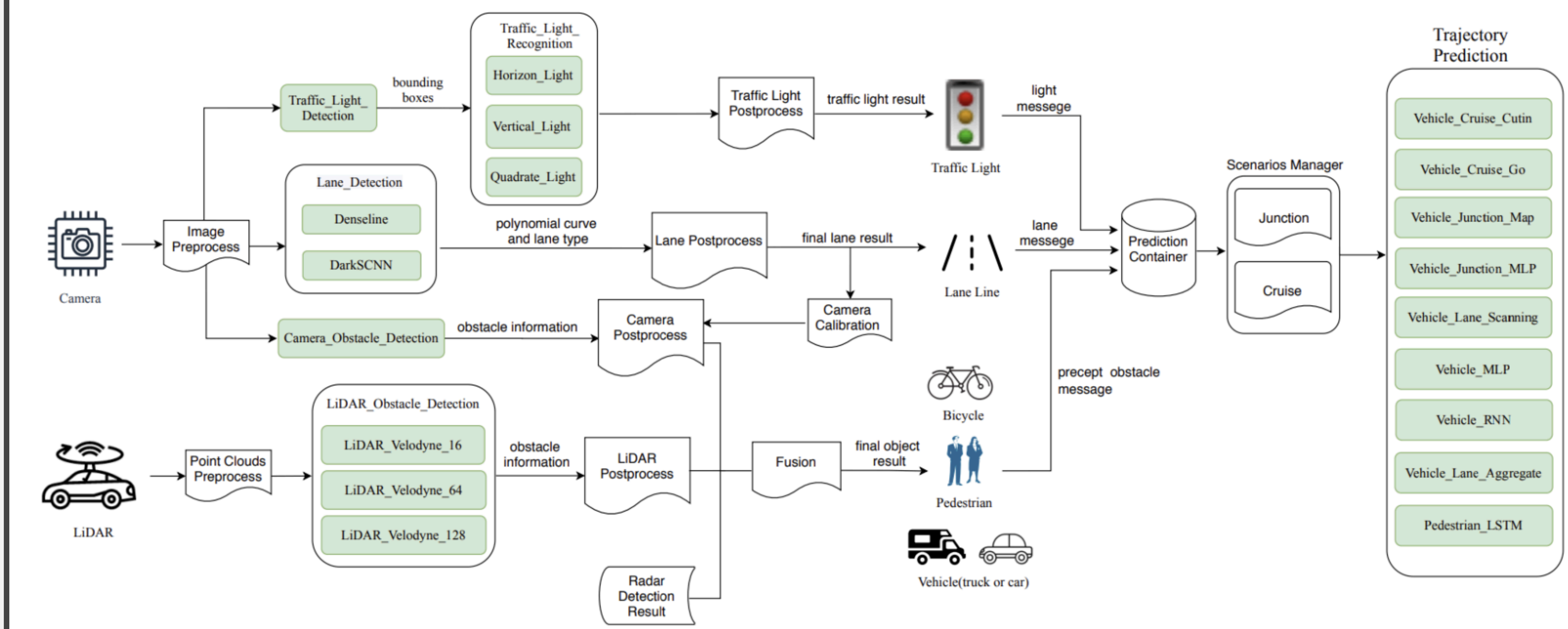
Apollo Hardware/Vehicle Overview



Apollo Perception Module



Apollo ML Models



Source: Zi Peng, Jinqiu Yang, Tse-Hsun (Peter) Chen, and Lei Ma. 2020. A First Look at the Integration of Machine Learning Models in Complex Autonomous Driving Systems: A Case Study on Apollo. In Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), https://doi.org/10.1145/ 3368089.3417063

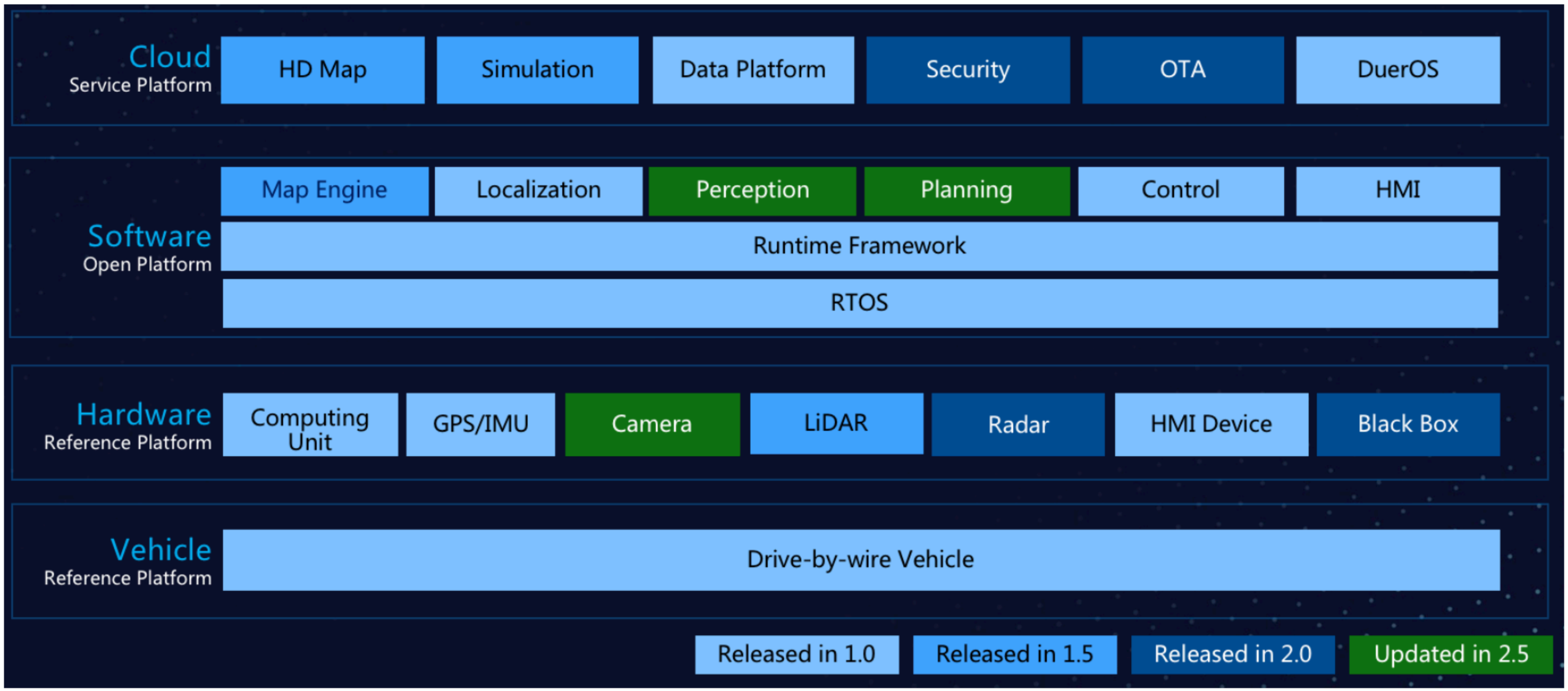
Apollo Software Stack

<table>
<tr><td>Cloud Service Platform</td><td>HD Map</td><td>Simulation</td><td>Data Platform</td><td>Security</td><td>OTA</td><td>DuerOS</td><td>Volume Production Service Components</td><td>V2X Roadside Service</td></tr>
<tr><td rowspan="3">Open Software Platform</td><td>Map Engine</td><td>Localization</td><td>Perception</td><td>Planning</td><td>Control</td><td>End-to-End</td><td>HMI</td><td rowspan="3">V2X Adapter</td></tr>
<tr><td colspan="7">Apollo Cyber RT Framework</td></tr>
<tr><td colspan="7">RTOS</td></tr>
<tr><td>Hardware Development Platform</td><td>Computing Unit</td><td>GPS/IMU</td><td>Camera</td><td>LiDAR</td><td>Radar</td><td>Ultrasonic Sensor</td><td>HMI Device</td><td>Black Box</td><td>Apollo Sensor Unit</td><td>Apollo Extension Unit</td><td>V2X OBU</td></tr>
<tr><td>Open Vehicle Certificate Platform</td><td colspan="6">Certified Apollo Compatible Drive-by-wire Vehicle</td><td colspan="2">Open Vehicle Interface Standard</td></tr>
</table>

Major Updates in Apollo 3.5

Source: https://github.com/ApolloAuto/

Feature Evolution (Software Stack View)



Source: https://github.com/ApolloAuto/apollo

Software Architecture

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.
[Bass et al. 2003]

Note: this definition is ambivalent to whether the architecture is known, or whether it's any good!

