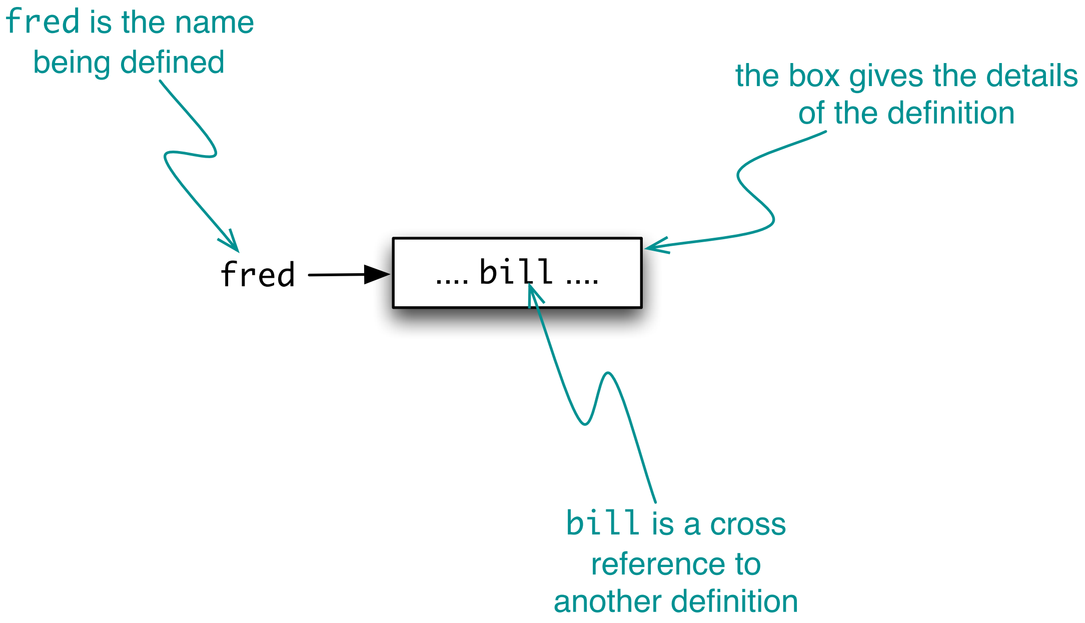
The Design Rationale for Name Space Management in Spice

One of the hardest issues of programming language design is managing names. Programs consist of many (possibly very many) named definitions, every name being different. When programs get large, finding memorable but distinct names for definitions can become very burdensome. If several people are collaborating on the program, it can be intolerable.

The basic solution is to break up the global collection of definitions. We put them into separate groups called packages with special rules for cross-referencing.

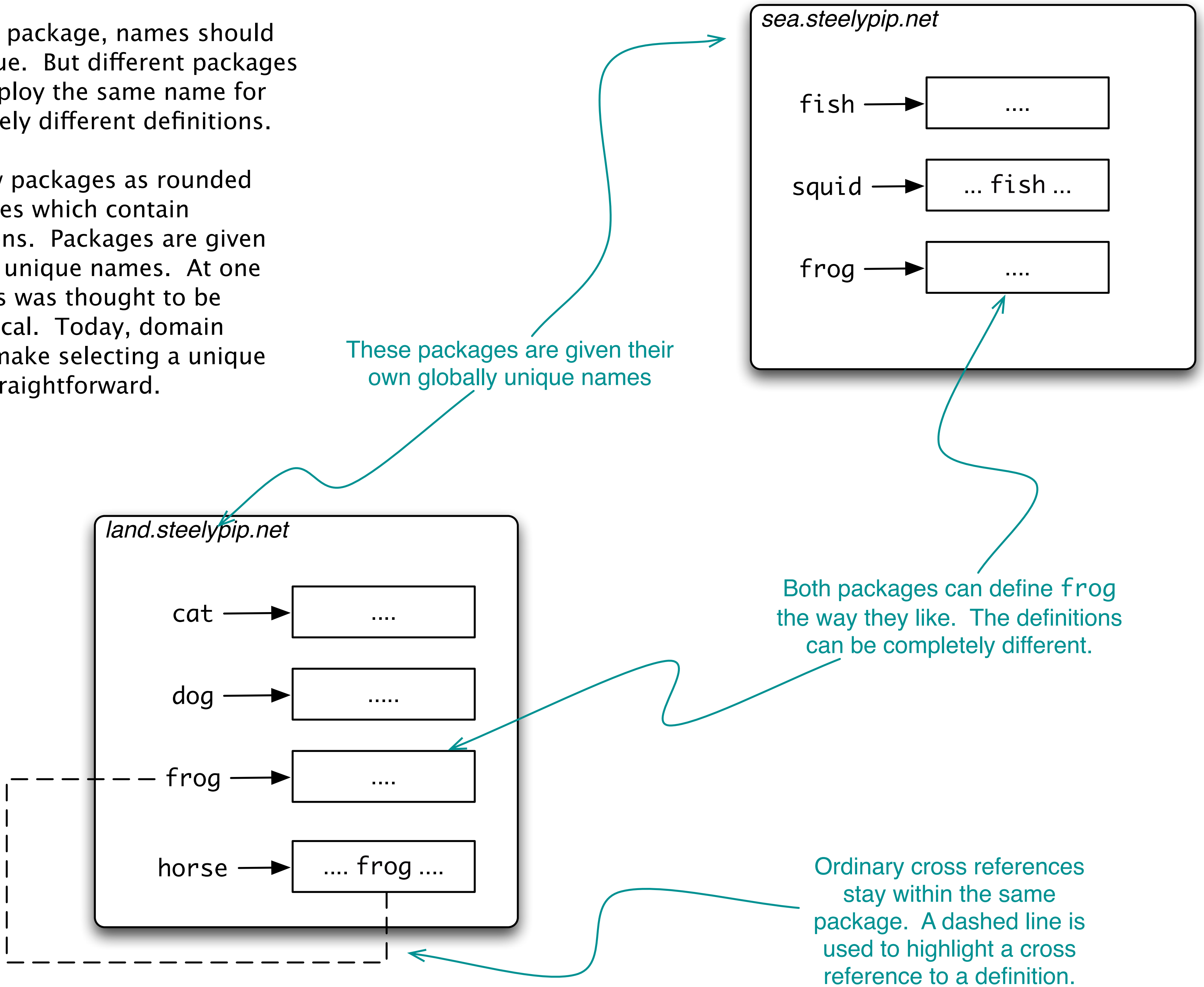
We draw a named definition as shown right. The thick arrow points to the contents of the definition and may contain cross-references to other definitions.


