Problem: Fat and Expensive Components

- Consider a banking system

- Bank clerks from different bank branches use web-based clients to interact with the bank server via (amongst other components) an Information component – this permits clerks to make a wide range of different queries to the server

- The component doesn’t contain any data itself – but provides links to a wide range of separate databases used by different parts of the Bank (mortgages, personal loans, personal accounts each have separate databases of different legacies)

- These links are very expensive to establish (it takes time to make a connection to a database, particularly the legacy ones)

- Problem: If there are 1000 bank clerks running these clients, we will need to create 1000 Information objects – this is a BIG waste, particularly as the number of clients actually using the Information object at any time is about 4

Object Pooling

- The object pooling service enables a set of component instances to be kept active in a pool that is shared by clients of the component.

- When a client instantiates a client component, the reference is actually linked to the pool – a new instance is not created.

- If a client calls an object of pooled class, the call is delegated to a member of the pool – if all the members of the pool are currently serving calls, then the call is queued until an object becomes available.

- You can administratively configure and monitor the pool maintained for a given component, specifying characteristics such as pool size and creation request time-out values.

- When the application is running, COM+ manages the pool for you, handling the details of object activation and reuse according to the criteria you have specified.
Object Pooling: Advantages

- You can achieve very significant performance and scaling benefits by reusing objects in this manner, particularly when they are written to take full advantage of reuse. With object pooling, you gain the following benefits:
  - You can speed object use time for each client, factoring out time-consuming initialization and resource acquisition from the actual work that the object performs for clients.
  - You can share the cost of acquiring expensive resources across all clients.
  - You can pre-allocate objects when the application starts, before any client requests come in.
  - You can govern resource use with administrative pool management—for example, by setting an appropriate maximum pool level, you can keep open only as many database connections as you have a license for.
  - You can administratively configure pooling to take best advantage of available hardware resources—you can easily adjust the pool configuration as available hardware resources change.

Object Pooling: Constraints

- Pooled objects must meet certain requirements to enable a single object instance to be used by multiple clients.
  - For example, they can't hold client state or have any thread affinity.
  - Pooled objects are COM+ components—so can be transactional. This can make things complicated, but COM+ handles it for us.

Object Pooling Code

```csharp
[ObjectPooling(Enabled=true, MinPoolSize=2, MaxPoolSize=5, CreationTimeOut=20000)]
public class TestObjectPooling : ServicedComponent
{
// methods
// go here
public override void Activate()
{
  // Called when removed from the pool.
}
public override void Deactivate()
{
  // Called before deactivating or placing back in pool.
}
public override bool CanBePooled()
{
  // Called after Deactivate. Indicate your vote here.
  return true;
}
}
```

Says TestObjectPooling is a COM+ component

UML2

- The situation of multiple clients sharing a single server can be represented in standard UML using multiplicities
- However, that representation doesn’t convey the platform specific detail with the precision we require
  - E.g., how do we express a pool of size 5 using multiplicities? If we made Information’s multiplicity 5, this could be interpreted to mean that each client has a reference to 5 Information components—not what we mean at all
  - So—we use tags again—using a straightforward mapping of the .NET attribute types to tag names

```
<<COM++>>
ClerkClient

<<COM++>>
Information
(ObjectPoolingEnabled=true,
 ObjectPoolingMinPoolSize=2,
 ObjectPoolingMaxPoolSize=5)
```

1 or 5 or what?!
Problem: Skinny and Idle Components

- Consider the Banking example but with the opposite kind of Information Component

- Imagine that the Information Component interacts with only one database
  - It is very cheap to instantiate (there are no legacy databases, only a very fast one)
  - It is cheap to destroy
  - It is expensive to have running constantly, as it takes up a lot of memory

- Performance would be improved if all the clients were to maintain references to Information components, but for an Information reference to be active only when it is needed, and deactivated the moment it is not needed.

Just in Time Activation

- The COM+ Just-in-Time (JIT) Activation service allows idle server resources to be used more productively.

- When a component is configured as JIT activated, COM+ can deactivate an instance of it while a client still holds an active reference to the object.

- The next time the client calls a method on the object, COM+ reactivates the object transparently to the client, just in time.

.NET and UML2

- In .NET

  [JustInTimeActivation(True)]
  class myClass : ServicedComponent { ... }

- In UML 2 Specialization

  <<COM>>
  ClerksClient

  <<COM>>
  Information
  (JustInTimeActivation = true)

.NET Enterprise Services

- Serviced Components
- Transactions
- Object Pooling
- Just In Time (JIT) Activation – Object Lifetimes
- Loosely Coupled Events
- Queued Components