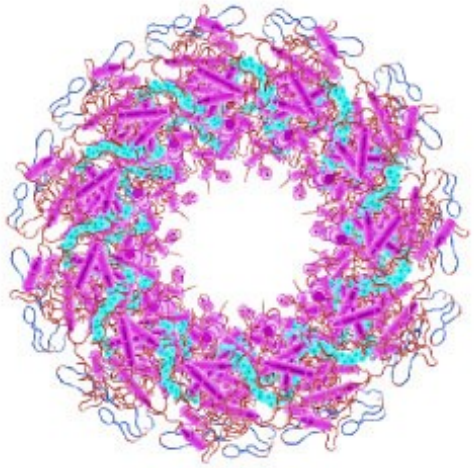
Enterprise software development
from a
biochemistry professor's
point of view



University of Montana

2019-09-13

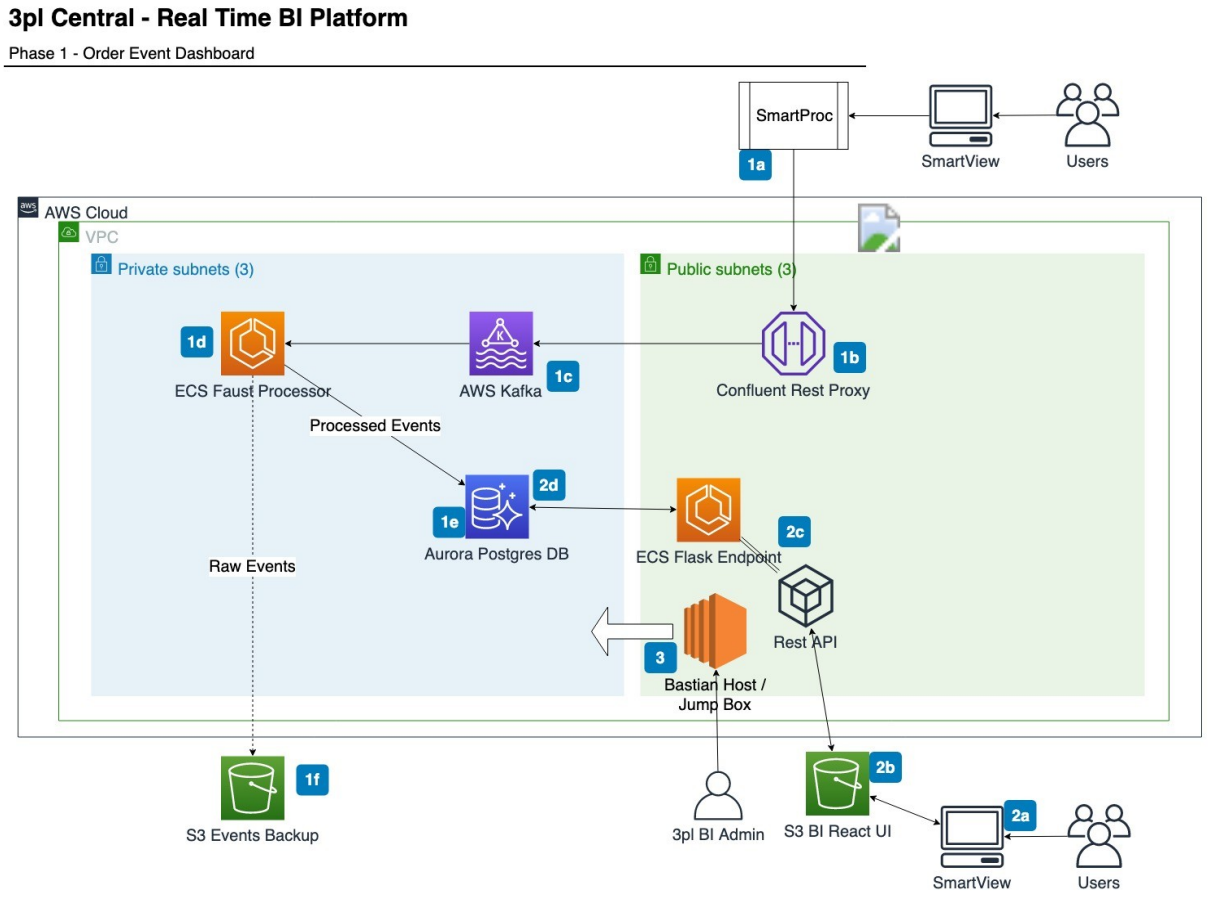
John T. Prince

Perspective

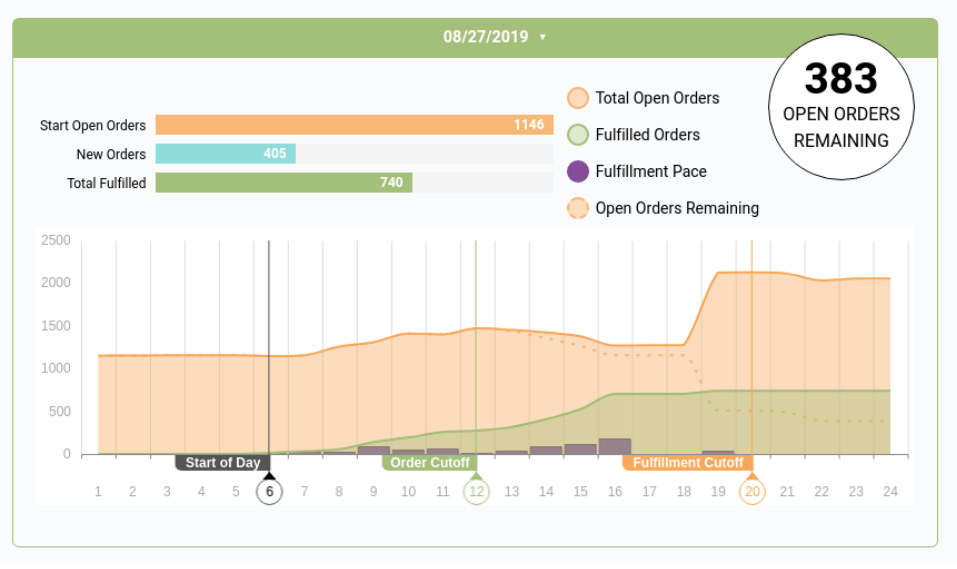
- **14 years academic software development**
  - **perl, C/C++, matlab, ruby, R**
- **5 years of enterprise software development**
  - **python, PHP, django, elasticsearch, kafka, faust, flask, AWS, docker, CI/CD**

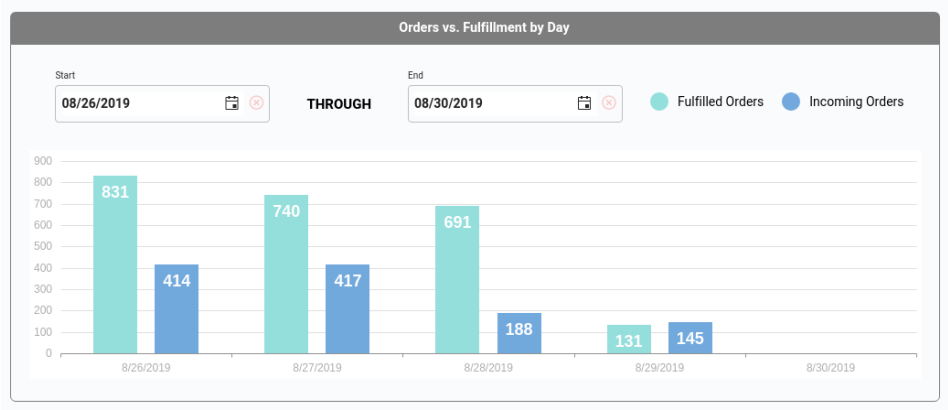
https://github.com/jtprince/portfolio/

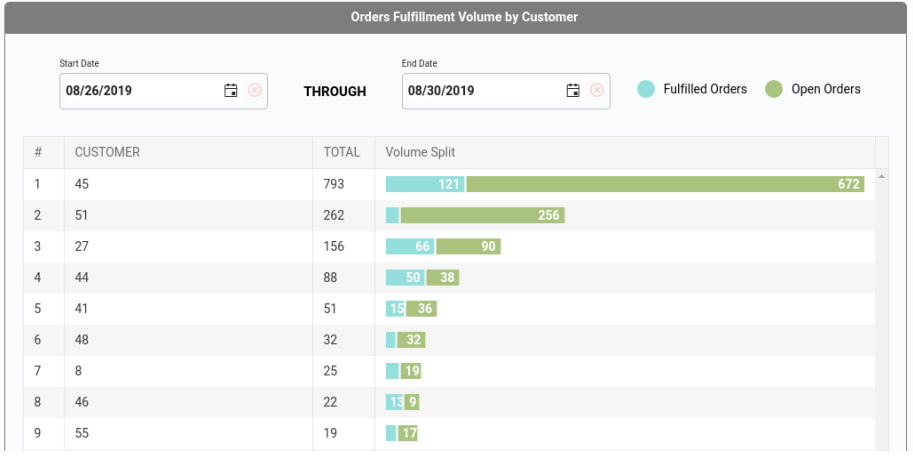
Event Driven Stream Processing



Order Volume Dashboard







#	CUSTOMER	TOTAL	Volume Split
1	45	793	121 / 672
2	51	262	256
3	27	156	66 / 90
4	44	88	50 / 38
5	41	51	15 / 36
6	48	32	32
7	8	25	19
8	46	22	13 / 9
9	55	19	17

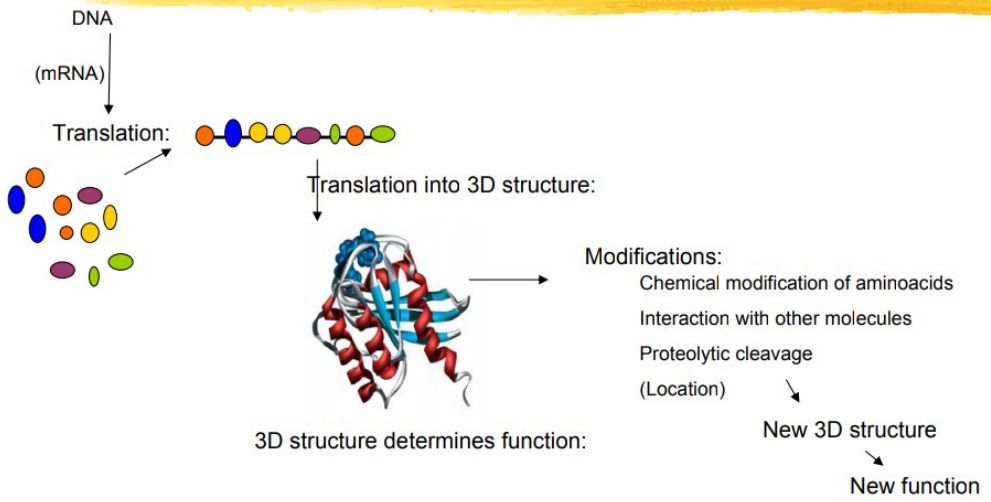
Purpose

Underscore good principles:

- **Useful in enterprise development**
- **Often neglected by junior developers**

Structure Determines Function

Protein structure determines function



- Proteins are single, unbranched chains of amino acid monomers
- There are 20 different amino acids
- The amino acid sidechains in a peptide can become modified, extending the functional repetoire of aminoacids to more than hundred different amino acids.
- A protein's amino acid sequence determines its three-dimensional structure (conformation)
- In turn, a protein's structure determines the function of that protein
- Conformation (=function) is dynamically regulated in several different ways

https://pdfs.semanticscholar.org/374b/647432f456a77b46f2b50e25ca71c3f606ab.pdf

Biochemistry in one minute

http://tiny.cc/biochem-1min

- **20 amino acids → Huge variety of function**
- **Patterns of structure**

Patterns and Conclusions

The fundamental units of cellular life:

