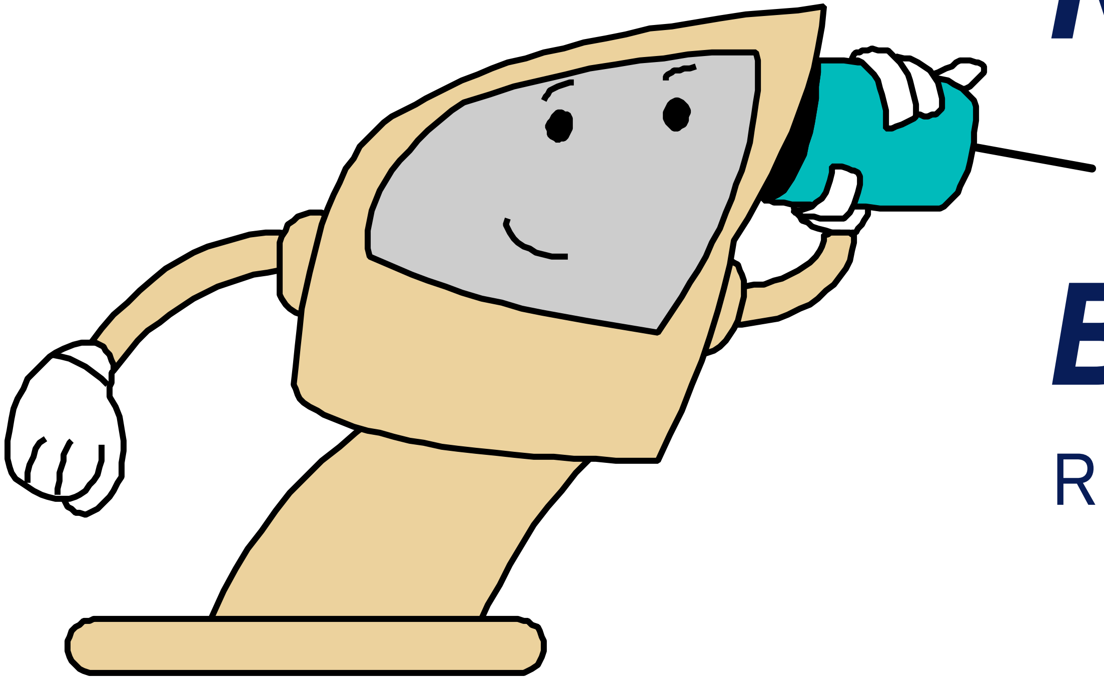


Naming and Binding

Richard Hayton

Context

- FlexiNet a Java middleware *framework*
  - It supports 'slot-in' components to support different abstractions
  - It is a project in its own right, and has been used for various other projects and investigations
    - FollowMe - mobile and persistent objects
    - Secure Sessions - exploring security mechanisms
    - Java Engineering - how to build with components
- It is now almost 2 years since its conception
  - I'm trying to tie up lots of loose ends
  - ... and write an architecture report

Recent Developments...

- In the last few months….
  - Transaction Integration with FlexiNet (Wu)
  - SSL Integration with FlexiNet (Laurence)
  - FlexiNet blueprints for binders (Peter)
  - Multicast in FlexiNet (Dave)
- Four people using and extending parts of FlexiNet they were previously unfamiliar with
  - Real ‘Power’ Users!
- This has lead to some useful feedback

Multiple Binding Protocols

- There is a requirement to manage different types of binding for use in different circumstances
  - Transactional .v. NonTransactional References
  - Insecure .v. Authenticated .v. Encrypted References
  - Multicast .v. Unicast References

- FlexiNet is capable of supporting types of binding
  - but up until now there were only ad-hoc mechanisms for manage the additional complexity
  - *very* steep learning curve

Binders we have built

● Green	REX over UDP	
● Yellow	REX over TCP	
● Rose	REX over TCP with SSL	
● Lemon	REX over TCP with SSL & mobility	
● Blue	REX over UDP with	mobility
● Magenta	RRP over TCP	(mobility)
● Crimson	RRP over TCP with SSL	(mobility)
● Burgundy	RRP over TCP using Blueprints	
● Purple	Same domain binder	
● Black	RMP over UDP for multicast	

Too Many Binders?

