FactoryFactories and so can you

Chris Gerpheide, CTO Bespoke

chris@be-spoke.io @phoxicle

2020-01-25

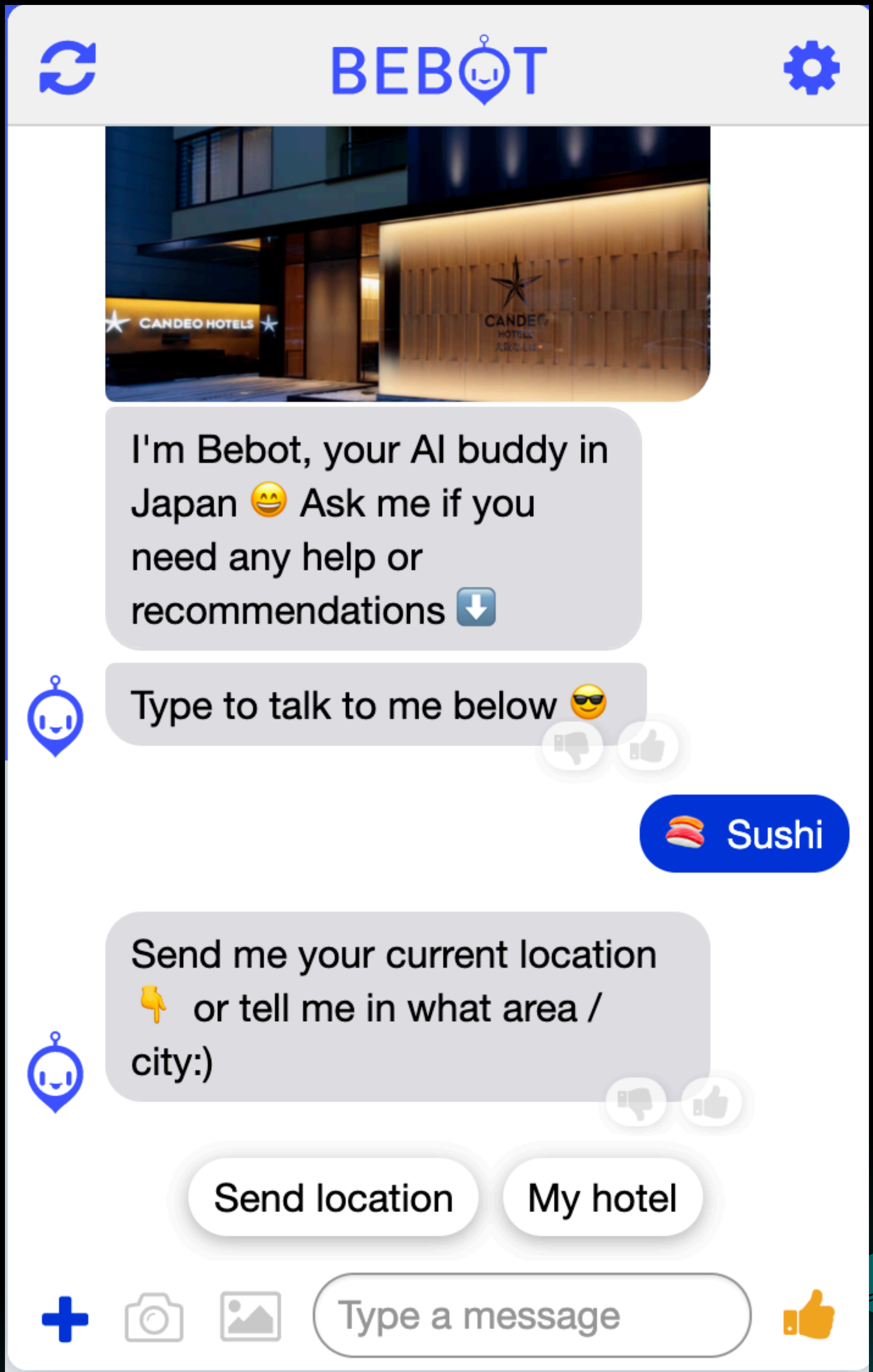
MLT Women in Machine Learning

Chris Gerpheide

