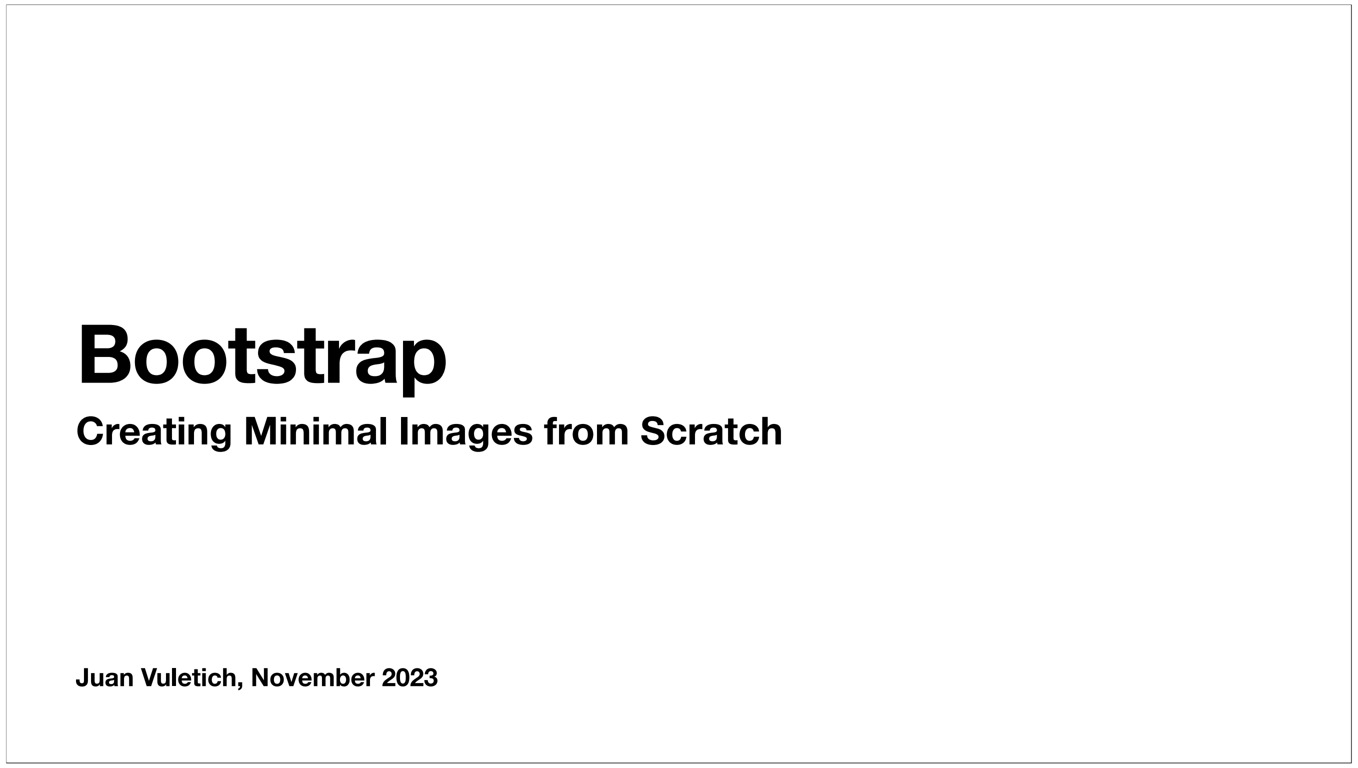
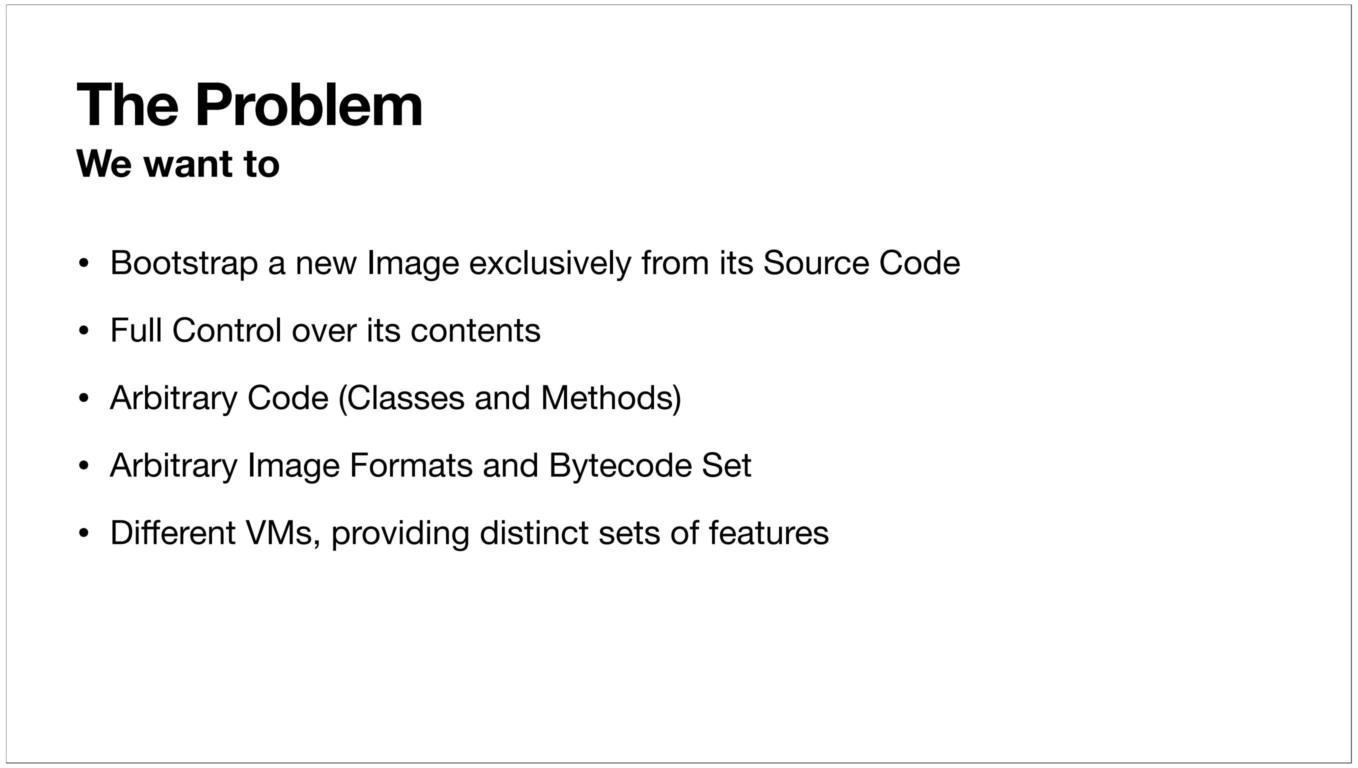


Bootstrap

Creating Minimal Images from Scratch

Juan Vuletich, November 2023

Hello all. I am Juan Vuletich.
I want to start by thanking you for being here today. I want to thank FAST, Universidad Nacional de Quilmes and the Public Universities in Argentina that host us every year for this conference.
I want to thank Máximo and Hernán for using Cuis for teaching, for building and maintaining CuisUniversity, and for helping many people use it. I want to thank Máximo for starting this movement that today means that Buenos Aires is world class Smalltalk center.
I want to thank Hilaire for DrGeo, for adopting Cuis for building DrGeo, for writing The Cuis Book, and for so many nice words about Cuis. Thank you all.
This talk is about a project I have been working on for most of this year. It is about bootstrapping Smalltalk images from scratch. It is part of a larger effort to make Cuis meet the needs of commercial projects, like those developed at LabWare. So, I want to thank LabWare for sponsoring this project.



The Problem

We want to

- Bootstrap a new Image exclusively from its Source Code
- Full Control over its contents
- Arbitrary Code (Classes and Methods)
- Arbitrary Image Formats and Bytecode Set
- Different VMs, providing distinct sets of features

What is the problem we are trying to solve here?

