VA DERS

(Q: What does it say?)

VA DERS

VA DERS

(A: “Space Invaders”)

am U us

am U us

(A: “Ambiguous”)

convention

—

configuration

convention

—

configuration

(A: “Convention over configuration”)

Welcome!

A Grok Talk

Brandon Craig Rhodes

November 2007

What's a convention?

Conventions are traditionally *extras*.

Conventions begin as *optional* practices to keep code sane

“We always capitalize the names of classes in the code we write for this department.”

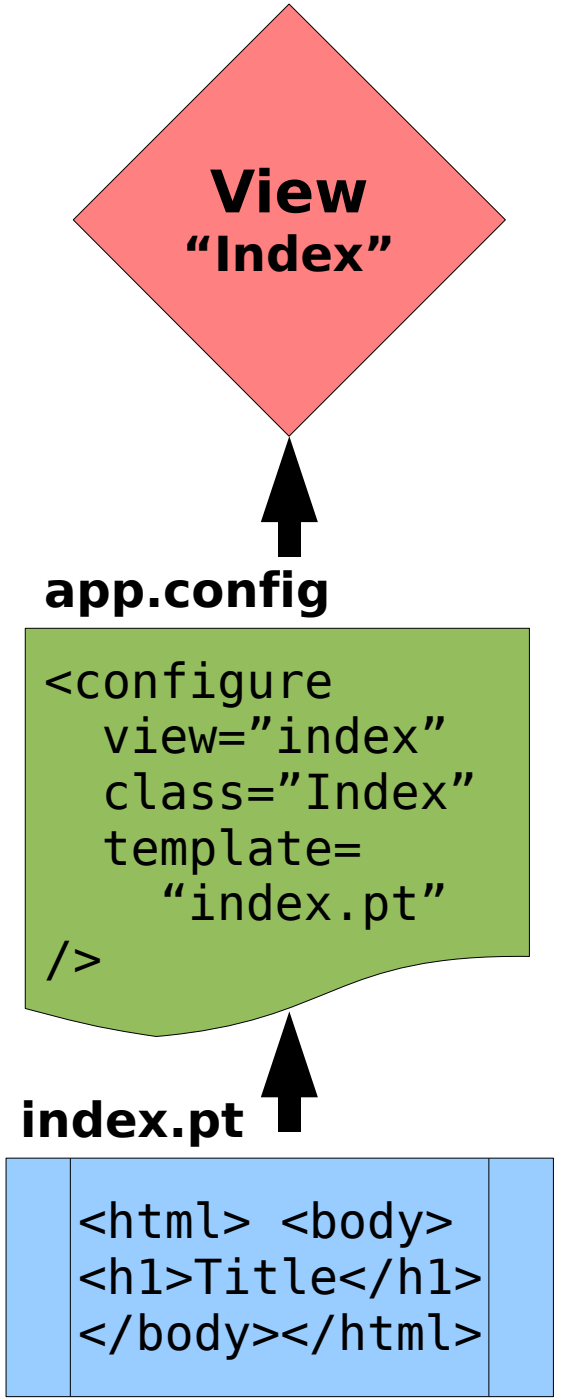
“If you have a class *Foo*, name its corresponding page template *foo.pt*.”

“We're putting cover sheets on all the TPS reports before they go out.”

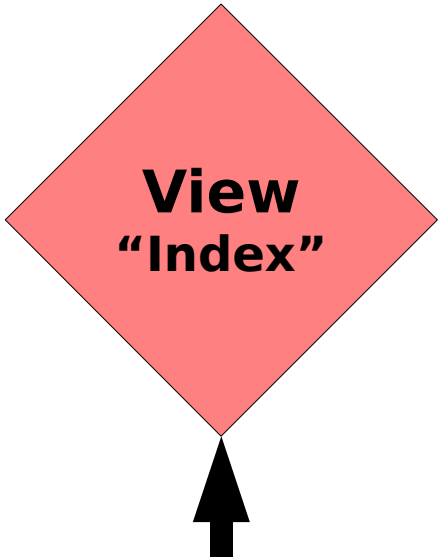
What if … ?

What if your web framework used conventions rather than config files?