- There are a lot of binders!
  - Lots of potential complexity
  - Adding an extra dimension doubles the number
- There is a lot of common functionality
  - Extra binder classes can be avoided by configuration
  - Recently the degree of configuration has increased

A binder may be configured to define a new protocol

- Problem: This won't be wire compatible with the old one
- Solution: Relate protocols to binder *instances* not binder classes

Rationalized Naming: Aims

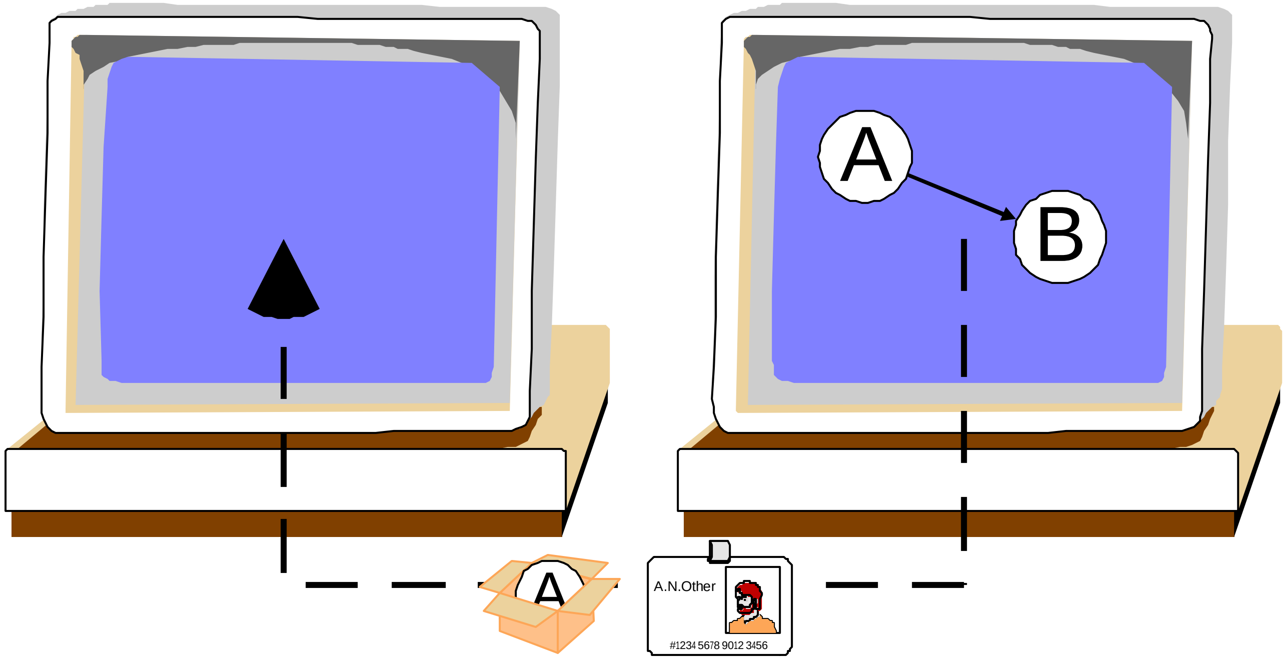
- support multiple binders per protocol
  - course grain QoS
- support multiple protocol per binder class
  - configure rather than re-implement
- allow runtime protocol definition
  - support for negotiation and generics
- allow runtime resolution of 'foreign' names
  - load the appropriate code and go
- support smart proxies?
  - Application specific binders

Back to basics...

- When are names generated?
- When are they used?
- Might we use them for anything else?