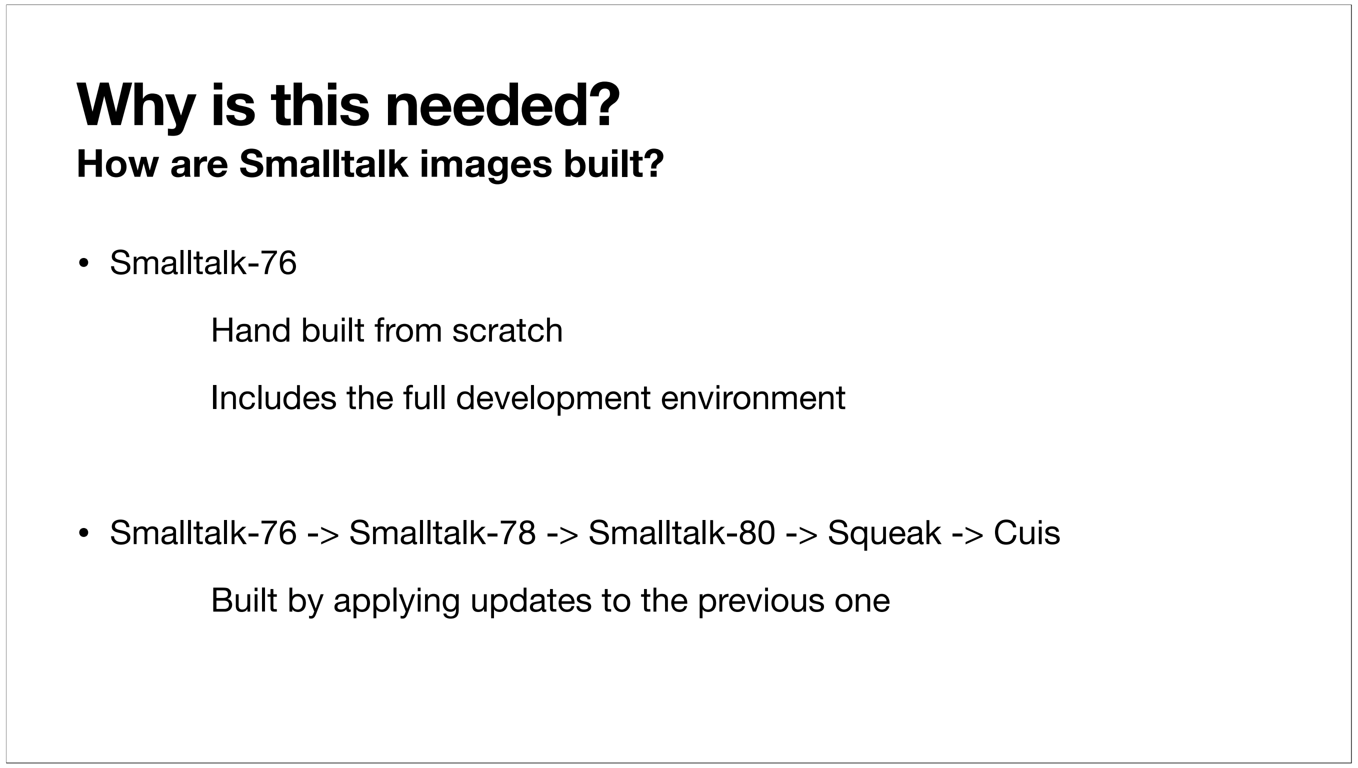
We want to Bootstrap new Images exclusively from their Source Code
We want Full Control over its contents
We want to be able to include Arbitrary Code (Classes and Methods)
We want to support Arbitrary Image Formats and Bytecode Sets
And we also want to support different VMs, that provide possibly different sets of features

We want to Bootstrap a new Image
Why?

- Maybe we are developing end user applications and need a runtime that:
  - Doesn't include development tools
  - Doesn't include a Compiler
  - Doesn't include any source code at all
- Maybe because we are building a new Smalltalk System and want:
  - Reproducible builds
  - To enable inspecting it, and know where each bit comes from

Additionally, we want a new image that

- Is small and completely under our control
- It only includes what we explicitly say
- It is not restricted by the design of some "Mother Image"
- The tools to build it are under our control to evolve and be enhanced
- It doesn't pull the Gorilla and whole Jungle when we just want the Banana



Why is this needed?

How are Smalltalk images built?

- Smalltalk-76

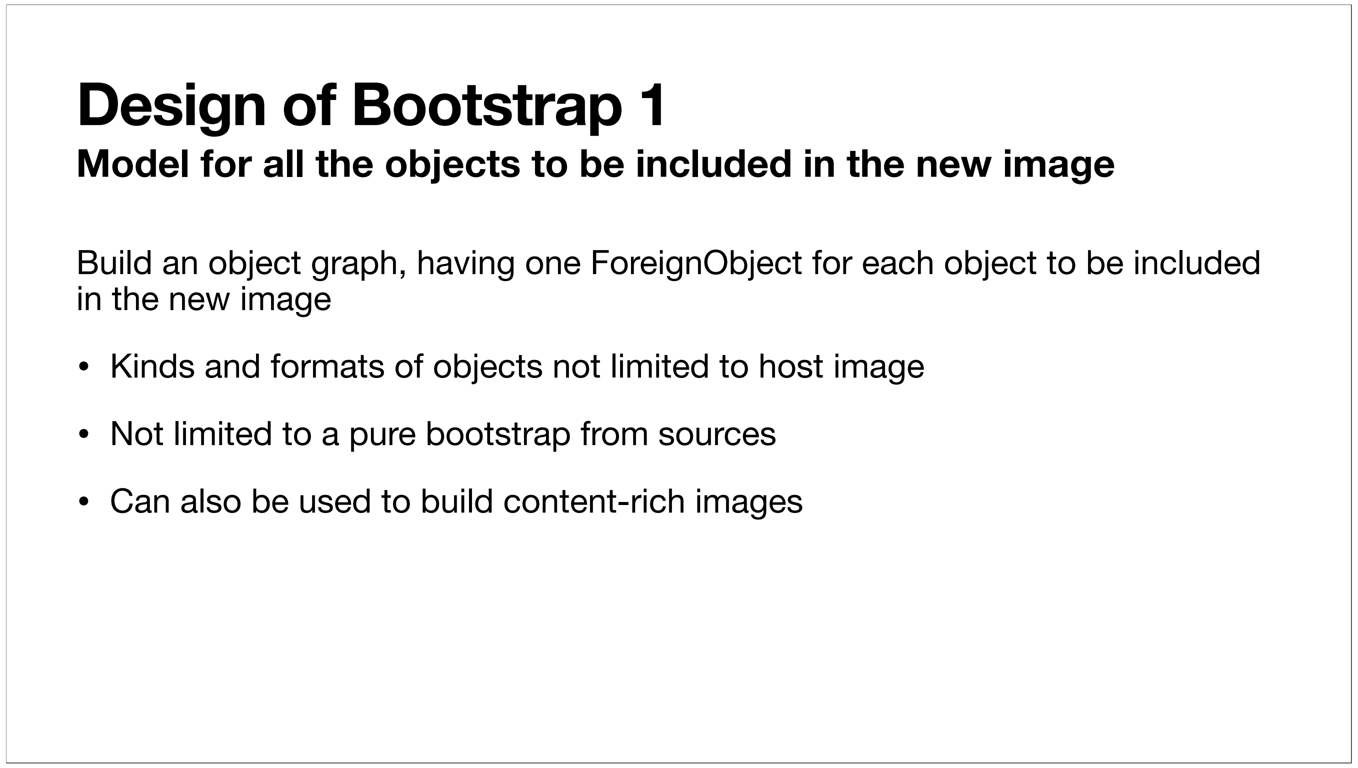
    Hand built from scratch

    Includes the full development environment

- Smalltalk-76 -> Smalltalk-78 -> Smalltalk-80 -> Squeak -> Cuis

    Built by applying updates to the previous one

So why is this needed? Well, the last ancestor of Cuis Smalltalk that was bootstrapped from scratch was Smalltalk-76. Since then, every released image of Smalltalk-80, Squeak and Cuis was derived by applying updates to the previous one.