fred is the name being defined
the box gives the details of the definition
fred
.... bill ....
bill is a cross reference to another definition

Packages Isolate Definitions

Within a package, names should be unique. But different packages may employ the same name for completely different definitions.

We draw packages as rounded rectangles which contain definitions. Packages are given globally unique names. At one time this was thought to be impractical. Today, domain names make selecting a unique name straightforward.


sea.steelypip.net
fish
....
squid
... fish ...
frog
....
These packages are given their own globally unique names
land.steelypip.net
cat
....
dog
.....
frog
....
horse
.... frog ....
Both packages can define frog the way they like. The definitions can be completely different.
Ordinary cross references stay within the same package. A dashed line is used to highlight a cross reference to a definition.

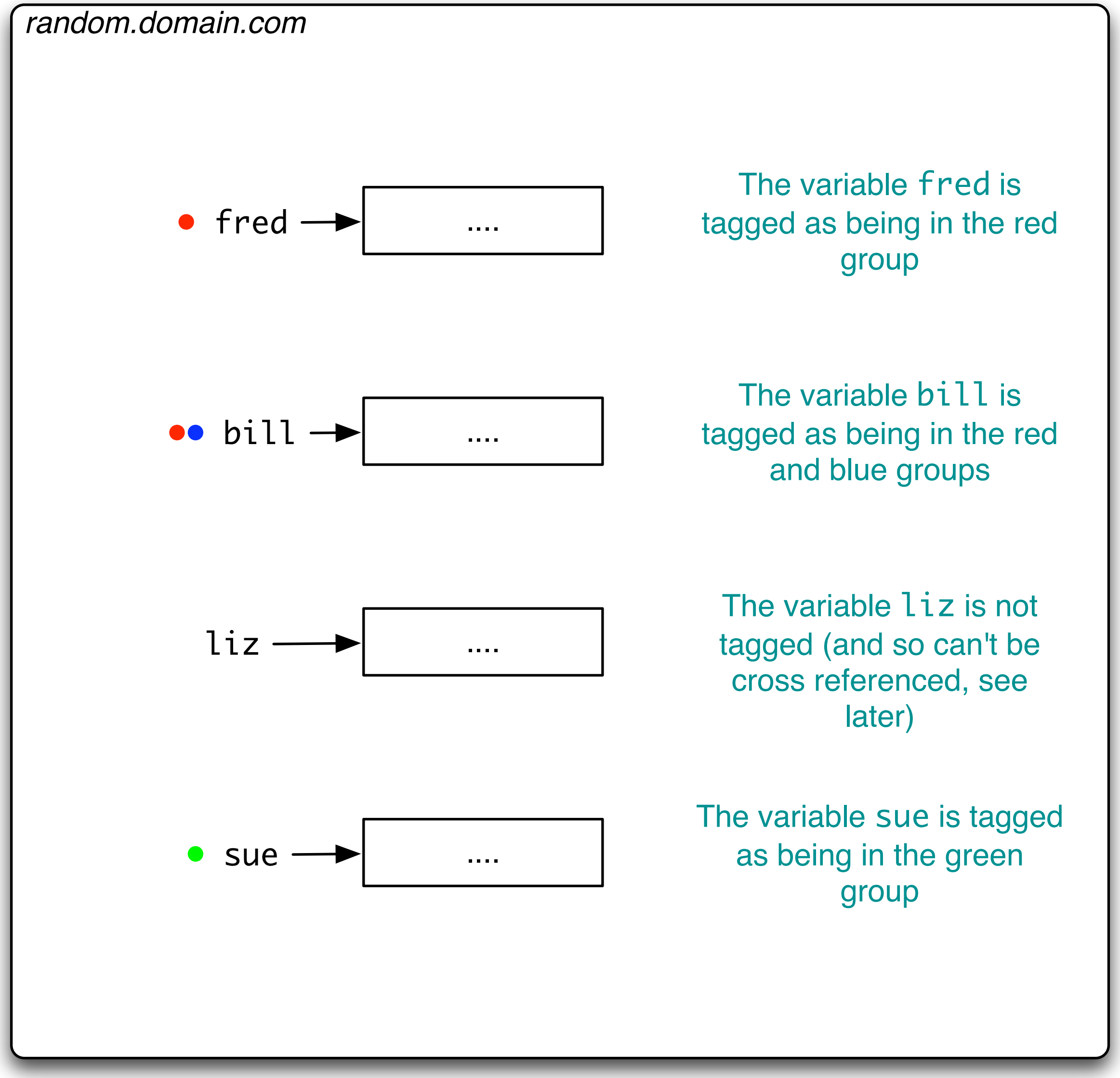
Categorising Definitions

We try to put related definitions into the same package, aiming to keep cross-references within the one package. Nevertheless, we must still be able to make cross-references to definitions in other packages.