- are Symmetrical
- Repeat themselves in the same way
- are Composable
  - Larger structures are composed of smaller units
- Interface well with one another
- Influence the behavior of other units in well-defined ways

Thesis

***How* you structure your code has enormous functional implications**

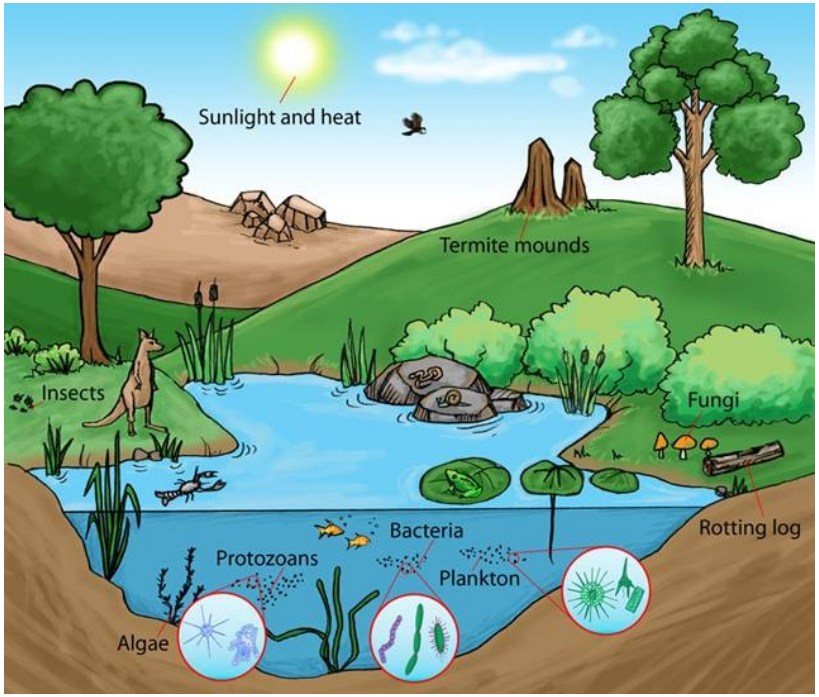
...at every level

Overview

- Software as an ecosystem
- Best coding practices
  - DRY (Don’t repeat yourself)
  - Single purpose (no side effects)
  - Abstractions, not append or indices
- Design principles
  - Hard shell, soft insides (function interfaces)
  - Modeling is hard (the world is a network)

ECOSYSTEM

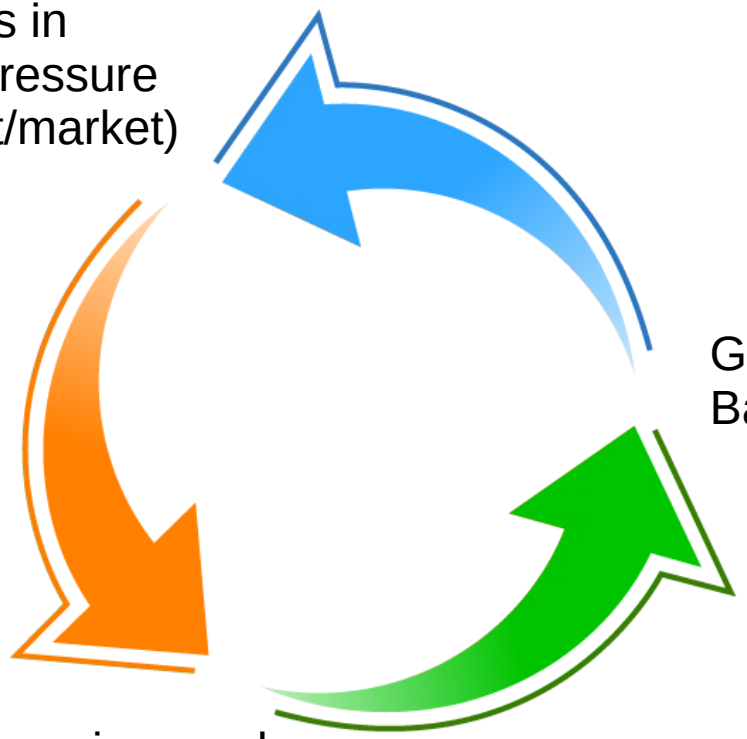
Software Ecosystem



- **Code is constantly being written and then dying off.**
- **Weather patterns change in the short and long-term**
- **Fit code contributes to a flourishing ecosystem**

Evolution of code

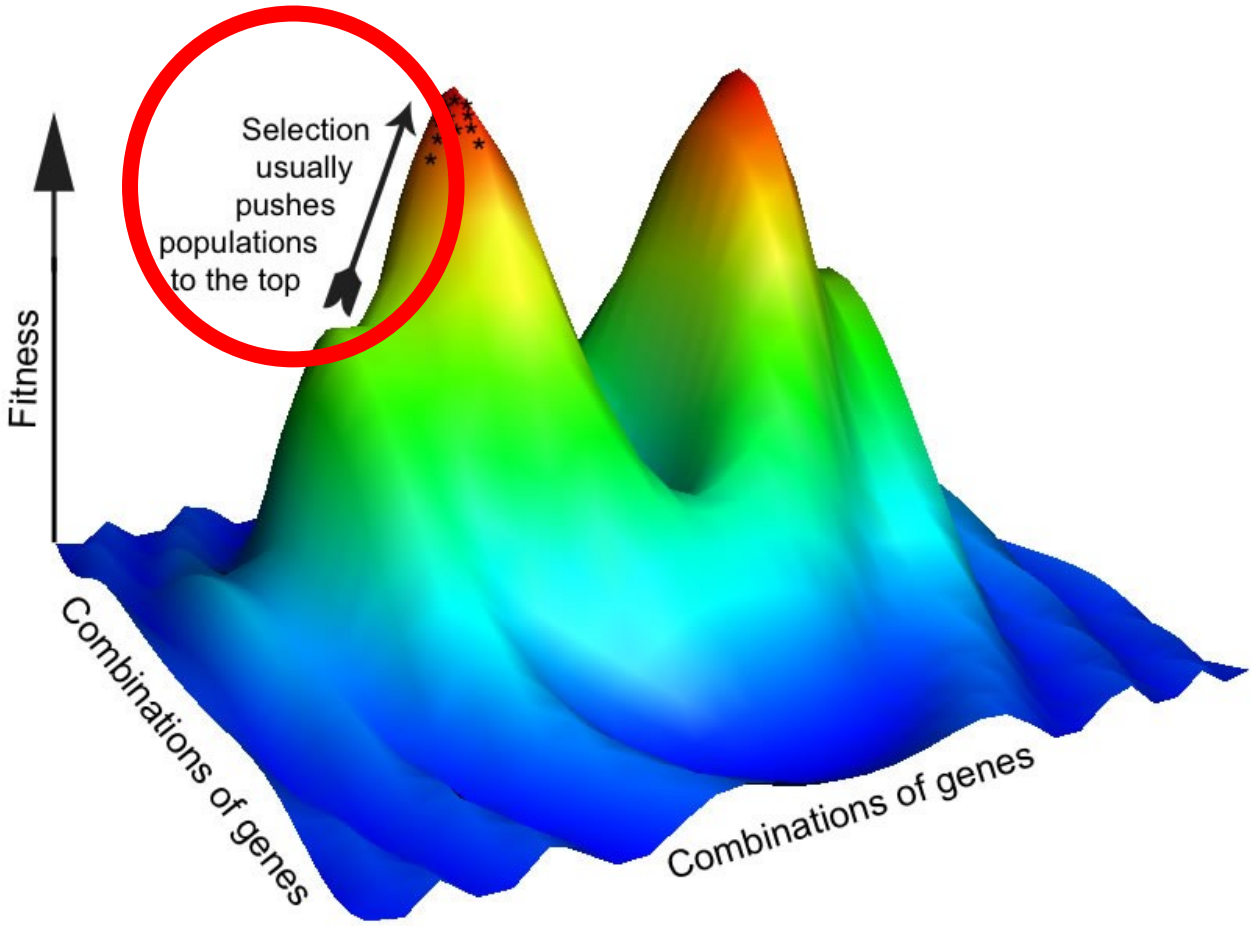
Code changes in response to pressure (management/market)



Good code survives
Bad code is discarded

Tests, code review, and performance promote code fitness

“Fit” code needs selective pressure

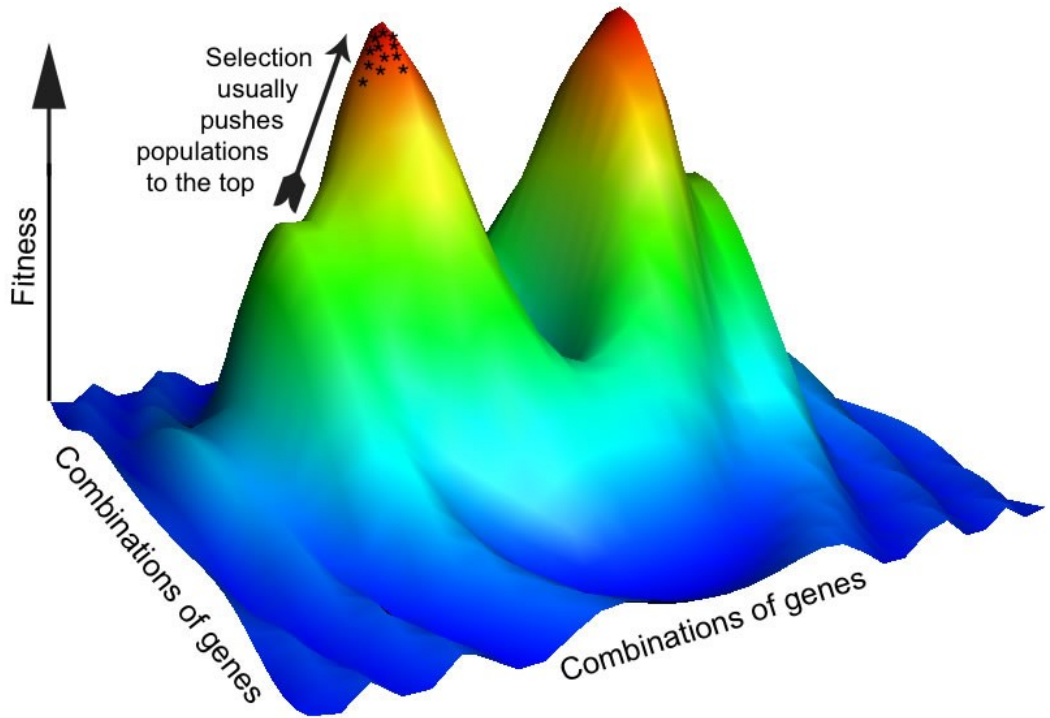


What is Code Fitness?

performance * robustness * modifiable/maintainability

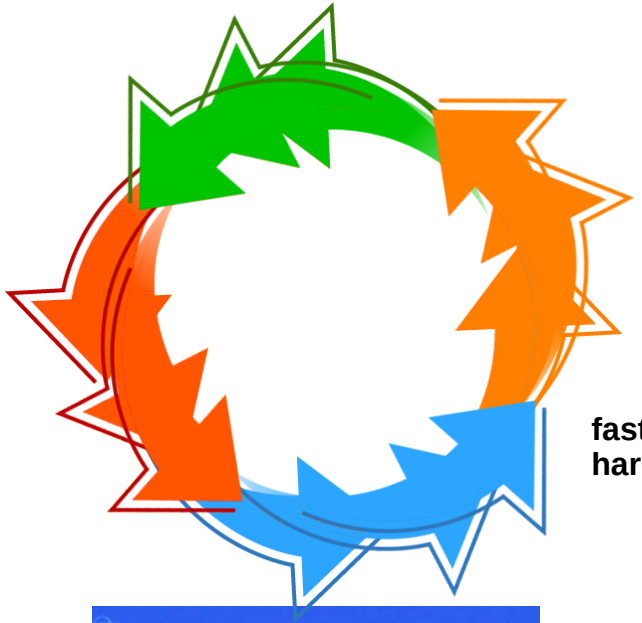
What drives code fitness?

