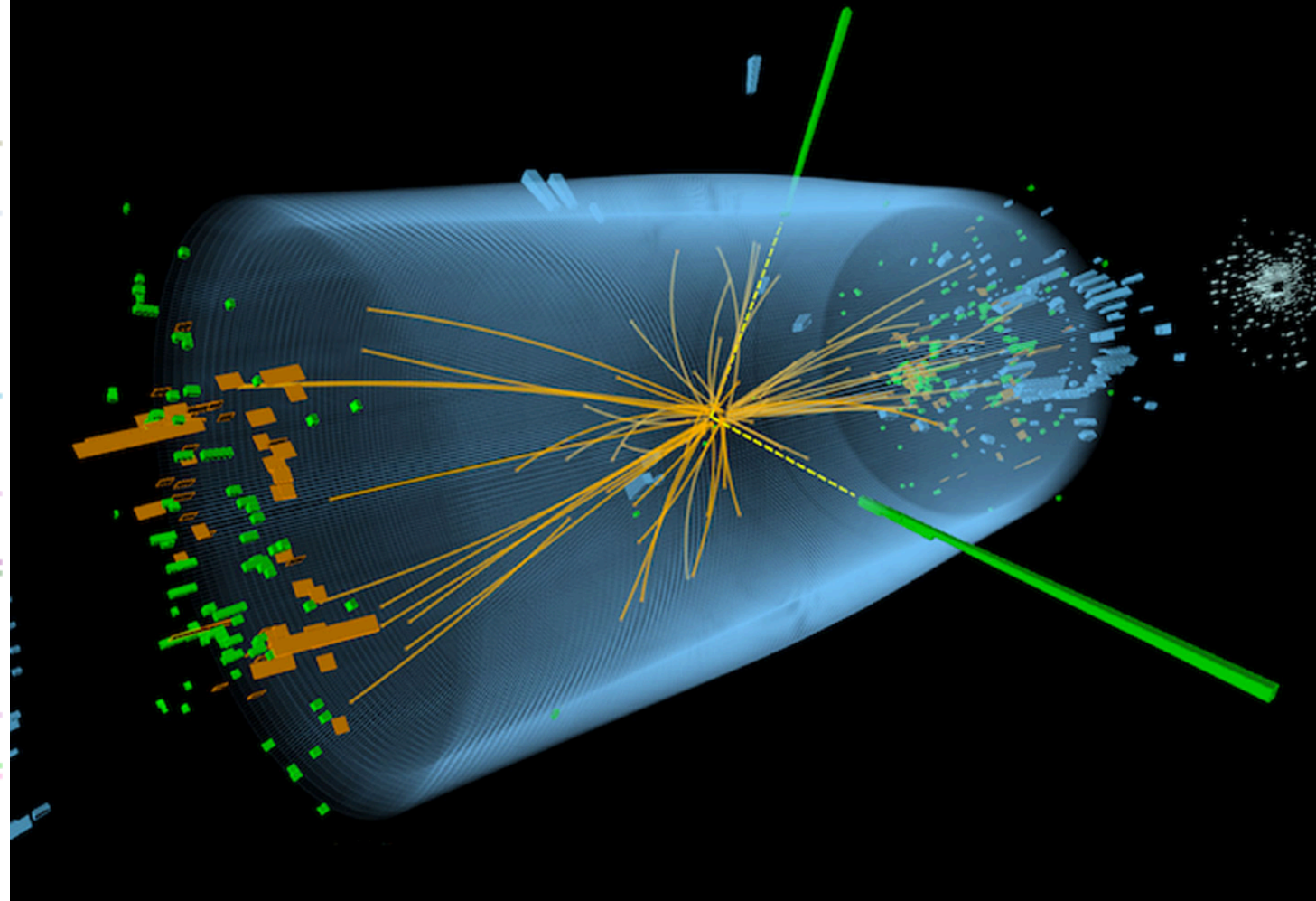