Software Design vs. Architecture

Levels of Abstraction

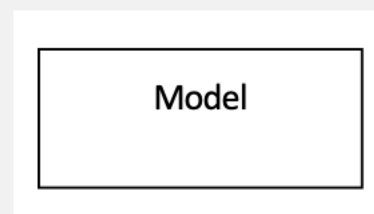
- Requirements
  - high-level “what” needs to be done
- Architecture (High-level design)
  - high-level “how”, mid-level “what”
- OO-Design (Low-level design, e.g. design patterns)
  - mid-level “how”, low-level “what”
- Code
  - low-level “how”

Design vs. Architecture

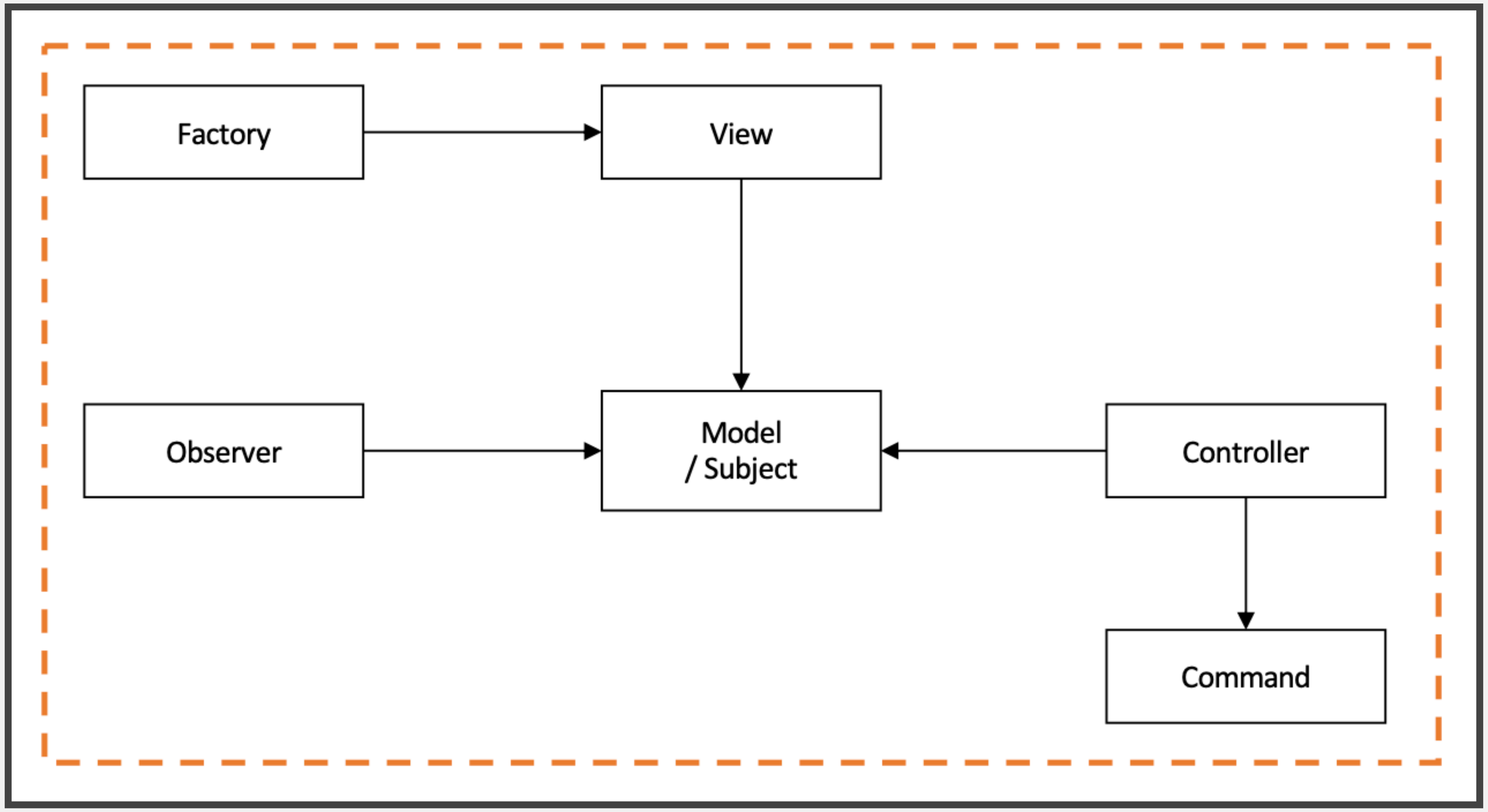
- Design Questions
  - *How do I add a menu item in VSCode?*
  - *How can I make it easy to add menu items in VSCode?*
  - *What lock protects this data?*
  - *How does Google rank pages?*
  - *What encoder should I use for secure communication?*
  - *What is the interface between objects?*

- Architectural Questions
  - *How do I extend VSCode with a plugin?*
  - *What threads exist and how do they coordinate?*
  - *How does Google scale to billions of hits per day?*
  - *Where should I put my firewalls?*
  - *What is the interface between subsystems?*