Design of Bootstrap 1

Model for all the objects to be included in the new image

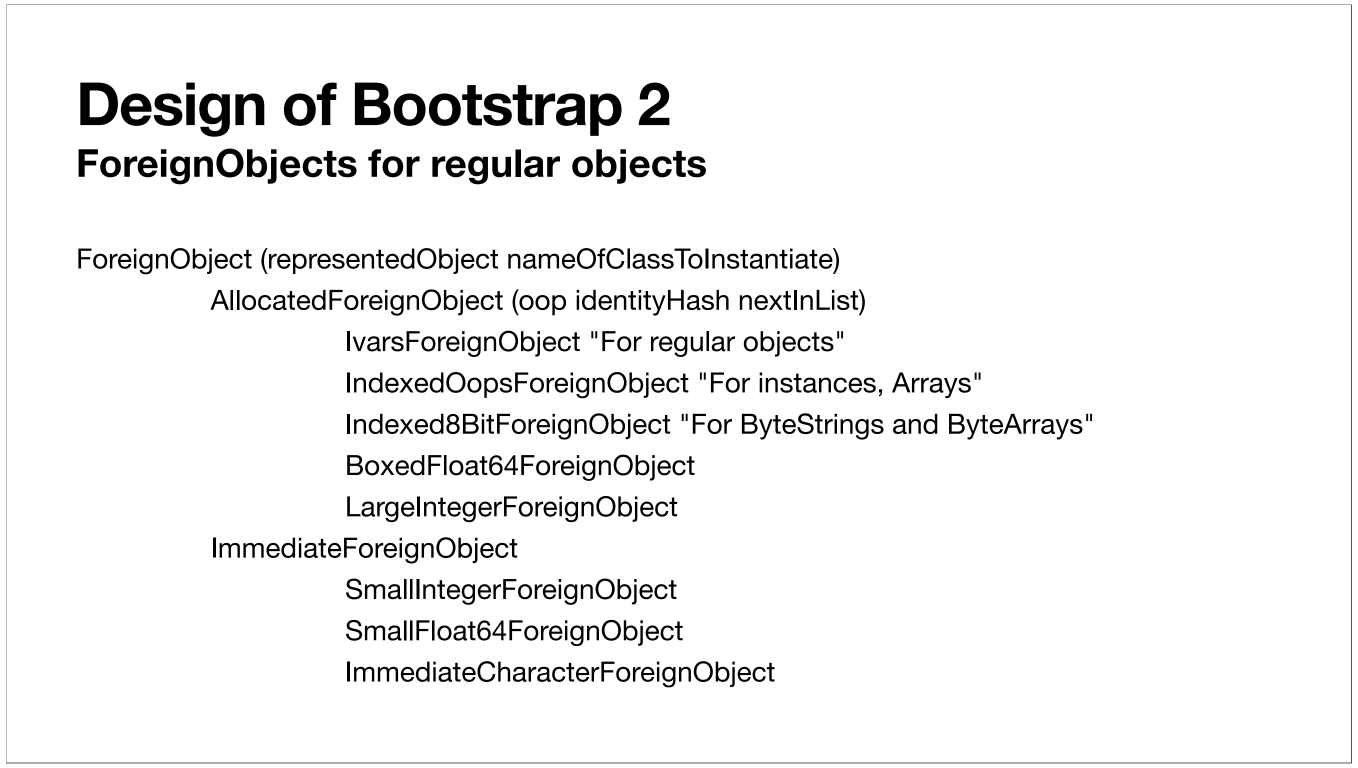
Build an object graph, having one ForeignObject for each object to be included in the new image

- Kinds and formats of objects not limited to host image
- Not limited to a pure bootstrap from sources
- Can also be used to build content-rich images

In order to build a Smalltalk model of everything that will be part of our new image, we need to describe the graph of objects it will include. For each of these objects we need to store some information that is usually hidden by the VM in a running system, but that is essential if we are to save a new Smalltalk image that can later run.

For this, we create a ForeignObject for each object that will be created in the new image, and an object graph is built with them.

There are many kinds of ForeignObjects. This reflects the fact that some objects have instance variables, while others have indexed slots, like Arrays. Still other objects are made of bytes or other numeric values. And yet others have their value encoded in pointers to it, like SmallIntegers, and they hold no state at all.



Design of Bootstrap 2

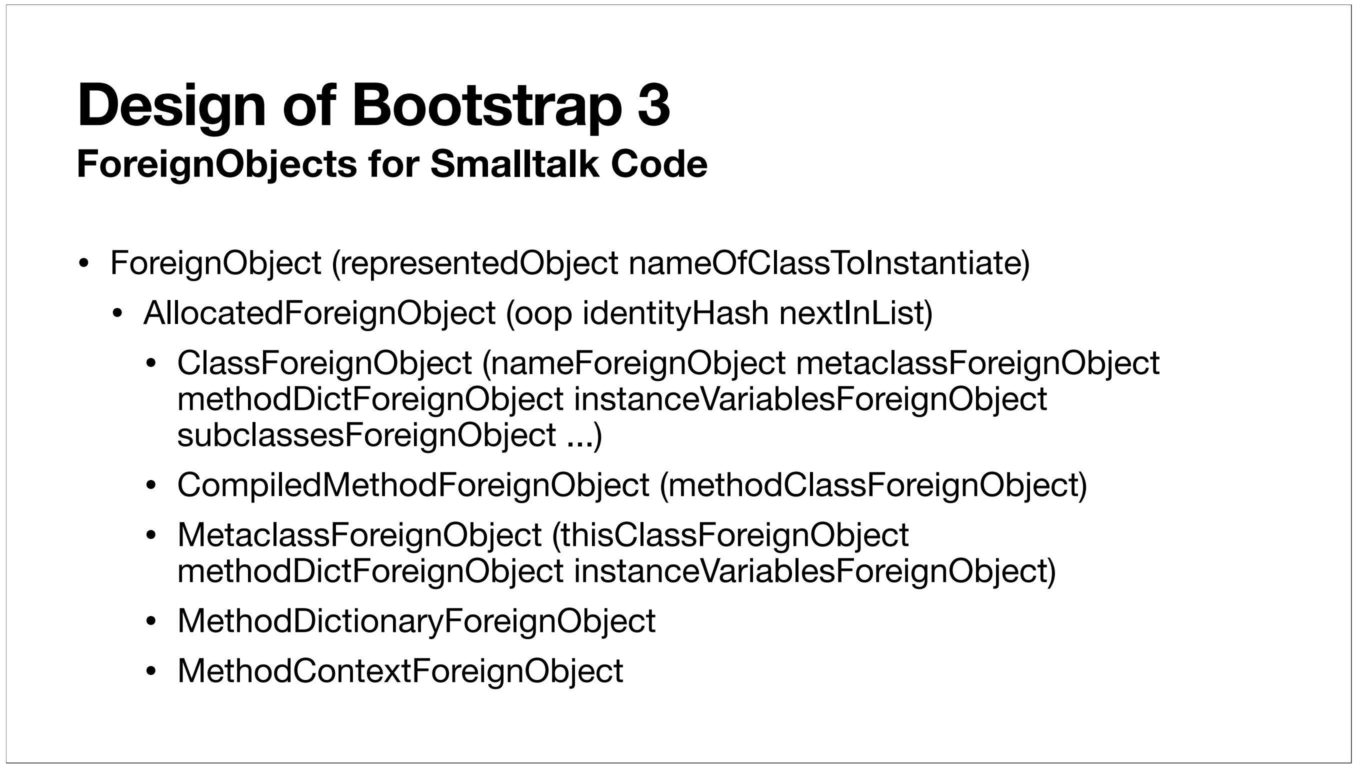
ForeignObjects for regular objects

```
ForeignObject (representedObject nameOfClassToInstantiate)
        AllocatedForeignObject (oop identityHash nextInList)
                IvarsForeignObject "For regular objects"
                IndexedOopsForeignObject "For instances, Arrays"
                Indexed8BitForeignObject "For ByteStrings and ByteArrays"
                BoxedFloat64ForeignObject
                LargeIntegerForeignObject
        ImmediateForeignObject
                SmallIntegerForeignObject
                SmallFloat64ForeignObject
                ImmediateCharacterForeignObject
```

These are the ForeignObject classes used to model regular Smalltalk objects. Each of them knows how to create a real object of its kind when serializing the object file. As you can see, AllocatedObjects hold the pointer in memory where their real object will reside, and the identityHash it will use. ImmediateForeignObject has neither, because immediate objects don't use memory as they are encoded in the very pointers referencing them.

This means that creating these ForeignObjects and creating the object graph is also allocating memory in the new image. The moment that memory is allocated and pointers are assigned to, is when we build this graph.

Also, these ForeignObjects form a linked list, in the order as they were allocated, that is also the order in which they will be stored in the memory of the new image, in the image file.