But the definitions within a package are not all of the same status. Some definitions are important ones designed for external consumption. Other definitions are purely internal and created only to simplify the program.

When we cross reference between packages, we will typically want to ignore internal definitions.

To handle this, we can put tags on definitions to group them into categories. These are shown pictorially as coloured bullets. You can put more than one bullet on each definition.


random.domain.com
fred
....
The variable fred is tagged as being in the red group
bill
....
The variable bill is tagged as being in the red and blue groups
liz
....
The variable liz is not tagged (and so can't be cross referenced, see later)
sue
....
The variable sue is tagged as being in the green group

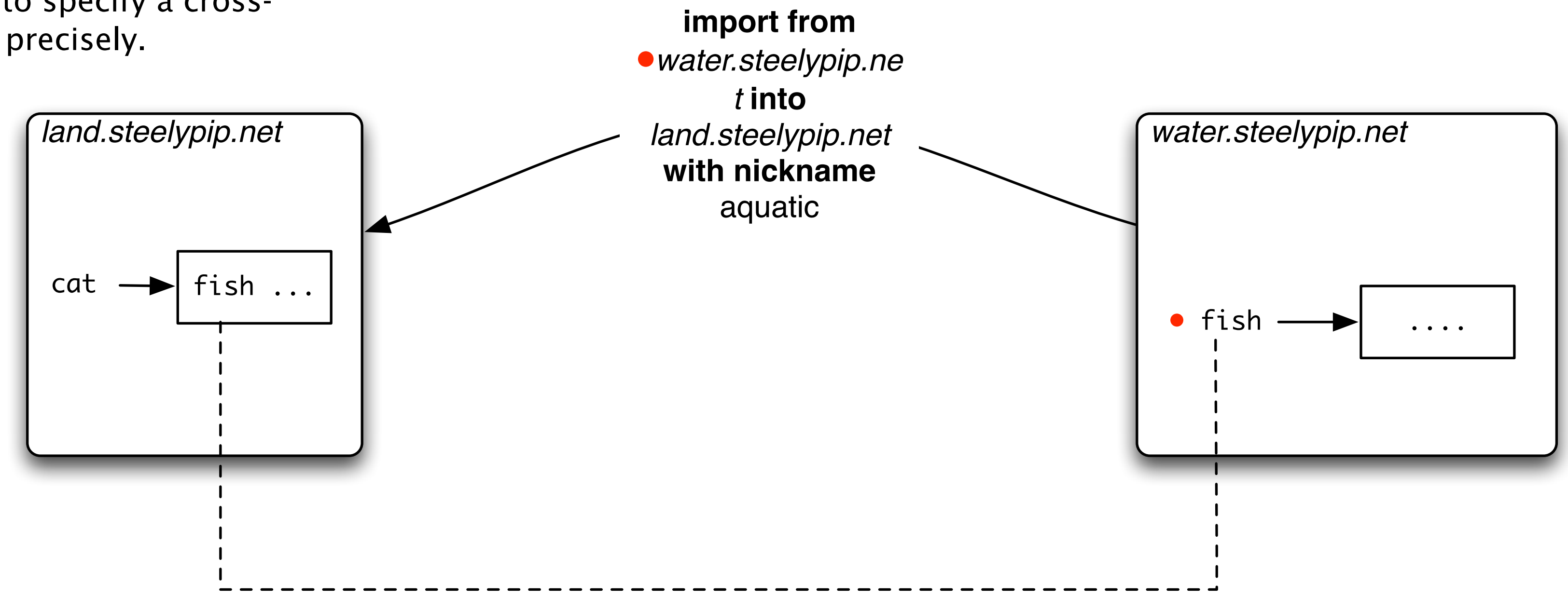
In a typical program, there are a small number of categories (bullets) in regular use. Typical categories are public, deprecated, and private. These categories are chosen to remind people who is allowed to change the definitions and who is allowed to use the definitions. But programmers are allowed to invent as many new categories as they like, which can becomes useful as a program matures and changes.

Cross-references Between Packages

Because an ordinary cross-reference is supposed not to see a definition in a different package, we need to add a special mechanism for letting one package see the definitions of another package.

This mechanism is called an import. An import makes the definitions of a particular category in another package available for cross-referencing.

Imports are optionally given nicknames, which are used when you need to specify a cross-reference precisely.