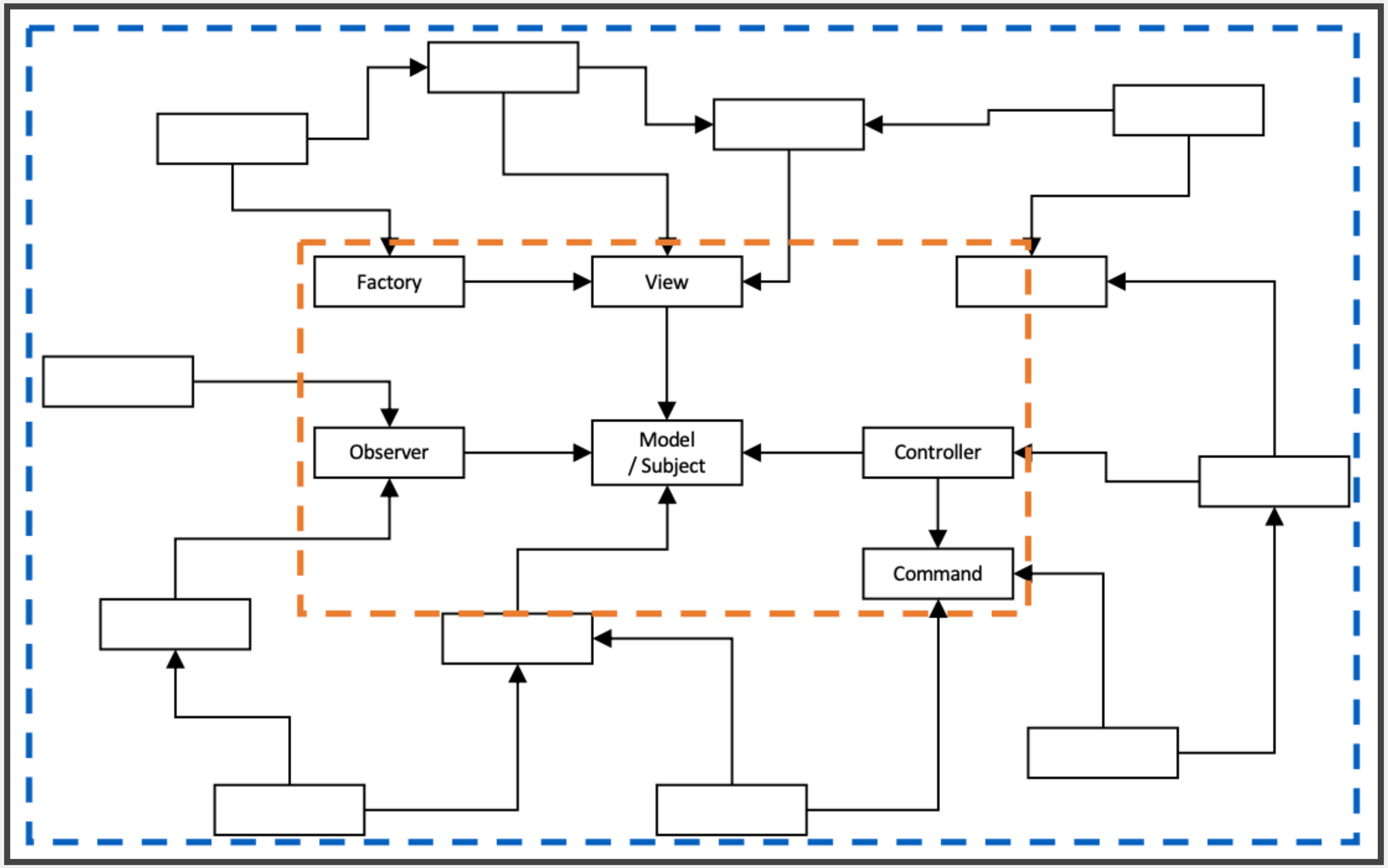
Objects



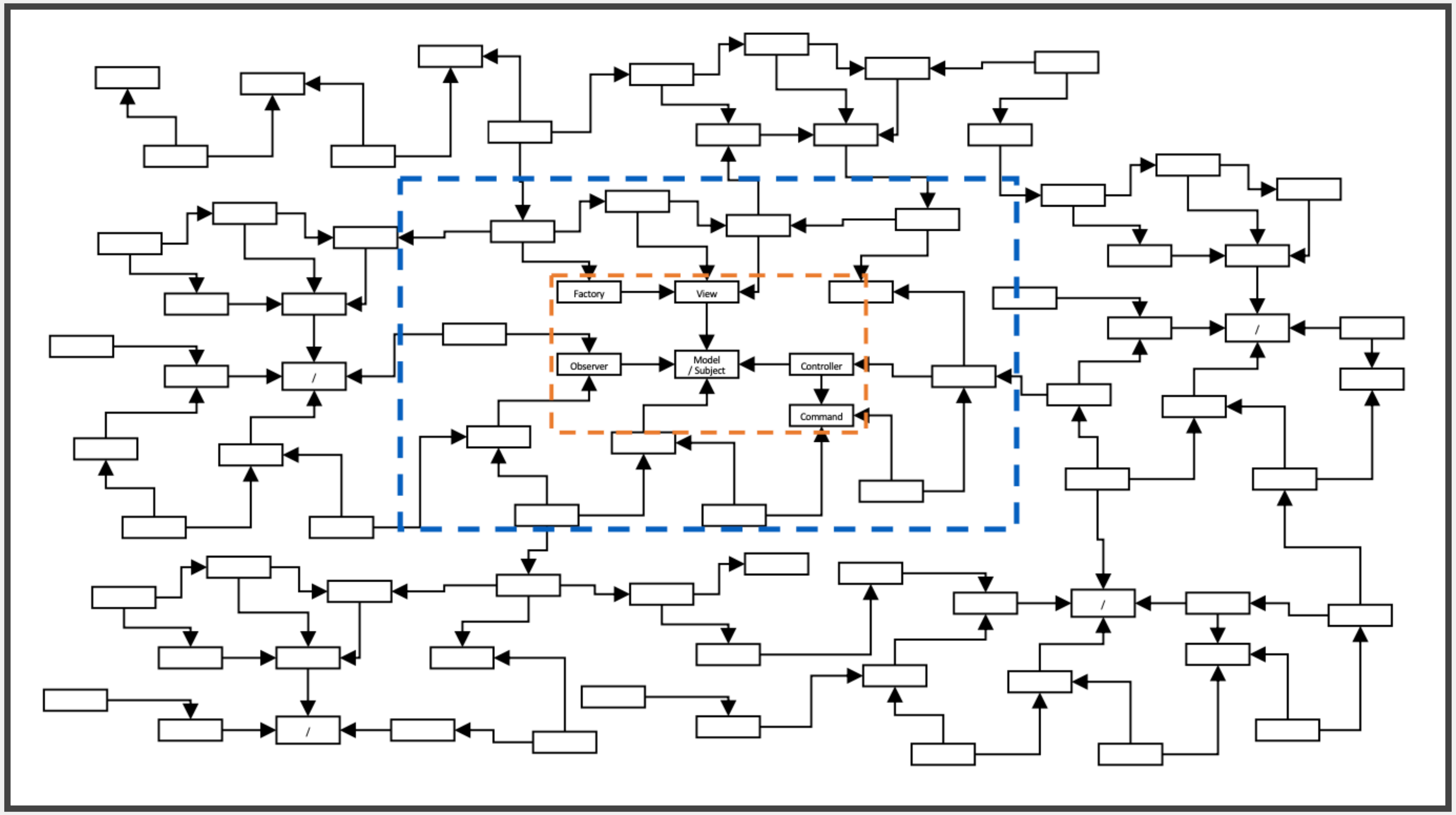
Design Patterns



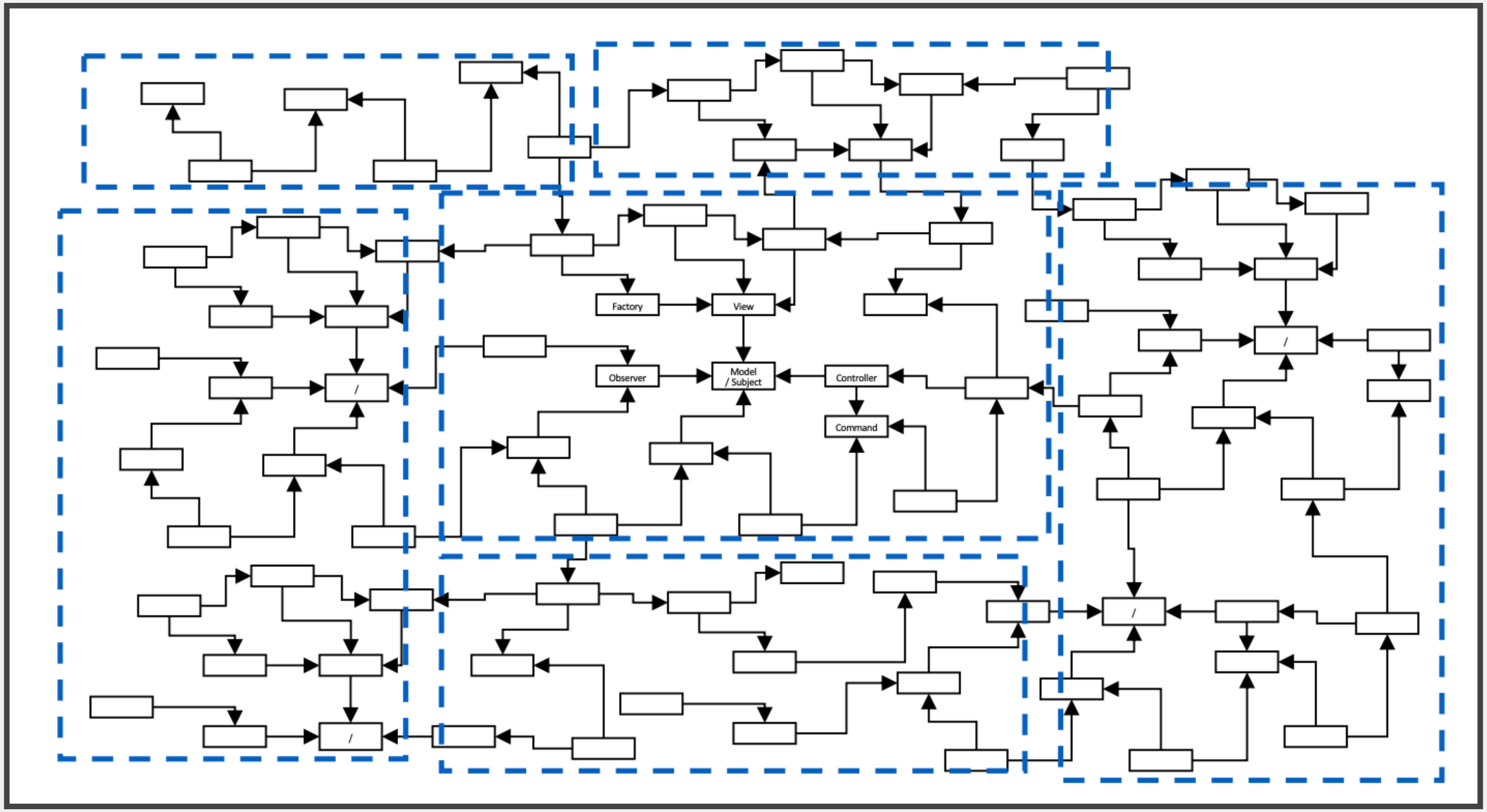
Design Patterns



