JavaScript Spessore

A Thick Shot of Objects, Metaobjects, & Protocols

by Reginald "raganwald" Braithwaite

JavaScript Spessore

A Thick Shot of Objects, Metaobjects, & Protocols

Reginald Braithwaite

Also By Reginald Braithwaite

Kestrels, Quirky Birds, and Hopeless Egocentricity

What I've Learned From Failure

How to Do What You Love & Earn What You're Worth as a Programmer

CoffeeScript Ristretto

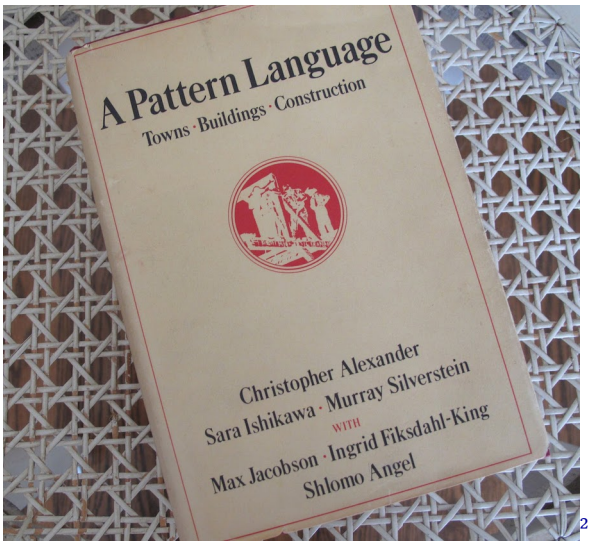
JavaScript Allongé

Contents

Prefaces

Foreword

There are many ways to write books about Architecture. One might attempt to catalogue a broad variety of problems and their solutions, at scales ranging from designing communities down to items of furniture, as Christopher Alexander did in A Pattern Language[1]:



[2]

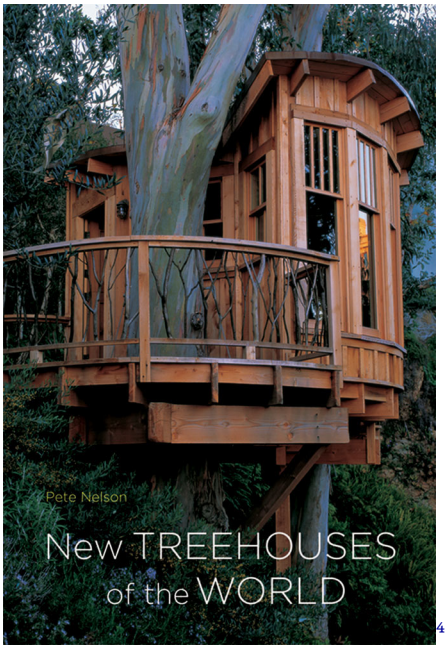
New Treehouses of the World

One might also pick a particular domain and then survey how different architects approached solving a common set of problems, highlighting the variety of possible styles. New Treehouses of the World[3] takes this approach:

[1] http://www.amazon.com/gp/product/0195019199/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0195019199&linkCode=as2&tag=raganwald001-20

[2] http://www.amazon.com/gp/product/0195019199/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0195019199&linkCode=as2&tag=raganwald001-20

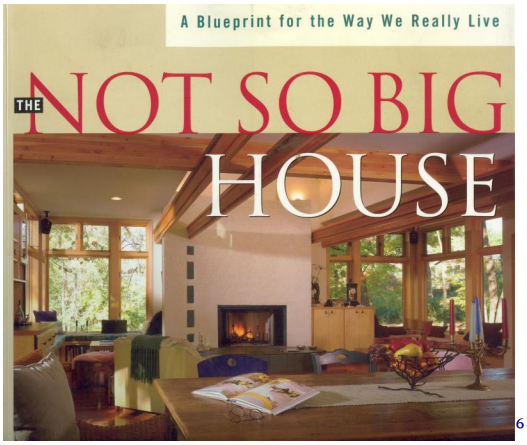
[3] http://www.amazon.com/gp/product/0810996324/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0810996324&linkCode=as2&tag=raganwald001-20



[4]