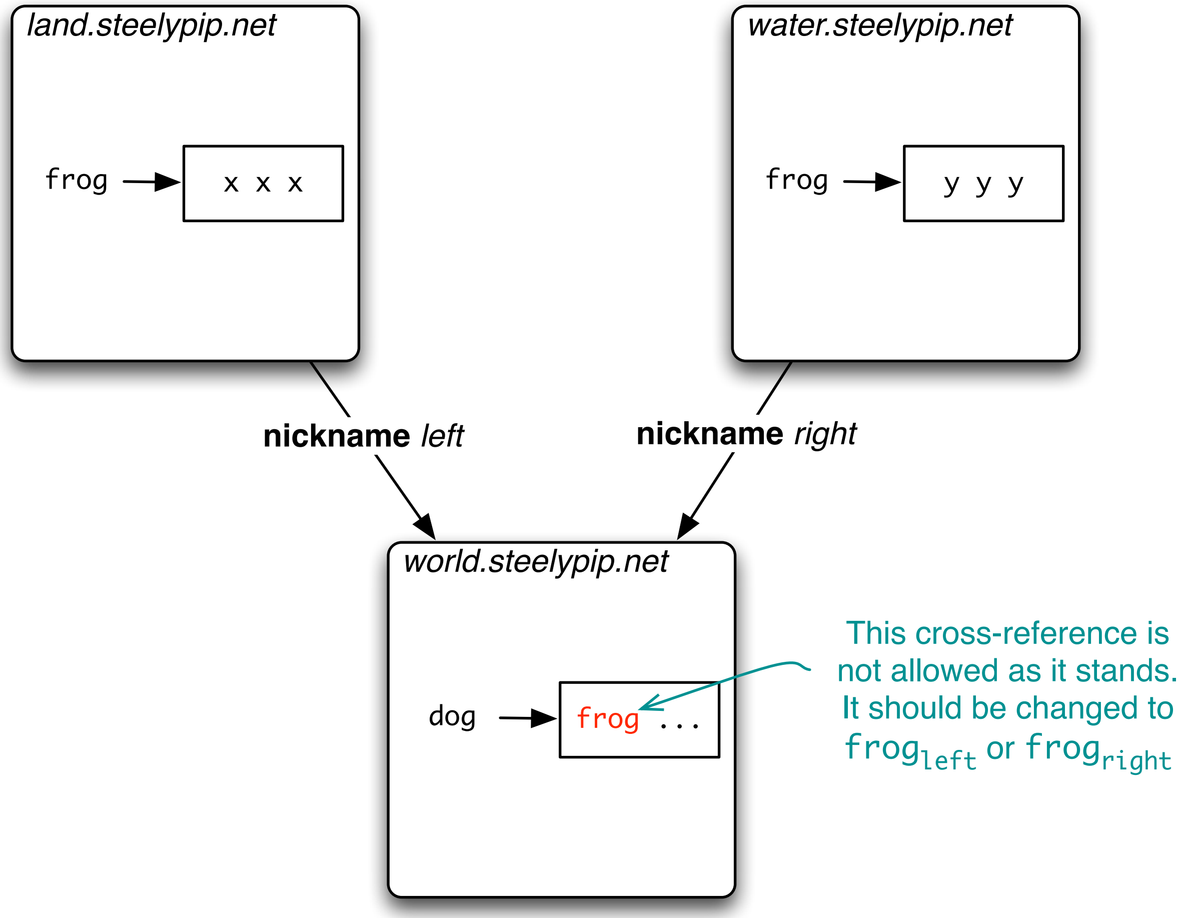
Matching Rules

Matching a cross-reference to the correct definition means searching at all the local definitions and then all the imported definitions. Local definitions are always preferred to imported definitions.

However, we refuse to chose between alternative imported definitions. If two imports offer different definitions with the same name, that cross-reference is deemed illegal!