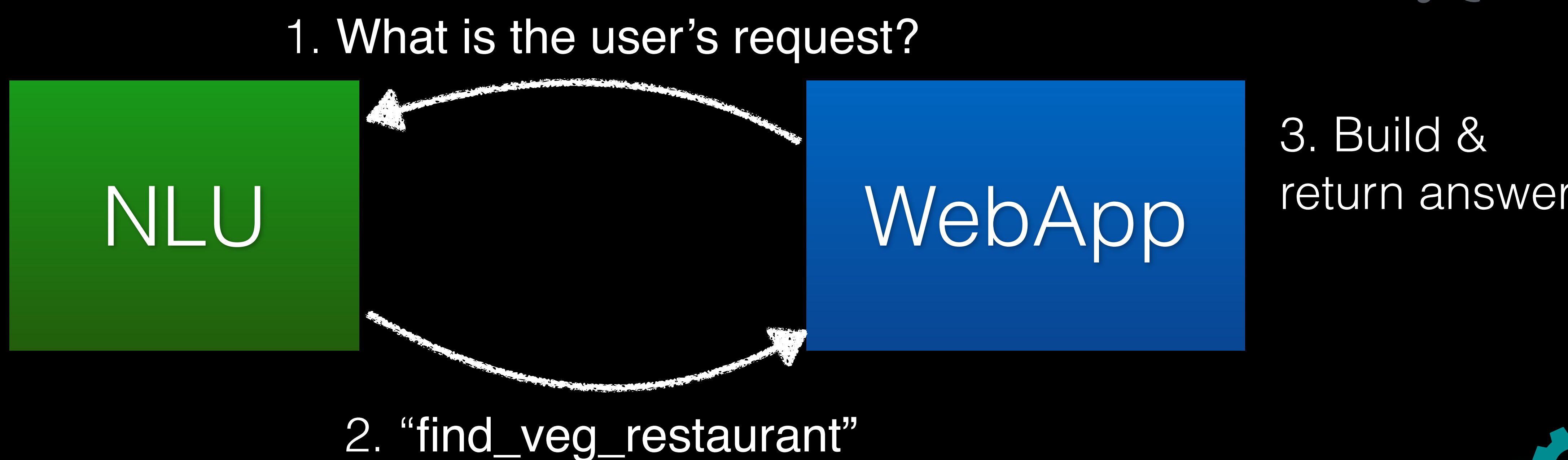
- CTO at Bespoke
- *Previous*:
  - B.A. Mathematics
  - M.Sc. Computer Science
  - 4 publications on software quality
  - Engineering Manager at AWS