Design of Bootstrap 3

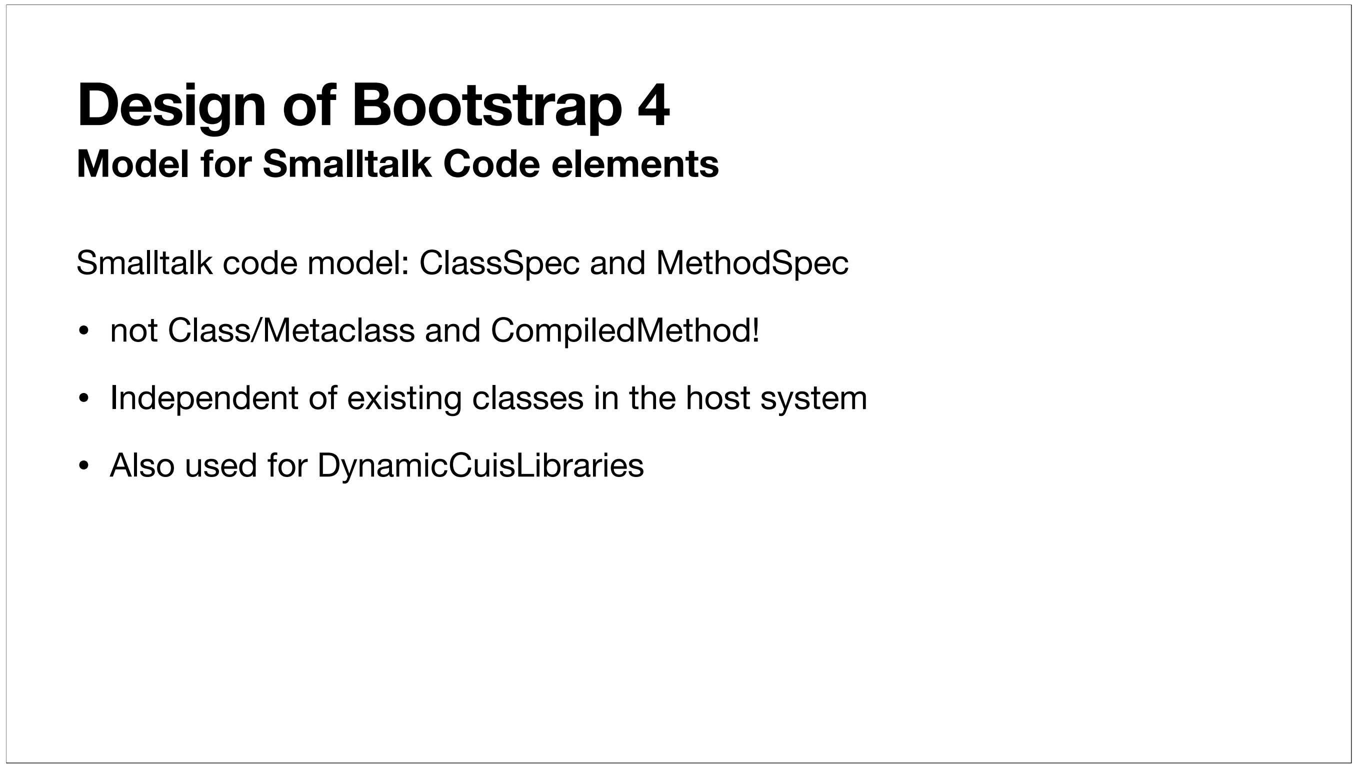
ForeignObjects for Smalltalk Code

- ForeignObject (representedObject nameOfClassToInstantiate)
  - AllocatedForeignObject (oop identityHash nextInList)
    - ClassForeignObject (nameForeignObject metaclassForeignObject methodDictForeignObject instanceVariablesForeignObject subclassesForeignObject ...)
    - CompiledMethodForeignObject (methodClassForeignObject)
    - MetaclassForeignObject (thisClassForeignObject methodDictForeignObject instanceVariablesForeignObject)
    - MethodDictionaryForeignObject
    - MethodContextForeignObject

These are the ForeignObjects used to create Smalltalk code entities in the new image. The represented object (that is held in the 'representedObject' instance variable) will not be a class or a method, but instances of new classes called ClassSpec and MethodSpec. Still, these ForeignObjects will create the real classes and methods in the new image.

The most important ones are those for Classes, Metaclasses and CompiledMethods. Also we build ForeignObjects for MethodDictionaries. These ForeignObjects form the metamodel of the new Smalltalk image. The object graphs they form is not unlike that of existing Classes, Metaclasses, MethodDictionaries and CompiledMethods. But they form a completely separated object graph.

Besides as they will be used for serializing them into a new image file, and not for running them in the current image and VM, the services they provide are completely different. These are not classes and methods!



Design of Bootstrap 4

Model for Smalltalk Code elements

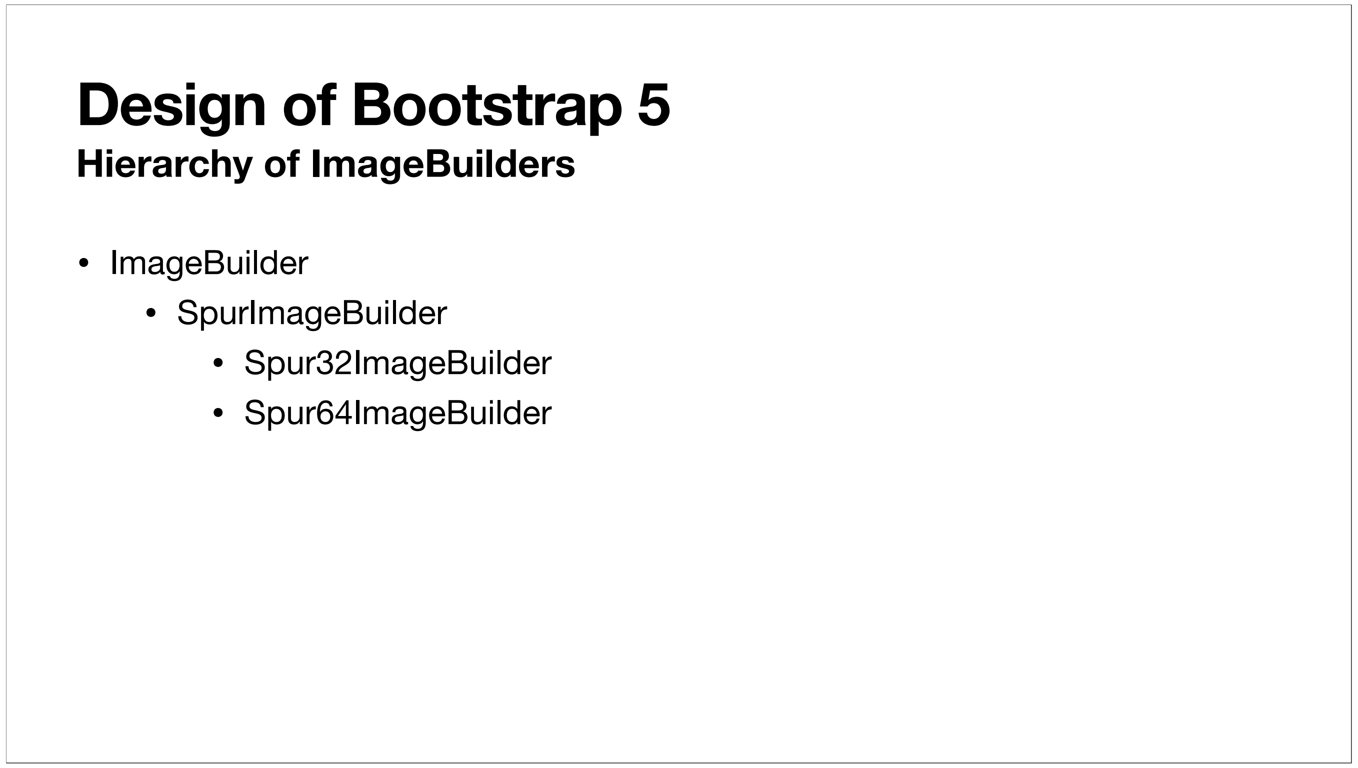
Smalltalk code model: ClassSpec and MethodSpec

- not Class/Metaclass and CompiledMethod!
- Independent of existing classes in the host system
- Also used for DynamicCuisLibraries

We want to build a new Smalltalk image that includes essentially Smalltalk code. We could think that our existing Smalltalk image already includes a model for Smalltalk code, right? That is true, but we don't need or want to be able to run this new code in our old image. Bootstrap includes a new object model for Smalltalk code that is completely separated and independent from the existing classes and methods of the image that runs the Bootstrap tool. The model for the code in the new image is not a set of Classes and CompiledMethods, but a set of ClassSpecs and MethodSpecs. These include all the required information to be later materialized as Classes and Methods into a running system, but they are not part of the host image metamodel.