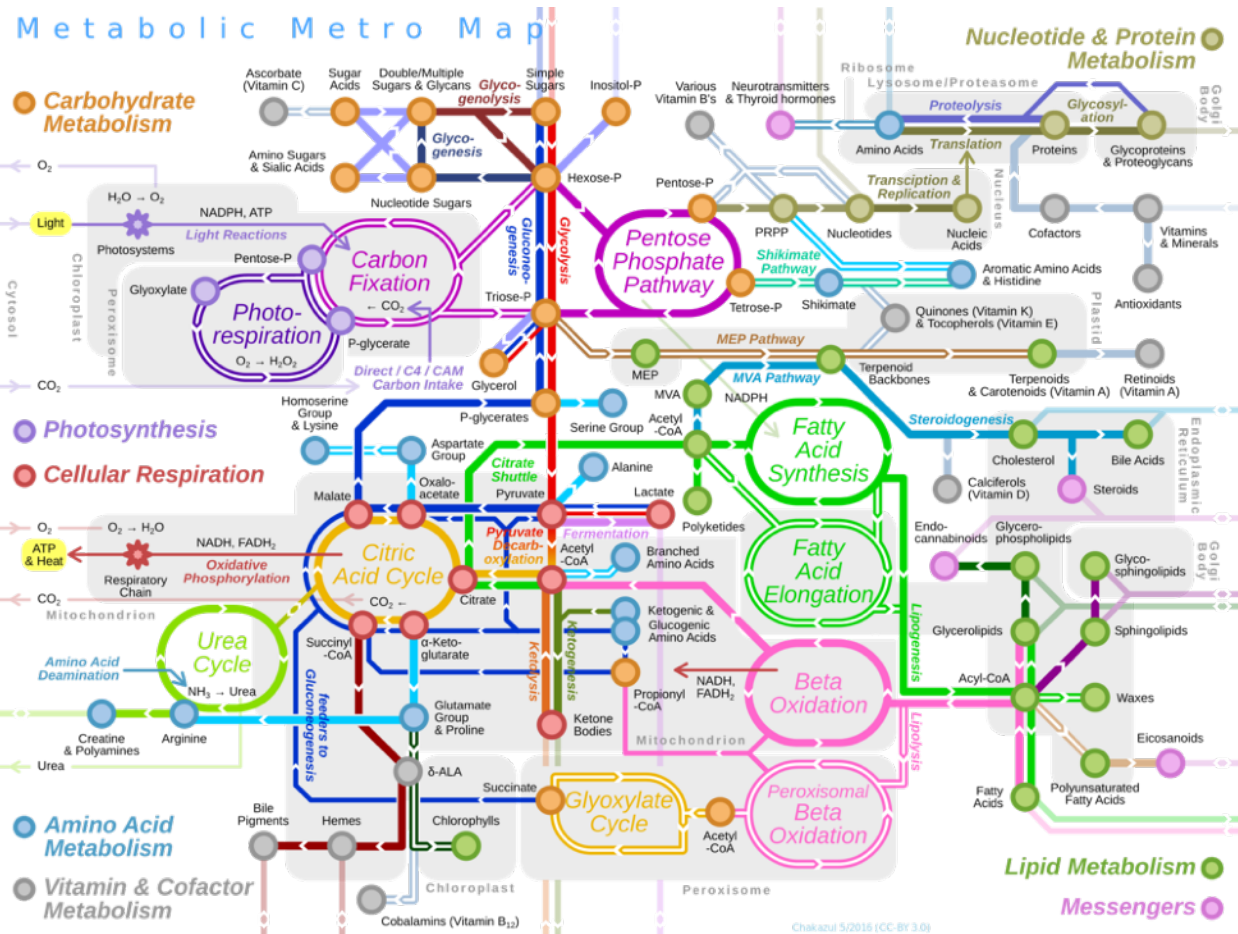