Bebot

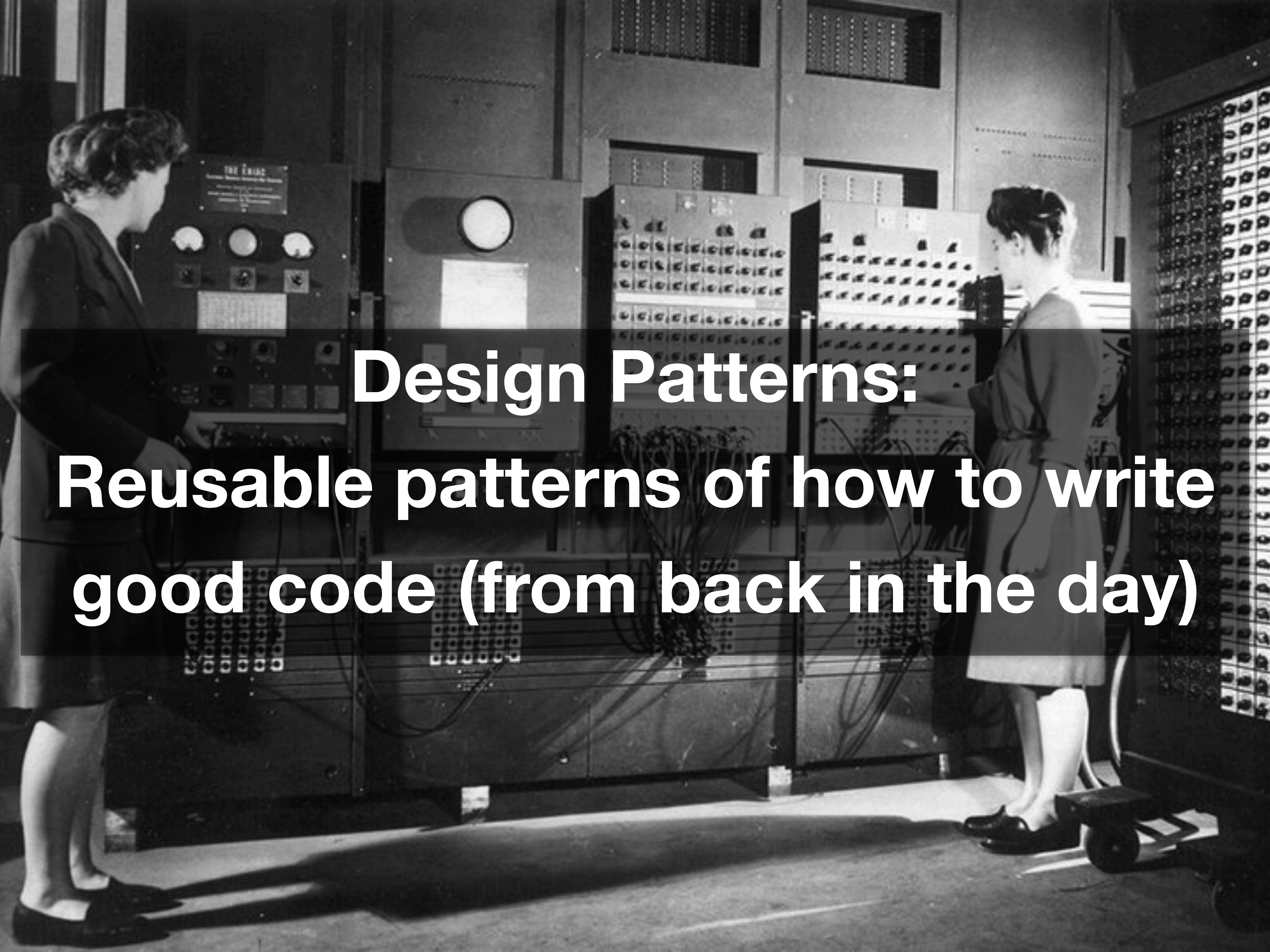
- Chatbot for tourists
- Custom bots for each client, users use the bot for free
- Clients include Narita airport, Tokyo Station, JP gov't (disaster relief), Tokyo metro
- Machine learning for NLU



Production code
Owned by **R&D team**



https://www.smithjournal.com.au/


Design Patterns:
Reusable patterns of how to write good code (from back in the day)

```
query = "SELECT " + t1_col1 + "," + t1_col2 + "," +
    t2_col1 + " FROM " + t1_name + " JOIN " +
    t2_name + " ON (" + t1_idcol + " = " + t2_idcol +
    ") ORDER BY " + t1_col1
```

```
query = "SELECT {},{},{} FROM {} JOIN {} ON ({} = {}) ORDER BY {}"
    .format(t1_col1, t1_col2, t2_col1, t1_name,
        t2_name, t1_idcol, t2_idcol,  t1_col1)
```

```
query = QueryBuilder(t1_name)
          .columns(t1_col1, t1_col2, t2_col1)
          .join(t2_name, t1_idcol, t2_idcol)
          .order_by(t1_col1)
          .build()
```