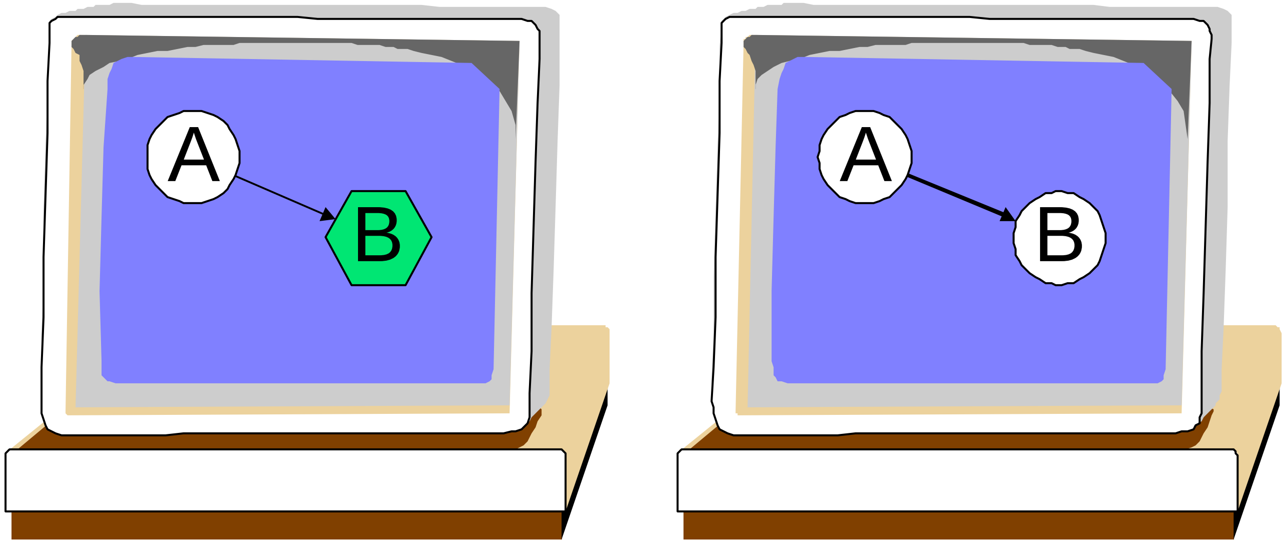
FlexiNet Naming

- Names are generated to be passed in place of objects



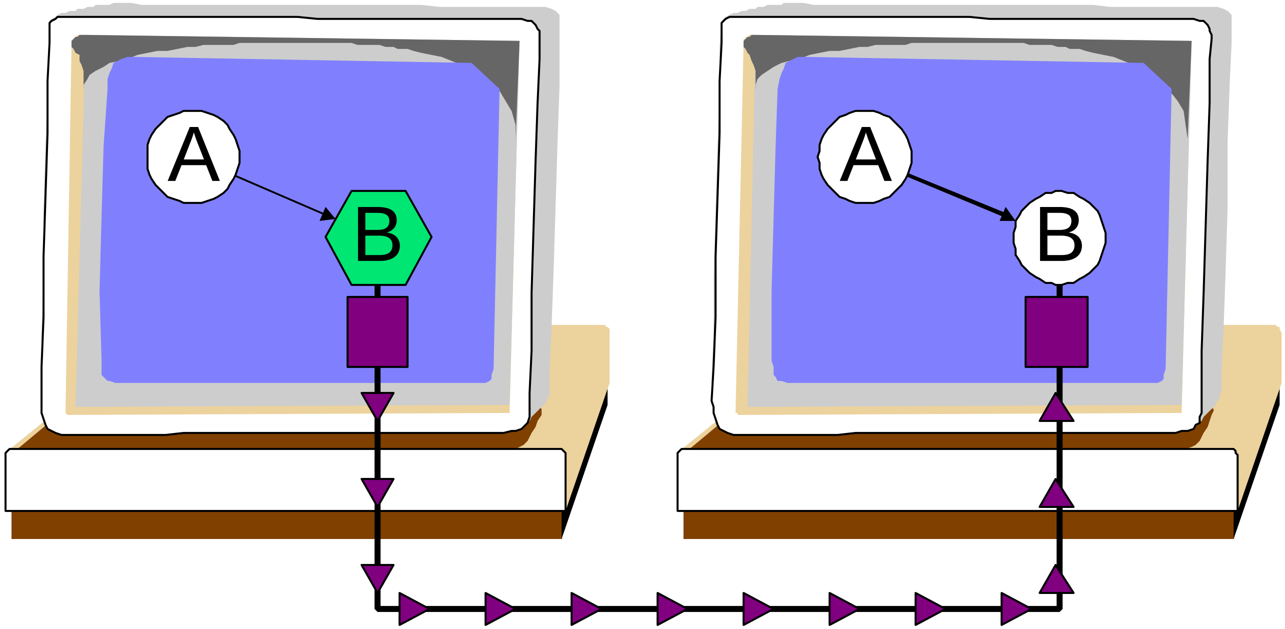
FlexiNet Naming

- On the client, a proxy is created to represent the original object
  - usually, the proxy is a ‘stub’ object