This means that the classes and methods that will be included in the new Smalltalk image we are building don't need to resemble those in our existing system. We could even model new kinds of classes, not only those supported by our host VM. For instance, it is trivial to model immediate Floats or Characters in a system that doesn't support them. This feature could have made the migration of Squeak and Cuis, to Closures and Spur much easier and less hacky in the past, if we had something like this.

Additionally, references from these objects are controlled carefully, so there's no risk of unwanted objects leaking into the new image.



Design of Bootstrap 5

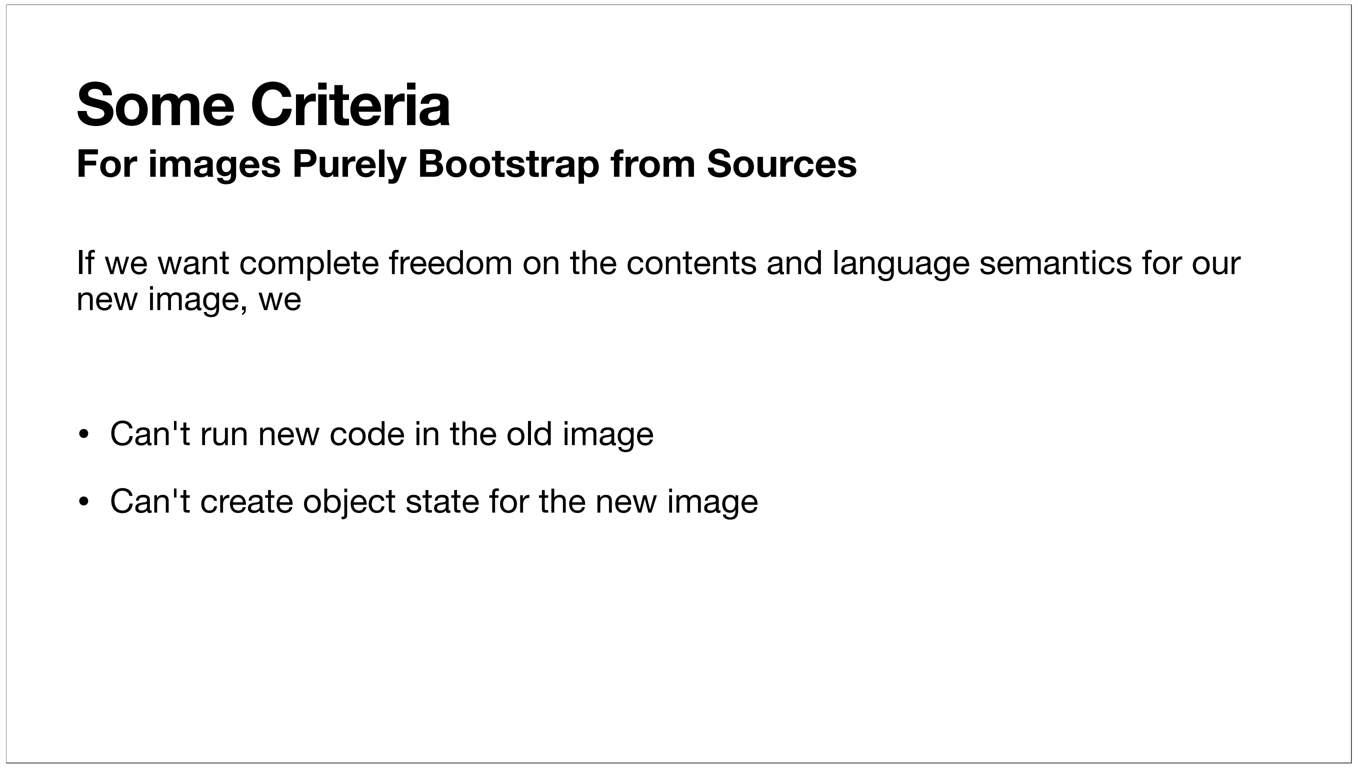
Hierarchy of ImageBuilders

- ImageBuilder
    - SpurImageBuilder
        - Spur32ImageBuilder
        - Spur64ImageBuilder

In order to turn this object graph into a new SmalltalkImage comprised of the real objects we want, we need an ImageBuilder. The main purpose of an ImageBuilder is to traverse the collection of ForeignObjects, and for each of them write the bits for the object header, and then write the outward pointers or literal bytes for the object contents.

Format specific ImageBuilders also know how to build format specific data structures, the image file header and other details.

Currently we have ImageBuilders for 32 bit and 64 bit Spur image formates.



Some Criteria

For images Purely Bootstrap from Sources

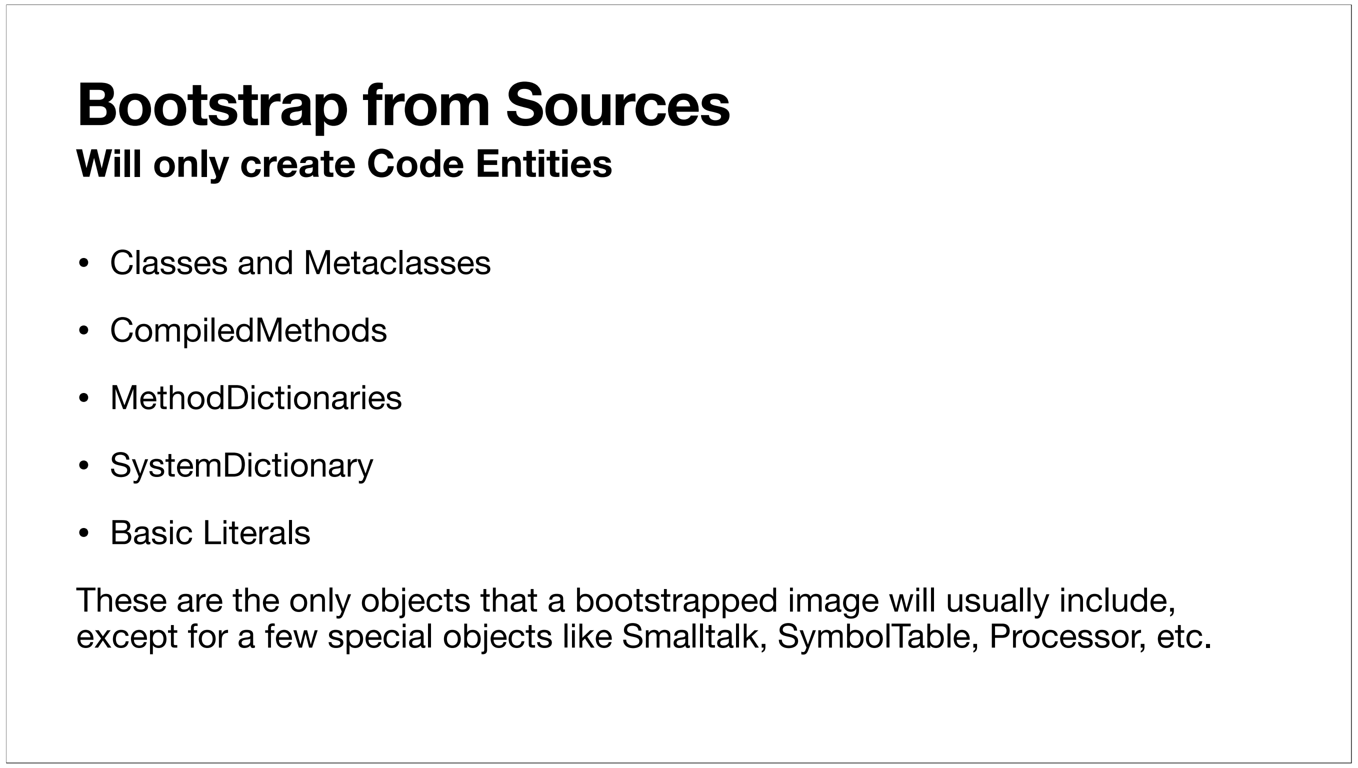
If we want complete freedom on the contents and language semantics for our new image, we

- Can't run new code in the old image
- Can't create object state for the new image

The new images could be very different from our existing image. Not only it will include a different set of classes and methods, but it could even run on a VM with different execution semantics. This means we can not run any code that belongs in the new image until it is built, saved to disk, and run by itself on its appropriate VM.

We can not run any of that code in our tool. We may not even have execution semantics for that.

It also means that we can't create arbitrary objects for the new image in our build tool, because the build tool runs in the "old" image, and doesn’t include those classes.