Focus on the ***UX*** of your code

(Or the Coder Experience, if you prefer)

- Spend time on this
- …And write a lot of tests

Object-oriented design patterns

- Builder
- Factory
- Facade
- Decorator
- Adapter

Builder (1/5)

Simplify creation of complex objects

```
query = "SELECT " + t1_col1 + "," + t1_col2 + "," +
    t2_col1 + " FROM " + t1_name  + " JOIN " +
    t2_name + " ON (" + t1_idcol + " = " + t2_idcol +
    ") ORDER BY " + t1_col1
```

```
query = "SELECT {},{},{} FROM {} JOIN {} ON ({} = {}) ORDER BY {}"
    .format(t1_col1, t1_col2, t2_col1, t1_name,
        t2_name, t1_idcol, t2_idcol,  t1_col1)
```

```
query = QueryBuilder(t1_name)
          .columns(t1_col1, t1_col2, t2_col1)
          .join(t2_name, t1_idcol, t2_idcol)
          .order_by(t1_col1)
          .build()
```

Keeps track of state

Return `self` to chain

```python
class QueryBuilder:

    def __init__(self, table):
        self.table = table
        self.cols = []
        self.joins = []
        self.orderbys = []

    def columns(self, *args):
        self.cols = ', '.join(args)
        return self

    def join(self, other_table, col, other_col):
        self.joins.append('JOIN {} ON {}={}'.format(other_table, col, other_col))
        return self

    def order_by(col):
        self.orderbys.append(col)
        return self
```

Validation

Merge all the state

```
class QueryBuilder:
    ...

    def build():
        # validation
        if not self.cols:
            raise Exception('Must specify columns')
        self.cols = unique(self.cols)

        # build
        q  = 'SELECT {} '.format(','.join(self.columns))
        q += 'FROM {} '.format(self.table)
        if self.joins:
            q += '{} '.format(' '.join(self.joins))
        if self.orderbys:
            q += ' ORDER BY {} '.format(','.join(self.orderbys))
        return q
```

Builder

```
model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

Factory (2/5)

Interface for creating related objects

```python
original_text = 'Hello! how are you todya? :)'

res = predict(original_text)
if not res:
    sanitized_text = make_ascii(original_text.lower())
    res = predict(sanitized_text)
if not res:
    res = predict(typo_correct(original_text))
if not res:
    san_and_typo_text = typo_correct(sanitized_text)
    res = predict(san_and_typo_text)

return res
```

```python
original_text = 'Hello! how are you todya? :)'

text_factory = TextFactory(original_text)

res = predict(original_text)
if not res:
    res = predict(text_factory.sanitized())
if not res:
    res = predict(text_factory.typo_corrected())
if not res:
    res = predict(text_factory.sanitized_and_typo_corrected())

return res
```

Nice to use with `@property` annotation too

```python
class TextFactory:

    def __init__(self, text):
        self.raw = text
    @property
    def sanitized(self):
        return make_ascii(self.raw.lower())
    @property
    def typo_corrected(self):
        return typo_correct(self.raw)
    @property
    def sanitized_and_typo_corrected(self):
        return typo_correct(self.sanitized())
```

Optional: Add caching

```python
class TextFactory:

    def __init__(self, text):
        self.raw = text

    def sanitized(self):
        if not self._sanitized:
            self._sanitized = make_ascii(self.raw.lower())
        return self._sanitized

    def typo_corrected(self):
        if not self._typo_corrected:
            self._typo_corrected = typo_correct(self.raw)
        return self._typo_corrected

    def sanitized_and_typo_corrected(self):
        if not self._sanitized_and_typo_corrected:
            self._sanitized_and_typo_corrected = typo_correct(self.sanitized())
        return self._sanitized_and_typo_corrected
```

Factory