A simple example:


View
“Index”
app.config
<configure
view=”index”
class=”Index”
template=
“index.pt”
/>
index.pt
<html> <body>
<h1>Title</h1>
</body></html>

A class you've written



app.config

```
<configure
  view=”index”
  class=”Index”
  template=
    “index.pt”
/>
```

The configuration file you need to hook them up!

index.pt

A page template you've written

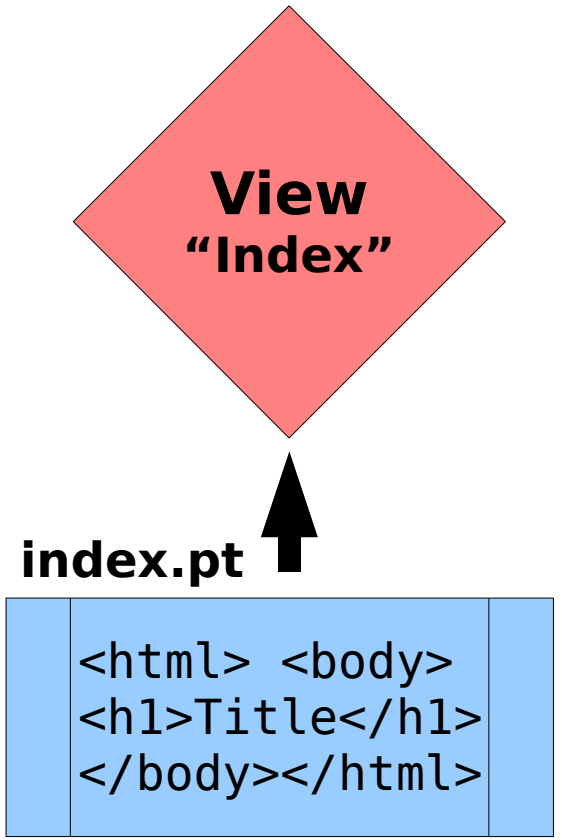
```
<html> <body>
<h1>Title</h1>
</body></html>
```

Conventions are traditionally *ignored* by the computer

But, what if … ?

What if the framework *assumed*, in the absence of other configuration, that *Index* goes with *index.pt*?

Result:



All explicit configuration has disapperaed

Advantages Ensue!

Advantages of CoC:

1. Less repetition

Advantages of CoC:

1. Less repetition
2. Conventions get used, because they matter

Advantages of CoC:

1. Less repetition
2. Conventions get used, because they matter
3. No loss of flexibility

So, who does CoC?

Actually a *very* old Computer Science concept

Example: FORTRAN assumed, in the absence of a declaration, that i and j were integers, and that x and y were floating-point

But when did it take off for web applications?

2004

Python frameworks:

2005 — Django, Turbogears

2006 — Pylons, Grok

Q:

Why talk about Grok?

It's only a year old!

Django and TurboGears are each two years old

A:

Because Grok is built atop Zope 3

Zope 3 started in 2001, production release 2004

Grok brings ease of configuration to an *existing* framework

Zope 3 is more mature

Zope 3 is powerful

Zope 3 provides
a powerful component
framework

But

It requires configuration

Raw Zope 3
requires you to use
its Zope Configuration
Markup Language
(ZCML)

```
<configure
   xmlns="http://namespaces.zope.org/zope"
   i18n_domain="zope"
   >
  <permission
     id="zope.Public"
     title="[public-permission] Public"
     description="Special permission indicating unconditional access.
                  Public resources are always accessible."
     />
  <utility
     component=".vocabulary.PermissionsVocabulary"
     name="Permissions"
     />
  <utility
     component=".vocabulary.PermissionIdsVocabulary"
     name="Permission Ids"
     />
  <include file="globalmodules.zcml" />
  <include file="_protections.zcml" />
  <utility
     provides=".interfaces.IAuthentication"
     component=".principalregistry.principalRegistry"
     />
```

But not Grok!