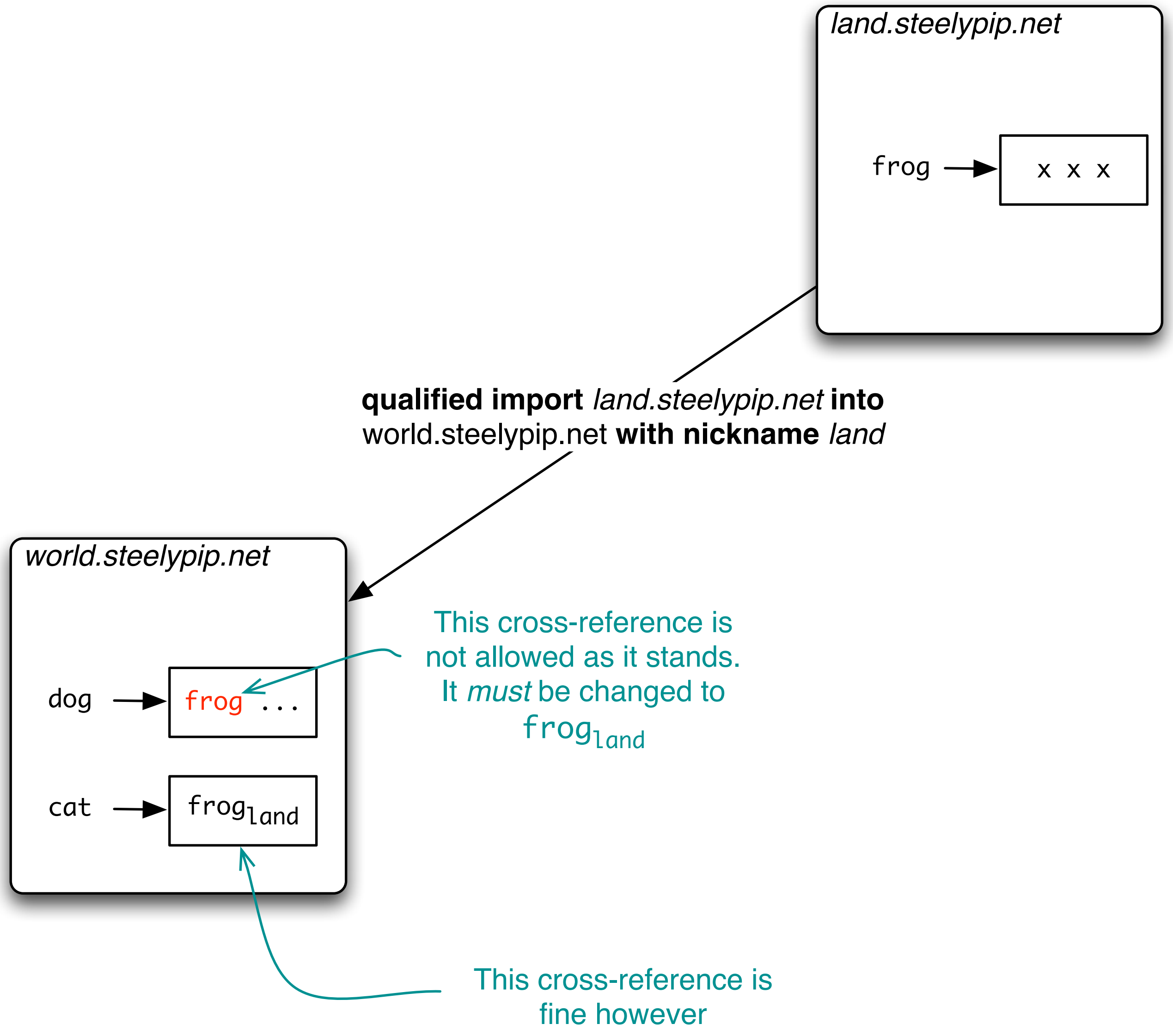
This is a very important policy decision. Firstly, it doesn't prevent imports just because of a small risk of an accidental match. Secondly, it exposes and prevents actual ambiguity. But the price is that some cross-references need to be more complicated.

To make these cross-references work, we require that they specify the nickname of the import they use. We call these *qualified* cross-references.



Mandatory Nicknames: Qualified Imports

Cross-references do not have to specify a nickname for ordinary imports. But an import can be marked as demanding them, in effect only permitting qualified cross-references.