```
original_text = 'Hello! how are you todya? :)'

text_factory = TextFactory(original_text)

res = predict(original_text)
if not res:
    res = predict(text_factory.sanitized())
if not res:
    res = predict(text_factory.typo_corrected())
if not res:
    res = predict(text_factory.sanitized_and_typo_corrected())

return res
```

Facade (3/5)

Unified, higher-level interface to another system

```
s3.write('some_s3_bucket', 'corpora/1234/corpus.json', corpus)
s3.write('some_s3_bucket', 'models/classifier_1234.gz',
        gzip(pickle.dump(classifier)))
db.insert('database1', 'current_model_id', 1234)

...

model_id = db.get('database1', 'current_model_id')
corpus = json.loads(s3.read('some_s3_bucket',
    'corpora/{}/corpus.json'.format(model_id)))
classifier = pickle.loads(gunzip(s3.read('some_s3_bucket'),
        'models/classifier_{}.gz'.format(model_id)))
...
prediction = classifier.classify('foo')
```

```
data_facade = DataFacade(1234)

data_facade.put_corpus(corpus)
data_facade.put_classifier(corpus)
data_facade.put_current_model_id()
...
model_id = data_facade.get_current_model_id()
corpus = data_facade.get_corpus()
classifier = data_facade.get_classifier()
```

```python
class DataFacade:

    S3_BUCKET = 'some_s3_bucket'
    DATABASE = 'database1'

    def __init__(self, model_id):
        self.model_id = model_id

    def get_corpus(self):
        return json.loads(s3.read(S3_BUCKET,
            'corpora/{}/corpus.json'.format(self.model_id)))

    def put_corpus(self, corpus):
        s3.write(S3_BUCKET, 'corpora/{}/corpus.json'
            .format(self.model_id), corpus)

    def get_current_model_id(self):
        return db.get(DATABASE, 'current_model_id')

    def put_current_model_id(self):
        db.insert(DATABASE, 'current_model_id', self.model_id)

    def get_classifier(self):
        return pickle.loads(gunzip(s3.read(S3_BUCKET),
                'models/classifier_{}.gz'.format(self.model_id)))

    def put_classifier(self, classifier):
        s3.write(S3_BUCKET, 'models/classifier_{}.gz'.format(self.model_id),
                gzip(pickle.dump(classifier)))
```

Different services

Handles file names + types

Facade

```
data_facade = DataFacade(1234)

data_facade.put_corpus(corpus)
data_facade.put_classifier(corpus)
data_facade.put_current_model_id()
...
model_id = data_facade.get_current_model_id()
corpus = data_facade.get_corpus()
classifier = data_facade.get_classifier()
```

Decorator (4/5)

Attach additional responsibilities to an object dynamically

```python
def train():

    start_time = time()

    model.train()

    print("Finished training in {} secs".format(time() - start_time ))
```

```python
@timer
def train():
    model.train()
```

```python
def timer(func):

    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time()
        value = func(*args, **kwargs)
        print("Finished {} in {} secs"
                .format(func.__name__, time() - start_time))
        return value

    return wrapper_timer
```

Decorator

```
@timer
def train():
    model.train()
```

Adapter (5/5)

Convert an object to an easier-to-use class

```
{
    'meta': {
        'id': 1234
        'date': '2020/01/25'
    },
    'cases': [
        [
            {
                'text': 'I really like pancakes',
                'source': 'twitter'},
            {
                'value': 'positiveSentiment',
                'labeller': 'Chris'
            },
        ],
        ...
    ]
}
```