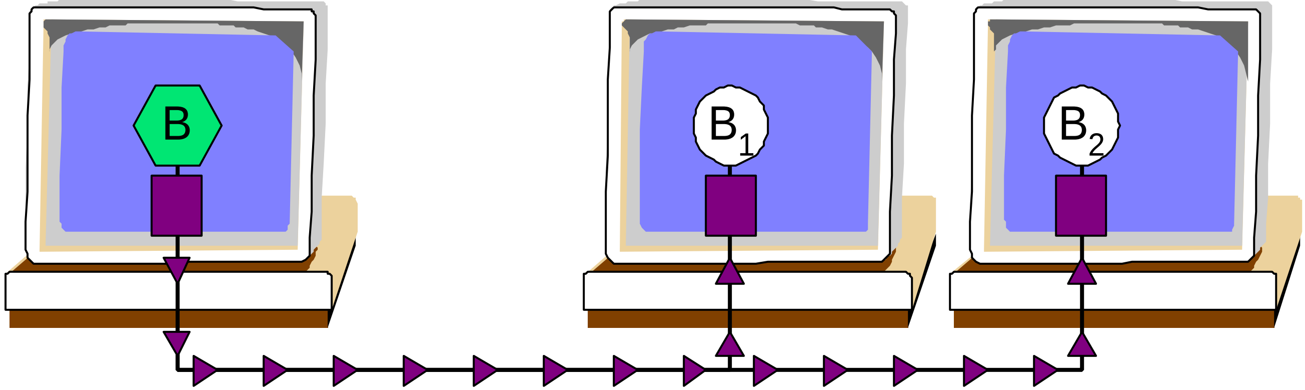
FlexiNet Naming

- The proxy acts ‘like’ the original object
  - e.g. by implementing remote method invocation