- **Performance/robustness in the “wild”**
- **Testing**
  - **Unit**
  - **Integration**
  - **End-to-end**
- **Code linting**
  - **Style and format**
- **Peer review**
  - **Readability**
  - **Sanity**



Short vs. long generation times

testing environment



fast adaptation and hardening

production environment

slow adaptation and hardening

Code without tests

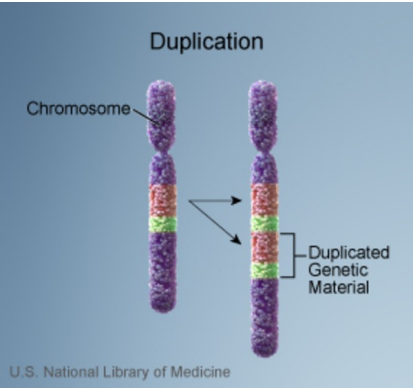
Dead on Arrival

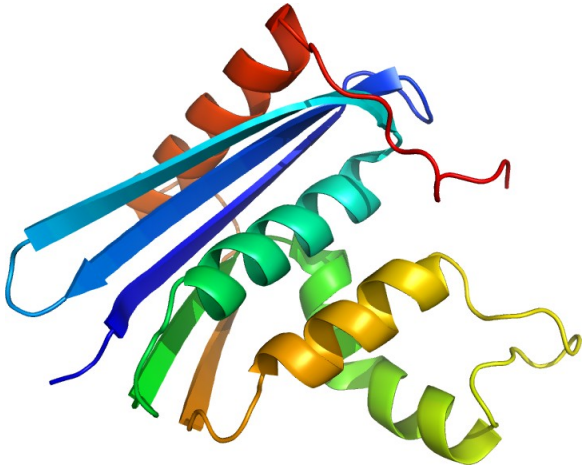
DRY

Unnecessary repetition is evil

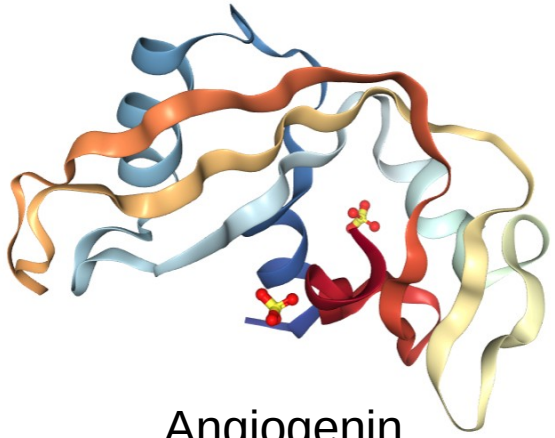
Why are you copying that chunk of code?