The Not So Big House

Or one might approach architecture from a single viewpoint, describing a highly specific and coherent style backed by personal experience, as Sarah Susanka did in The Not So Big House[5]:



[6]

[4] http://www.amazon.com/gp/product/0810996324/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0810996324&linkCode=as2&tag=raganwald001-20

[5] http://www.amazon.com/gp/product/1600851509/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=1600851509&linkCode=as2&tag=raganwald001-20

[6] http://www.amazon.com/gp/product/1600851509/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=1600851509&linkCode=as2&tag=raganwald001-20

JavaScript Spessore

JavaScript Spessore[7] follows this path. This book will not attempt to dictate the "one true definition" of object-oriented programming. It will not survey and catalog the many different programming languages and libraries people have used to solve software problems with objects.

Instead, JavaScript Spessore describes one way to design programs with objects as the central idea: Building software on top of a metaobject protocol. A metaobject protocol is a kind of abstraction for implementing object-oriented programming semantics. One program might be developed with pattern matching for methods, another might use state machines, a third might be heavily factored into aspects.

By implementing this metaobject protocol and semantics on top of it, we gain insight into how these semantics work and how they might be applied to solving problems we encounter as software developers.

[7] https://leanpub.com/javascript-spessore

Taking a page out of LiSP

Teaching Lisp by implementing Lisp is a long-standing tradition. We read book after book, lecture after lecture, blog post after blog post, all explaining how to implement Lisp in Lisp. Christian Queinnec's Lisp in Small Pieces[8] ("**LiSP**") is particularly notable, not just implementing a Lisp in Lisp, but covering a wide range of different semantics within Lisp.

LiSP's approach is to introduce a feature of Lisp, then develop an implementation. The book covers [Lisp-1 vs. Lisp-2[9], then discusses how to implement namespaces, building a simple Lisp-1 and a simple Lisp-2. Another chapter discusses scoping, and again you build interpreters for dynamic and block scoped Lisps.

Building interpreters (and eventually compilers) may seem esoteric compared to tutorials demonstrating how to build a blogging engine, but there's a method to this madness. If you implement block scoping in a "toy" language, you gain a deep understanding of how closures really work in any language. You gain some insight into the implications with respect to memory and performance. If you write a Lisp that rewrites function calls in Continuation Passing Style[10], you can't help but feel comfortable using JavaScript callbacks in Node.js[11].

The simple fact is that *implementing* a language feature teaches you a tremendous amount about how the feature works in a relatively short amount of time. And that goes double for implementing variations on the same feature–like dynamic vs block scoping or single vs multiple namespaces.

That being said, you get the most mileage out of implementing language semantics, not language syntax. Semantics are the *meanings* of programs, syntax is merely the appearance. Writing parsers for both `compose = (a, b) -> (c) -> a(b(c))` in CoffeeScript and `function comnpose (a, b) { return function (c) { return a(b(c)); }; }` in JavaScript wouldn't teach you nearly as much as writing the code that implements closures. And it's the closures that make `compose` work the way it does.

In this book, we are going to implement a number of different programming language semantics, all in JavaScript. We won't be choosing features at random; We aren't going to try to implement every possible type of programming language semantics. We won't explore dynamic vs block scoping, we won't implement call-by-name, and we will ignore the temptation to experiment with lazy evaluation.

We *are* going to implement different object semantics, implement different kinds of metaobjects, and implement different kinds of method protocols. We are going to focus on the semantics of objects, metaobjects, and protocols, because we're interested in understanding "object-oriented programming" and all of its rich possibilities.

---

[8] http://www.amazon.com/gp/product/B00AKE1U6O/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=B00AKE1U6O&linkCode=as2&tag=raganwald001-20

[9] A "Lisp-1" has a single namespace for both functions and other values. A "Lisp-2" has separate namespaces for functions and other values. To the extend that JavaScript resembles a Lisp, it resembles a Lisp-1. See The function namespace.

[10] https://en.wikipedia.org/wiki/Continuation-passing_style

[11] http://nodejs.org/about/

In doing so, we'll learn about the principles of object-oriented programming in far more depth than we would if we chose to implement a "practical" example like a blogging engine.

Disclaimer

Writing is a journey, not a destination. This sample documents the direction we're facing as we take the next step.