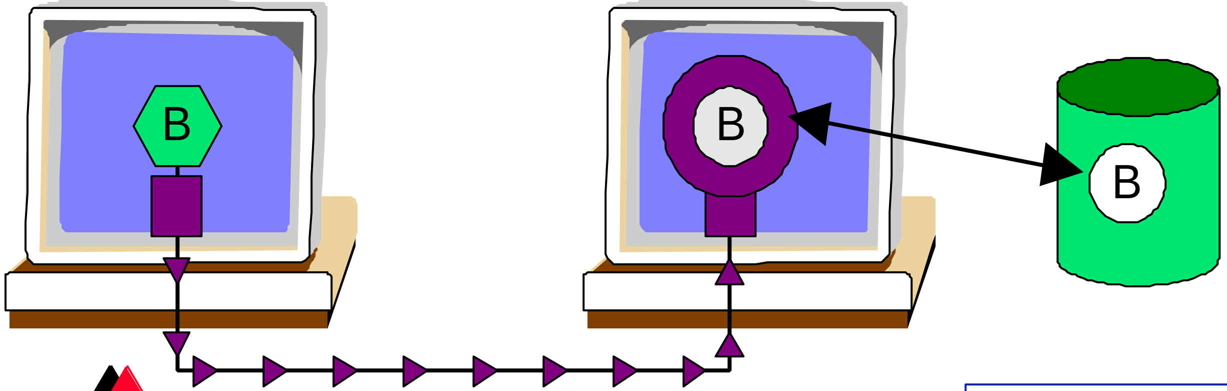


Other uses of Names

- Names for groups

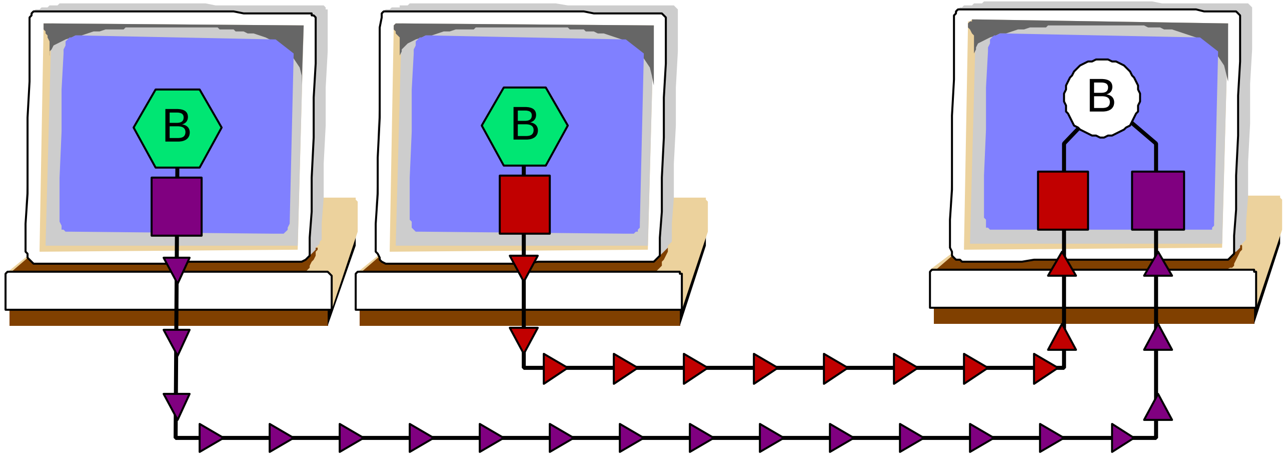


- Names for managed objects (e.g. Persistent)



Other uses of Names

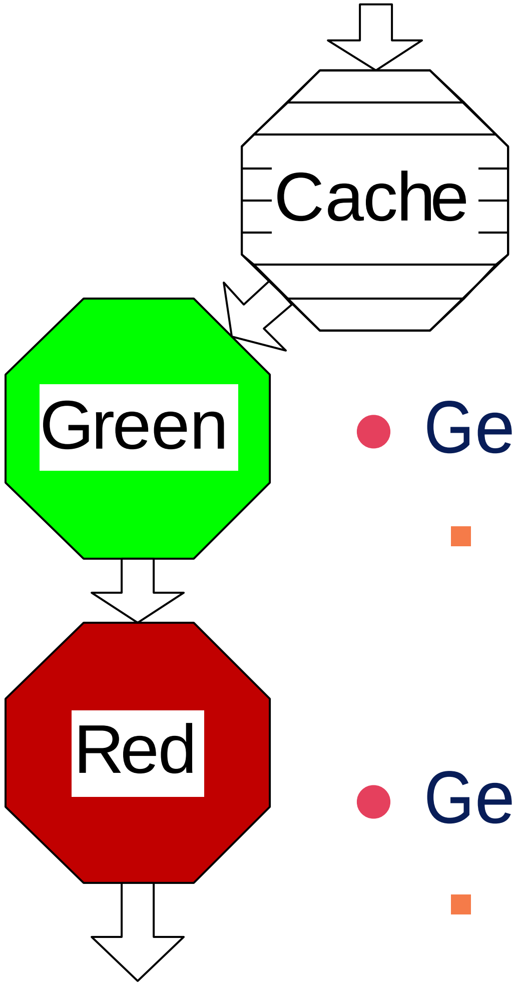
- Different names for the same object (different QoS)