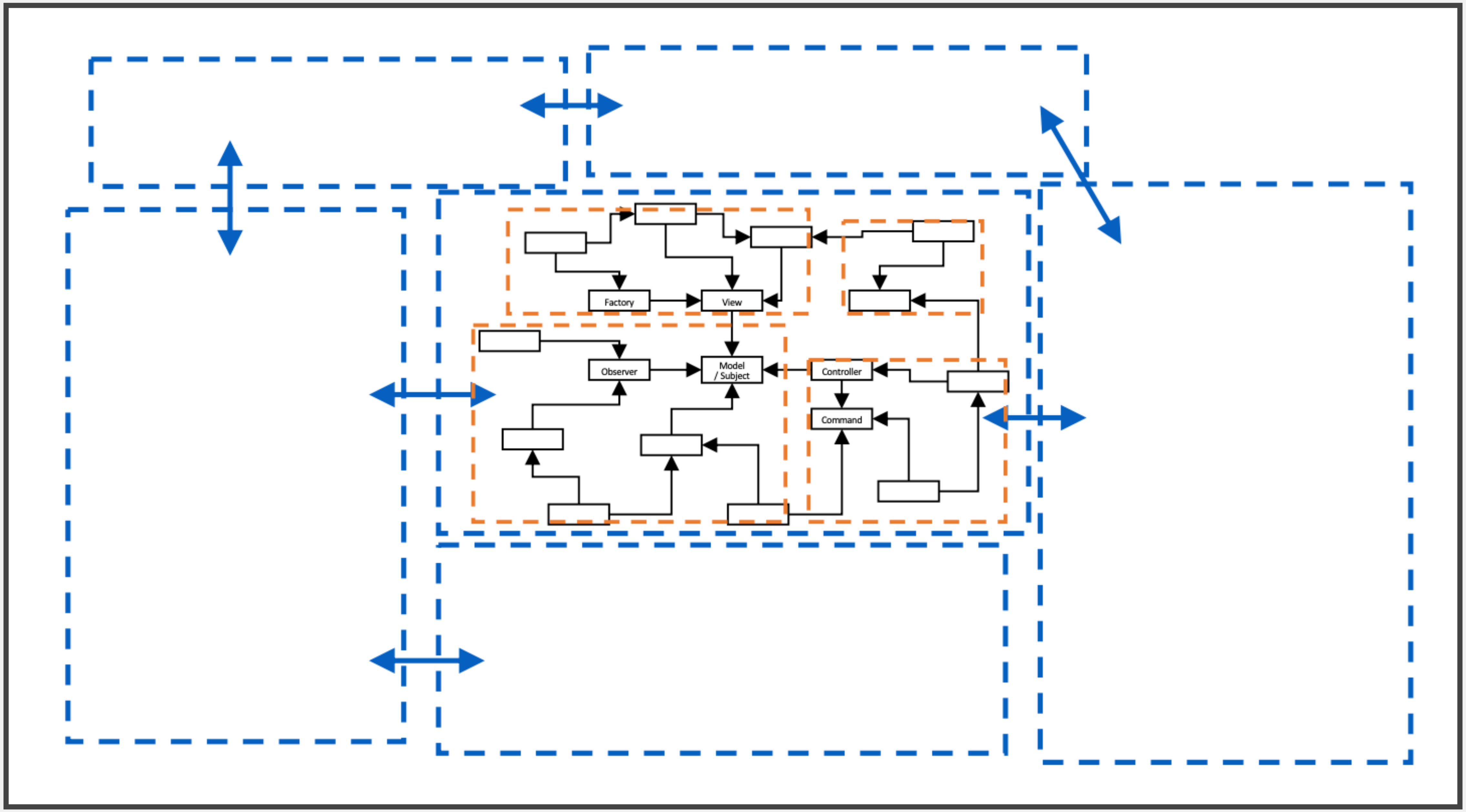
Design Patterns



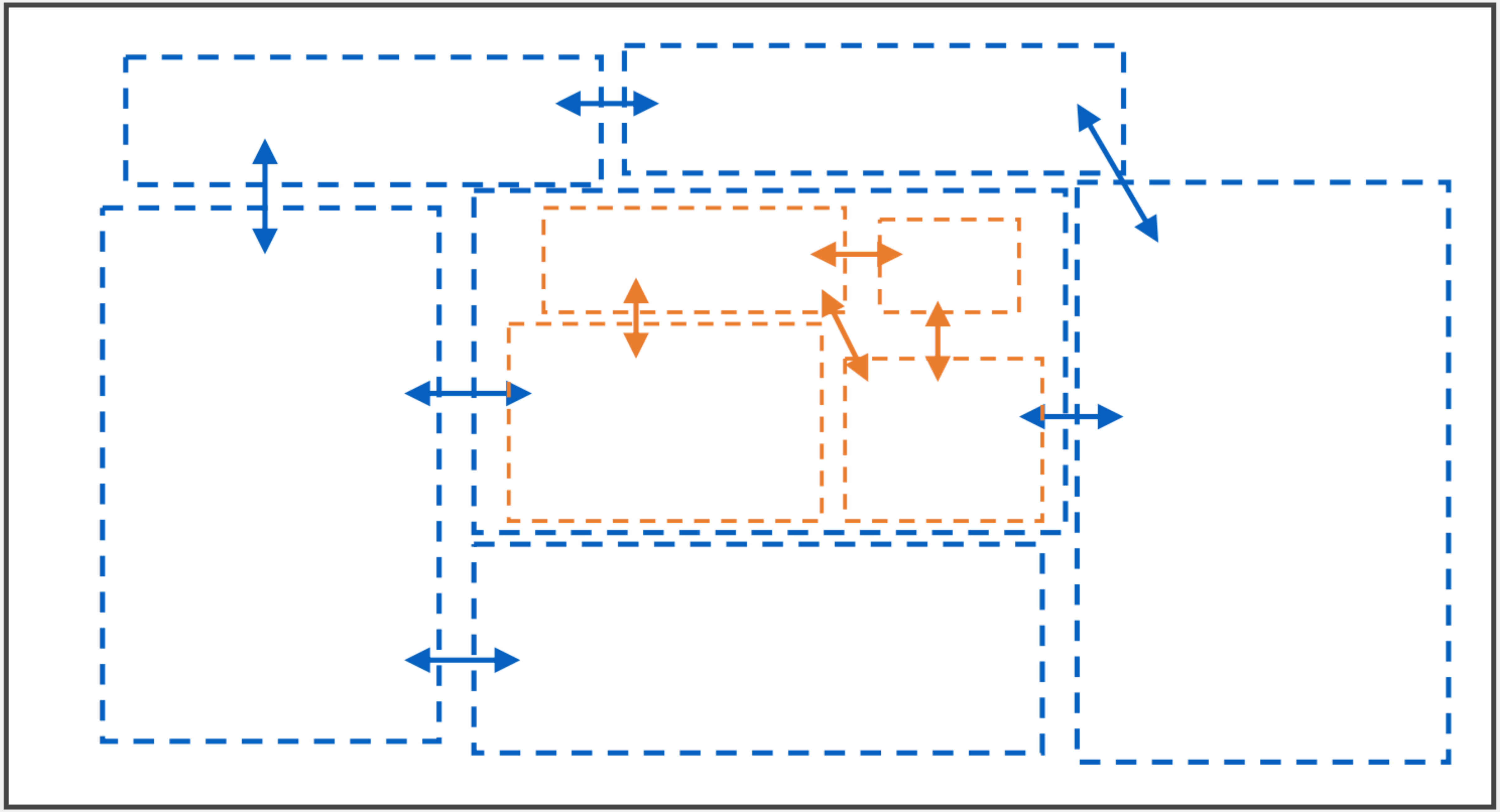
Architecture



Architecture



Architecture



Why Document Architecture?