```
data = json.loads(download_raw_data())

print('Processing ID {}'.format(data['meta']['id']))

inputs = []
labels = []

for item in data['cases']:
    if len(item) > 2:
        if 'text' in item[0] and 'value' in item[1]:
            cleaned_input = clean(item[0]['text'])
            inputs.append(cleaned_input)
            labels.append(item[1]['value'])

model.train(inputs, labels)
```

BESPOKE

```
data [illegible]a())
corpu[illegible] = CorpusJsonAdapter(data)

print[illegible]rpus.id))

model.train(corpus.inputs, corpus.labels)
```

or

```
data [illegible]ta())
corpu[illegible] = Corpus.from_json(data)

print[illegible]orpus.id))

model.train(corpus.inputs, corpus.labels)
```

All transformation logic in own class

Then work with this

```
class Corpus:

    def __init__(self, id):
        self.id = None
        self.inputs = []
        self.labels = []
```

```
class CorpusJsonAdapter(Corpus):

    def __init__(self, data):
        self.id = data['meta']['id']

        for item in data['cases']:
            if len(item) > 2:
                if 'text' in item[0] and 'value' in item[1]:
                    cleaned_input = clean(item[0]['text'])
                    self.inputs.append(cleaned_input)
                    self.labels.append(item[1]['value'])
```

Transformation logic in own method

```
class Corpus:

    def __init__(self, id):
        self.id = None
        self.inputs = []
        self.labels = []

    @staticmethod
    from_json(data):
        corpus = Corpus()
        corpus.id = data['meta']['id']

        for item in data['cases']:
            if len(item) > 2:
                if 'text' in item[0] and 'value' in item[1]:
                    cleaned_input = clean(item[0]['text'])
                    corpus.inputs.append(cleaned_input)
                    corpus.labels.append(item[1]['value'])
```

```python
corpus = CorpusJsonAdapter(data)
```

or

```python
corpus = Corpus.from_json(data)
```

or even...

```python
data_obj = RawData(data)
corpus = data_obj.to_corpus()
```

BESPOKE

Nested classes for all sub-structures

```python
class RawData:

    def __init__(self, data):
        self.meta = RawData.Meta(data['meta'])
        self.cases = [RawData.Case(d) for d in data['cases']]

    class Meta:
        def __init__(self, data):
            self.id = data['id']

    class Case:

        def __init__(self, data):
            self.textinfo = RawData.Case.Text(data[0])
            self.valueinfo = RawData.Case.Value(data[1])

            class Text:
                def __init__(self, data):
                    self.text = data.get('text')
                    self.source = data.get('source') # not used

            class Value:
                def __init__(self, data):
                    self.value = data.get('value')
                    self.labeller = data.get('labeller') # not used
```

(in all its glory)

```python
class RawData:
    ...

    def to_corpus():
        corpus = Corpus(self.meta.id)

        for case in self.cases:
            if case.textinfo.text and case.valueinfo.value:
                cleaned_input = clean(case.textinfo.text)
                corpus.inputs.append(cleaned_input)
                corpus.labels.append(case.valueinfo.value)
```

Adapter