Bootstrap from Sources

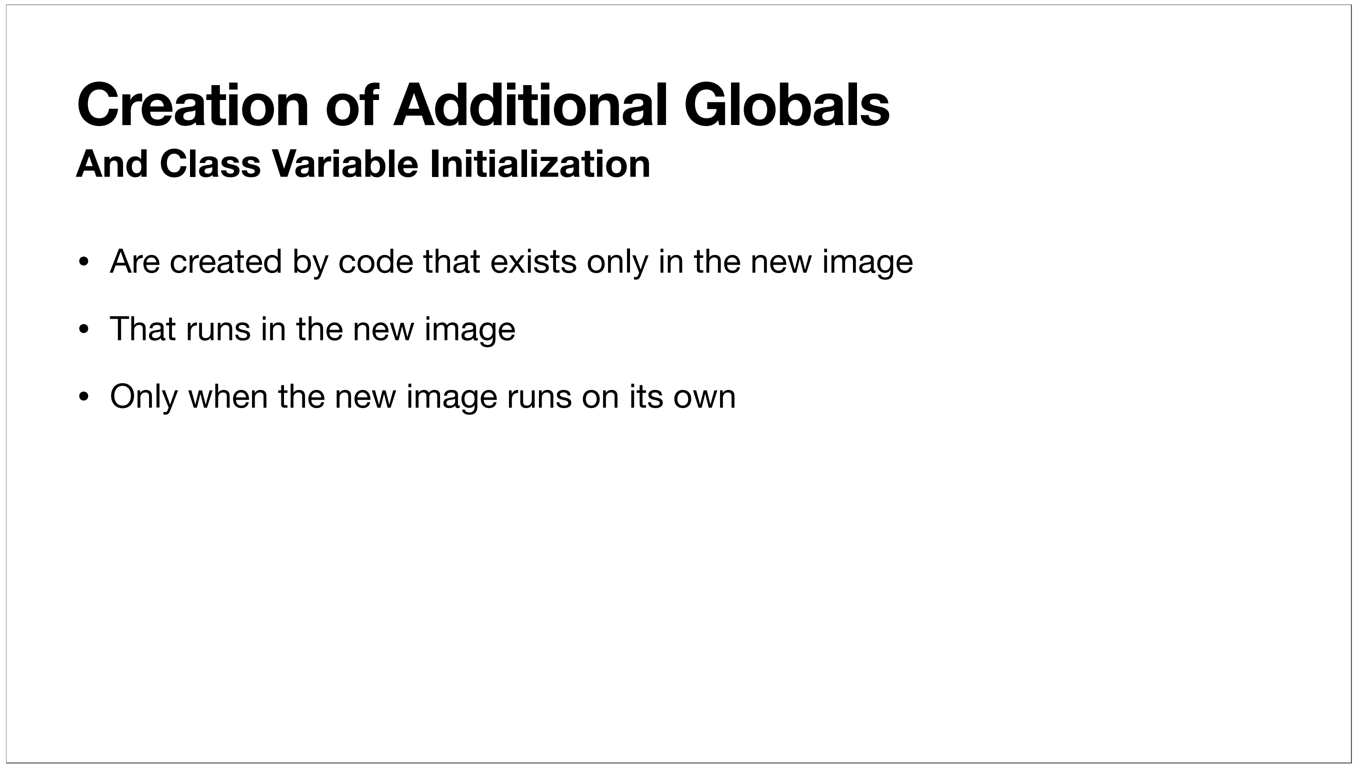
Will only create Code Entities

- Classes and Metaclasses
- CompiledMethods
- MethodDictionaries
- SystemDictionary
- Basic Literals

These are the only objects that a bootstrapped image will usually include, except for a few special objects like Smalltalk, SymbolTable, Processor, etc.

Because of all this, by default, ImageBuilder tool will only create a few specific kinds of objects. These are objects for code: Classes, Metaclasses, CompiledMethods, MethodDictionaries, SystemDictionary and basic Literals.

In short, our bootstrapped images will only include Smalltalk code, plus a few special objects, like the Smalltalk SystemDictionary, a SymbolTable, the Processor, and very few others, that are required by the VM to run.



Creation of Additional Globals

And Class Variable Initialization

- Are created by code that exists only in the new image
- That runs in the new image
- Only when the new image runs on its own

Any other object, for example, additional globals and class variables, must be created in the new image, by code included as part of the new image, and only when the new image is run by itself.

Demo time!

Yep. Let see it in action!

(Actually, the demo will be done after the presentation. Change of plans.)

Existing Alternatives

There have been several previous attempts at building new images in the Smalltalk-80 / Squeak / Pharo / Cuis lineage. When applied to our stated problem, each of these approaches has specific problems. Let's review them.

SystemTracer
Smalltalk-80 / Ted Kaehler

+ Clones a running image from within itself

+ Can convert to a different image format

- Can not specify what is to be included

- Existing and new image are same, any code is ran on "both"

- Brittle and hacky (for instance, recreation of hashed collections)

- Any difference in Image/VM capability is handled by complicated specific code

MicroSqueak

Squeak / John Maloney

+ Uses a separate MObject hierarchy

- Part of the host image metamodel

- Many parts of the image can not be made different

- Brittle and hacky

- New image is in the same format as host image

Spoon
Squeak / Craig Latta

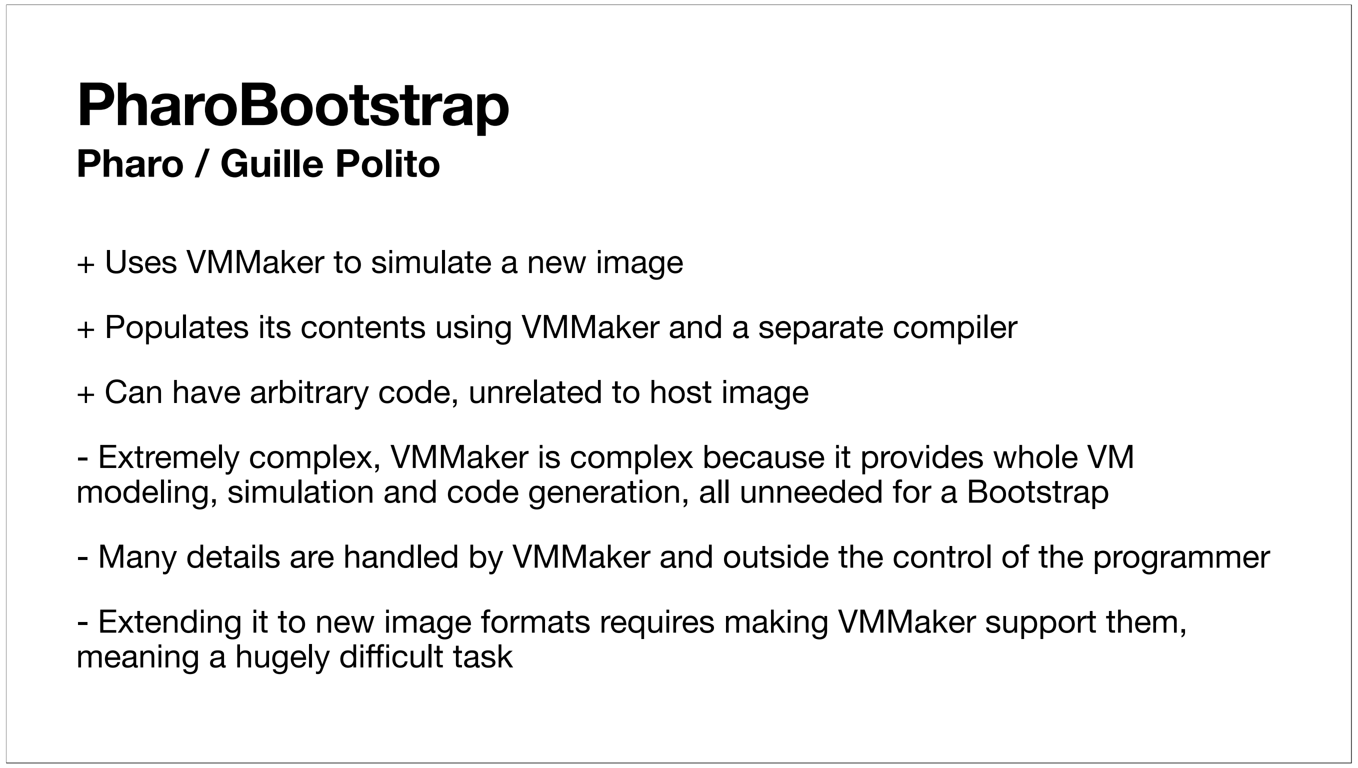
+ Minimal runtime loads code on demand from a separate running image