Grok is friendly

More specifically

Grok
is a friendly
cave man

Grok wields a club

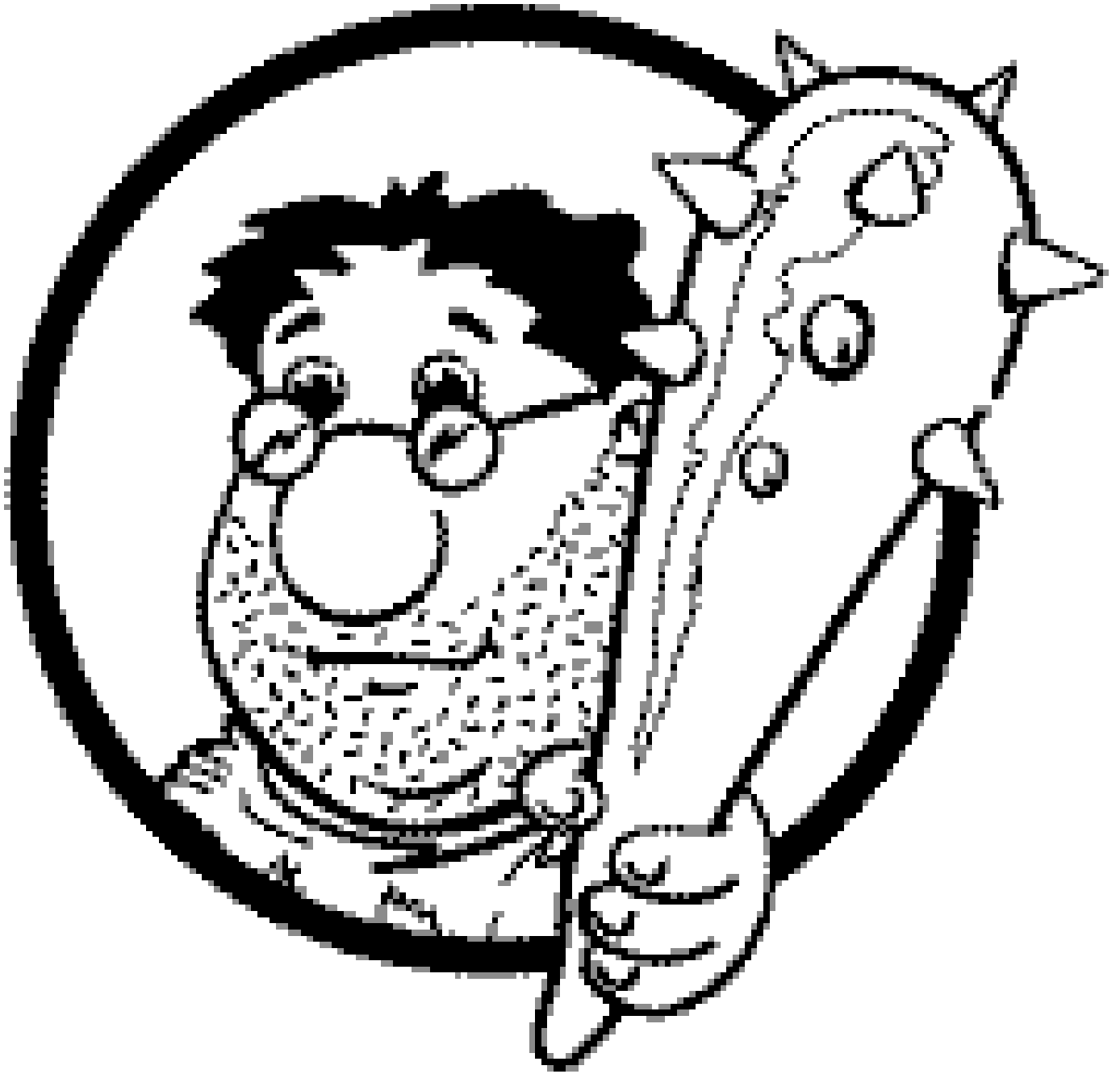
In
fact

Grok wields
a *large*
club

Grok uses his club to *smash* ZCML


GROK

Grok offers us his club



So that our web apps can be configured using convention instead of XML

Let's create a Grok instance!

(Brandon, pause the slides, and show how to create a Grok instance. Name it “MyApp”.)

The instance comes with a web page already displaying!

What's the formula?