Code duplication allows divergence

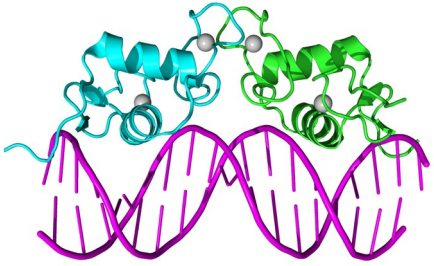




Ribonuclease-H



Angiogenin



A Tale of Two Codebases

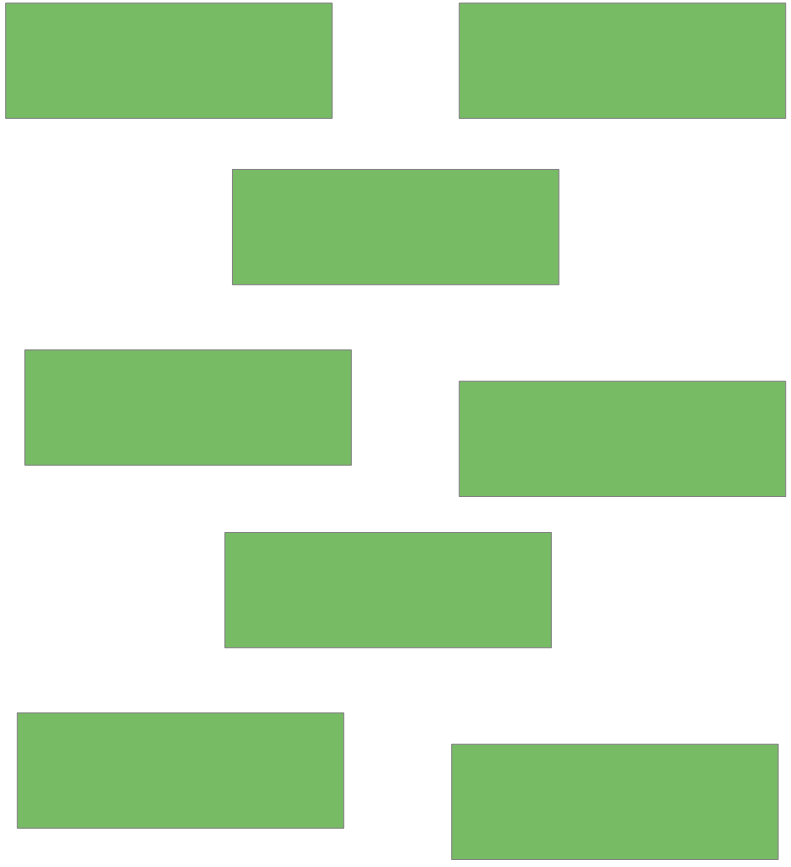
No code duplication

Code duplication

A Tale of Two Codebases

No code duplication

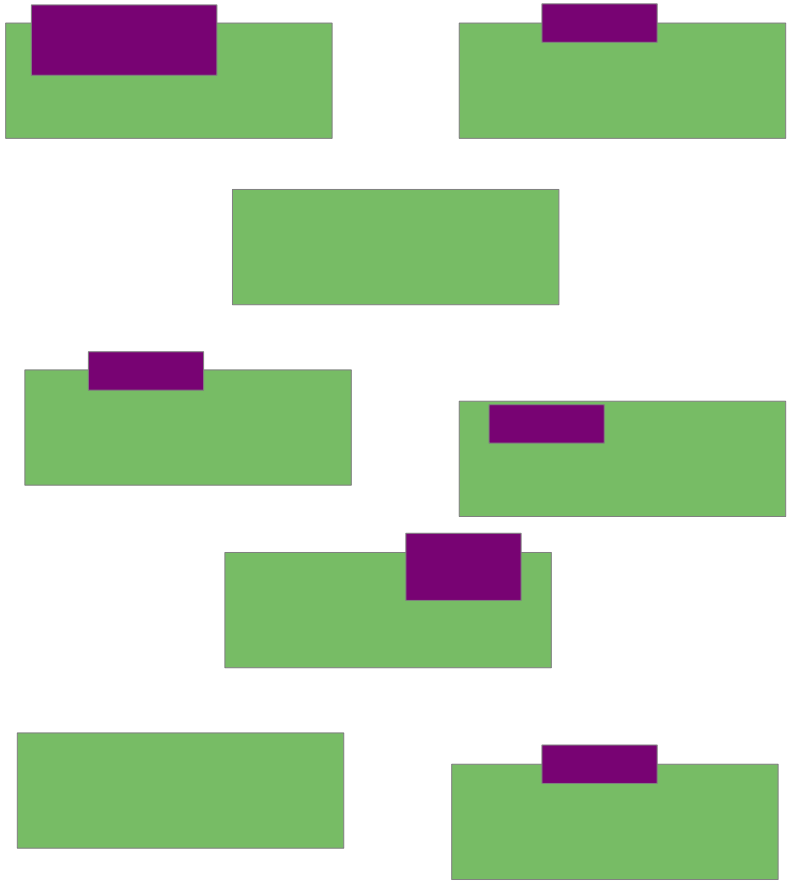
Code duplication



Bug Fix

No code duplication

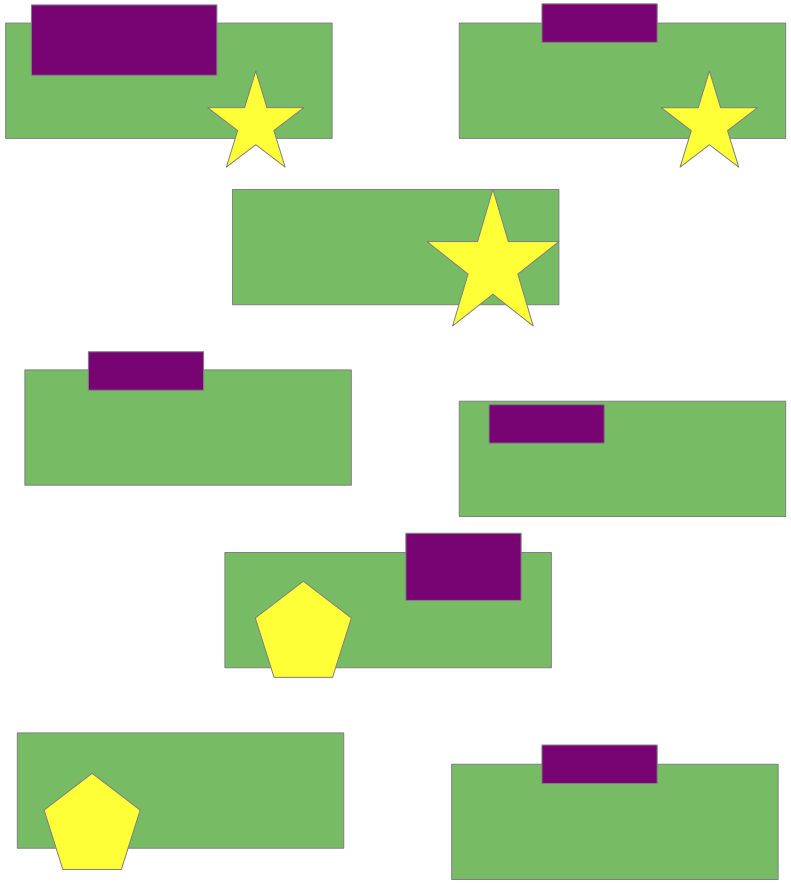
Code duplication



Feature addition

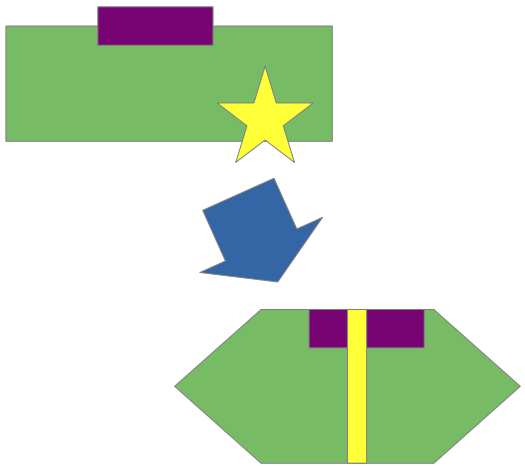
No code duplication

Code duplication

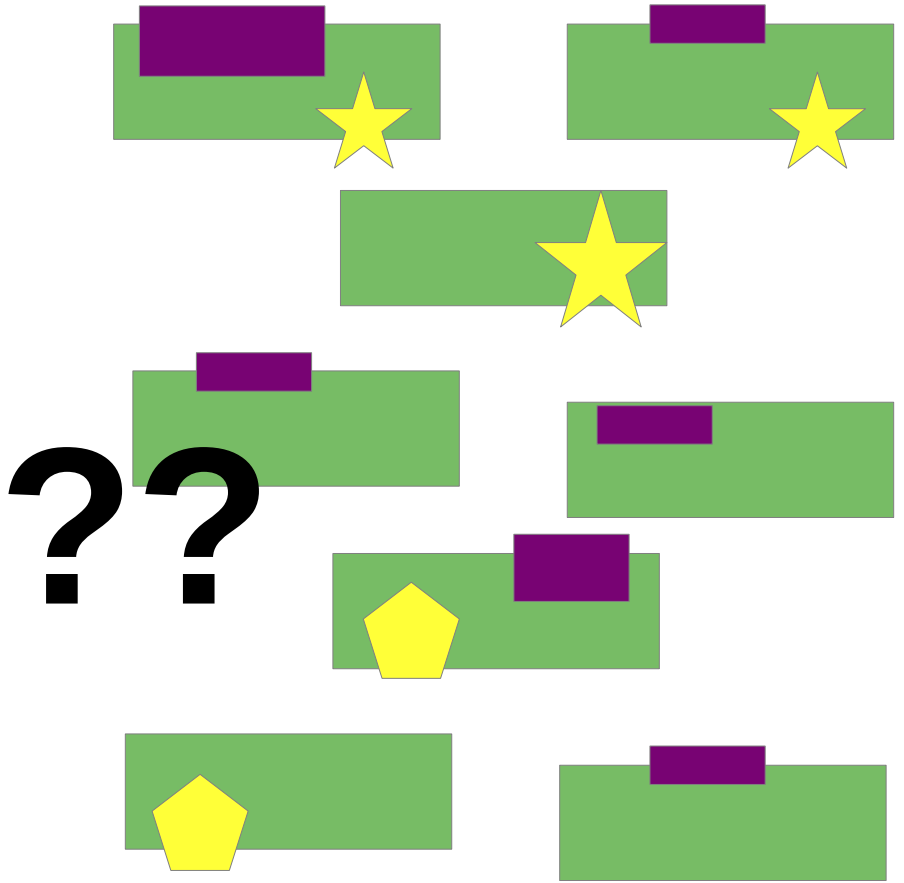


Refactoring

No code duplication

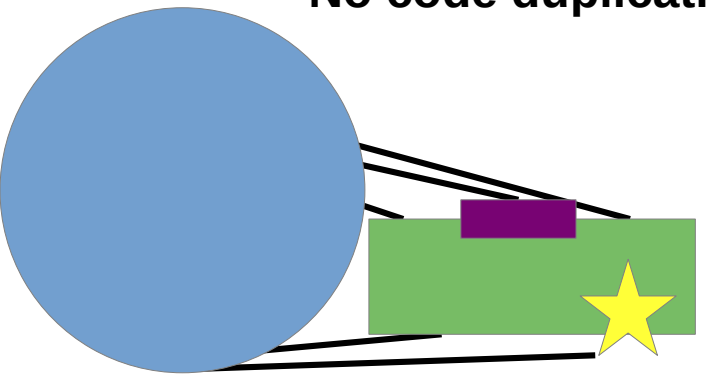


Code duplication

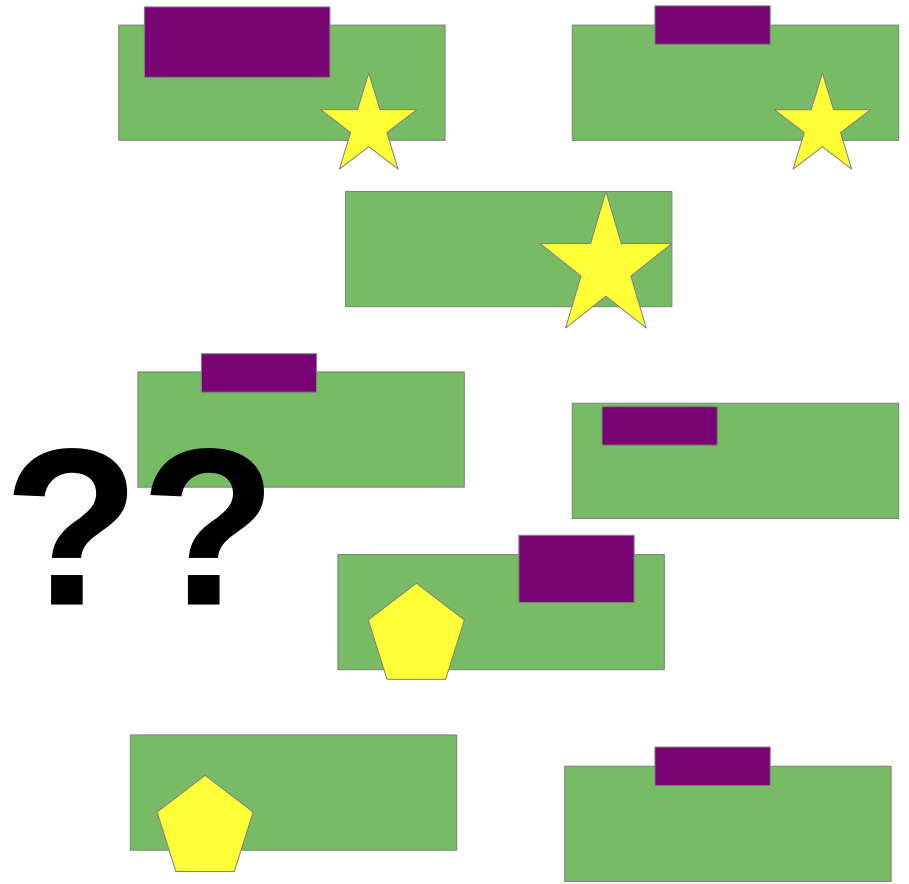


Interfacing

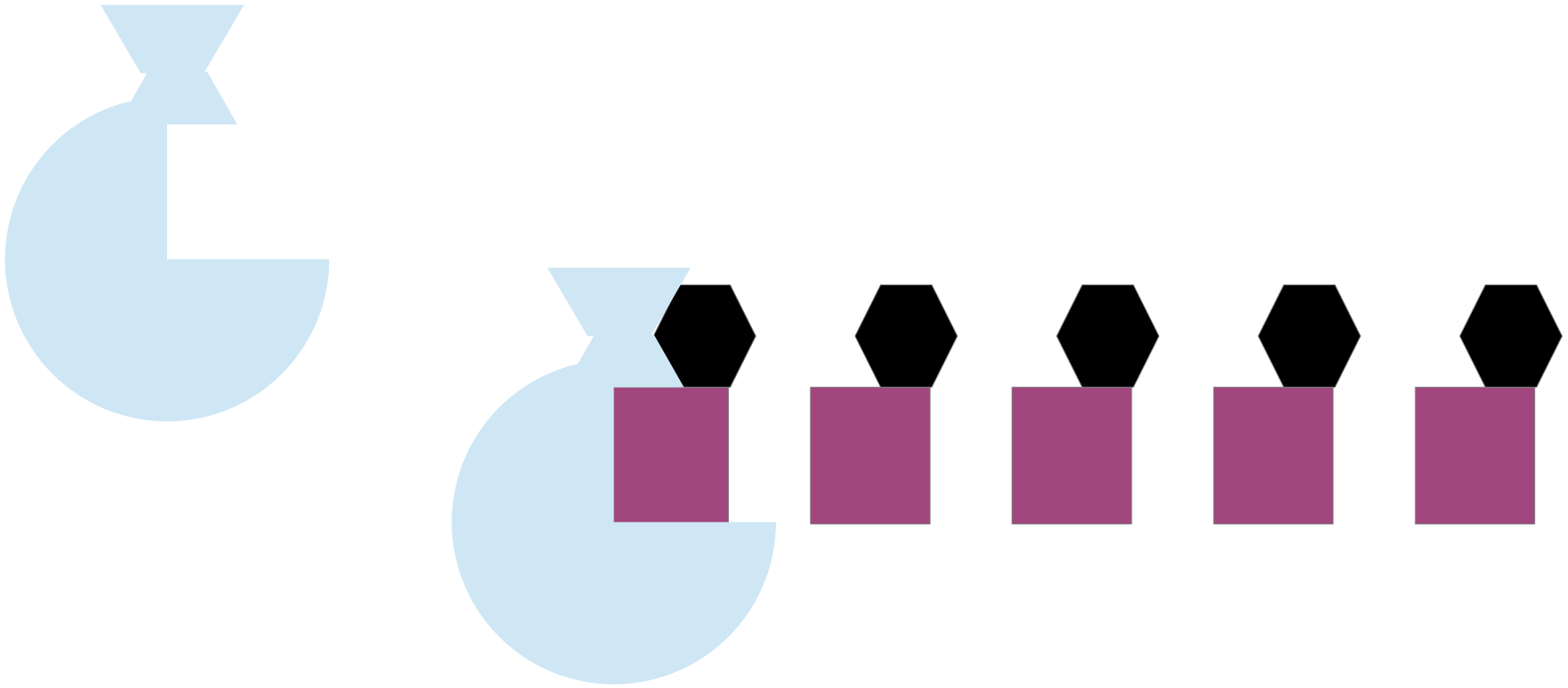
No code duplication



Code duplication

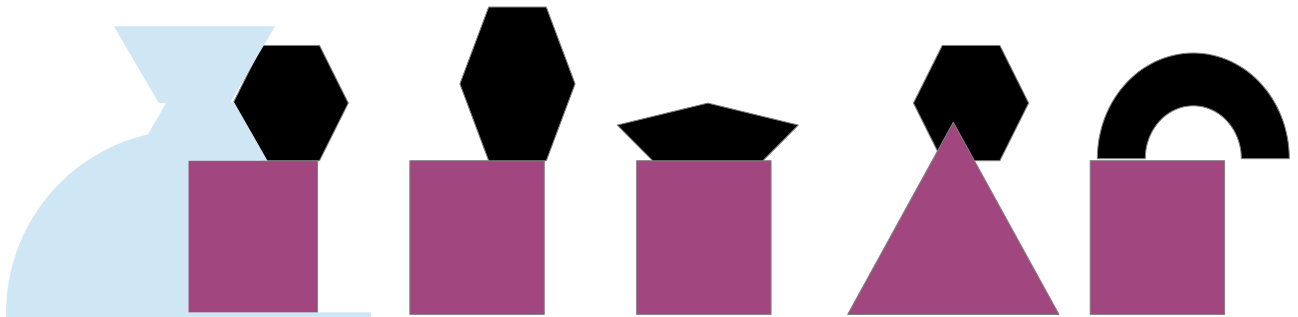


Isomorphism is crucial to automation



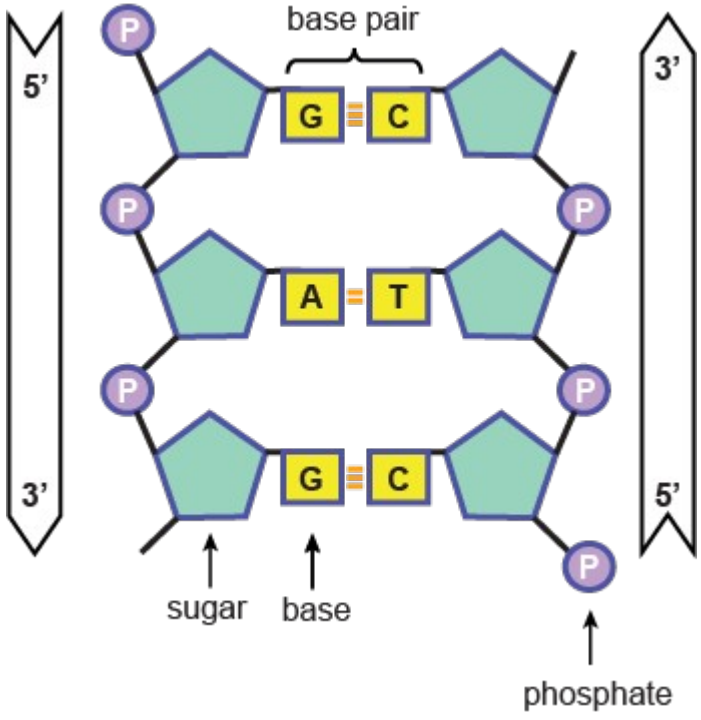
Isomorphic == same or similar “shape”

Different “shape” means special handling



Isomorphic == same or similar “shape”