```
corpus = CorpusJsonAdapter(data)
```

```
corpus = Corpus.from_json(data)
```

```
data_obj = RawData(data)
corpus = data_obj.to_corpus()
```

Summary

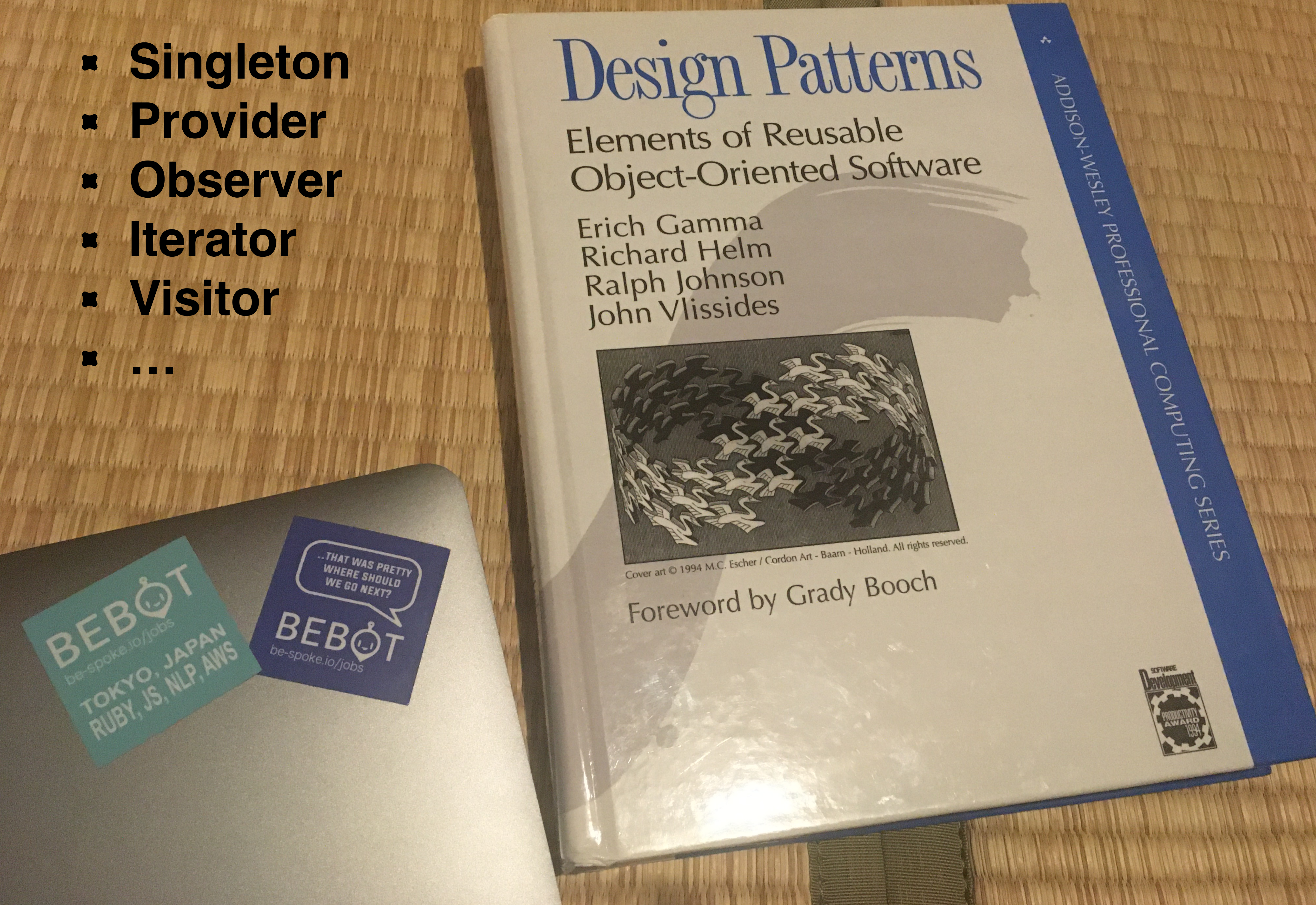
- Builder
- Factory
- Facade
- Decorator
- Adapter

BESPOKE

Focus on the UX of your code

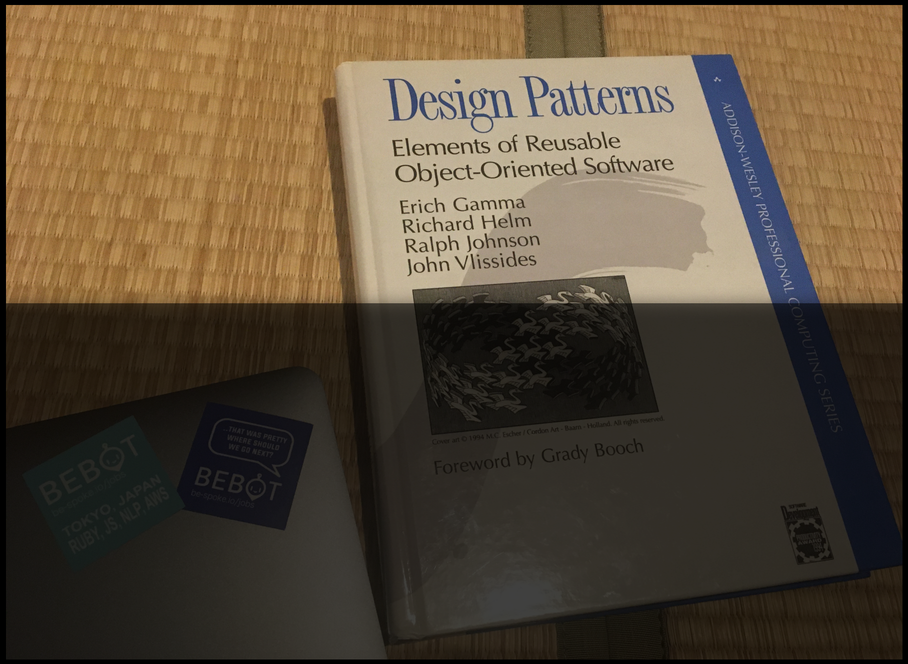
```python
@timer
def classify(text):
    query = QueryFactory(text)