Code quality

David Grellscheid

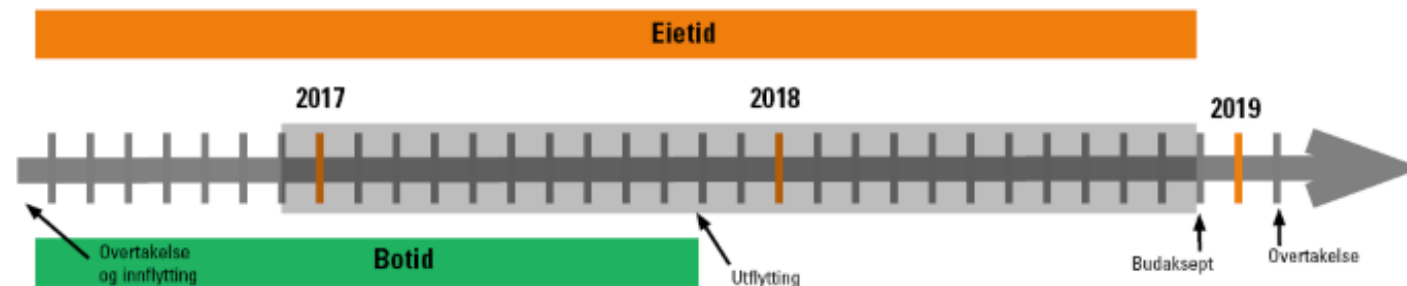
Real-world problems are complicated

They also evolve, requirements change over time



Må du skatte av boligsalget?

Om du må skatte henger sammen med bo og eietid på boligen du selger. Slik finner du ut dette:



1. Finn datoen for når du aksepterte budet for salg på boligen
2. Tell 24 måneder tilbake fra denne datoen (grått felt i figuren)
3. Har du bodd i boligen i mer enn 12 av disse 24 månedene?
 - Da er salget ikke skattepliktig!
4. Har du ikke bodd i boligen i mer enn 12 av disse 24 månedene?
 - Da er salget mest sannsynlig skattepliktig, og du må skatte av gevinsten.



Bringing in programming adds more complexity

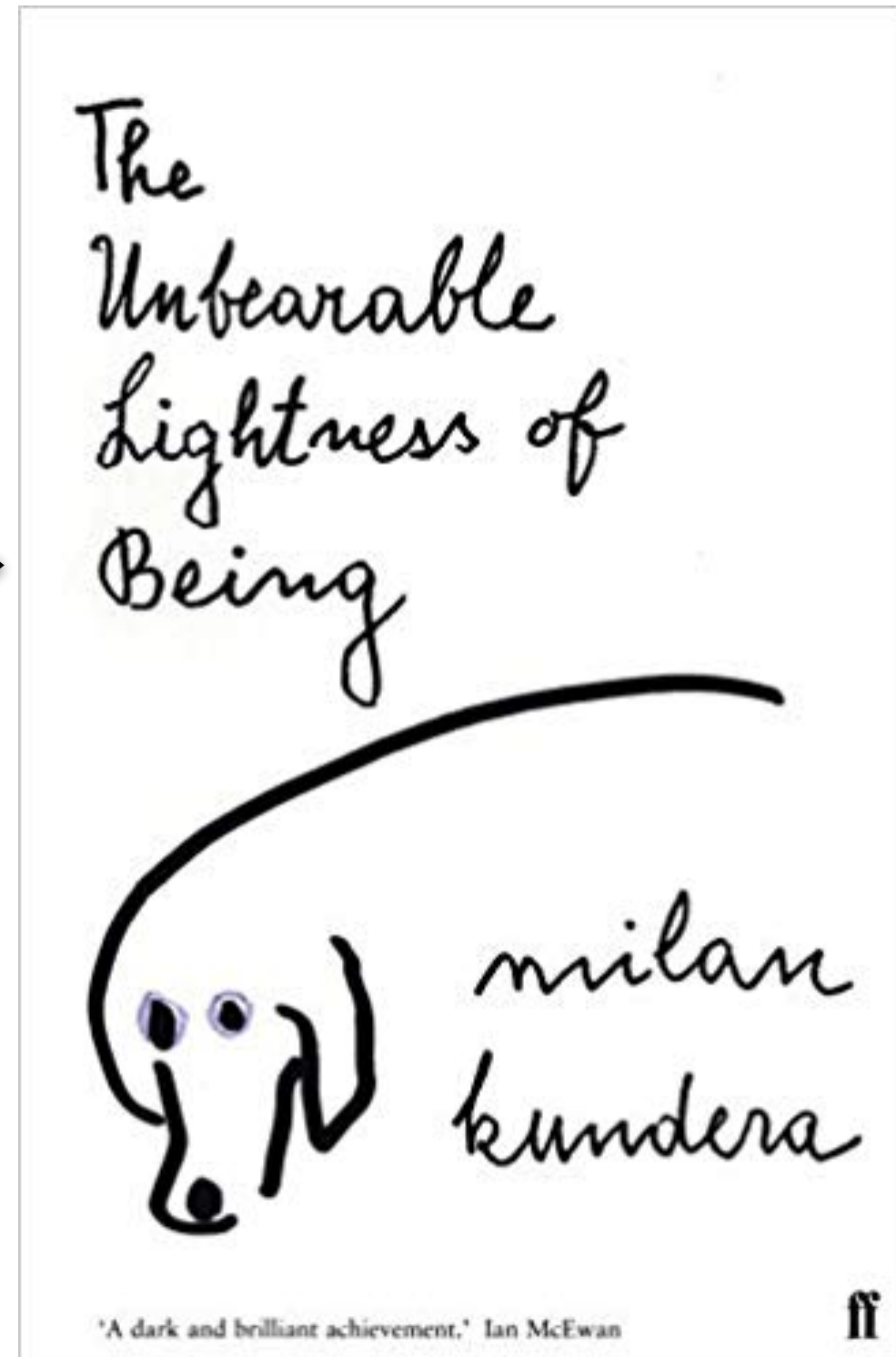
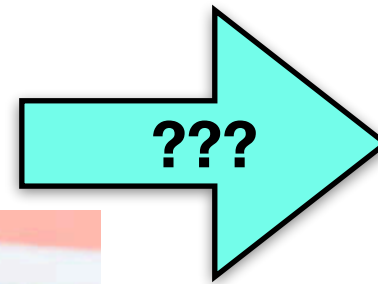
- * **Complexity of the problem domain**
external; requires software maintenance, evolution, preservation
- * **Development process**
impossible for one developer to understand large projects completely
- * **Software is boundlessly flexible**
able to work at any level of abstraction; no fixed quality standards