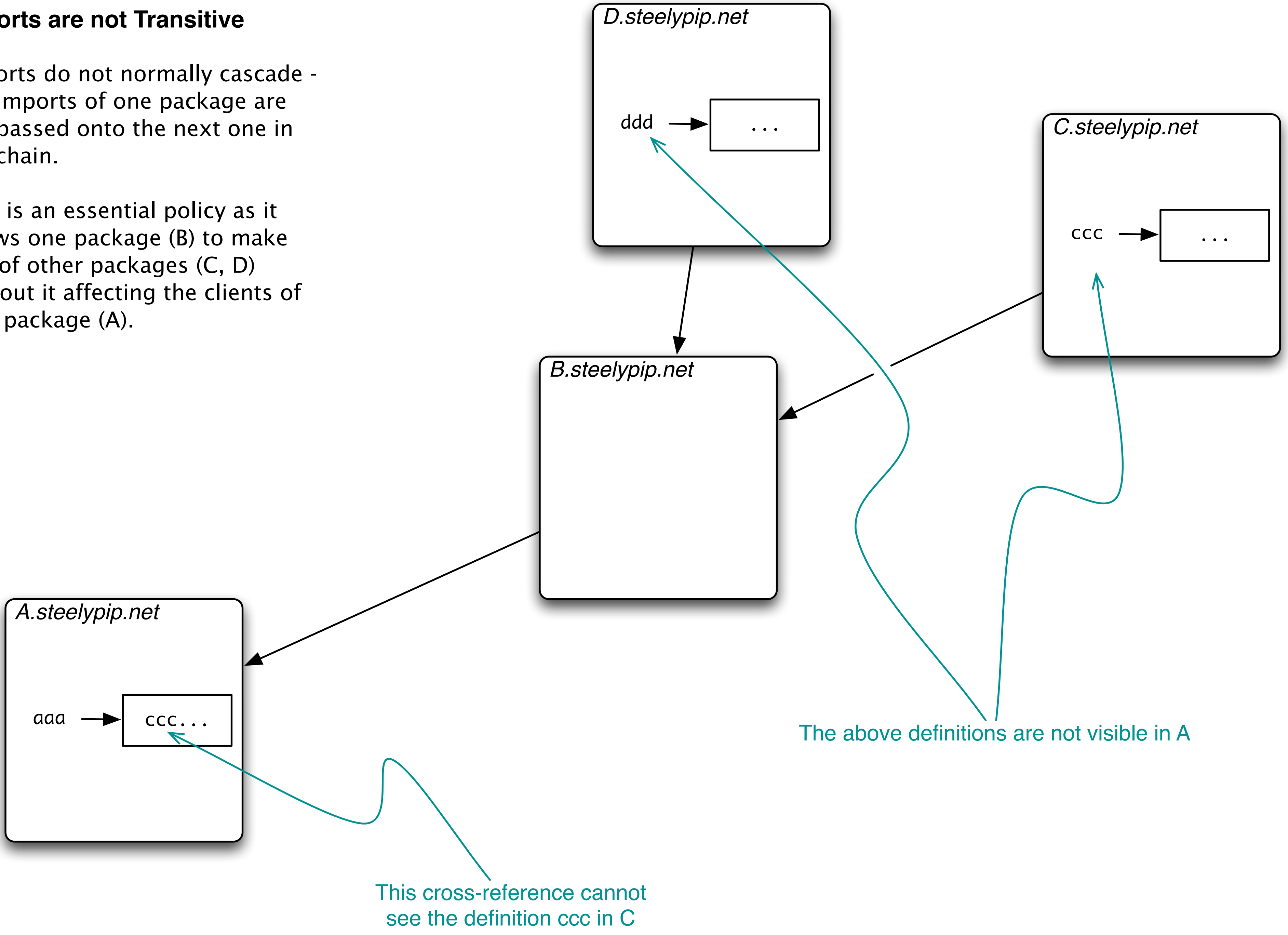


land.steelypip.net
frog
x x x
qualified import *land.steelypip.net* **into** world.steelypip.net **with nickname** *land*
world.steelypip.net
dog
frog ...
cat
frog$_{land}$
This cross-reference is not allowed as it stands. It *must* be changed to frog$_{land}$
This cross-reference is fine however

Imports are not Transitive

Imports do not normally cascade - the imports of one package are *not* passed onto the next one in the chain.

This is an essential policy as it allows one package (B) to make use of other packages (C, D) without it affecting the clients of that package (A).



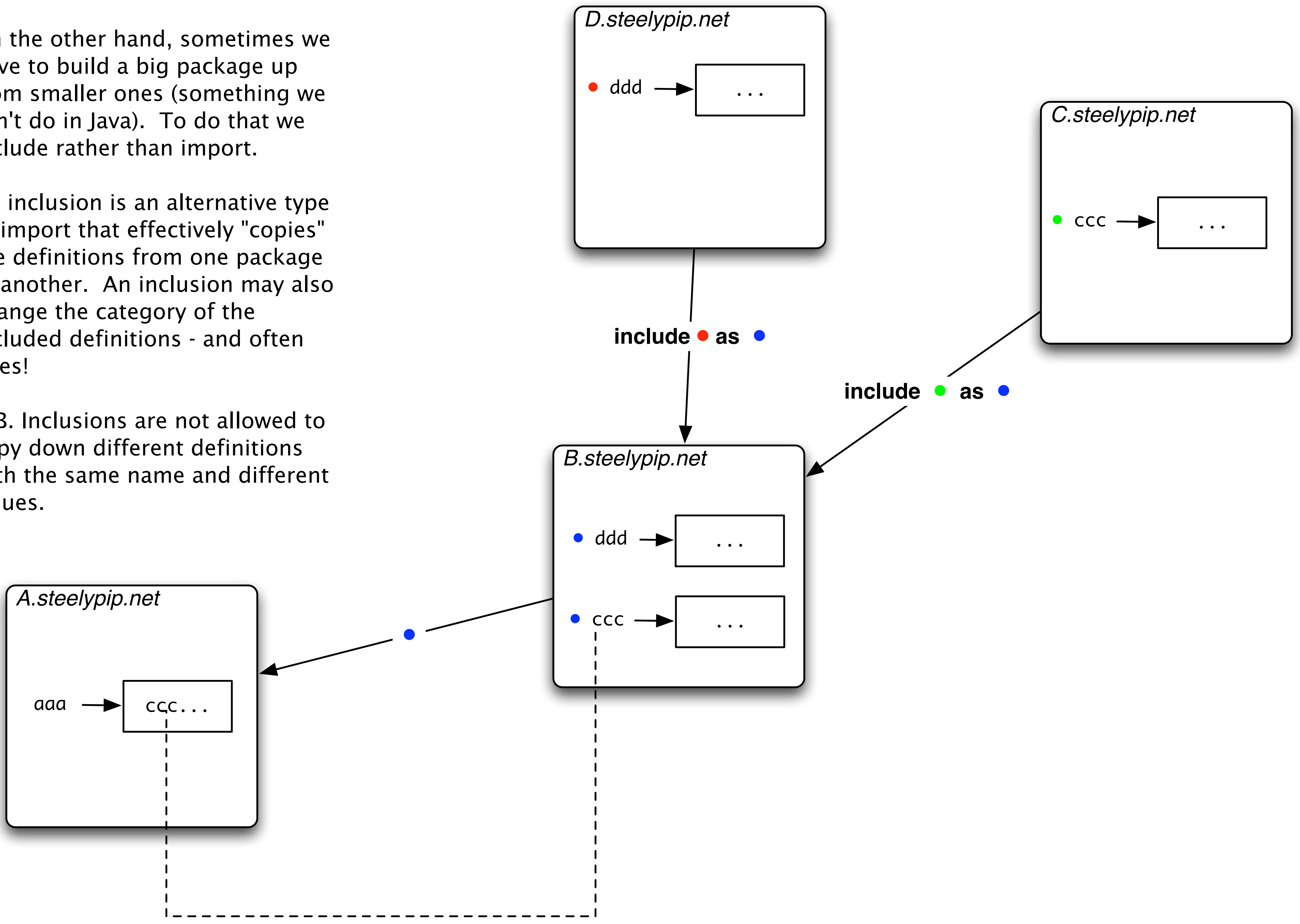
D.steelypip.net
ddd
. . .
C.steelypip.net
ccc
. . .
B.steelypip.net
A.steelypip.net
aaa
ccc. . .
The above definitions are not visible in A
This cross-reference cannot see the definition ccc in C

Building Packages Incrementally

On the other hand, sometimes we have to build a big package up from smaller ones (something we can't do in Java). To do that we include rather than import.