Nature has figured this out



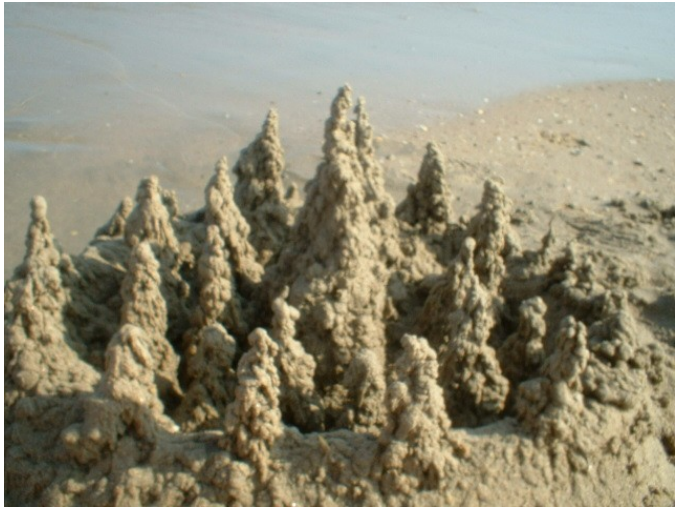
Isomorphic == same or similar “shape”

http://tiny.cc/dna-replication

Logical legos

Simple, composable, well-defined pieces allow for automatable, scalable systems

versus



Consequences

Repeating code unnecessarily often results in loss of:

- **modifiability**
- **maintainability**
- **robustness**
- **composability**
- **automation**

Example

```
def get_status(connection):

    if connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_IDLE:
        return "TRANSACTION_STATUS_IDLE"
    elif connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_ACTIVE:
        return "TRANSACTION_STATUS_ACTIVE"
    elif connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_INERROR:
        return "TRANSACTION_STATUS_INERROR"
    elif connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_UNKNOWN:
        return "TRANSACTION_STATUS_UNKNOWN"
    else:
        return "unknown transaction status"
```

DRY

```
CONNECTION_STATUSES = {
    psql.extensions.TRANSACTION_STATUS_IDLE: "TRANSACTION_STATUS_IDLE",
    psql.extensions.TRANSACTION_STATUS_ACTIVE: "TRANSACTION_STATUS_ACTIVE",
    psql.extensions.TRANSACTION_STATUS_INTRANS: "TRANSACTION_STATUS_INTRANS",
    psql.extensions.TRANSACTION_STATUS_UNKNOWN: "TRANSACTION_STATUS_UNKNOWN",
}

def get_status(connection):
    return CONNECTION_STATUSES.get(connection.status, "unknown transaction status")
```

Duplication fosters divergence

```
def get_status(connection):

    if connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_IDLE and os.environ['SOMETHING'] != 'dog':
        return "TRANSACTION_STATUS_IDLE"
    elif connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_ACTIVE:
        raise RuntimeError("Cannot handle these yet!")
        return "TRANSACTION_STATUS_ACTIVE"
    elif connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_INERROR:
        print("why do we do this?")
        return "TRANSACTION_STATUS_INERROR"
    elif connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_UNKNOWN:
        connection.status.clear()
        return "TRANSACTION_STATUS_UNKNOWN"
    else:
        return "unknown transaction status"
```

Duplication fosters divergence

```
def get_status(connection):

    if connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_IDLE and os.environ['SOMETHING'] != 'dog':
        return "TRANSACTION_STATUS_IDLE"
    elif connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_ACTIVE:
        raise RuntimeError("Cannot handle these yet!")
        return "TRANSACTION_STATUS_ACTIVE"
    elif connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_INERROR:
        print("why do we do this?")
        return "TRANSACTION_STATUS_INERROR"
    elif connection.info.transaction_status == psql.extensions.TRANSACTION_STATUS_UNKNOWN:
        connection.status.clear()
        return "TRANSACTION_STATUS_UNKNOWN"
    else:
        return "unknown transaction status"
```

Can this code be more DRY?

```
CONNECTION_STATUSES = {
    psql.extensions.TRANSACTION_STATUS_IDLE: "TRANSACTION_STATUS_IDLE",
    psql.extensions.TRANSACTION_STATUS_ACTIVE: "TRANSACTION_STATUS_ACTIVE",
    psql.extensions.TRANSACTION_STATUS_INTRANS: "TRANSACTION_STATUS_INTRANS",
    psql.extensions.TRANSACTION_STATUS_UNKNOWN: "TRANSACTION_STATUS_UNKNOWN",
}

def get_status(connection):
    return CONNECTION_STATUSES.get(connection.status, "unknown transaction status")
```

DRYest?

```
POSTGRES_TRANSACTION_STATUSES = ('IDLE', 'ACTIVE', 'INTRANS', 'UNKNOWN')
TRANSACTION_STATUS_BASE = "TRANSACTION_STATUS_"

CONNECTION_STATUSES = {
    getattr(psql.extensions, TRANSACTION_STATUS_BASE + status): TRANSACTION_STATUS_BASE + status
    for status in POSTGRES_TRANSACTION_STATUSES
}

def get_status(connection):
    return CONNECTION_STATUSES.get(connection.status, "unknown transaction status")
```

DRYest?

```
POSTGRES_TRANSACTION_STATUSES = ('IDLE', 'ACTIVE', 'INTRANS', 'UNKNOWN')
TRANSACTION_STATUS_BASE = "TRANSACTION_STATUS_"

CONNECTION_STATUSES = {
    getattr(psql.extensions, TRANSACTION_STATUS_BASE + status): TRANSACTION_STATUS_BASE + status
    for status in POSTGRES_TRANSACTION_STATUSES
}

def get_status(connection):
    return CONNECTION_STATUSES.get(connection.status, "unknown transaction status")
```

This is not “clickable” code

“Clickable” code

```
POSTGRES_TRANSACTION_STATUSES = ('IDLE', 'ACTIVE', 'INTRANS', 'UNKNOWN')
TRANSACTION_STATUS_BASE = "TRANSACTION_STATUS_"

CONNECTION_STATUSES = {
    getattr(psql.extensions, TRANSACTION_STATUS_BASE + status): TRANSACTION_STATUS_BASE + status
    for status in POSTGRES_TRANSACTION_STATUSES
}

def get_status(connection):
    return CONNECTION_STATUSES.get(connection.status, "unknown transaction status")
```

```
CONNECTION_STATUSES = {
    psql.extensions.TRANSACTION_STATUS_IDLE: "TRANSACTION_STATUS_IDLE",
    psql.extensions.TRANSACTION_STATUS_ACTIVE: "TRANSACTION_STATUS_ACTIVE",
    psql.extensions.TRANSACTION_STATUS_INTRANS: "TRANSACTION_STATUS_INTRANS",
    psql.extensions.TRANSACTION_STATUS_UNKNOWN: "TRANSACTION_STATUS_UNKNOWN",
}

def get_status(connection):
    return CONNECTION_STATUSES.get(connection.status, "unknown transaction status")
```

DRYing code leads to higher abstractions

all?	drop	find_all	max_by
any?	drop_while	find_index	member?
chain	each_cons	first	min_by
chunk	each_entry	flat_map	minmax_by
chunk_while	each_slice	grep	none?
collect	each_with_index	group_by	partition
collect_concat	each_with_object	include?	reject
count	entries	inject	select
cycle	filter	lazy	take_while
detect	find	map	uniq

NO SIDE EFFECTS

Single Purpose

Each unit of code should have a single purpose and zero (or least possible number of) side effects

undesired side effect

```
def sort_it_with_side_effects(values):
    values.sort()
    return values


values = [3, 1, 2]
sorted_values = sort_it_with_side_effects(values)
# sorted_values = [1, 2, 3]
# values = [1, 2, 3]
```

single purpose, no side effects

```
def sort_it_no_side_effects(values):
    sorted_values = sorted(values)
    return sorted_values


sorted_values = sort_it_without_side_effects(values)
# sorted_values = [1, 2, 3]
# values = [3, 1, 2]
```

ABSTRACTIONS

Avoid `append` in loops

Naive modification of values in a list

```
collector = []
for val in values:
    new_val = val * 2
    collector.append(new_val)
```

Use list comprehension

```
collector = []
for val in values:
    new_val = val * 2
    collector.append(new_val)
```

```
collector = [val * 2 for val in values]
```

`map` in most languages

```
collector = values.map {|val| val * 2 }
```

Avoid list indices in loops

Naive implementation

Sliding window:

```
values = [1, 2, 4, 5, 6, 7]

windows = []
for index in range(0, len(values) - 2):
    window = (values[index], values[index + 1], values[index + 2])
    windows.append(window)

# windows -> [(1, 2, 4), (2, 4, 5), (4, 5, 6), (5, 6, 7)]
```

```
from toolz.itertoolz import sliding_window
list(sliding_window(3, values))
```

```
from toolz.itertoolz import sliding_window
list(sliding_window(3, values))
```

```
from cytoolz.itertoolz import sliding_window
list(sliding_window(3, values))
```

Combine values from a list

```
def combine(values, other_values):

    len_values = len(values)
    len_other_values = len(other_values)

    long_length = len_values if len_values > len_other_values else len_other_values

    combined = []
    for index in range(long_length):
        value, other_value = None, None

        if index < len_values:
            value = values[index]

        if index < len_other_values:
            other_value = other_values[index]

        combined.append((value, other_value))

    return combined
```

zip_longest

```
from itertools import zip_longest

zip_longest(values, other_values)
```

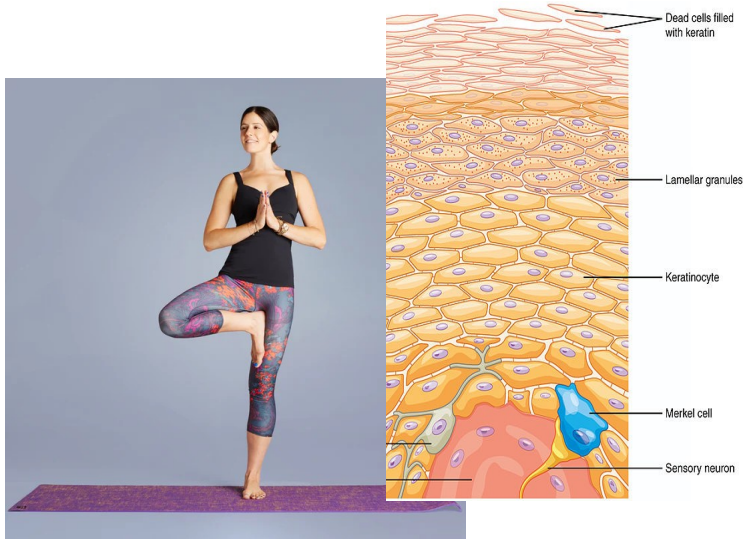
Iterable abstractions

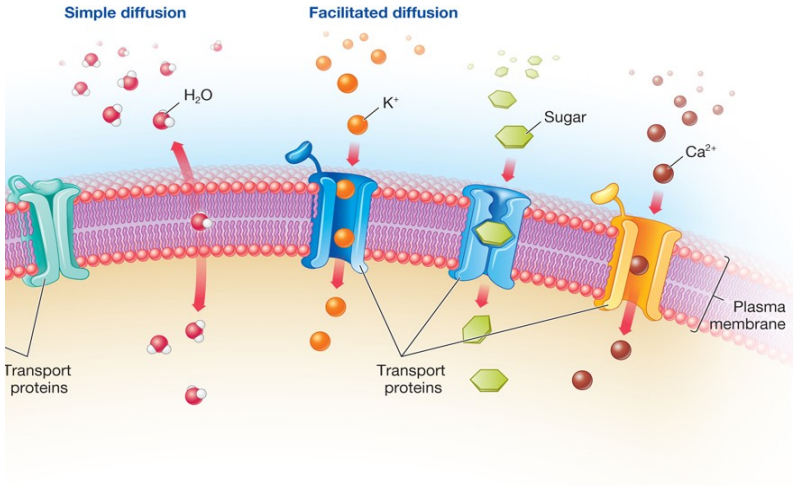
- Guarantee the length of the returned object
- (typically) guarantee no side effects
- Require far less code to specify
- Handle all edge cases
- Are typically much faster (C level code)