- Blueprint for the system
  - Artifact for early analysis
  - Primary carrier of quality attributes
  - Key to post-deployment maintenance and enhancement
- Documentation speaks for the architect, today and 20 years from today
  - As long as the system is built, maintained, and evolved according to its documented architecture
- Support traceability.

Views & Purposes

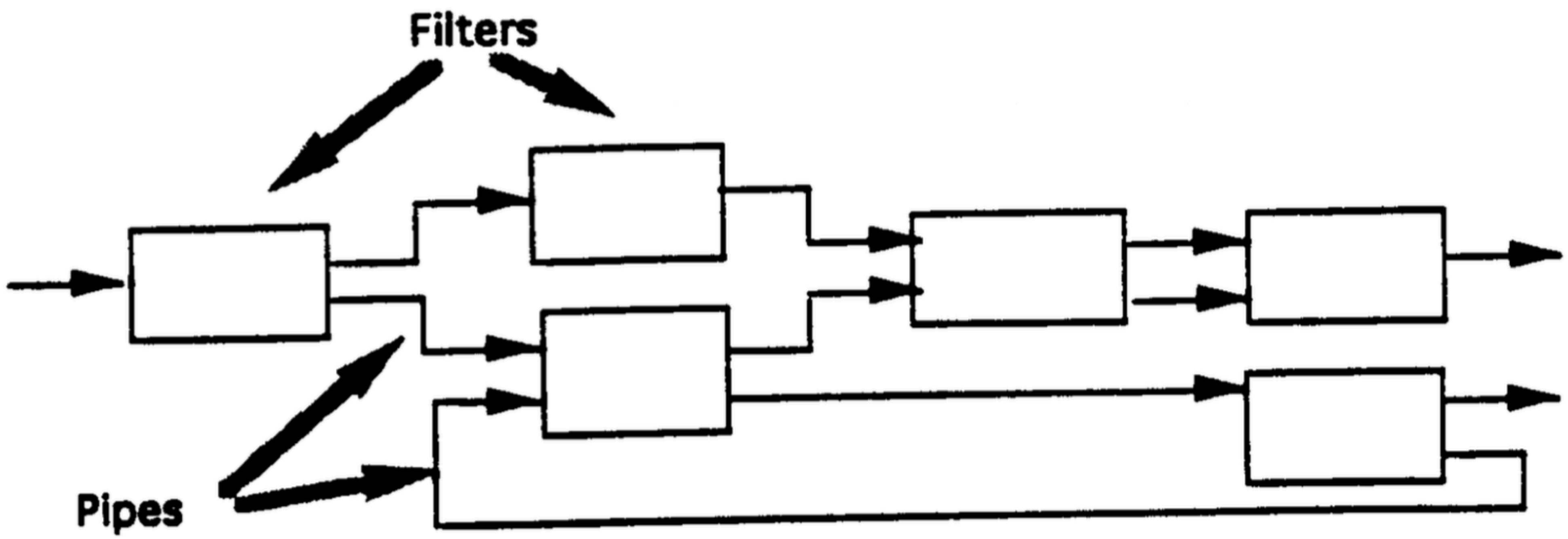
- Every view should align with a purpose
- • Views should only represent information relevant to that purpose
  - Abstract away other details
  - Annotate view to guide understanding where needed
- • Different views are suitable for different reasoning aspects (different quality goals), e.g.,
  - Performance
  - Extensibility
  - Security
  - Scalability
  - ...

Common Views in Documenting Software Architecture

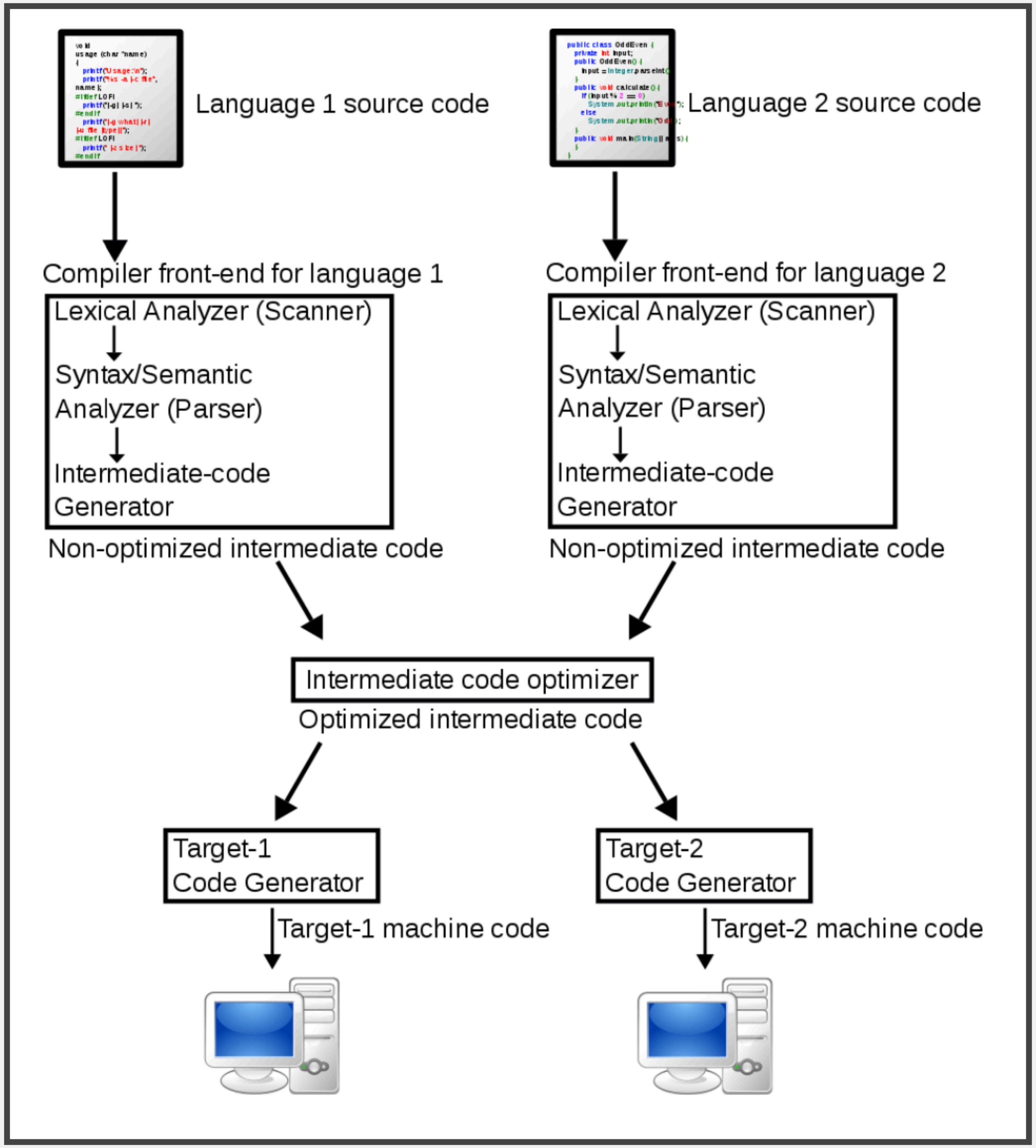
- Static View
  - Modules (subsystems, structures) and their relations (dependencies, ...)
- Dynamic View
  - Components (processes, runnable entities) and connectors (messages, data flow, ...)
- Physical View (Deployment)
  - Hardware structures and their connections