    pipeline = Pipeline()
        .naive_bayes_classifier(query.typo_corrected)
        .string_similarity_classifier(query.sanitized)
        .naive_bayes_classifier(query.named_entities_substituted)
        .rnn_classifier(query.sanitized)

    return pipeline.first_with_probability_above(.80)
```

- Singleton
- Provider
- Observer
- Iterator
- Visitor
- …



Senior programmer

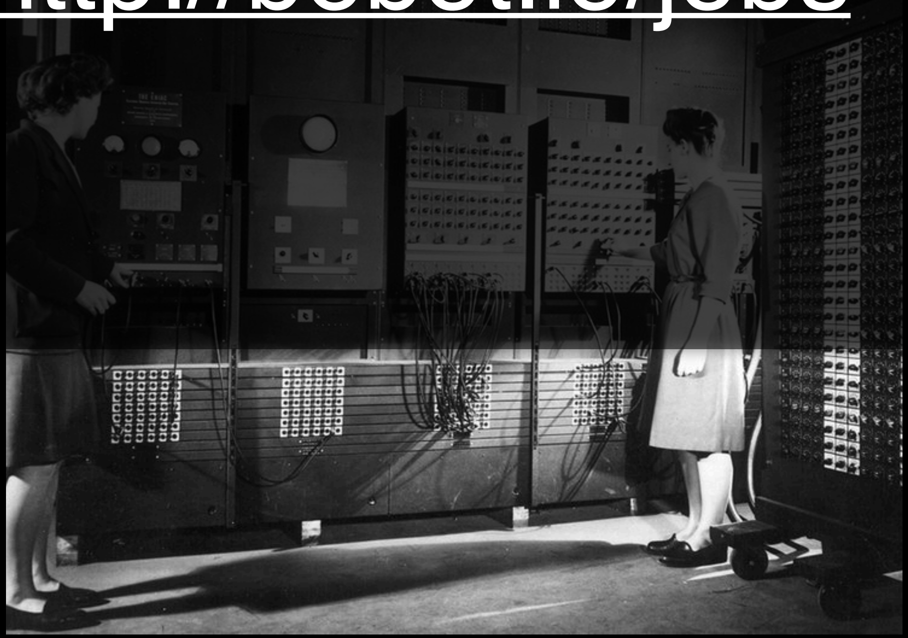
Junior programmer



We’re hiring!

- R&D Team Lead
- R&D Engineers

http://bebot.io/jobs



FactoryFactories and so can you

Chris Gerpheide, CTO Bespoke
chris@be-spoke.io @phoxicle
2020-01-25
MLT Women in Machine Learning