An inclusion is an alternative type of import that effectively "copies" the definitions from one package to another. An inclusion may also change the category of the included definitions - and often does!

N.B. Inclusions are not allowed to copy down different definitions with the same name and different values.

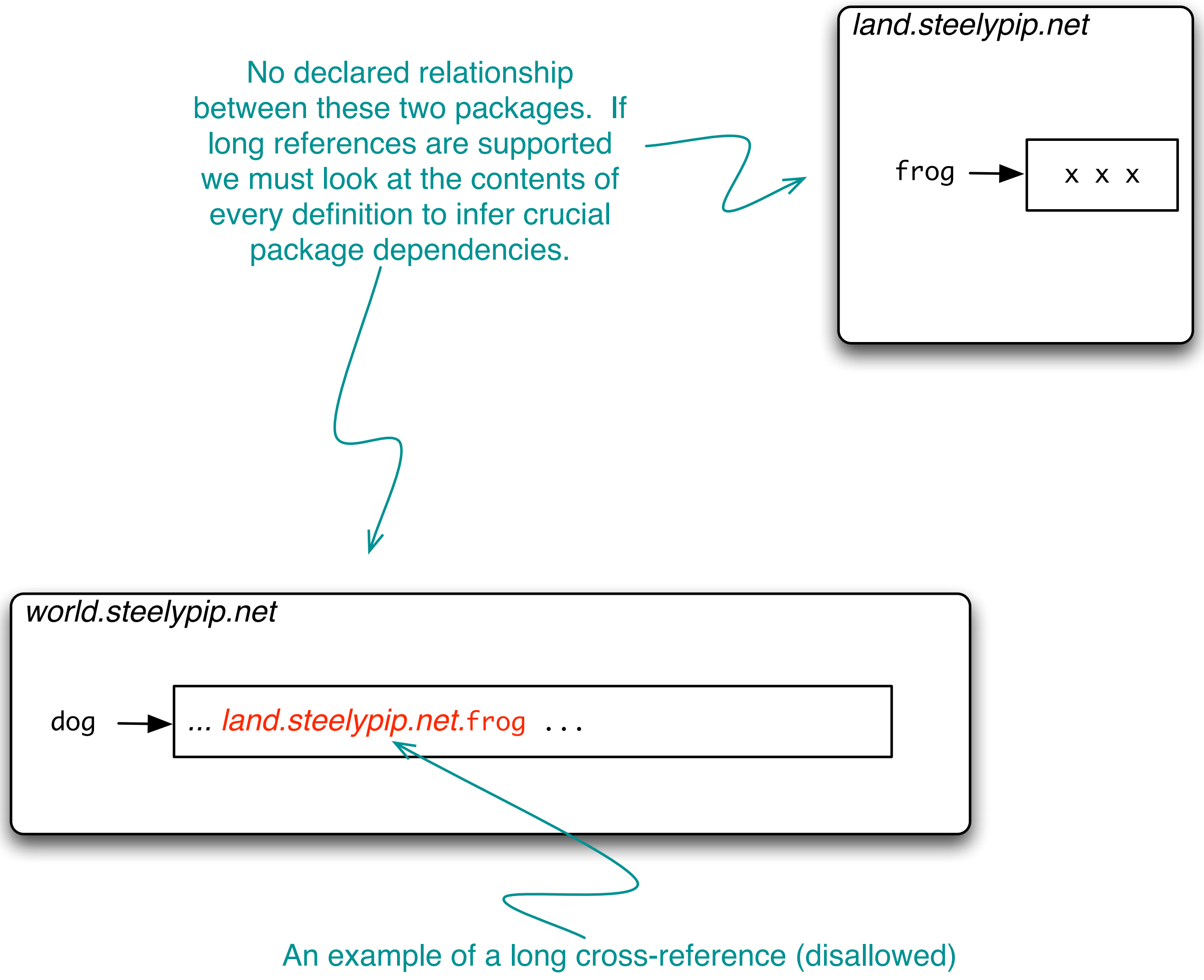


What is not Supported

In other programming languages, it is common to support an explicit "long" form of cross-reference. These are a bad idea because they bury the relationship between packages deep into the contents of the definitions.

We argue that these require sophisticated support from an IDE in order to make them safe. But if you have a sophisticated IDE you have a better option available; the visual suppression of nicknames and the automatic maintenance of interfaces.

We also do not support imports at a finer grain size than a category. We may do this at a later date.


land.steelypip.net
frog
x x x
No declared relationship between these two packages. If long references are supported we must look at the contents of every definition to infer crucial package dependencies.
world.steelypip.net
dog
... land.steelypip.net.frog ...
An example of a long cross-reference (disallowed)

How This Works in Spice

It is helpful to see examples of the above in the actual syntax of a programming language.