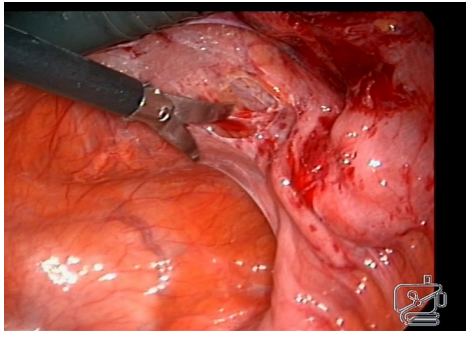
MODELING

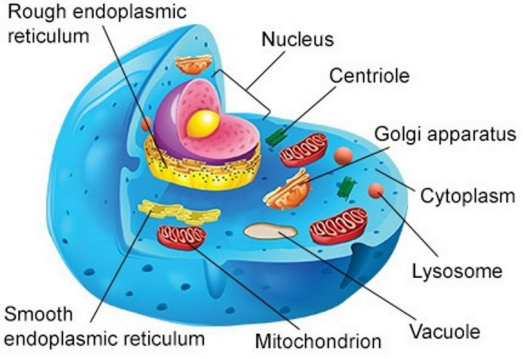
HARD SHELL
SOFT INSIDE

Hard Shell, Squishy Insides









Stringent public interface

```
BASE_SCHEMA = Schema(
    {
        Required("facility_id"): Dequote(str),
        Required("start_of_day"): All(Coerce(int), Clamp(min=1, max=23)),
        Required("fulfillment_cutoff"): All(Coerce(int), Clamp(min=1, max=23)),
        Required("order_cutoff"): All(Coerce(int), Clamp(min=1, max=23)),
        Required("facility_timezone"): TimeZoneify(str),
        "tpl_id": Dequote(str),
    }
)
DATE_RANGE_FRAGMENT = {Required("start_date"): DateObj(str), Required("end_date"): DateObj(str)}

DATE_SCHEMA = BASE_SCHEMA.extend({Required("date"): DateObj(str)})
DATE_RANGE_SCHEMA = BASE_SCHEMA.extend(DATE_RANGE_FRAGMENT)
```

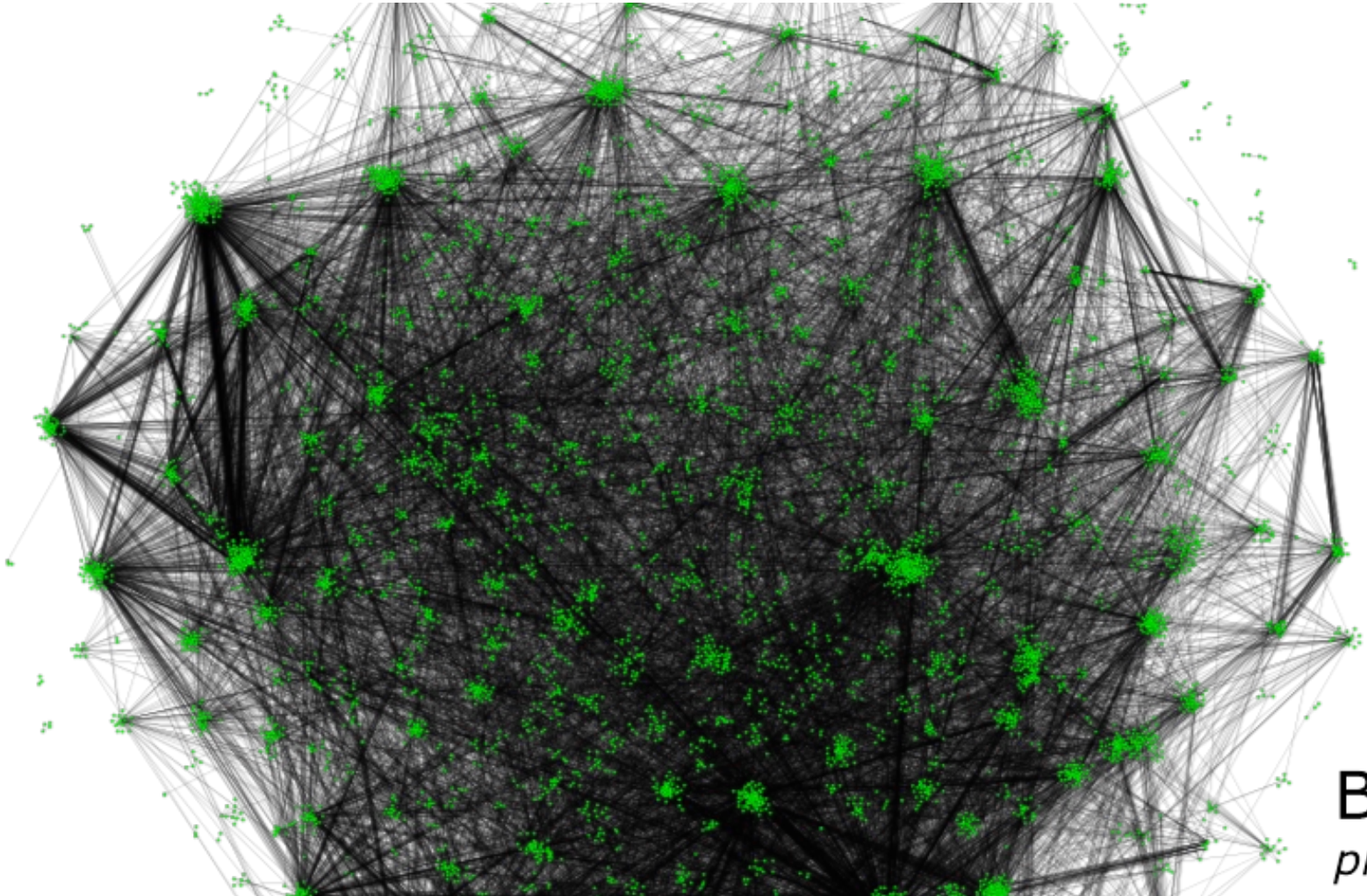
```
@app.route(f"{PATH_PREFIX}/daily-fulfillment", methods=["GET"])
@as_json
@expect(DATE_RANGE_SCHEMA, "args")
def daily_fulfillment(args):
    tpl_id = _get_tpl_id(request.headers)
```

Internal interfaces assume skilled handlers

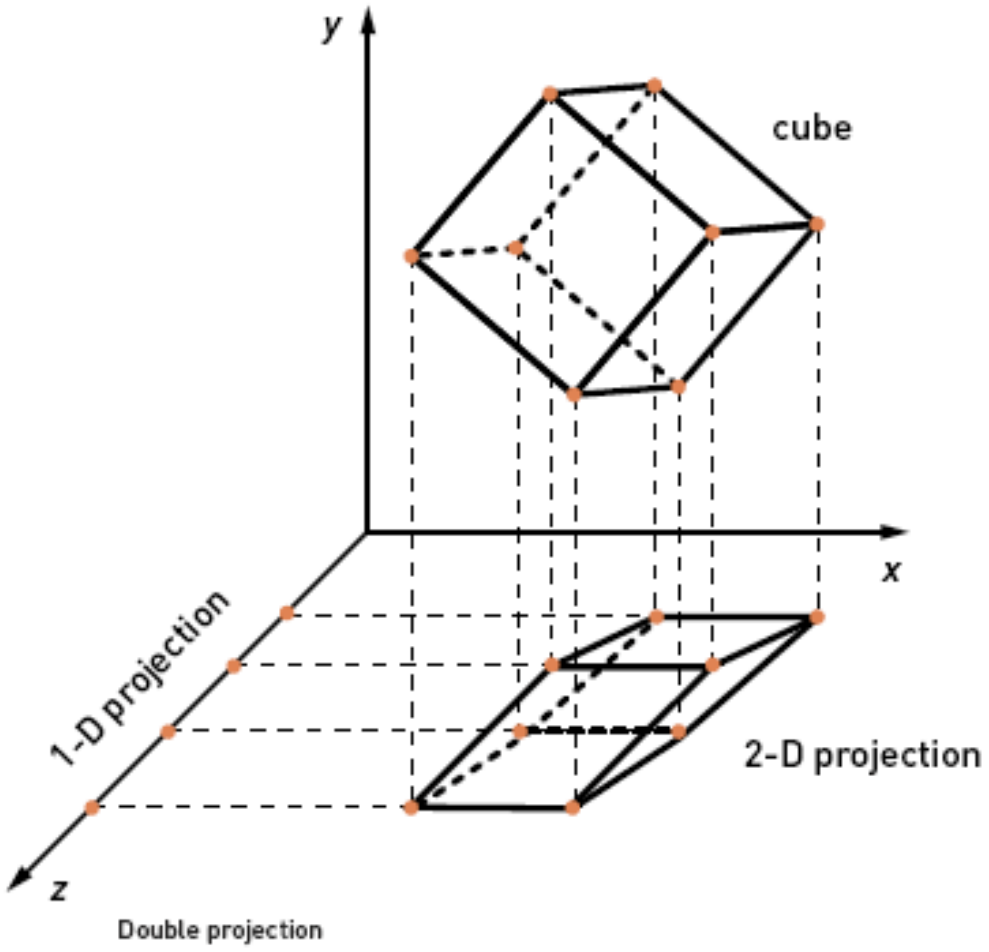
```
def __init__(self, display_hours, master_accumulator, null_after=None, plot_prepper=None):
    """
    Args:
      display_hours (list): An object parallel to the datetimes fed to the
          master accumulator.
      master_accumulator (MasterAccumulator): An object that can give a
          list of `accumulators`.
      null_after (datetime): A timezone aware UTC datetime.  If None, then no
          filtering will occur.  If True, then current datetime will be used.
    """
    self.display_hours = display_hours
    self.master_accumulator = master_accumulator
    self.plot_prepper = PlotPrepper() if plot_prepper is None else plot_prepper
    self.null_after = null_after
```

MODELING IS HARD

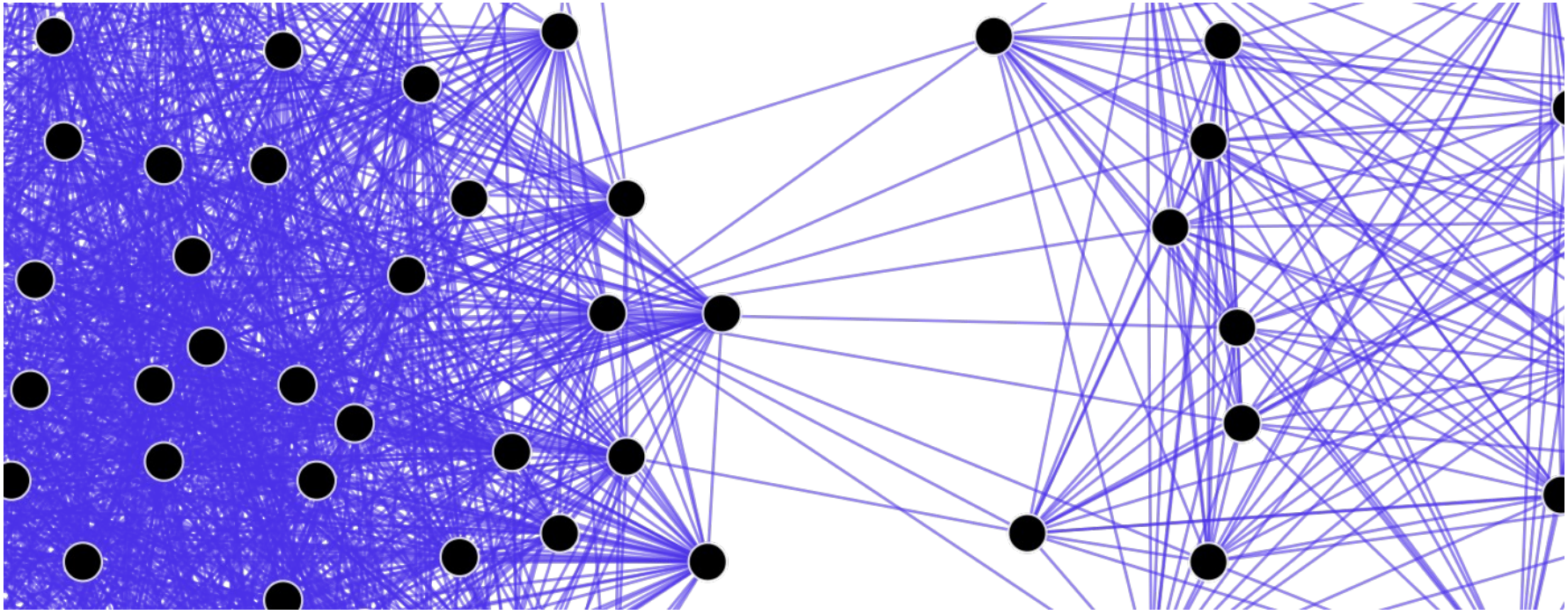
Real world interactions are like a network