This sample "document" is provided to illustrate direction JavaScript Spessore[12] is taking. **It is not held out to contain any of the actual book's content**. "Purchases" are being offered to people whose primary motivation is to support the book and encourage me to get it done.

Your purchase does include the right to download any and all versions of the book, as per Leanpub's lean publishing model. You will never be asked to pay more. You have the right to a 100% no-questions-asked refund if you are not satisfied with the progress of the book:

> Within 45 days of purchase you can get a 100% refund on any Leanpub purchase, in two clicks. We process the refunds manually, so they may take a few days to show up. See full terms[13].

> If you buy a Leanpub book you get all the updates to the book for free! All books are available in PDF, EPUB (for iPad) and MOBI (for Kindle). There is no DRM. There is no risk, just guaranteed happiness or your money back.

That being said, writing like this cannot be rushed: progress may be slow relative to a "Learn JavaScript OO in 21 Days" type of book. If you would prefer that there be a substantial amount of work completed, please be patient and check back in a month or so.

Please also bear in mind that pricing and bundling may vary over time. The book may be offered at any price in the future or even be free to read or free to share at some point in the future.

[12] https://leanpub.com/javascript-spessore
[13] https://leanpub.com/terms#returns

What is Object-Oriented Programming?

"OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."

—Dr. Alan Kay

What is an Object?

Consider Smalltalk's definition of an object:

> A Smalltalk object can do exactly three things: Hold state (references to other objects), receive a message from itself or another object, and in the course of processing a message, send messages to itself or another object.â€"Smalltalk on Wikipedia[14]

Objects seem simple enough: They hold state, they receive messages, they process those messages, and in the course of processing messages, they also send messages. This brief definition implies an important idea: Objects can *change state* in the course of processing messages. They do this directly by removing, changing, or adding to the references they hold.

messages and method invocations

A nine year-old messenger boy

Smalltalk speaks of "messages." The metaphor of a message is very clear. A message is composed and sent from one entity (the "sender") to one entity (the "recipient"). The sender specifies the identity of the recipient. The message may contain information for the recipient, instructions to perform some task, or a question to be answered. An immediate reply may be requested, or the sender may trust the message's recipient to act appropriately. The metaphor of the "message" emphasizes the arms-length relationship between sender and recipient.[15]

[14] https://en.wikipedia.org/wiki/Smalltalk

[15] More exotic messaging protocols are possible. Instead of a message being couriered from one entity to another entity, it could be posted on a public or semi-private space where many recipients could view it and decide for themselves whether to respond. Or perhaps there is a dispatching entity that examines each message and decides who ought to respond, much as an operator might direct your call to the right person within an organization.

Popular languages don't usually discuss messages. Instead, they speak of "invoking methods." The word "invoke" means to conjure up and to declare in law. "Invoking a method" has a much more imperative implication than "sending a message." It implies that the entity doing the invoking knows exactly what is going to happen. The relationship is not arms-length.

A method is a kind of recipe. It represents how an object will respond to a message. In JavaScript, methods are functions. In other languages, methods are a separate kind of thing than functions.

references and state

You can imagine sending a message to a mathematician: "What's the biggest number: one, five, or four?" You can do that is JavaScript:

```
1  Math.max(1, 5, 4)
2    //=> 5
```

Although this is a message, it is unsatisfying to think of `Math` as an object, because it doesn't have any state. Objects were invented fifty years ago[16] to model entities when building simulations. When making a simulation, an essential design technique is to make provide entity with its own independent decision-making ability.

Let's say we're modeling traffic. In real life, each car has its own characteristics like maximum speed. Each car has a driver with their own particular style of driving, and of course different cars have different destinations and perhaps different senses of urgency. Each driver independently responds to the local situation around their car. The same goes for traffic lights, roads… Everything has its own independent behavior.

The way to build a simulation is to provide each simulated entity such as the cars, roads, and traffic lights, with their own little programs. You then embed them in a simulated city with a supervisory program doling out events such as rain. Finally, you start the simulation and see what happens.

The key need is to be able to have entities be independent decision-making units that respond to events from outside of themselves. In essence, we're describing computing units. Computation has, at its heart, a program and some kind of storage representing its state. Although impractical, each entity in a simulation could be a Turing Machine with a long tape.