A typical named definition with the name highlighted in magenta and the contents of the definition marked in orange $\longrightarrow$

```
define cube( n ) =>
    n * n * n
enddefine
```

cross-references underlined $\longrightarrow$

```
define sixth_power( n ) =>
    cube( n ) * cube( n )
enddefine
```

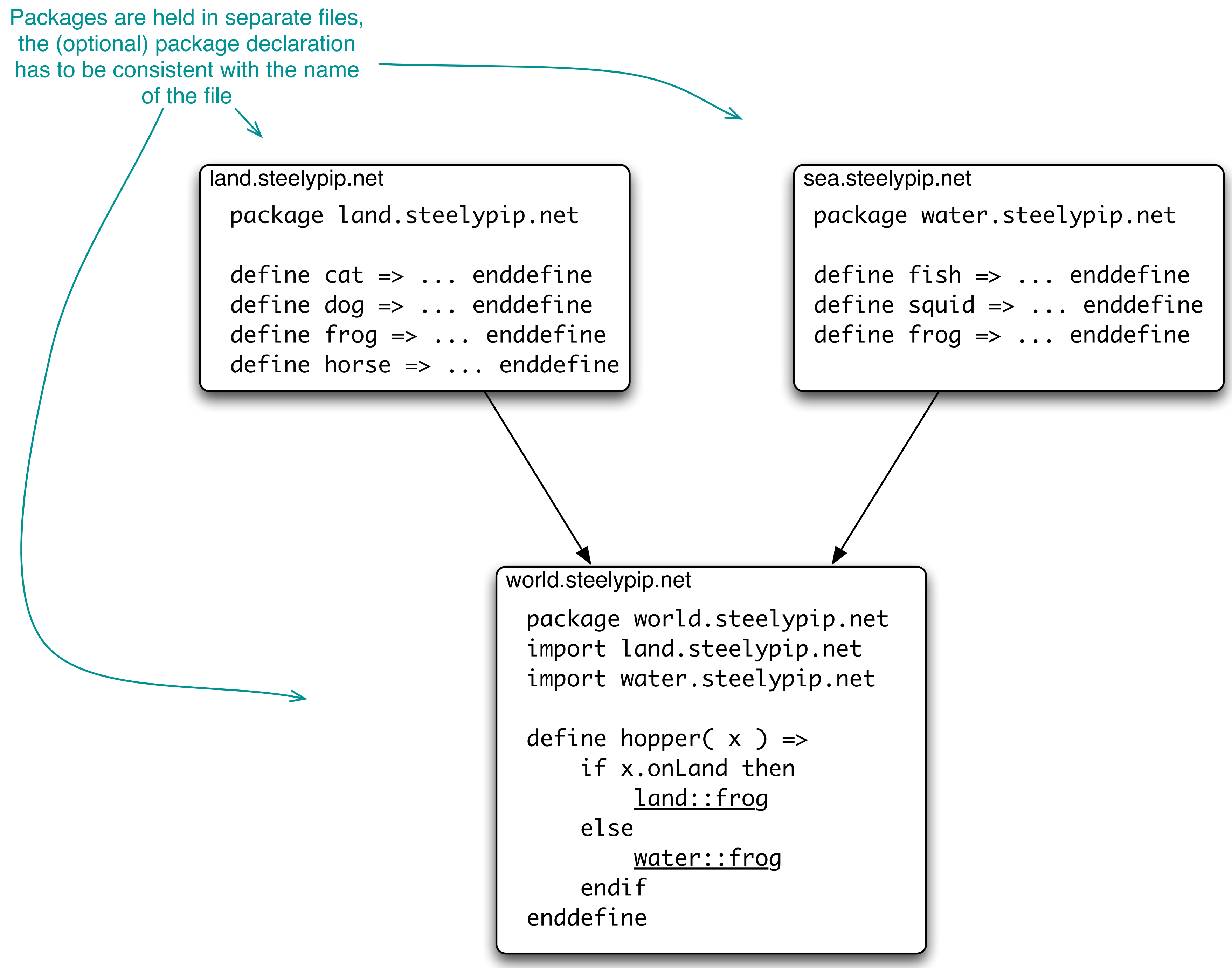
An import takes the default nickname of its first part, in this case "sort". The qualified cross reference is underlined. By default the "public" category is imported. $\longrightarrow$

```
import sort.steelypip.net;

define s3(a, b, c) =>
    {a, b, c}.sort::shortSort() ...
enddefine
```

A definition can be assigned to another category by prefixing it with an annotation in brackets. In this case we put "dup" in the private category $\longrightarrow$

```
[private]
define dup(x) =>
    x, x
enddefine
```


Packages are held in separate files, the (optional) package declaration has to be consistent with the name of the file
land.steelypip.net
sea.steelypip.net
world.steelypip.net

```
package land.steelypip.net

define cat => ... enddefine
define dog => ... enddefine
define frog => ... enddefine
define horse => ... enddefine
```

```
package water.steelypip.net

define fish => ... enddefine
define squid => ... enddefine
define frog => ... enddefine
```

```
package world.steelypip.net
import land.steelypip.net
import water.steelypip.net

define hopper( x ) =>
    if x.onLand then
        land::frog
    else
        water::frog
    endif
enddefine
```