Common Software Architectures

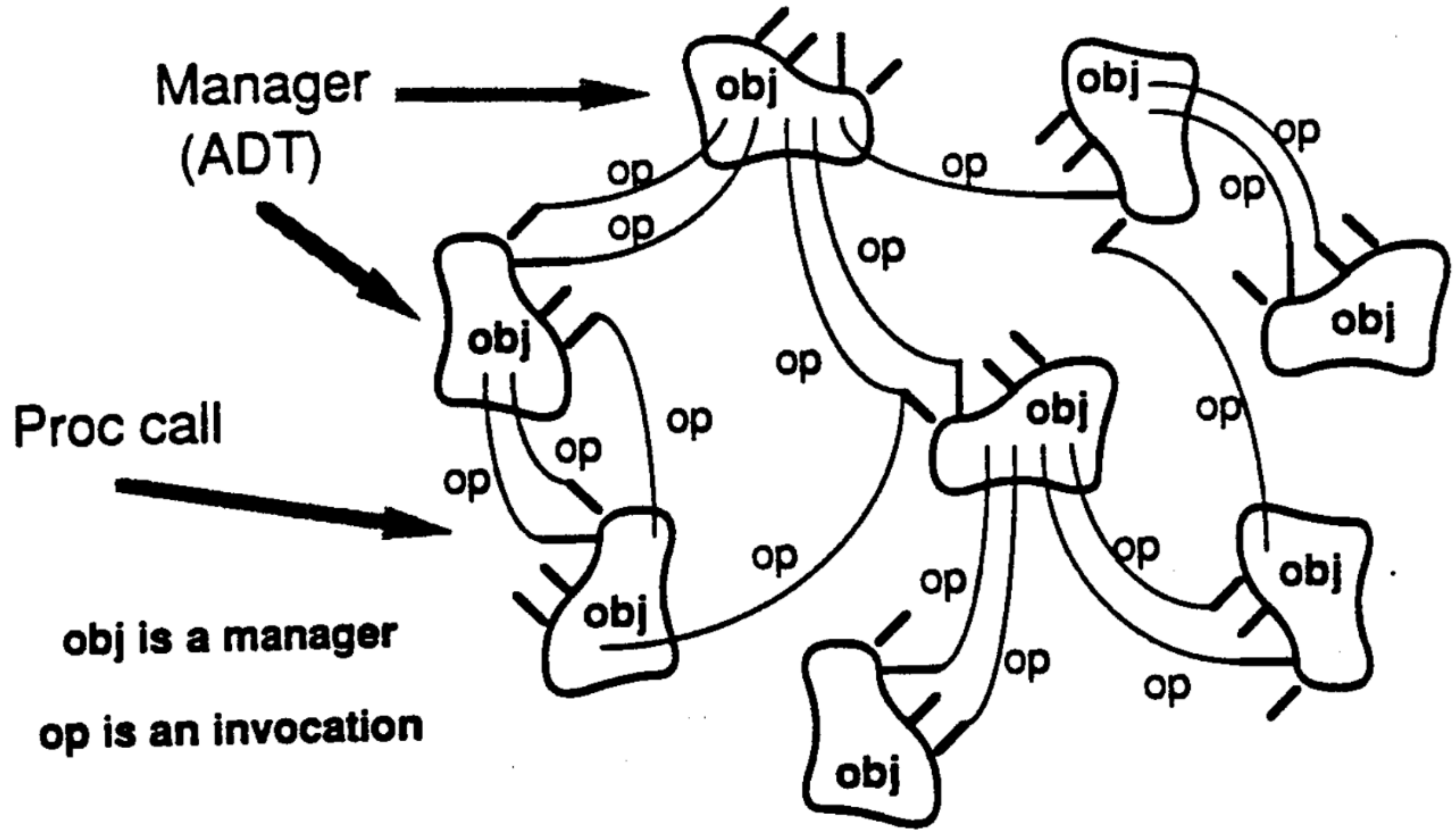
1. Pipes & Filters



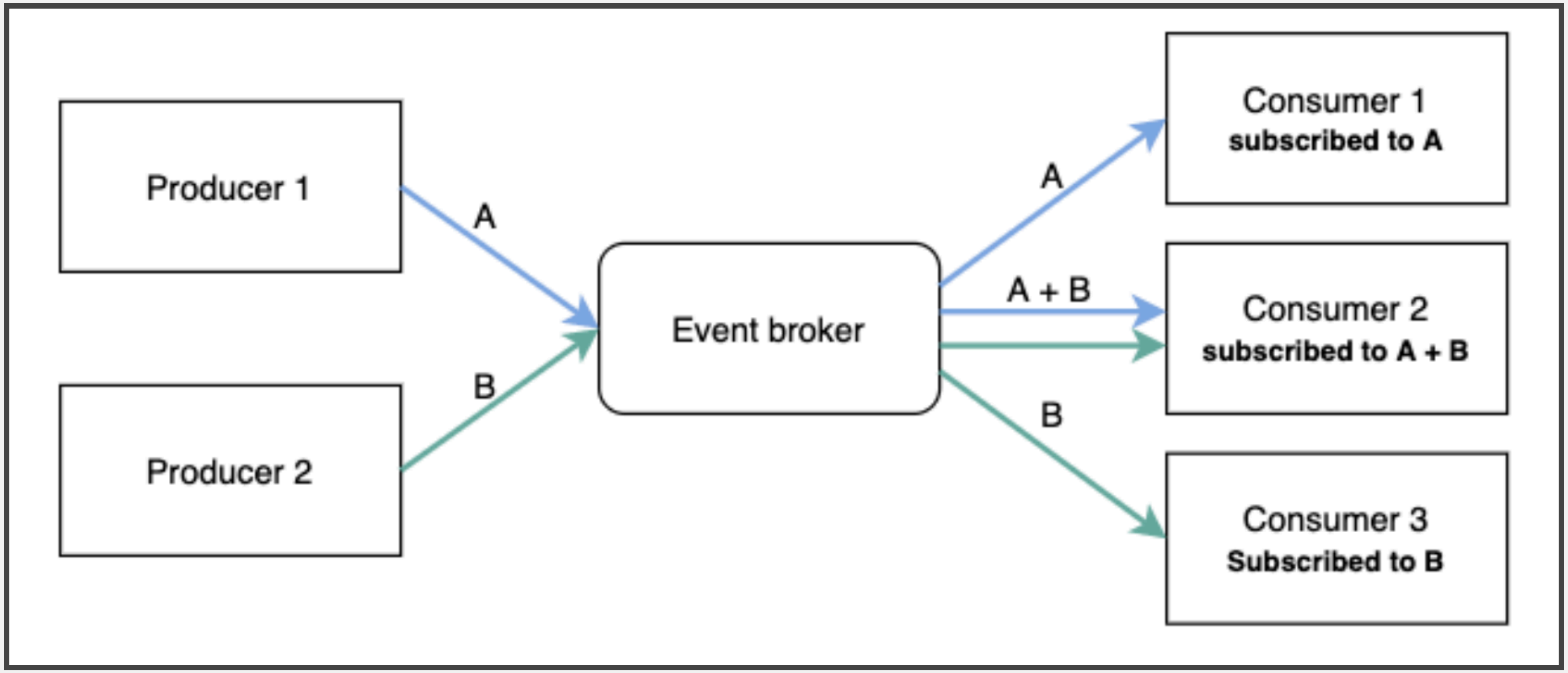
Pipes & Filters Example: Compilers



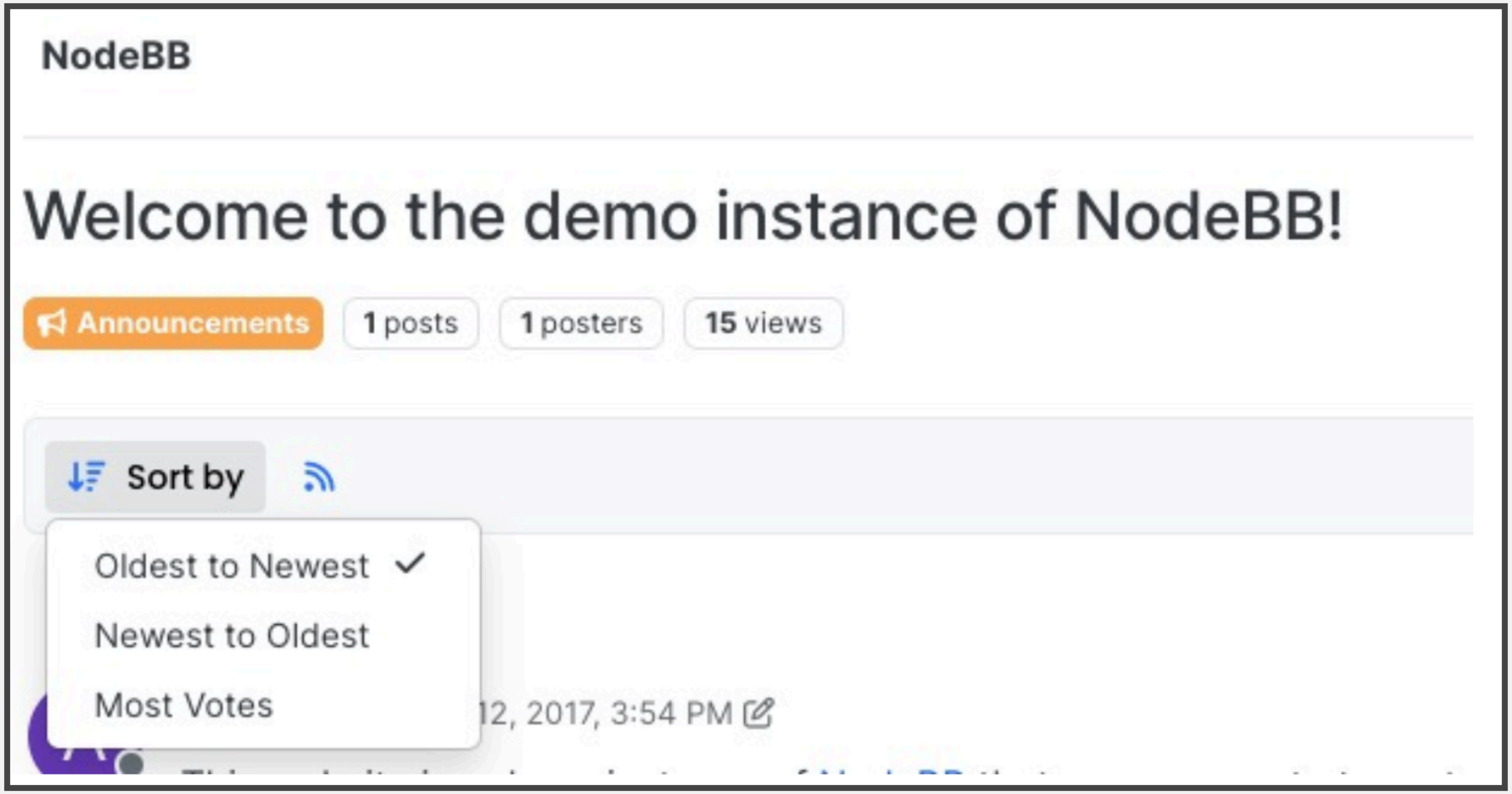
2. Object Oriented Organization



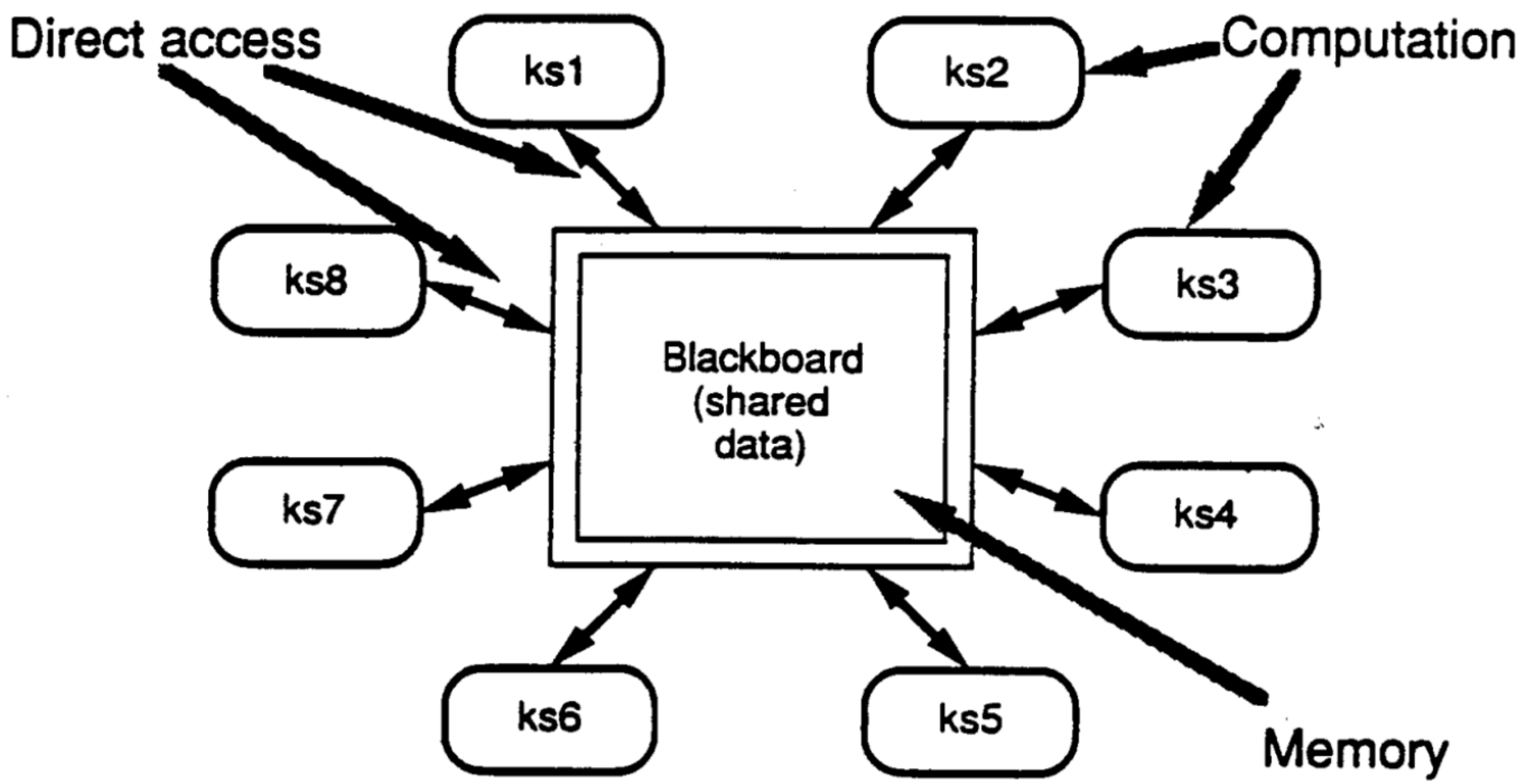
3. Event-Driven Architecture



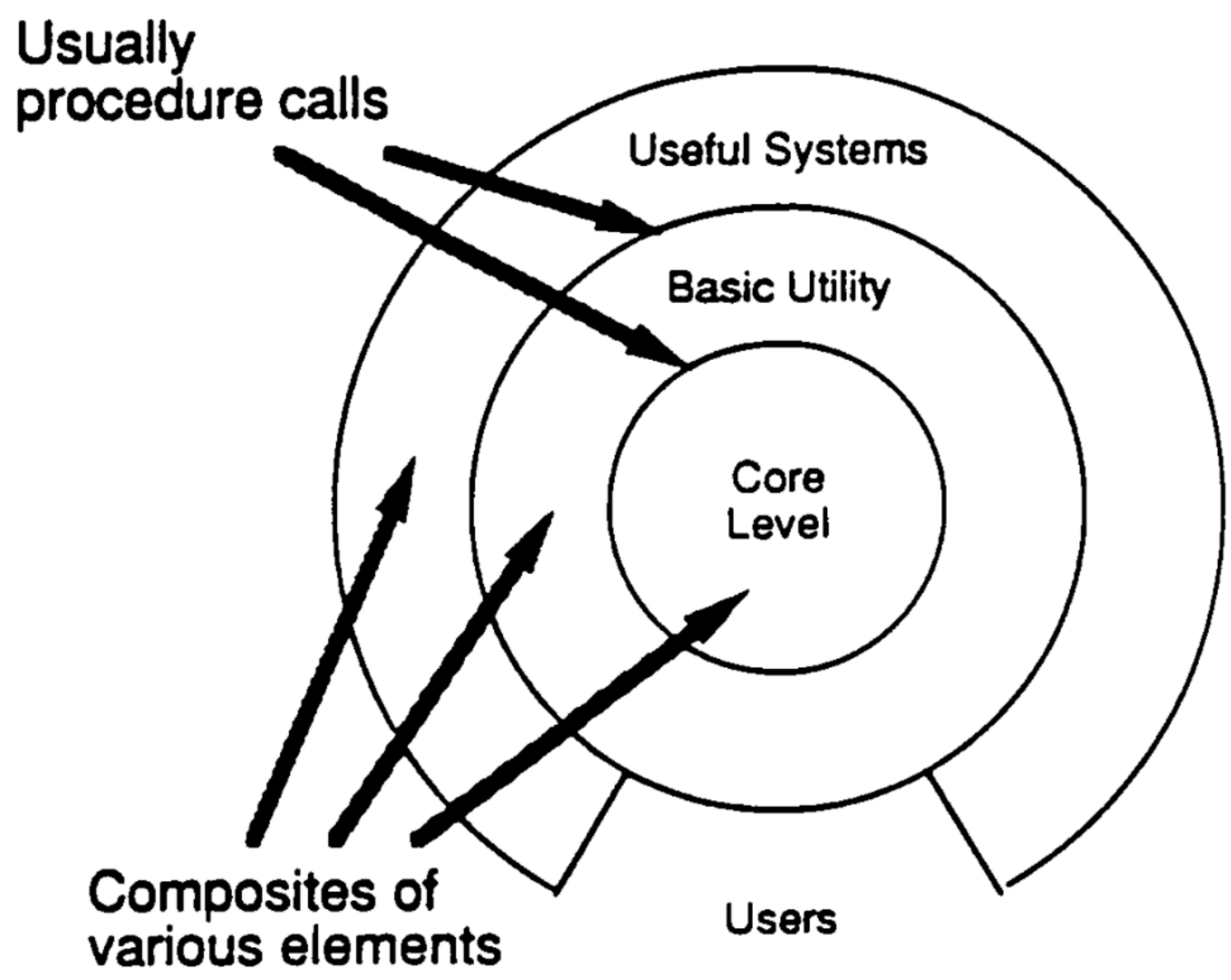
Example: HTML DOM + Javascript