Names as Objects

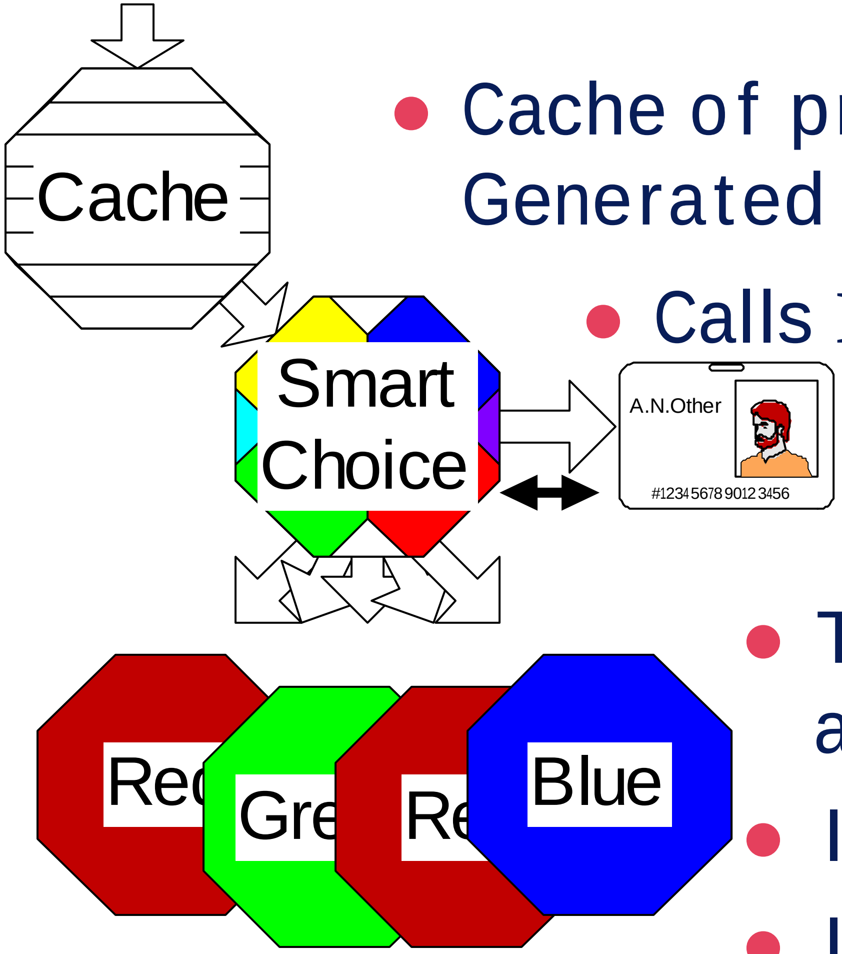
- FlexiNet names have always been constructed in an object oriented fashion
  - naming class hierarchy
    - TrivName,GreenName,MobileName all subclass of Name
  - Component class hierarchy
    - TrivName contains an Address
    - UDPEndpoint,TCPEndpoint are both subclasses of Address
  - Generic use of Names
    - binders, caches, explicit binding etc. all in terms of superclass
- Can names be *real* objects?
  - Add code as well as data
    - e.g. `name.resolve()`

Generating Names



- Cache of previously Generated Names
- Generates a name
  - if it can meet QoS requirements
- Generates a name
  - if it can meet QoS requirements

Resolving Names



- Cache of previously Generated Names
  - Calls `Name.Resolve(…)`
- The name may make use of an existing Resolver
- It may create a new Resolver
- It may resolve itself unaided

Benefits

- A Name may choose which resolver to use
  - not tied to one resolver per name class
- A name may store QoS requirements
  - allows names to be used for explicit binding
- A name may resolve itself
  - a 'smart' name
  - allows server code to execute on the client

Smart Proxies

- What are they?
  - A piece of code supplied by the server to run on the client and locally manage calls to the server
- What might they do?
  - Anything that is better done at the client side
    - Caching
    - Add client contextual information (ID, Thread etc)
    - Rebinding to one of a number of replica servers
    - Rebinding to a mobile server
  - Smart proxies are *application specific* resolvers
    - They can do exactly the same things as a resolver
    - But are easier to write

Using Smart Proxies

- What is required
  - A generator that generates names for Smart Proxies
  - A resolver that resolves names to Smart Proxies

- Approach
  - We arrange that a SmartProxy *is* also a Name
    - don't need to generate names - the application does it