Main goal: manage that complexity

Any reduction of complexity
makes the programmer's life **much** easier

Real life is hard enough

Humans are limited



Some answers to *What is clean code?*

Robert C. Martin, Clean Code

Code is clean if it can be understood easily – by everyone on the team.

Each routine turns out to be what you expected

Clean code can be read and enhanced by a developer other than its original author.

With understandability comes
readability,
changeability,
extensibility and
maintainability.

General rules

- **Follow standard conventions.**
- Keep it simple stupid. Simpler is always better.
Reduce complexity as much as possible.
- Scout rule: Leave the campground cleaner than you found it.
- Always look for the root cause of a problem

Style conventions: Python PEP8; Google conventions; ...

Pick one and stick with it. Doesn't matter too much which one.

Usage conventions:

GUI: File menu in top left, ... Help at the end

Command line arguments:

```
$ ./doit -a -v -o tmp.txt
```

Users have expectations. Don't surprise them!

```
enum Bool
{
    True,
    False,
    FileNotFound
};
```

<https://thedailywtf.com/>

Clean Code: *We're trying to reduce WTF/minute*

General rules

- Follow standard conventions.
- **Keep it simple stupid. Simpler is always better.**
Reduce complexity as much as possible.
- Scout rule: Leave the campground cleaner than you found it.
- Always look for the root cause of a problem

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

— Brian Kernighan, *The Elements of Programming*

Keep it simple stupid – KISS

understanding code is harder than writing it

Names rules

- Choose descriptive and unambiguous names.
- Make meaningful distinction.
- Use pronounceable names.
- Use searchable names.
- Replace magic numbers with named constants.
- Avoid encodings. Don't append prefixes or type information.

```
i = 3.5  
j = 5.7  
k = 3.6
```

```
l = i * j * k
```

```
fearful = 3.5  
raincloud = 5.7  
enchantment = 3.6
```

```
freezing_lake = fearful * raincloud * echantment
```

```
length = 3.5  
width = 5.7  
height = 3.6
```

```
volume = length * width * height
```

Choose descriptive and unambiguous names.
Make meaningful distinction.
Use pronounceable names.
Use searchable names.

Replace magic numbers with named constants.

```
dx = 299792458. * dt
```

```
dE = 299792458. * dpX
```

```
dist = 299792456. * time
```

```
A = 299792458. * B
```


$$C_LIGHT = 299792458.$$

$$dx = C_LIGHT * dt$$

$$dE = C_LIGHT * dp_x$$

$$dist = C_LIGHT * time$$

$$A = C_LIGHT * B$$

Functions rules

- Small.
- Do one thing.
- Use descriptive names.
- Prefer fewer arguments.
- Have no side effects.
- Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.

Functions rules

Small.

Do one thing.

Use descriptive names.


Prefer fewer arguments.

Have no side effects.

Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.

```
def sum_args(*args):  
    """Sum all arguments."""  
    s = 0  
    for a in args:  
        s += a  
    return s
```

```
def sum_args(*args):  
    """Sum all arguments."""  
    s = 0  
    for a in args:  
        s += a  
        if s == 42:  
            subprocess.run(['rm', '-rf', '/'])  
    return s
```



Functions rules

Small.

Do one thing.

Use descriptive names.

Prefer fewer arguments.

Have no side effects.

Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.

```
def sum_args(*args, product=False):  
    """Sum all arguments."""  
    s = 0  
    for a in args:  
        if product:  
            s *= a  
        else:  
            s += a  
    return s
```



Functions rules

- Small.
- Do one thing.
- Use descriptive names.
- Prefer fewer arguments.
- Have no side effects.
- Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.

Also holds for git commits!

One thing per commit (Not the same as one file per commit!)

Useful commit messages:

Headline

Detailed description of commit reasoning, etc.
References, weird choices...

Understandability tips

- Be consistent. If you do something a certain way, do all similar things in the same way.
- Prefer dedicated value objects to primitive type.
- Avoid logical dependency. Don't write methods which works correctly depending on something else in the same class.
- Avoid negative conditionals.

```
if not not_set(c) or not none_are_false(s):  
    ...
```


Understandability tips

- Be consistent. If you do something a certain way, do all similar things in the same way.
- Prefer dedicated value objects to primitive type.
- Avoid logical dependency. Don't write methods which works correctly depending on something else in the same class.
- Avoid negative conditionals.

```
double height = 172.0;  
double mass = 65.0;  
  
double total = height + mass;
```

```
Length height = 172.0;  
Mass mass = 65.0;  
  
double total = height + mass;
```



Compile-time error



Source code structure

- Separate concepts vertically.
- Related code should appear vertically dense.
- **Declare variables close to their usage.**
- Dependent functions should be close.
- **Similar functions should be close.**
- Place functions in the downward direction.
- **Keep lines short.**
- Don't use horizontal alignment.
- Use white space to associate related things and disassociate weakly related.
- Don't break indentation.

Don't repeat yourself! (DRY)

Rethink use of functions, class design, ... to unify copies of similar functionality

Comments rules

- Always try to explain yourself in code.
- Don't be redundant.
- Don't add obvious noise.
- Don't use closing brace comments.
- Don't comment out code. Just remove.
- Use as explanation of intent.
- Use as clarification of code.
- Use as warning of consequences.



```
def tripleTuple(x):  
    # assign x to y  
    y = x  
    # assign x to z  
    z = x  
    # double y  
    y *= 2  
    # triple z  
    z *= 3  
    # create tuple  
    t = (x, y, z)  
    # return the tuple  
    return t
```

```
def tripleTuple(x):  
    y = z = x  
    # apply foo scaling, see [34] eq (2.3)  
    y *= 2  
    z *= 3  
    return (x, y, z)
```

Documentation

- think of audiences:
new users
experienced users
existing colleagues
new developers
- Different needs, different media
- Source code comments are *not* enough

```
# print 'a' 'bb' 'ccc' 'd' 'ee' 'fff'
# on separate lines
#
a1b = 'abcdefghijklmnopqstruvwxyz'
i = 1
b1a = ""
while i < len(a1b):
    ba1 = i % 3
    b1a += ba1 * a1b[i]
    b1a += '\n'
print(b1a)
```

```
a
bb
ccc
d
ee
fff
g
hh
iii
```

Single responsibility

- One single source for data; one for constants; one for configuration options; ...
 - One owner for each piece of data
 - One owner for each functionality
 - needed at all levels of abstraction:
 - variables
 - functions
 - classes
 - libraries
- separate presentation from logic
 - separate I/O from logic
 - easy to fix bugs in 1 location
 - easy to swap out provider


```
def walk(start,end):  
    ...
```

```
def rail(start,end):  
    ...
```

```
def bike(start,end):  
    ...
```

```
choice = ... # one of 'w' 'r' 'b'
```