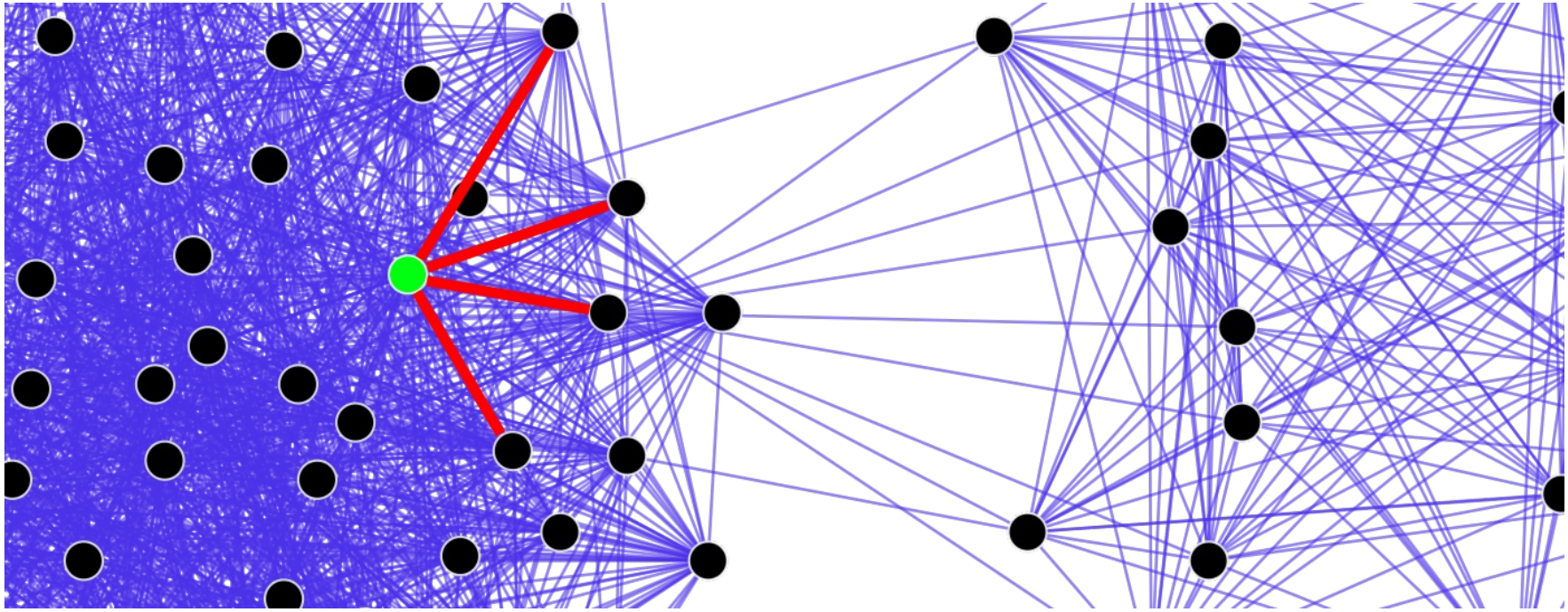
Modeling is the art of projection



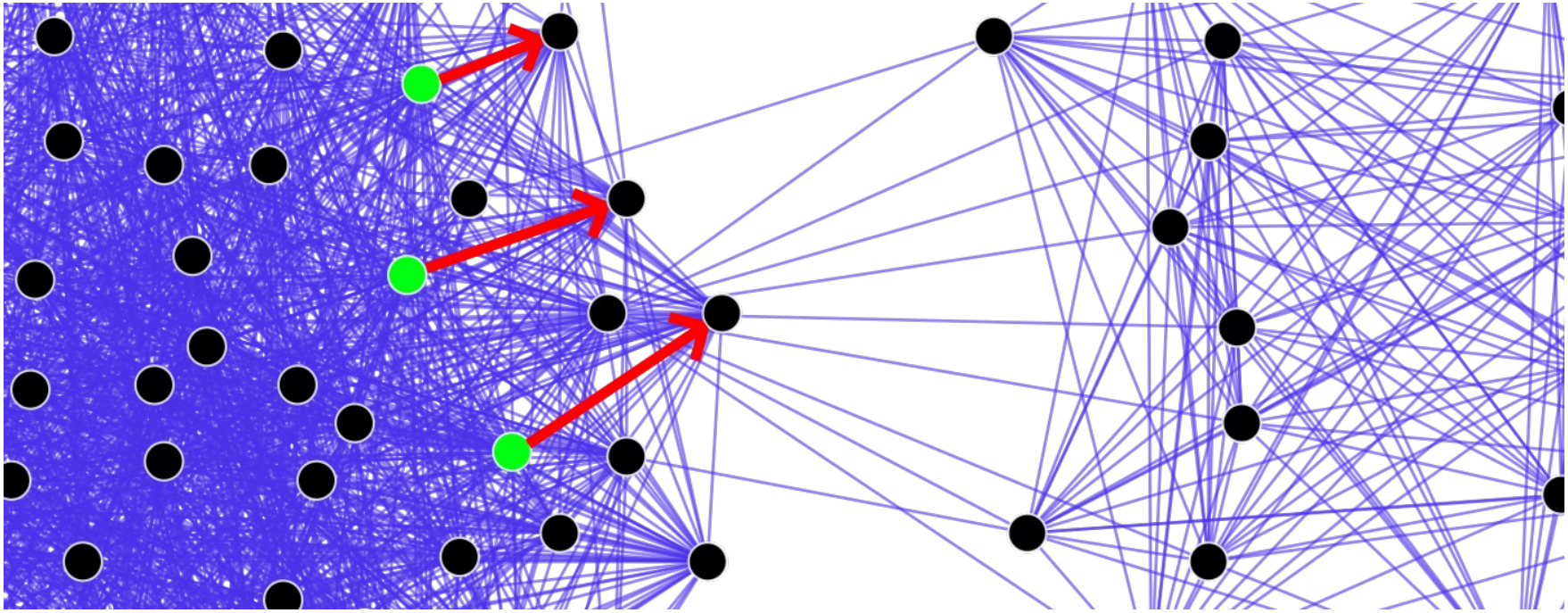
Reality is (typically) a network



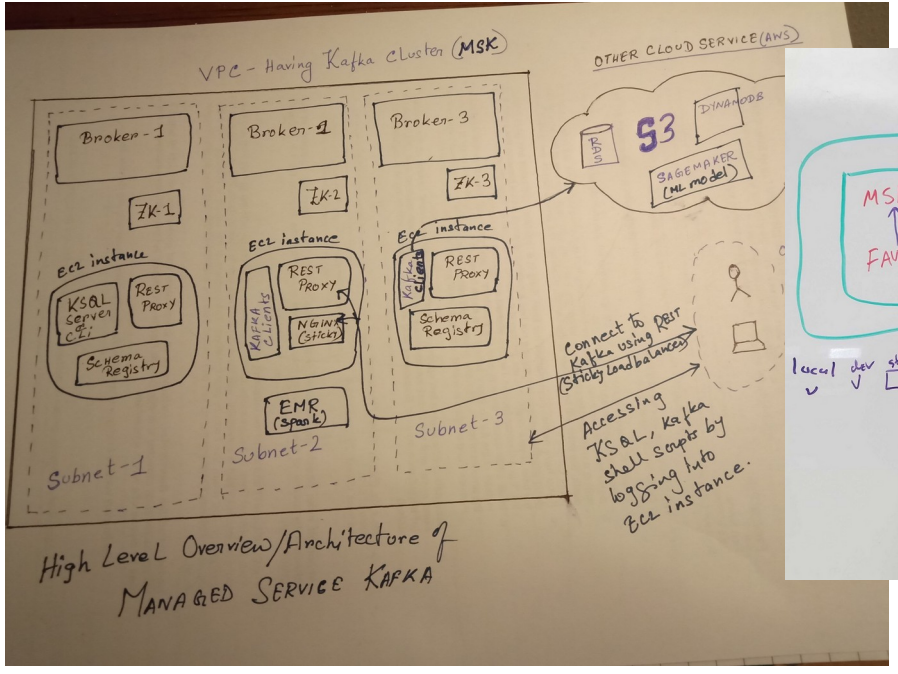
We emphasize certain relationships

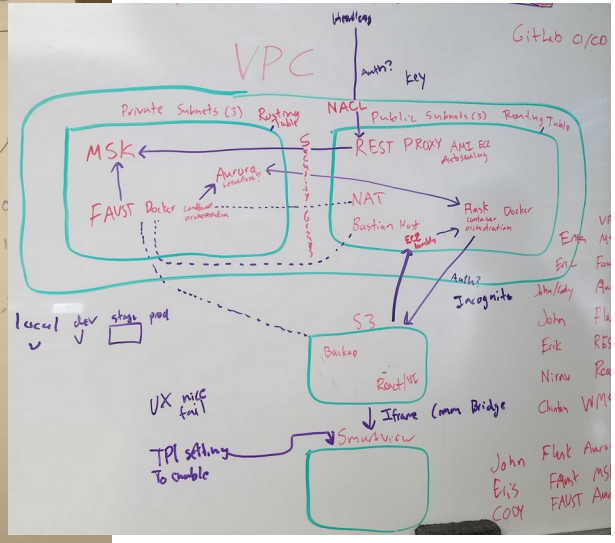


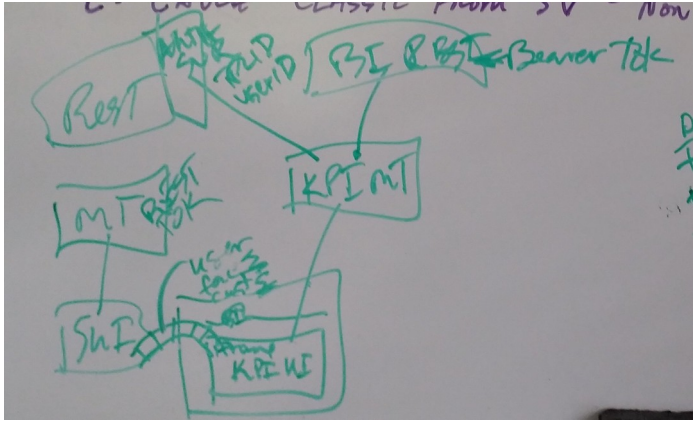
Restrictions limit complexity

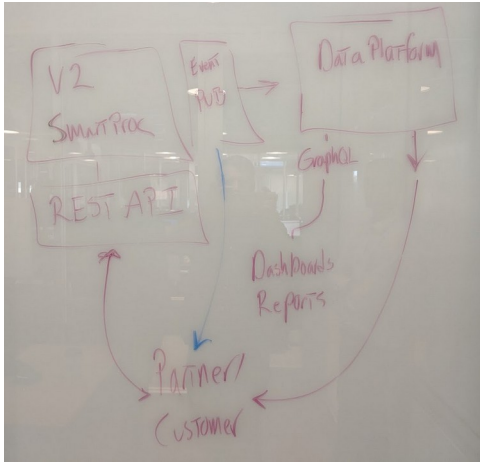


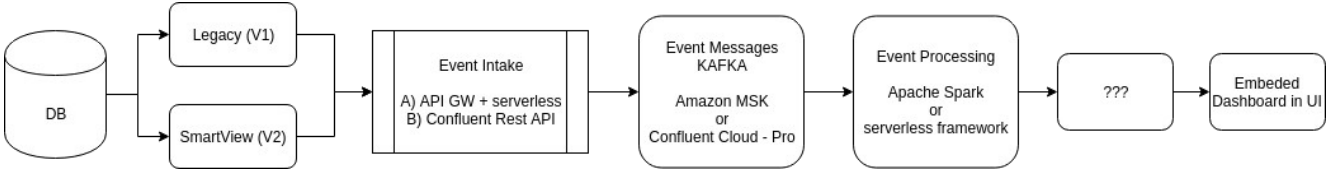
Diagramming helps











***How* you structure your code has enormous functional implications**

...at every level

Acknowledgements

- 3PL Central
  - Event diagram: Erik Cornelsen and Brant Snow
  - Help with examples: Cody Holdaway
  - UI: Nirav Adatiya, Tyson Wanlass, Brian Sevy
  - Sergiy Gorodnichiy
- Doba and CruxConnect
  - Jeff Fischer (MX), Dustin Chadwick (MX), Jason Weir (Owlet)
- UVU
  - Joshua Prince (second year CS student)
  - Melissa Snow (zoology)