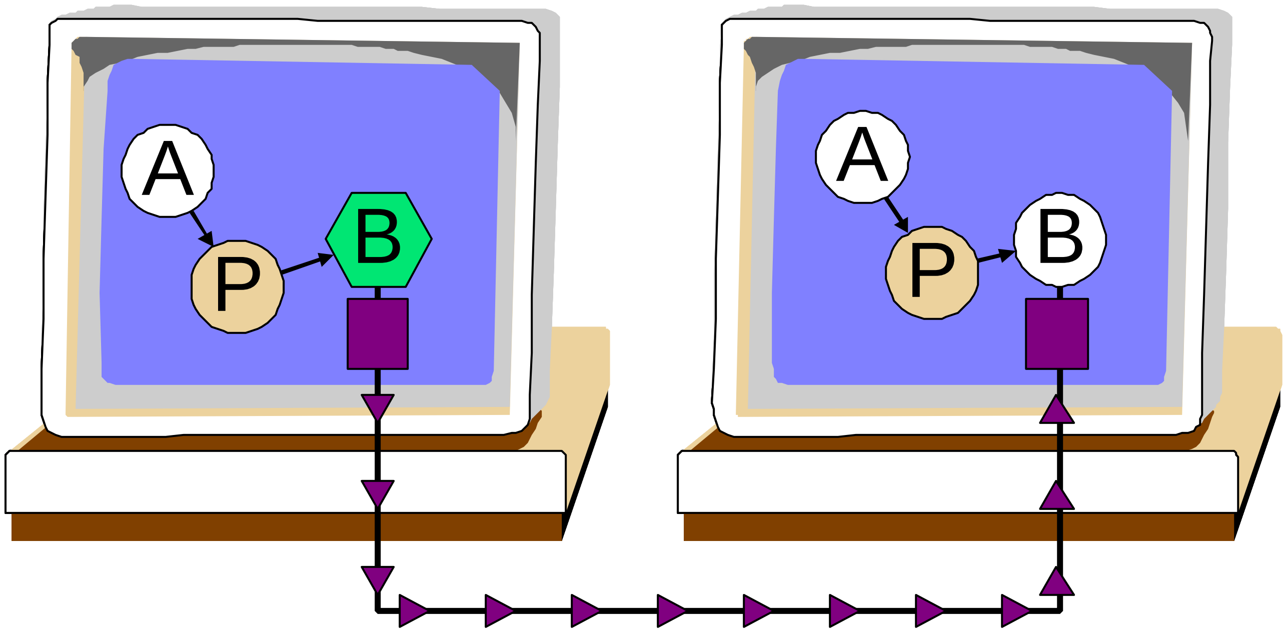
  - We implement Name.resolve() to return 'this'
    - don't need a resolver to resolve names

An example smart proxy

-
  ```
  Class Aproxy extends SmartProxy implements A
  {
     private A remoteA;

     public int add(int a,int b)
     {
       System.out.println(“Call add()”);
       return remoteA.add(a,b);
     }
  }
  ```

Using the Smart Proxies



- Isn't a smart proxy just pass by value?
  - Yes EXCEPT that like a normal proxy, there is only one proxy to a particular interface per-client
  - I.e. If P is passed to the client a second time, a reference to the first copy is passed instead