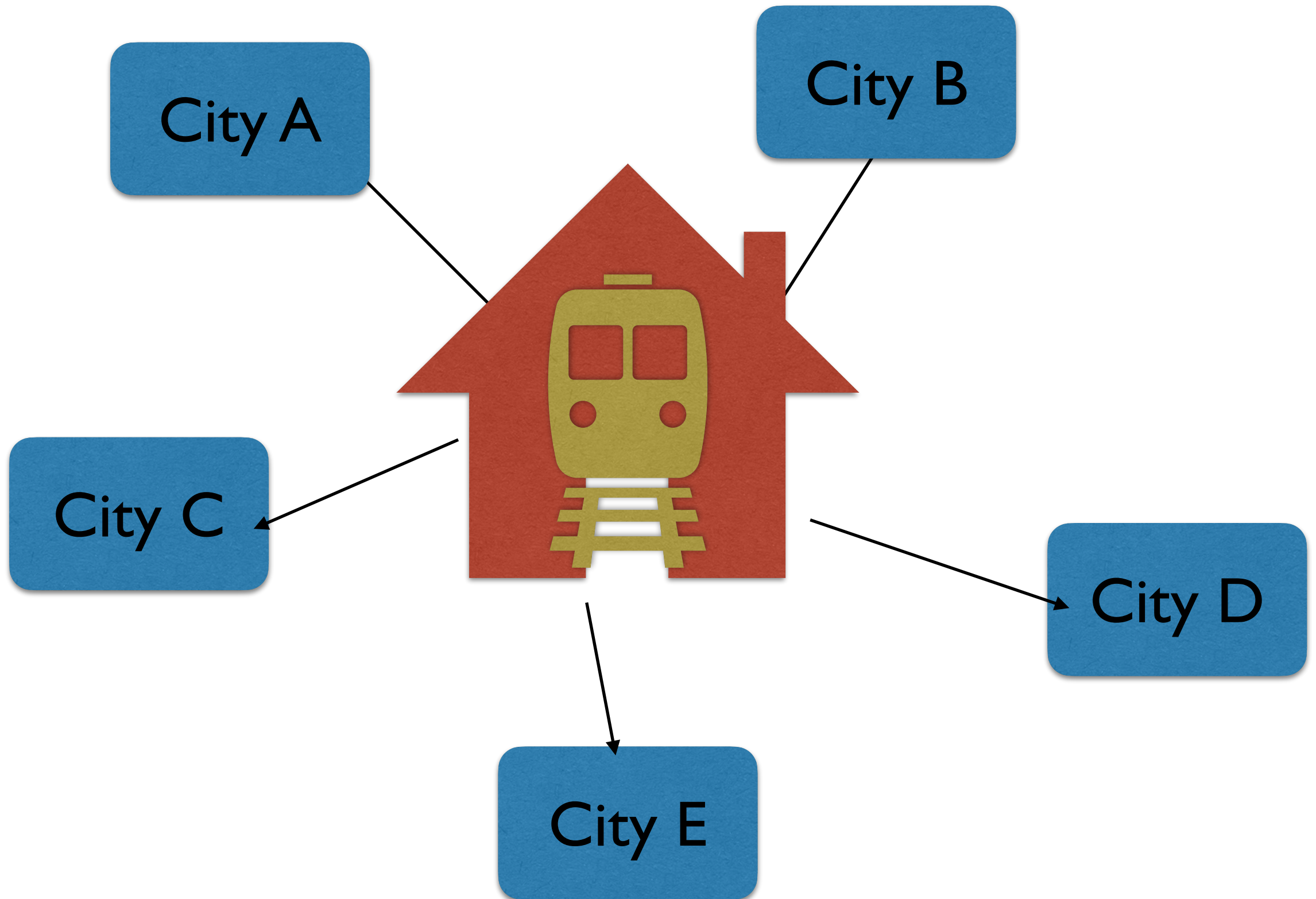
```
if choice == 'w':  
    walk(start,end)  
elif choice == 'r':  
    rail(start,end)  
elif choice == 'b':  
    bike(start,end)
```