Grok's Threefold Way

Grok's Threefold Way

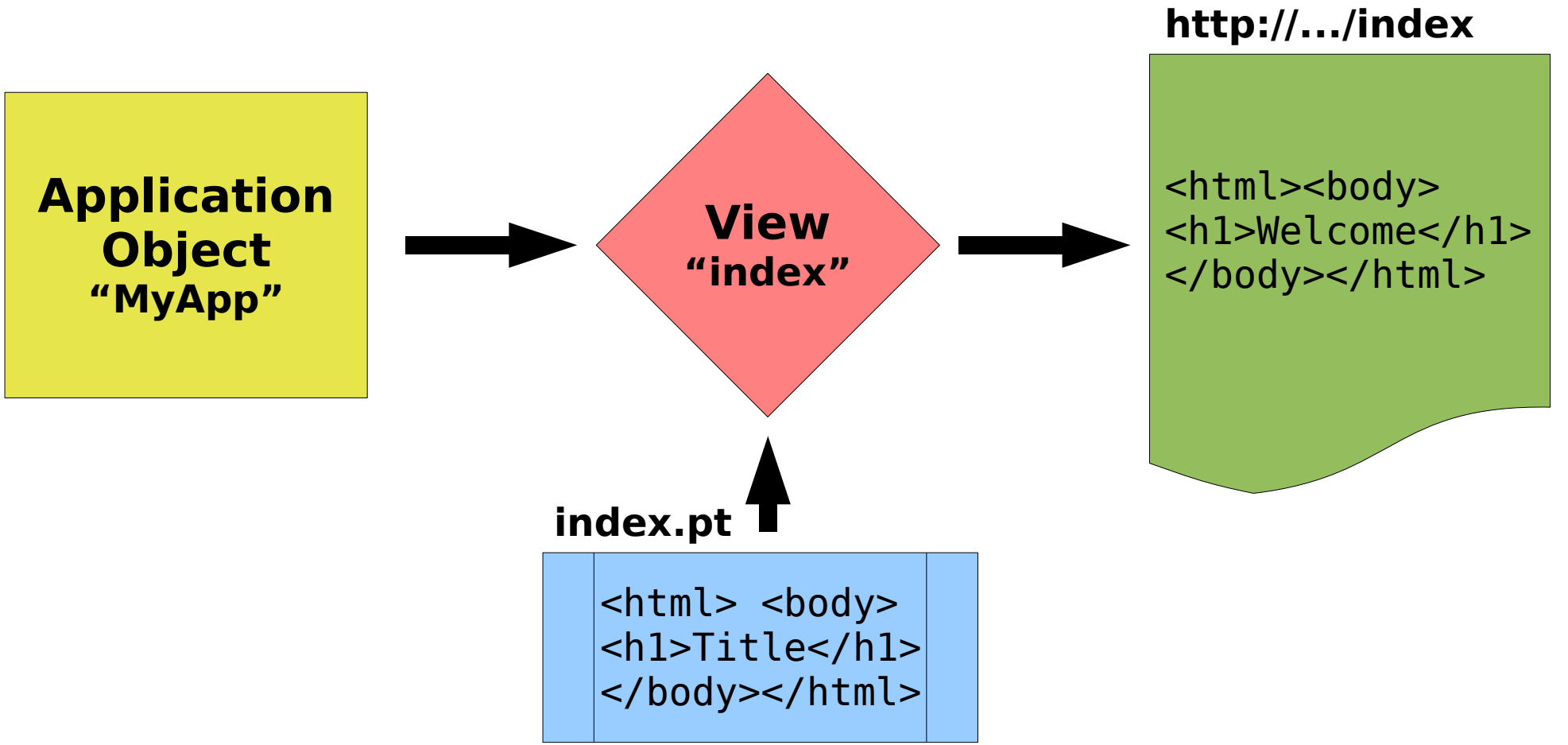
1. An *object* at the URL

Grok's Threefold Way

1. An *object* at the URL
2. A *view* for that object

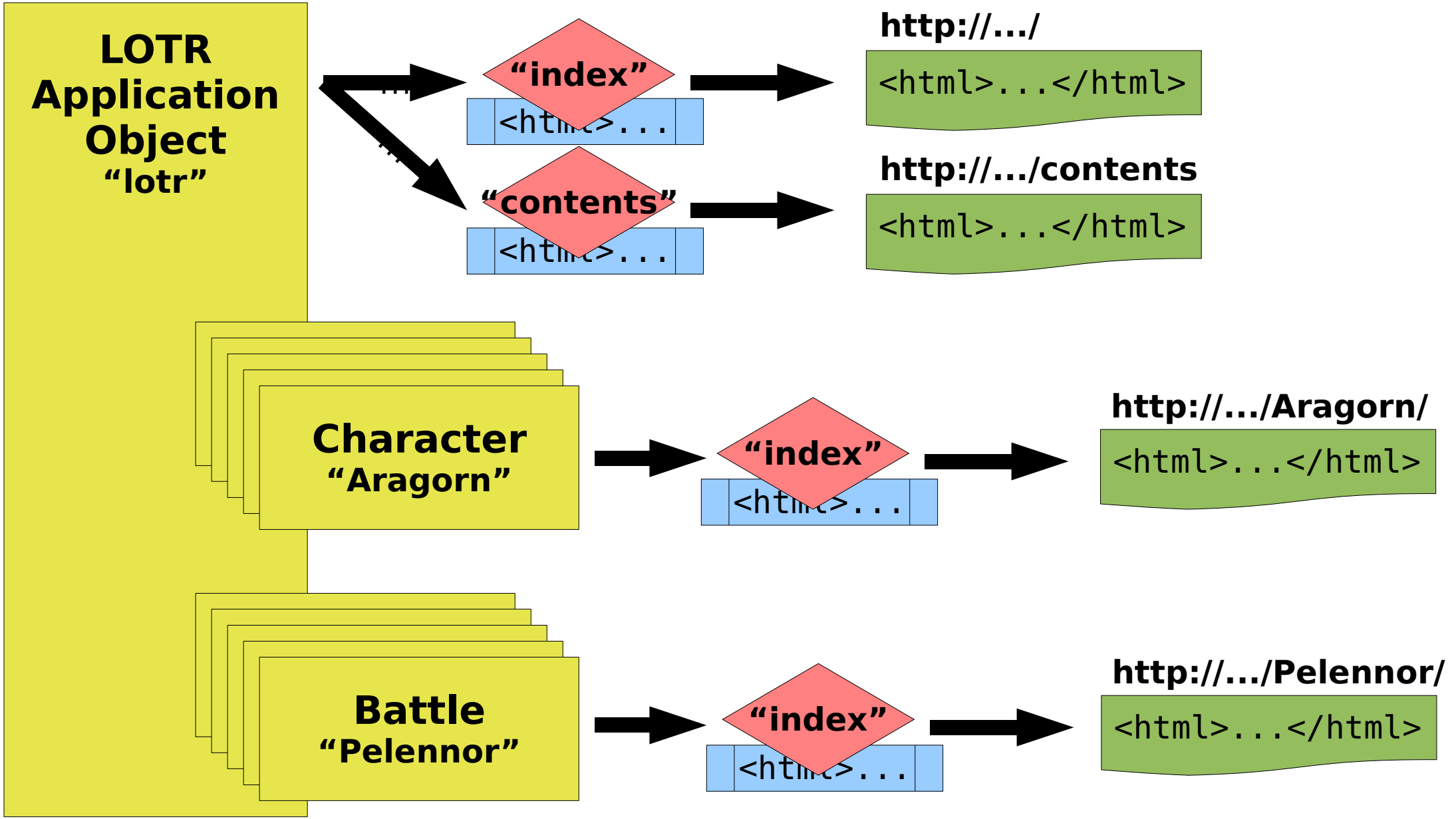
Grok's Threefold Way

1. An *object* at the URL
2. A *view* for that object
3. A *template* for the view


Application Object
“MyApp”
View
“index”
http://.../index
<html><body>
<h1>Welcome</h1>
</body></html>
index.pt
<html> <body>
<h1>Title</h1>
</body></html>

Let's add some
more Python objects,
more Grok Views,
and more templates

(Brandon, go add some models and further views to your application)


LOTR
Application
Object
“lotr”
“index”
<html>...
http://.../
<html>...</html>
“contents”
<html>...
http://.../contents
<html>...</html>
Character
“Aragorn”
“index”
<html>...
http://.../Aragorn/
<html>...</html>
Battle
“Pelennor”
“index”
<html>...
http://.../Pelennor/
<html>...</html>

Isn't that fun?

Time for one more topic

What should it be?

There are several directions we could go

I could show you how easy it is to process form data

We could, in several seconds, have an XML-RPC interface to our models working

We could explore how Zope can generate forms *for you*

An illustration could be made of how our bare application logic itself benefits from being in a component framework

But instead:

We will look at the contract between a View and a Template

And explore *two* forms
that the contract
can take

Let's look back
at our code...

Our View classes are pretty anemic

While our templates are out surfing our raw application objects!