Image credits

- https://upload.wikimedia.org/wikipedia/commons/f/fa/RNase_H_fix.png
- https://ebbailey.files.wordpress.com/2011/04/biodnabinding.png
- http://slideplayer.com/slide/7712010/25/images/47/Example:+10,000+years+ago+the+Colorado+River+separated+two+squirrel+populations..jpg
- https://cosbiology.pbworks.com/f/1268762052/13.03.GeographicIsolationMap.JPG
- http://larryfrolich.com/Evolution/NaturalSelection/adaptive_landscape_labelled.jpg
- https://cdn.zmescience.com/wp-content/uploads/2018/03/D095D0BAD0BED0BBD0BED0B3D0B8D187D0BDD0BE_D180D0B0D0B2D0BDD0BED0B2D0B5D181D0B8D0B53.jpg
- https://www.livescience.com/53822-dodo-birds-were-smart.html
- https://cf.ltkcdn.net/kids/images/std/239664-731x450-snail.jpg
- https://www.news-medical.net/life-sciences/What-is-Elastin.aspx
- https://i.ytimg.com/vi/1LqyOX12_nQ/maxresdefault.jpg
- https://harveyblackauthor.files.wordpress.com/2012/05/berlin-may-1945-0062.jpg
- https://digital.wwnorton.com/ebooks/epub/bionowcore/OPS/images/Pg_43.jpg
- https://farm3.staticflickr.com/2194/2351631527_8bc3f50263_z.jpg?zz=1
- https://www.brothers-brick.com/2008/11/26/kings-castle-a-crowning-achievement/
- https://ghr.nlm.nih.gov/art/large/chromosomalduplication.jpeg

Thank You!

Questions / comments?

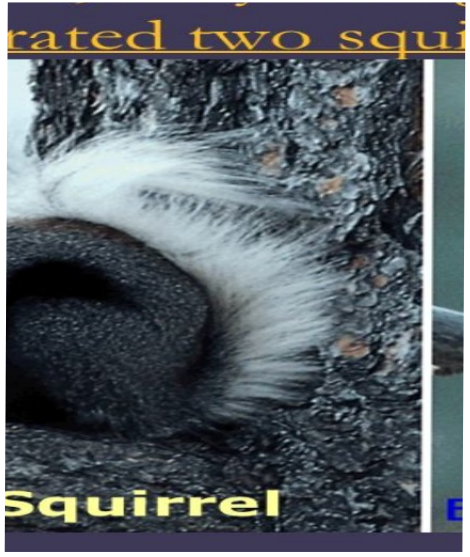
Appendix

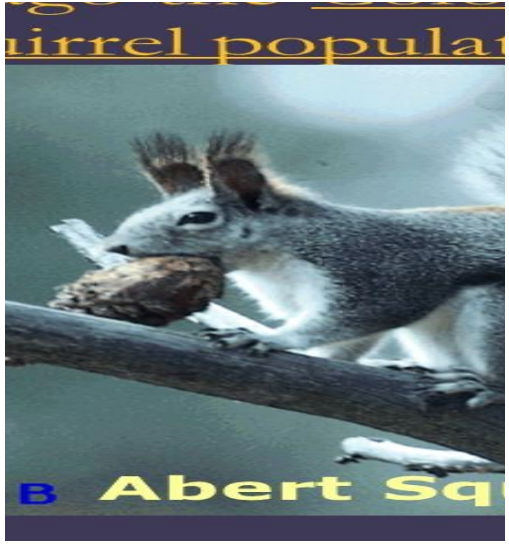
An eternal debate

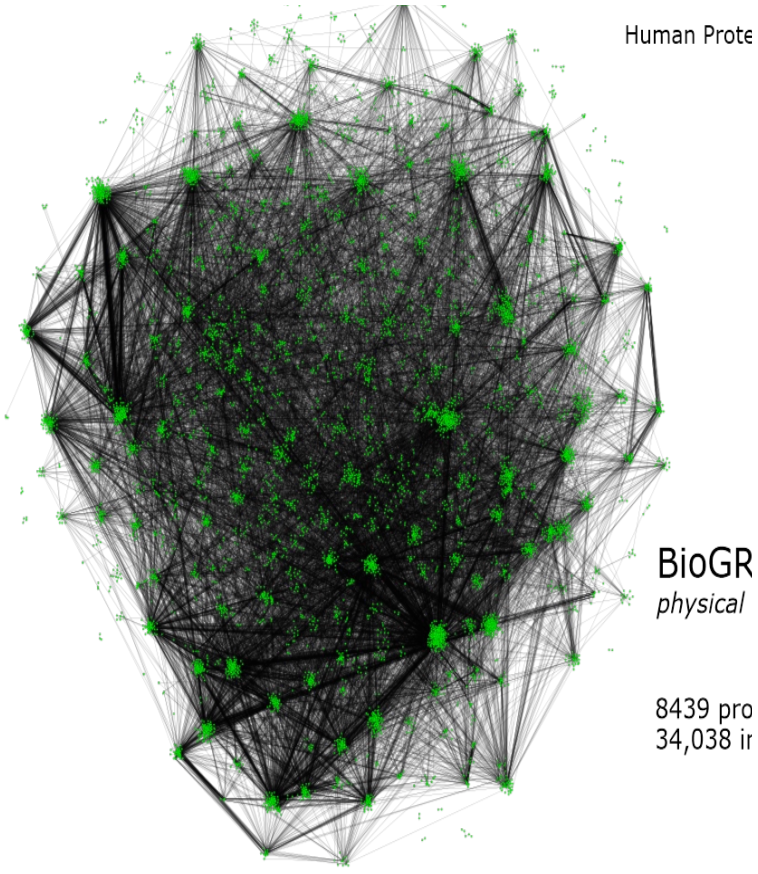
	New Jersey style [UNIX, Bell Labs]	MIT style [Multics]
Simplicity	No.1 consideration Implementation > Interface	Interface > Implementation
Correctness	mostly	100%
Consistency	mostly	100%
Completeness	de facto	mostly

https://www.slideshare.net/jserv/iot-microkernel

Squirrel speciation



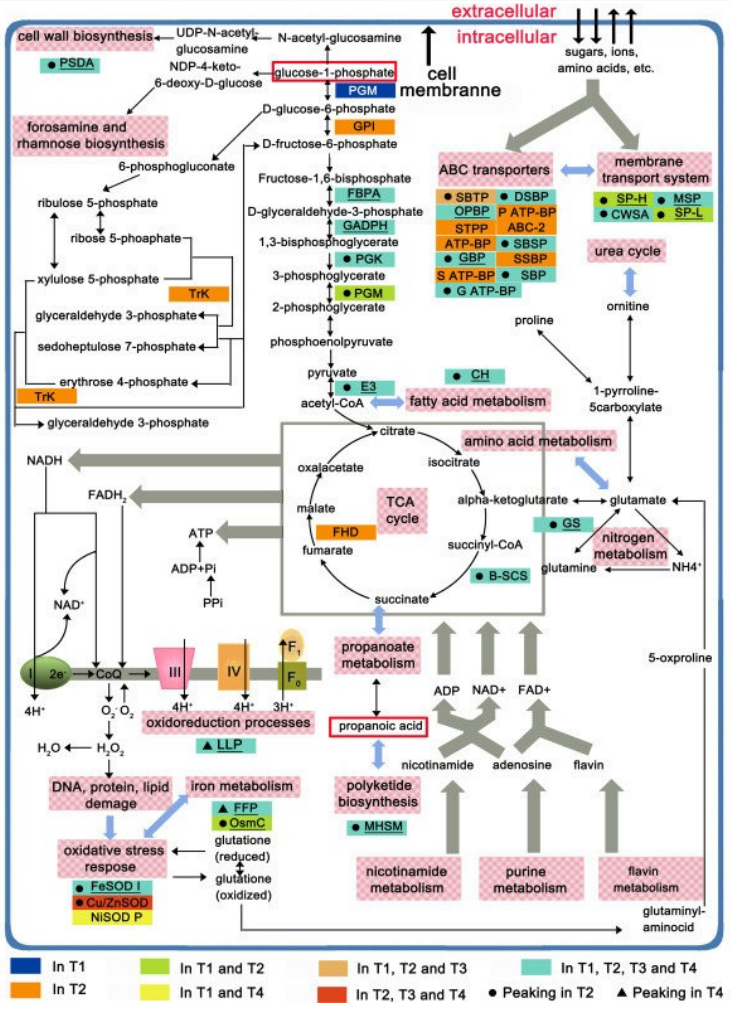




Human Prote

BioGR
physical

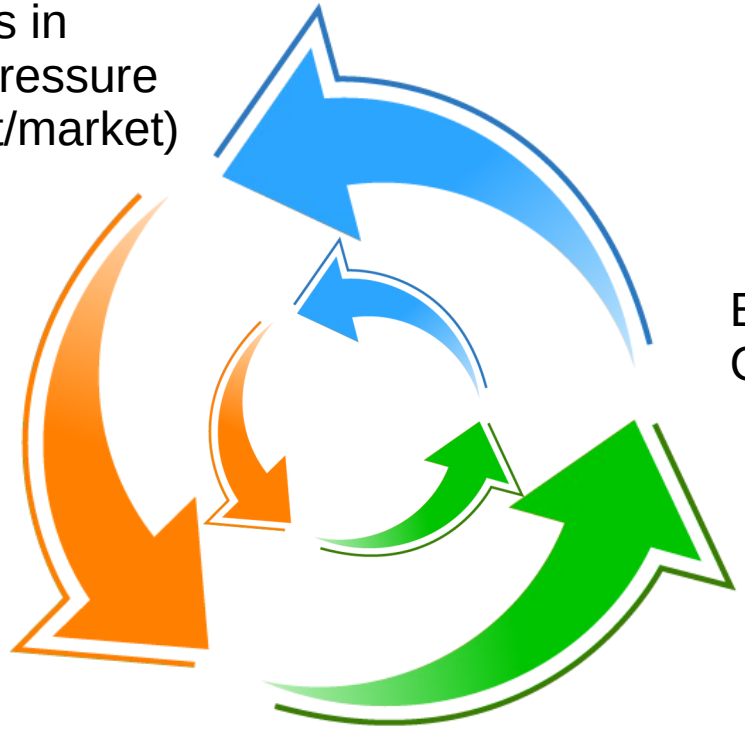
8439 pro
34,038 ir



Continuous Integration/Deployment

More iterations =~ fitter code

Code changes in
response to pressure
(management/market)



Bad code is discarded
Good code survives

Tests and code review
promote code fitness