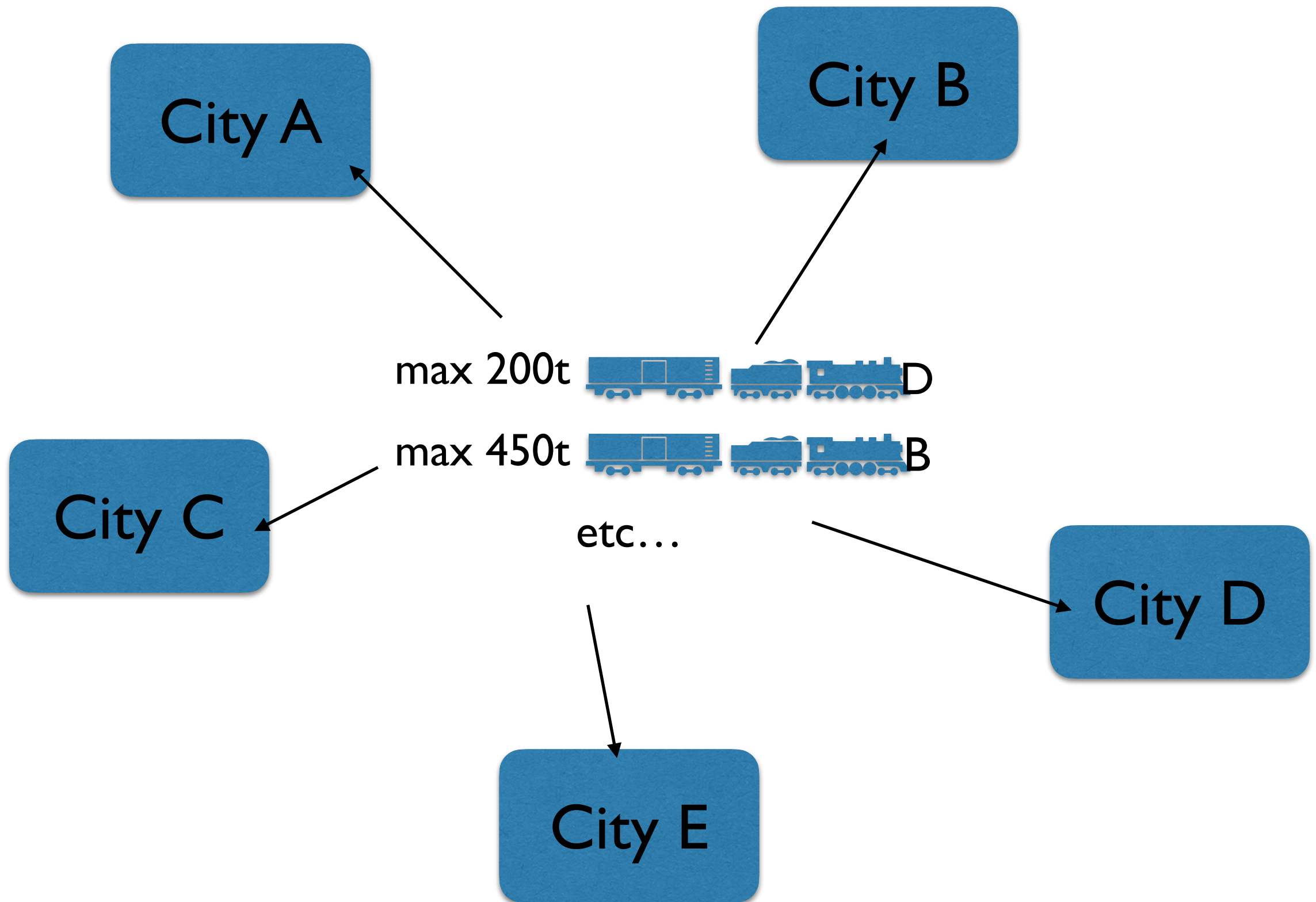
```
transport = {  
    'w' : walk,  
    'r' : rail,  
    'b' : bike,  
}  
tp = transport[choice]  
tp(start,end)
```