(Brandon, go show them your anemic View classes and your object-surfing templates)

Let's call templates which surf the raw application objects *“Muscular templates”*

Advantages of muscular templates:

1. Quick

Advantages of muscular templates:

1. Quick
2. Fast

Advantages of muscular templates:

1. Quick
2. Fast
3. Easy

But...

There are also disadvantages to muscular templates

Disadvantages of muscular templates:

1. Difficult to read

Disadvantages of muscular templates:

1. Difficult to read
2. … and thus, to audit

Disadvantages of muscular templates:

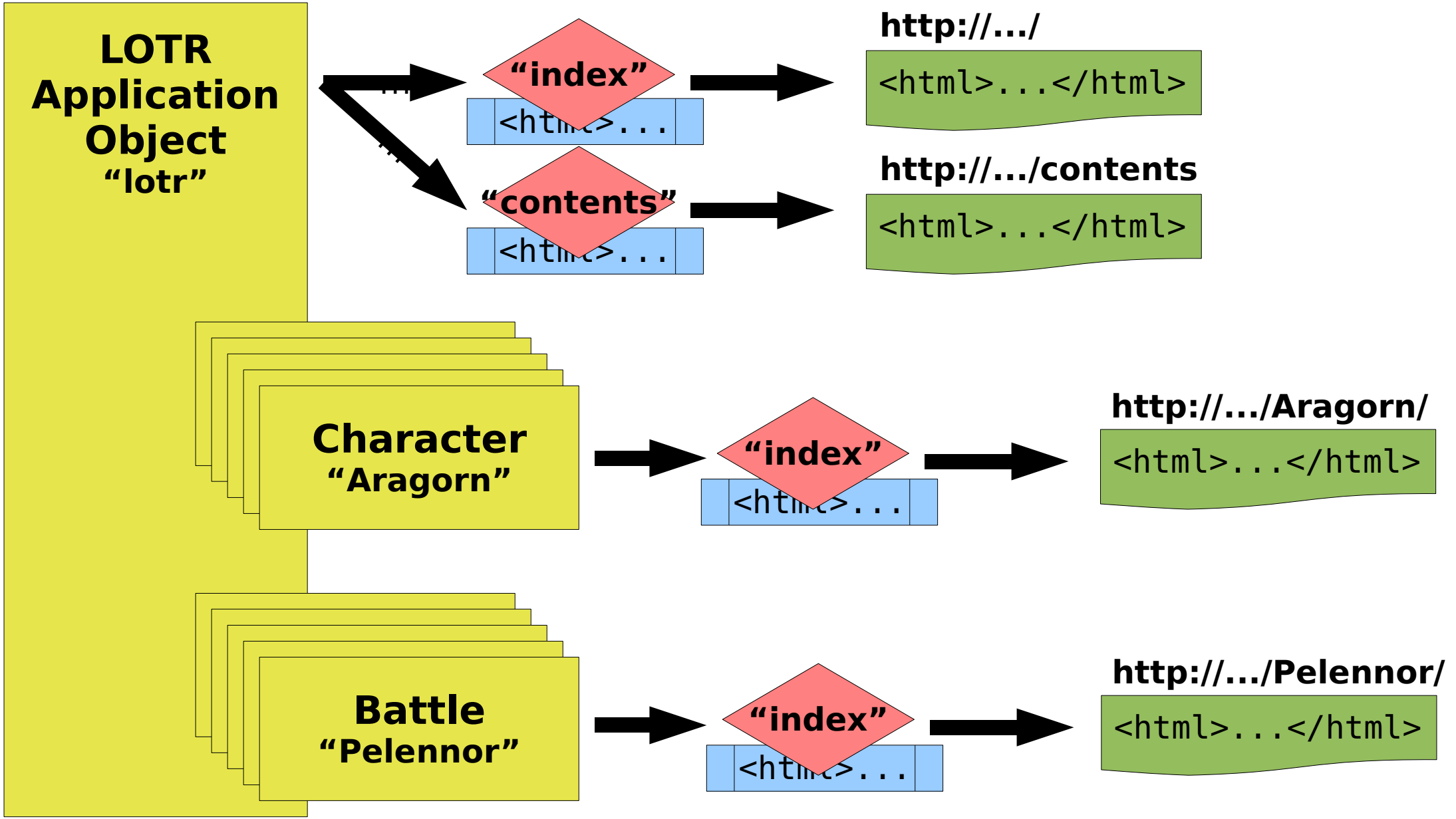
1. Difficult to read
2. … and thus, to audit
3. *They know your model*