4. Blackboard Architecture

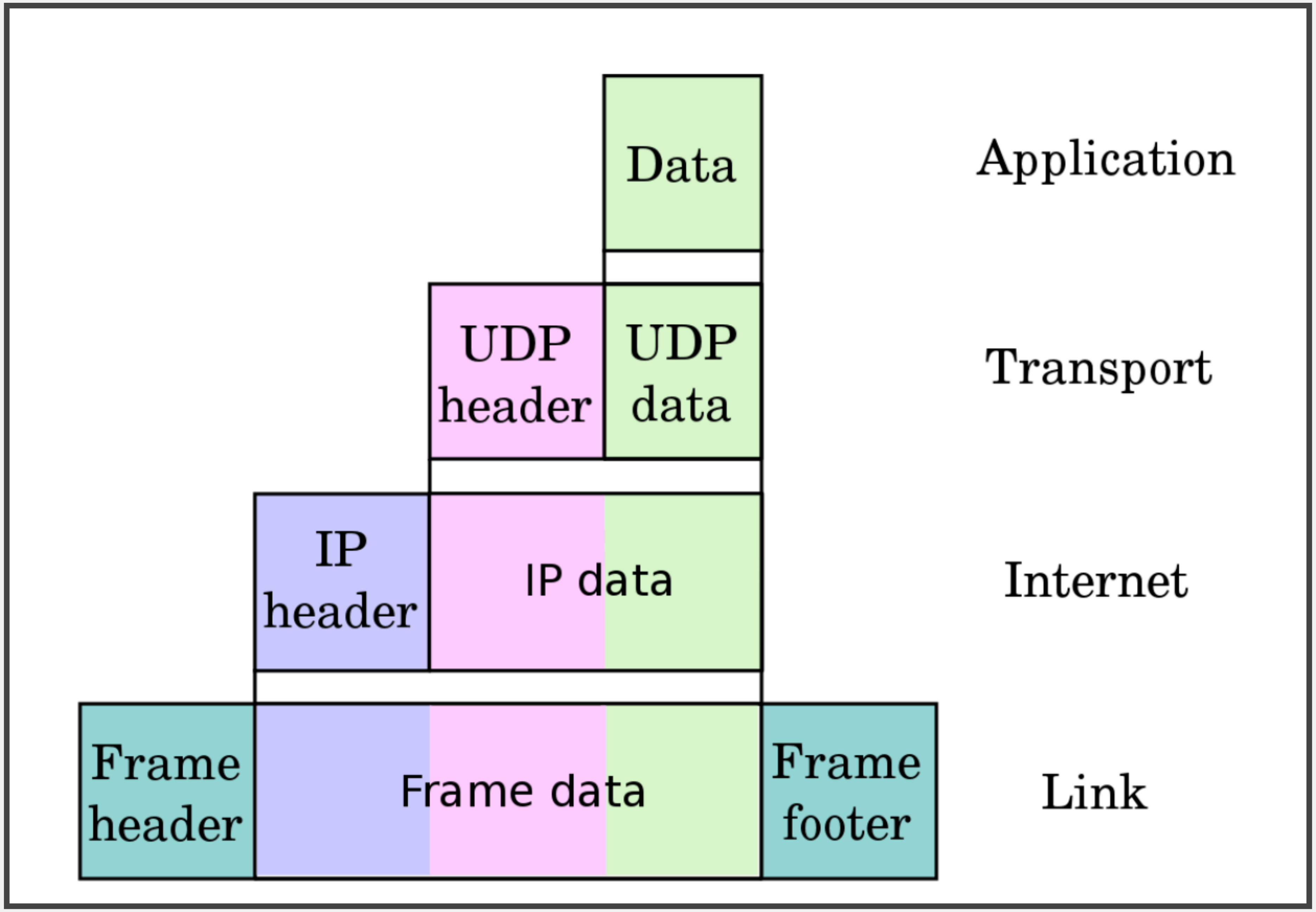


5. Layered Systems



© David Garlan and Mary Shaw, CMU/SEI-94-TR-021

Example Internet Protocol Suite



Guidelines for Selecting a Notation

- Suitable for purpose
- Often visual for compact representation
- Usually boxes and arrows
- UML possible (semi-formal), but possibly constraining
  - Note the different abstraction level – Subsystems or processes, not classes or objects
- Formal notations available
- Decompose diagrams hierarchically and in views
- Always include a legend
- Define precisely what the boxes mean
- Define precisely what the lines mean
- Do not try to do too much in one diagram
  - Each view of architecture should fit on a page
  - Use hierarchy