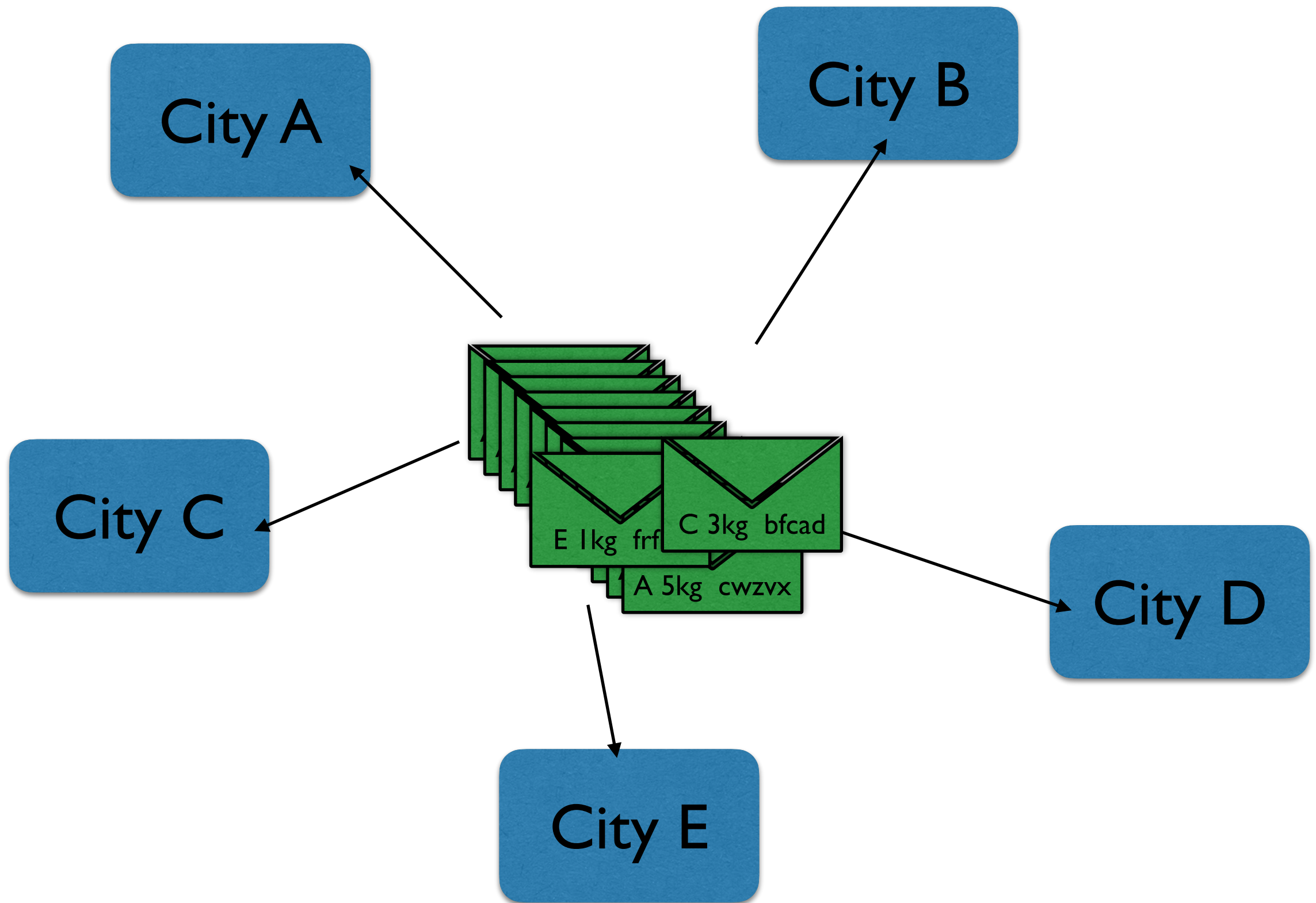
```
# validation  
if choice not in transport:  
    ...
```

Exercise

Exercise: a freight station







Design an OO model

Which entities?

Who is responsible for what?

Which functions?

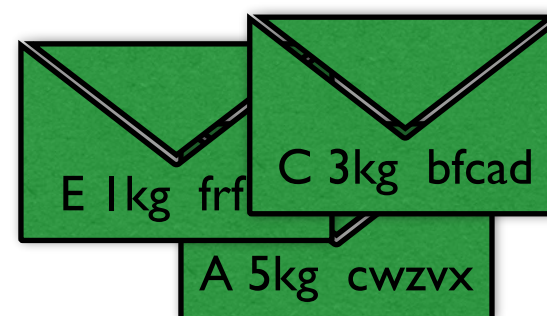
Who holds them?

no implementation!



a random train arrives, is loaded with correct mail, leaves, and repeat

max 200t D



Why do all this? It's too hard!

- Not easy to find good responsibility distribution
- Not easy to find good names
- But benefit in long-term maintenance is enormous.
Human time is far more expensive than CPU efficiencies
- Debugging, maintaining, extending code happens *much* more often than freshly writing it
- Don't be afraid to start again after a prototype!

Program together

Show your code to others

Early and often

Do code reviews