And thus when we think of "objects," we think of independent computing devices, each with their own storage representing their state: **An object is an entity that use handlers to respond to messages**. **It maintains internal state**, **and its handlers are responsible for querying and/or updating its state**.

We'll have a closer look at implementing objects in Javascript a little later in this book.

[16] https://en.wikipedia.org/wiki/Simula "The Simula Programming Language"

What is a Metaobject?

> In computer science, a metaobject is an object that manipulates, creates, describes, or implements other objects (including itself). The object that the metaobject is about is called the base object. Some information that a metaobject might store is the base object's type, interface, class, methods, attributes, parse tree, etc.â€"Wikipedia[17]

It is technically possible to write software just using objects. When we need behaviour for an object, we can give it methods by binding functions to keys in the object:

```
var sam = {
  firstName: 'Sam',
  lastName: 'Lowry',
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
  rename: function (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
}
```

We call this a "naÃ¨ve" object. It has state and behaviour, but it lacks any division of responsibility/

This lack of separation has two drawbacks. First, it intermingles properties that are part of the model domain (such as `firstName`) with methods (and possibly other properties, although none are shown here) that are part of the implementation domain. Second, when we needed to share common behaviour, we could have objects share common functions, but does it not scale: There's no sense of organization, no clustering of objects and functions that share a common responsibility.

singleton metaclasses

Metaobjects solve this problem by separating the domain-specific properties of objects from their behaviour and implementation-specific properties. In classic JavaScript, we can use a prototype for the behaviour:

[17] https://en.wikipedia.org/wiki/Metaobject

```
var eigenPrototype = {
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
  rename: function (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};

var sam = Object.create(eigenPrototype);
sam.firstName = 'Sam';
sam.lastName = 'Lowry';
```

Notice that a prototype provides value even when we have no intention of sharing behaviour: Its value is that it separates behaviour from our model's domain properties very neatly:

```
Object.getOwnPropertyNames(sam)
  //=> [ 'firstName', 'lastName' ]
```

Insulating our base objects from our metaobjects is important for hiding implementation details, a core "OO" value. When we provide a method like `fullName`, we're hiding details of how to compute a full name from other entities, all they need to know is the name of the message to send.

hiding implementation details

But we need to hide implementation details within an object as well. Consider logging.[18] If we wish to write to the console every time an object is renamed, we have many choices. Here are three:

First, we can rewrite our function:

```
eigenPrototype.rename = function (first, last) {
  console.log(this.fullName() + " is being renamed " + first + " " + last);
  this.firstName = first;
  this.lastName = last;
  return this;
};
```

Second, we can use a method combinator[19]:

[18] The canonical example of a cross-cutting concern. We could also consider implementing undo and transactions, they have similar characteristics.

[19] https://github.com/raganwald/method-combinators

```
var logsNameChange = before( function (first, last) {
  console.log(this.fullName() + " is being renamed " + first + " " + last);
});

eigenPrototype.rename = logsNameChange( function (first, last) {
  this.firstName = first;
  this.lastName = last;
  return this;
});
```

Third, we could use an aspect-oriented programming library like YouAreDaChef[20]:

```
YouAreDaChef(sam).before('rename', function (first, last) {
  console.log(this.fullName() + " is being renamed " + first + " " + last);
});
```

Never mind which choice we should exercise. The important thing is that whatever we do should be hidden from the `sam` object itself! All it "knows" is that it has a prototype. What happens inside that prototype ought to be opaque. Is the `rename` function rewritten? Is it recombined? Is there a `before_rename` property with list of functions to execute? That should be irrelevant to the `sam` object itself.

The basic principle of the metaobject is that we separate the mechanics of behaviour from the domain properties of the base object. Everything else, like how inheritance is implemented, or whether a method is being logged, is hidden from the object.

[20] https://github.com/raganwald/YouAreDaChef

Protocols

> A metaobject **protocol** provides the vocabulary to access and manipulate the structure and behavior of objects. Typical functions of a metaobject protocol include: Creating and deleting new classes; Creating new methods and properties; Changing the class structure so that classes inherit from different classes; Generating or modifying the code that defines the methods for the class.â€"Wikipedia[21]

We've seen that a metaobject separates an object's domain-specific properties from the implementation details of its behaviour.