+ Good to manually "grow" a subset of an existing image

- Doesn't give full control on image contents

- Code comes from a separate running image, not from source code

- New image is in the same format as mother image



PharoBootstrap

Pharo / Guille Polito

+ Uses VMMaker to simulate a new image

+ Populates its contents using VMMaker and a separate compiler

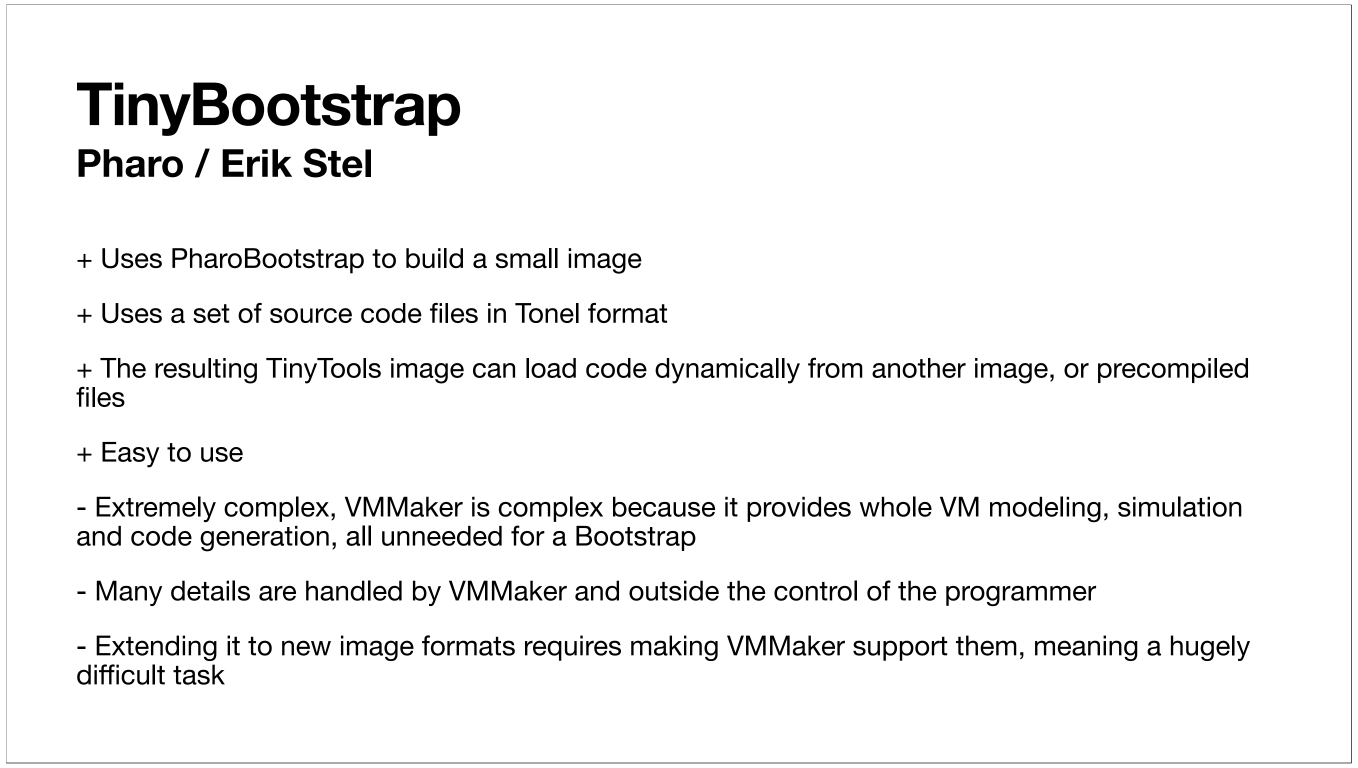
+ Can have arbitrary code, unrelated to host image

- Extremely complex, VMMaker is complex because it provides whole VM modeling, simulation and code generation, all unneeded for a Bootstrap

- Many details are handled by VMMaker and outside the control of the programmer

- Extending it to new image formats requires making VMMaker support them, meaning a hugely difficult task

Note: In the presentation it was stated: "Can not be extended to new image formats" and that was later corrected in the Q&A section by Esteban Lorenzano. This slide is corrected after that.



TinyBootstrap

Pharo / Erik Stel

+ Uses PharoBootstrap to build a small image

+ Uses a set of source code files in Tonel format

+ The resulting TinyTools image can load code dynamically from another image, or precompiled files

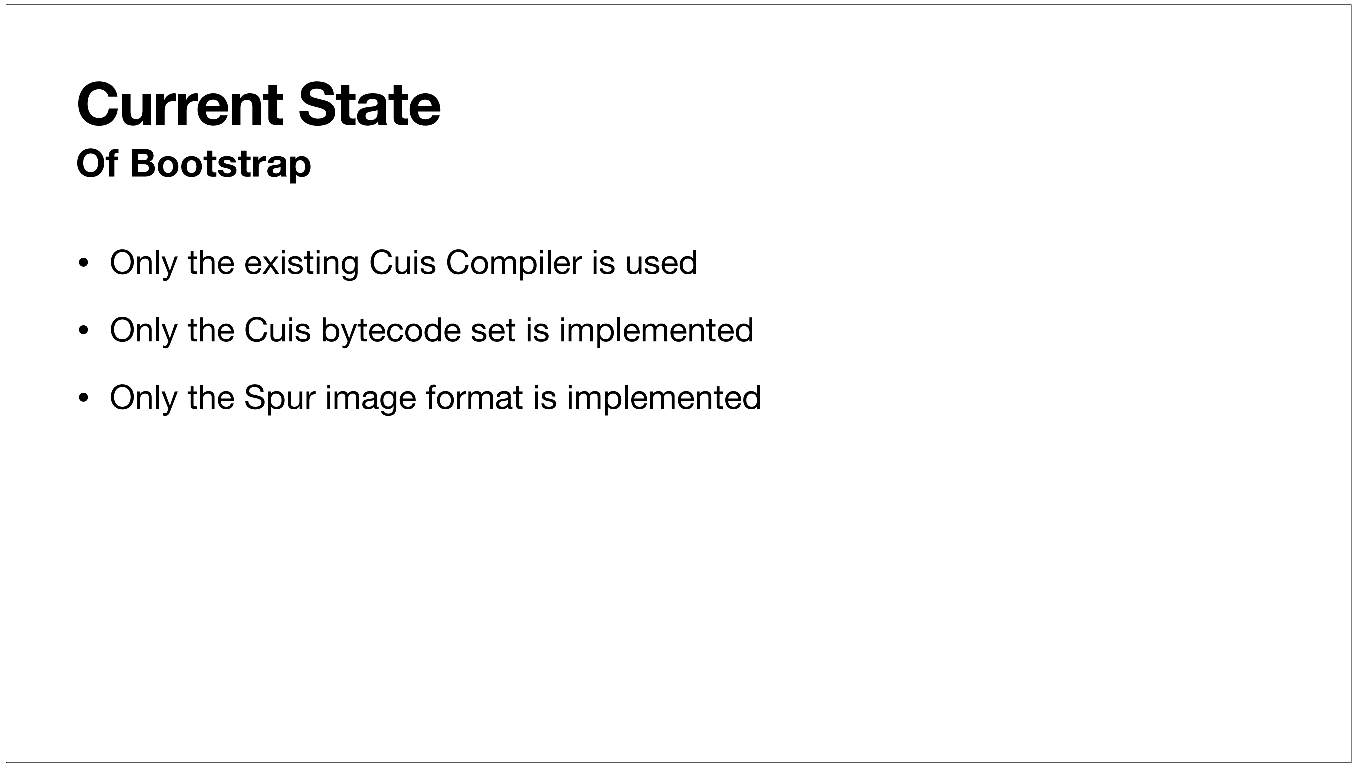
+ Easy to use

- Extremely complex, VMMaker is complex because it provides whole VM modeling, simulation and code generation, all unneeded for a Bootstrap

- Many details are handled by VMMaker and outside the control of the programmer

- Extending it to new image formats requires making VMMaker support them, meaning a hugely difficult task

Note: In the presentation it was stated: "Can not be extended to new image formats" and that was later corrected in the Q&A section by Esteban Lorenzano. This slide is corrected after that.