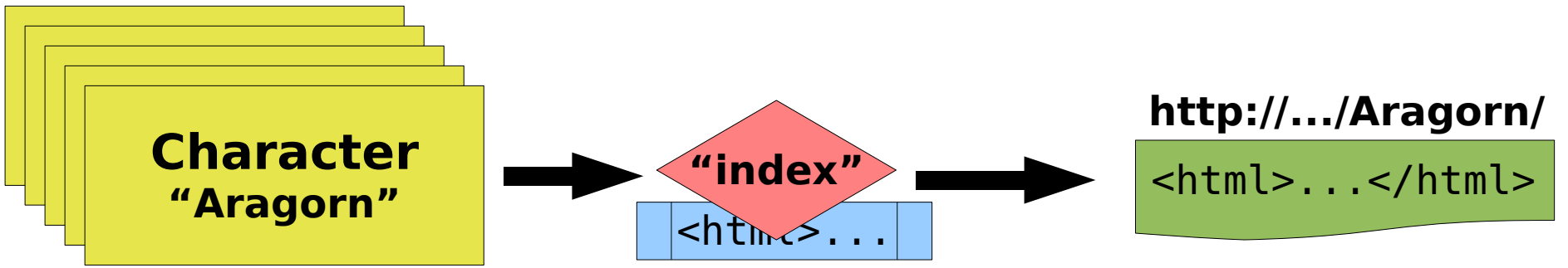
The alternative:

Muscular Views

Let's return to our application's design

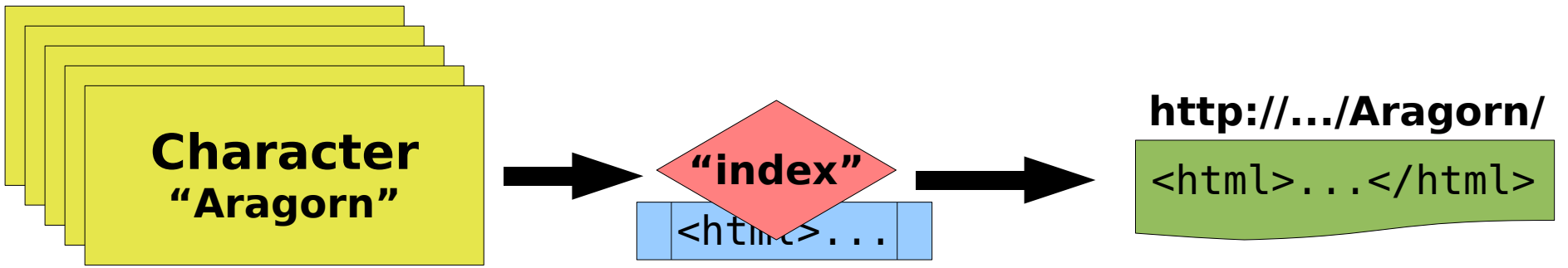
And look at one
of its pages


LOTR
Application
Object
“lotr”
“index”
<html>...
http://.../
<html>...</html>
“contents”
<html>...
http://.../contents
<html>...</html>
Character
“Aragorn”
“index”
<html>...
http://.../Aragorn/
<html>...</html>
Battle
“Pelennor”
“index”
<html>...
http://.../Pelennor/
<html>...</html>

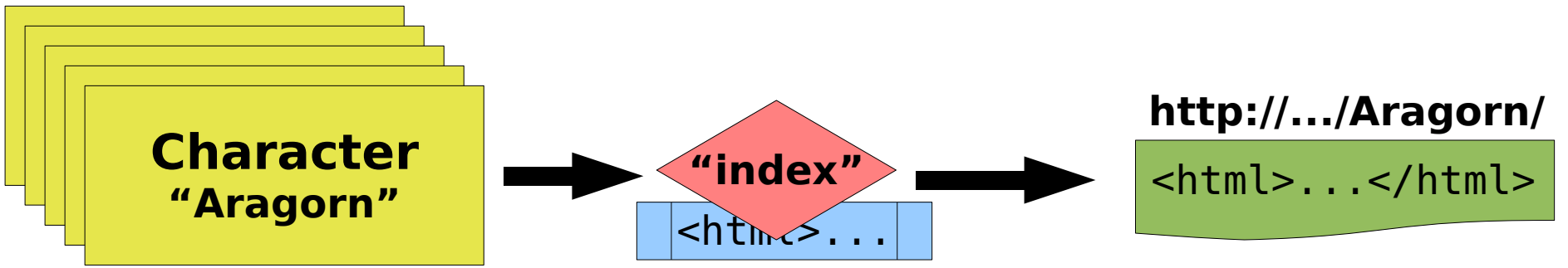

Character
“Aragorn”
“index”
<html>...
http://.../Aragorn/
<html>...</html>

The object and the view both have names in a traditional Zope page template...