A metaobject protocol (or just "protocol") is an interface for managing metaobjects. In the previous section, we said that an object should not know whether logging is implemented by writing:

```
eigenPrototype.rename = function (first, last) {
  console.log(this.fullName() + " is being renamed " + first + " " + last);
  this.firstName = first;
  this.lastName = last;
  return this;
};
```

Or:

```
var logsNameChange = before( function (first, last) {
  console.log(this.fullName() + " is being renamed " + first + " " + last);
});

eigenPrototype.rename = logsNameChange( function (first, last) {
  this.firstName = first;
  this.lastName = last;
  return this;
});
```

Or:

```
YouAreDaChef(sam).before('rename', function (first, last) {
  console.log(this.fullName() + " is being renamed " + first + " " + last);
});
```

[21] https://en.wikipedia.org/wiki/Metaobject

To be specific, we meant that a base object should be insulated from the implementation consequences of rewriting a method, applying a combinator to it, or applying "before advice" to it. But we go further: In addition to insulating the base object from the implications of this choice, we must insulate the programmer from needing to think through and possibly re-invent the implementation of any of these three choices.

A protocol permits the programmer to write:

```
MOP.before(sam, 'rename', function (first, last) {
  console.log(this.fullName() + " is being renamed " + first + " " + last);
});
```

The protocol abstracts the implementation detail away.

Objects and Methods

encapsulation

object-1s, object-2s, and primitive protocols

queries, updates, and degenerate methods

object composition and delegation

state machines and strategies

nouns, verbs and commands

pattern matching

method composition

immediate, forward, and late-binding

me, myself, and i

Metaobjects

JavaScript's Constructors

Metaobjects have one key responsibility: *Defining base object behaviour*. Although it's not strictly required, most object-oriented languages accomplish this by having metaobjects also construct each object.

In classic JavaScript, any function can be used to construct a new object with the use of the `new` keyword. All functions have a `prototype` property even if they aren't intended to be used as constructors. You can change a function's prototype is you wish.

Functions used to create objects are called *constructors*, and in classic JavaScript the *constructor* is responsible for initializing new objects, while the *prototype* is responsible for the behaviour of the object. When we use JavaScript's `new` operator on a function, the prototype of the object is initialized automatically to be the prototype of the constructor. Since initialization and the prototype flow from the constructor, it is tempting to think of the constructor as the "Queen of all Things" in JavaScript.

```
var ClassicJSConstructor = function () {};
ClassicJSConstructor.prototype.identity = 'classic';
var classicObject = new ClassicJSConstructor();
Object.getPrototypeOf(classicObject)
  //=> { identity: 'classic' }
classicObject instanceof ClassicJSConstructor
  //=> true
```

If we follow that reasoning, we think of constructors as our metaobjects, and consider the prototype as part of the constructor. JavaScript encourages this perspective by providing an `instanceof` operator that appears to test whether an object was created by a particular constructor.[22]

constructors are not metaobjects

This thinking is mistaken. Constructors are not metaobjects. The `new` operator isn't even necessary to create objects:

```
var NakedPrototype = {
  identity: 'naked'
};
var unconstructedObject = Object.create(NakedPrototype);
Object.getPrototypeOf(unconstructedObject)
  //=> { identity: 'naked' }
```

[22]Most OO programmers prefer using polymorphism to explicitly testing `instanceof`. Wide use of explicit type testing is generally a design smell, but nevertheless it is a useful tool in some circumstances.

Furthermore, the `instanceof` operator doesn't do what it advertises. It appears to test whether a constructor created an object. But it actually tests whether an object and a function have compatible `prototype` properties.

Here we are fooling the operator:

```
var NeverConstructedAnything = function () {};
NeverConstructedAnything.prototype = NakedPrototype;
unconstructedObject instanceof NeverConstructedAnything
  //=> true
```

It turns out that `instanceof` is fine when there is a 1:1 correspondence between constructors and prototypes, when we do not change prototypes dynamically, and when we use `new` for all objects, eschewing `Object.create`. But the moment we venture into deeper waters, they the required workarounds outweigh their convenience.