Current State

Of Bootstrap

- Only the existing Cuis Compiler is used
- Only the Cuis bytecode set is implemented
- Only the Spur image format is implemented

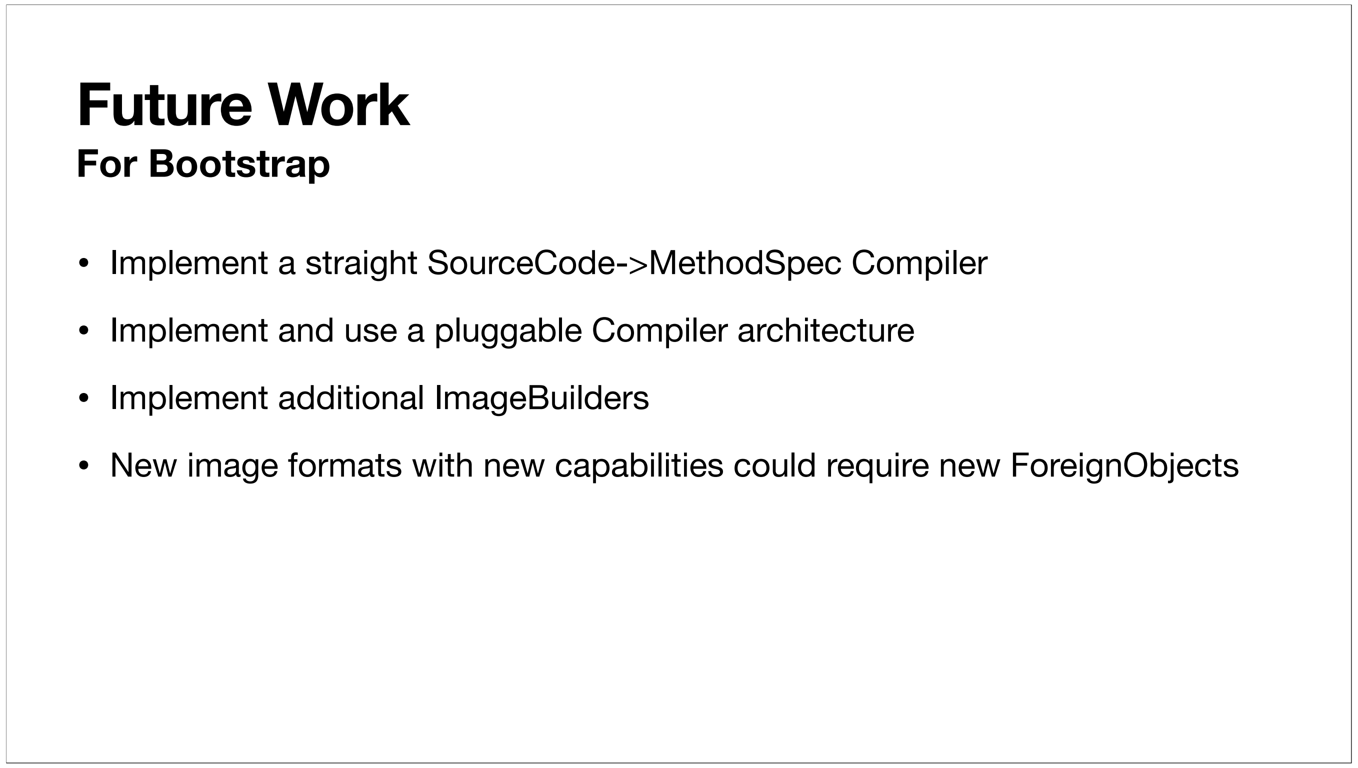
So, none of the existing approaches did what we wanted, so we needed to do it anew.

The Bootstrap framework is designed with all those ideas and is hopefully easy to extend, but so far we have only been using the existing Cuis compiler, and that means that right now we can not change language semantics. That would need pluggable compilers that generate different formats of CompiledMethods or enough information to materialize these CompiledMethods.

Right now we are only using the regular Cuis Compiler, and the standartd V3 bytecode set, and we only have builders for the Spur image formats. The knowledge of the details of the Spur image formats is well localized in the design. Supporting additional image formats is mostly writing a new builder for them.

Aso, as of today, ClassSpecs and ImageSpecs are created from existing classes and methods in the system. Complete separation of the contents of the new image in its own Source Code files requires additional work.

The system is designed for all that, and this is the first iteration.

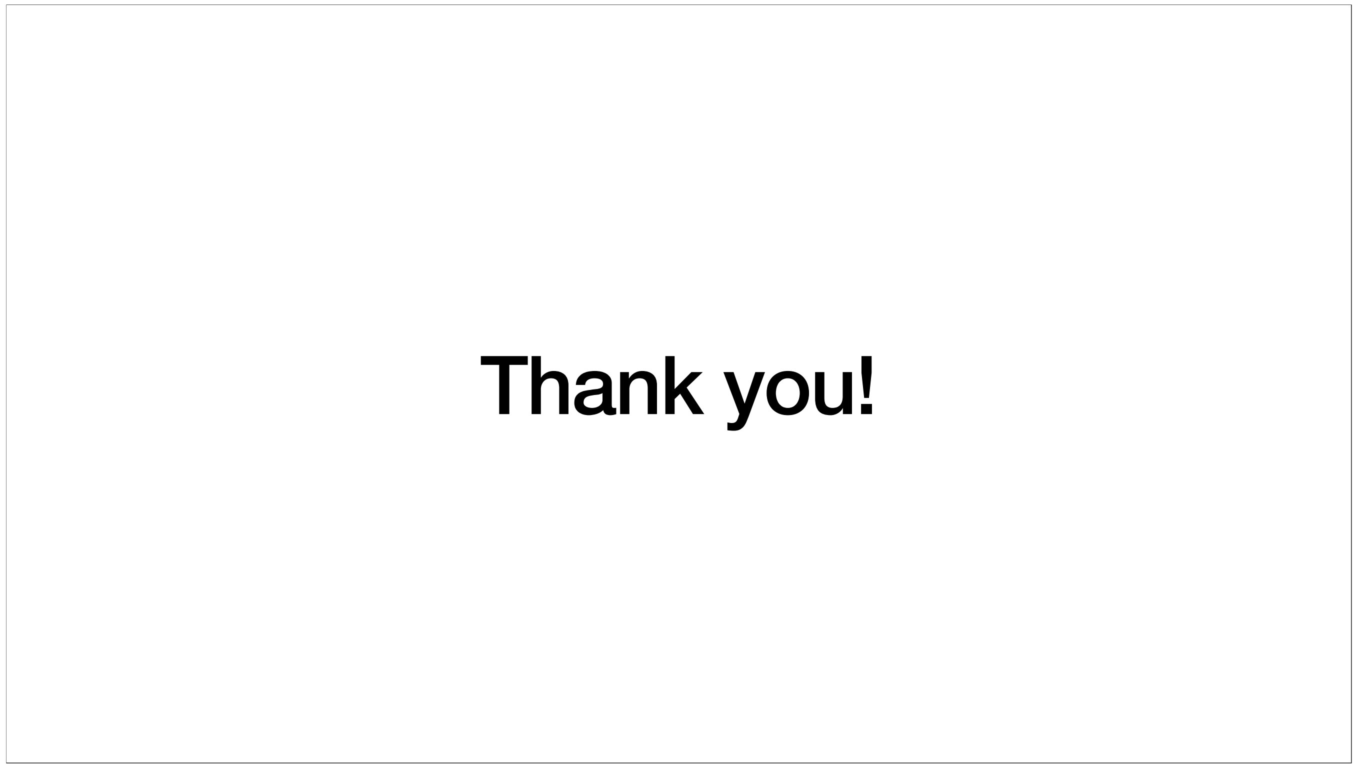


Future Work

For Bootstrap

- Implement a straight SourceCode->MethodSpec Compiler
- Implement and use a pluggable Compiler architecture
- Implement additional ImageBuilders
- New image formats with new capabilities could require new ForeignObjects

So, Future Work includes actually writing the missing parts. They include a separate Smalltalk Compiler that can compile SourceCode straight into ClassSpecs and MethodSpecs, using any desired bytecode set. It also includes writing ImageBuilders for supporting additional image formats and VMs, and most likely new kinds ForeignObject to create whatever new kinds of objects those new VMs will support.



Thank you!

That’s all. It is my hope that you find Bootstrap both interesting and useful.

Thanks for your time.