context



context view



But instead of using these default names, we can provide our own View namespace

We do so by creating a *namespace()* method on our View

(Brandon, go make your CharacterIndex View more muscular)

What are the advantages of muscular Views?

(Phrased differently: *“Why on earth would you ever do that?!”*)

(Because, of course, we've added more code to our app without adding any new functionality!)

Advantages:

1. Your models can now evolve without breaking your page templates

Advantages:

2. Page templates cannot reveal data not delivered explicitly in namespace()

Advantages:

3. If you write some fake Views returning static data from namespace(), your template writers can start work Day 1

Advantages:

4. Your Views are easy to test (call namespace() and check result), templates are too (try them with static data)

We'll conclude

Grok

Advantages:

1. Easy, fun
2.
3.
4.
5.

Advantages:

1. Easy, fun
2. Powerful framework
3.
4.
5.

Advantages:

1. Easy, fun
2. Powerful framework
3. Deploys with *buildout*
4.
5.

Advantages:

1. Easy, fun
2. Powerful framework
3. Deploys with *buildout*
4. Shares code with Plone
5.

Advantages:

1. Easy, fun
2. Powerful framework
3. Deploys with *buildout*
4. Shares code with Plone
5. Vibrant community

Of course, there are also disadvantages

Disadvantages

1. Not yet 1.0

Disadvantages

1. Not yet 1.0
2. Security could be easier

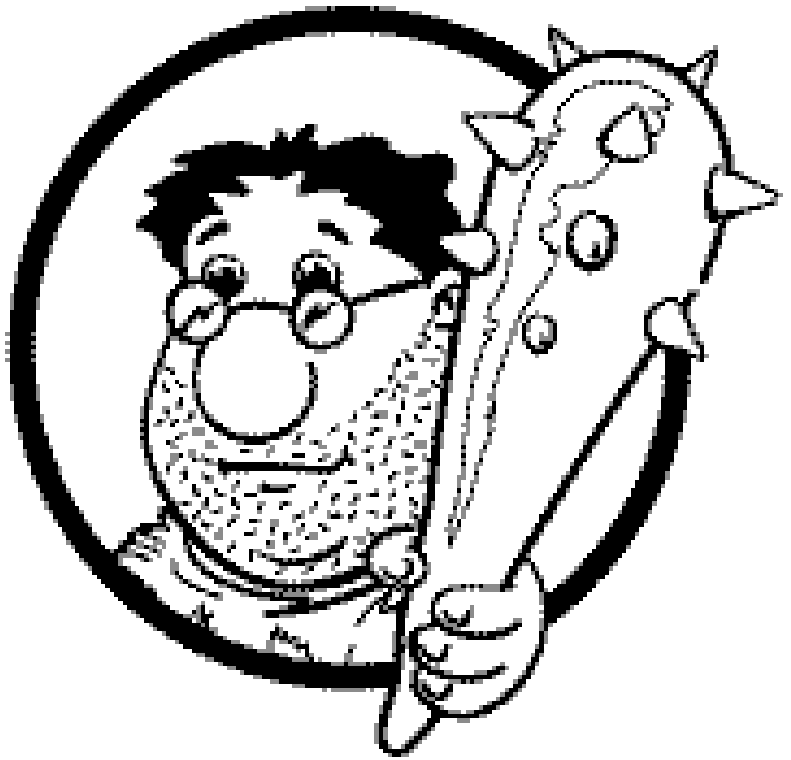
Disadvantages

1. Not yet 1.0
2. Security could be easier
3. Online docs still weak
   (buy PvW *Zope 3* book!)

Disadvantages

1. Not yet 1.0
2. Security could be easier
3. Online docs still weak
4. Community is 6h off

Thank you!



Any questions?