If we want to test compatibility between an object and a prototype, we can and should do so directly:

```
NakedPrototype.isPrototypeOf(unconstructedObject)
  //=> true
```

The prototype always defines the behaviour of an object. JavaScript's "constructors," `new` operator, and its `instanceof` operator are convenient for programming within a narrow band of conventions, but are unreliable in the general case.

Therefore, although constructors can be used create objects, we consider the object's prototype to be central to the idea of a metaobject. We do not rely on the `instanceof` operator and if we wish to test for prototype compatibility, we use the `isPrototypeOf` method instead.

Classes and Prototypes

Although classes and prototypes both are responsible for creating objects and managing their shared behaviour, classes aren't prototypes and prototypes aren't classes. The simplest way to see the difference is to think about their methods. An object's methods are the surest clue to its function and responsibilities.

Let's revisit an example prototype:

```
function MovieCharacter (firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
};

MovieCharacter.prototype.fullName = function () {
  return this.firstName + " " + this.lastName;
};
```

What are the prototype's methods?

```
Object.keys(MovieCharacter.prototype).filter(function (key) {
  return typeof(MovieCharacter.prototype[key]) === 'function'
});
  //=> [ 'fullName' ]
```

In JavaScript, `fullName` is a method of `MovieCharacter`'s prototype. The prototype's methods are the behaviour we're defining for `MovieCharacter` objects.

Now let's compare this to a "class." JavaScript doesn't have classes right out of the box, so we'll compare the prototype's methods to the methods of an equivalent Ruby class as an example:

```
class MovieCharacter

  def initialize(first_name, last_name)
    @first_name, @last_name = first_name, last_name
  end

  def full_name
    "#{first_name} #{last_name}"
  end

end
```

```
12
13 MovieCharacter.methods - Object.instance_methods
14 #=> [ :allocate, :new, :superclass, :<, :<=, :>, :>=, :included_modules, :include\
15 ?,
16       :name, :ancestors, :instance_methods, :public_instance_methods,
17       :protected_instance_methods, :private_instance_methods, :constants, :const_\
18 get,
19       :const_set, :const_defined?, :const_missing, :class_variables,
20       :remove_class_variable, :class_variable_get, :class_variable_set,
21       :class_variable_defined?, :public_constant, :private_constant, :module_exec,
22       :class_exec, :module_eval, :class_eval, :method_defined?, :public_method_de\
23 fined?,
24       :private_method_defined?, :protected_method_defined?, :public_class_method,
25       :private_class_method, :autoload, :autoload?, :instance_method,
26       :public_instance_method ]
```

In Ruby, `full_name` isn't a method of the `MovieCharacter` class, and unlike JavaScript, the class has lots and lots of methods that are specific to the business of being a meta-class that aren't shared by other objects.

JavaScript prototypes look just like ordinary objects, while Ruby classes don't look anything like ordinary objects. They're both metaobjects, but the two languages use completely different approaches. This is not surprising when you learn that Ruby was inspired by Smalltalk[23], a language that emphasized classes, while JavaScript was inspired by Self[24], a successor to Smalltalk that used prototypes instead of classes.

In some languages the difference between a class and a metaobjetc liek a prototype is even more pronounced. They have the notion of classes, but they don't have metaobjects you can access at runtime. C++[25], for example, allows you to define classes, the definitions are compiled into protocols for virtual functions that are late-bound, but there are no class objects in the system at run time. Such classes aren't metaobjects at all, so those classes are even more different than JavaScript's prototypes.

The takeaway is that there is no one "correct" design for metaobjects. The fundamental idea is that metaobjects manage creation and/or behaviour of a common set of objects, and that they are themselves objects a program can access and manipulate at runtime.

[23] https://en.wikipedia.org/wiki/Smalltalk

[24] https://en.wikipedia.org/wiki/Self_programming_language

[25] https://en.wikipedia.org/wiki/C%2B%2B

templates and creation by value

metaclasses and meta-metaobjects

immediate, forward, and late-binding of metaobjects

degenerate protocols

eigenclasses, again

prototypes vs. classes: metaobject-1s vs. metaobject-2s

contracts and liskov equivalence

metaobjects are not types, and types are not interfaces

Protocols

prototype chaining

single inheritance

mixins and multiple inheritance

resolution: merge, override, and final

template method protocols

method guards and contracts

early and late method composition

multiple dispatch and generic functions

pattern matching protocols