Generic Smart Proxies

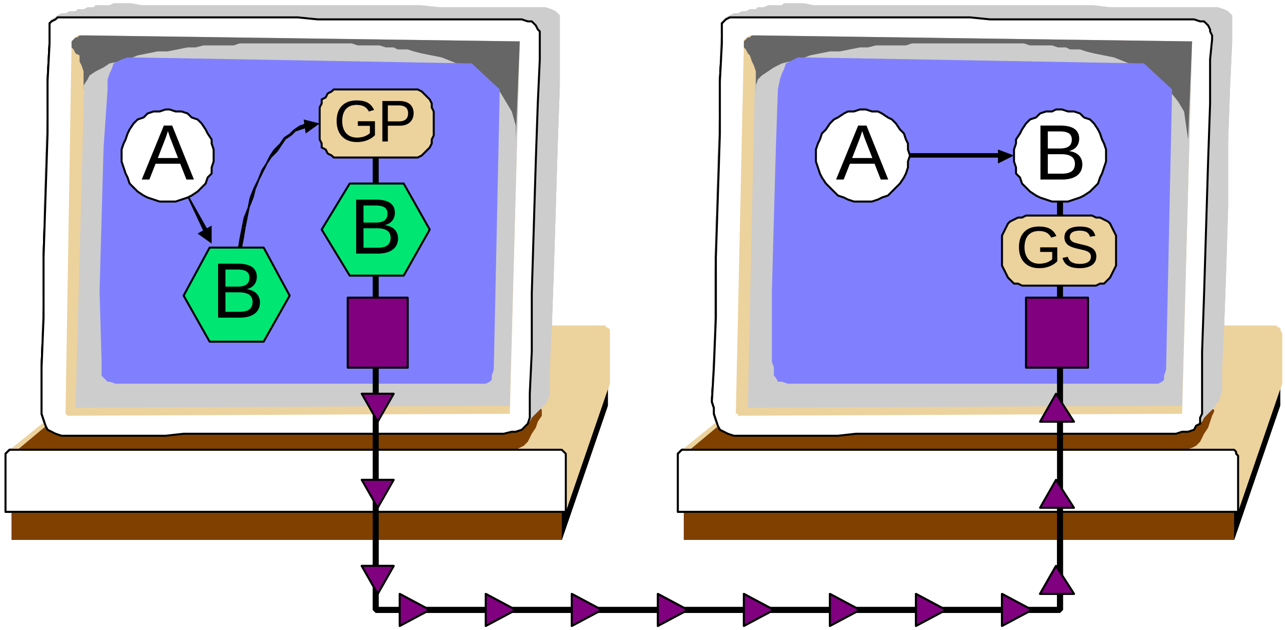
- Smart proxies are great for simple application reflection
  - but they are type specific
  - so you have to write one for each class proxied

- What about *generic* smart proxies?
  - I.e. a proxy that performs a type independent operations.
  - Useful for 'high level' reflection
    - transactions
    - auditing
    - replication
    - caching

Generic Smart Proxies

- What is required
  - A way of generating names for generic proxies
  - A name that creates a generic proxy on resolution
- Approach
  - We arrange that a GenericProxy *is* also a Name
    - we also need a generator to create them as required
  - We implement Name.resolve() to return stub+'this'
    - don't need a resolver to resolve names
  - We define an 'Invocation' class
    - and a generic call interface

Using Generic Proxies and Skeletons



- Generic Proxies work like an extra layer of a binder
  - The overhead is low
    - the second stub is effectively bypassed
  - The invocation object allows additional data to be passed

Example GenericProxy

```
class FooProxy extends SimpleGenericProxy
{
 FooProxy(Name n) { super(n);}
 FooProxy() {}

 void invoke(Invocation i)
 {
  i.push(“Using proxy”);
  super.invoke(i);
 }
}
```

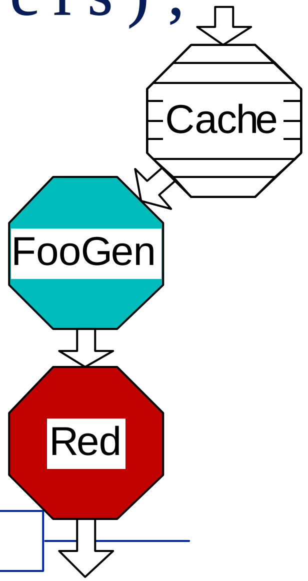
Example Generic Skeleton

```
class FooSkeleton implements GenericCall
{
  Object obj;
  FooSkeleton(Object o) {obj = o;}

 void invoke(Invocation i)
 {
  String msg = (String)i.pop();
  System.out.println(“msg ”+msg);
  i.invoke(obj);
 }
}
```

Example Generic Proxy Generator

```
class FooGen extends GenericProxyGenerator
{
 Name generateName(Object,obj,Class cls)
 {
   GenericCall skeleton = new FooSkeleton(obj);
   Name name = generateBaseName(skeleton,cls);
   return new FooProxy(name);
  }
}
```



Summary

- "Names as objects" is a flexible abstraction
  - Ease management of a large number of protocols
  - Allow smart proxies
- Smart Proxies are easy to use
  - at least, a lot easier that writing binders
  - provide a 'friendlier' API for reflection
- Explicit Binding fits well
  - Available since MOW v0.1
  - More useful when combined with Smart Proxies