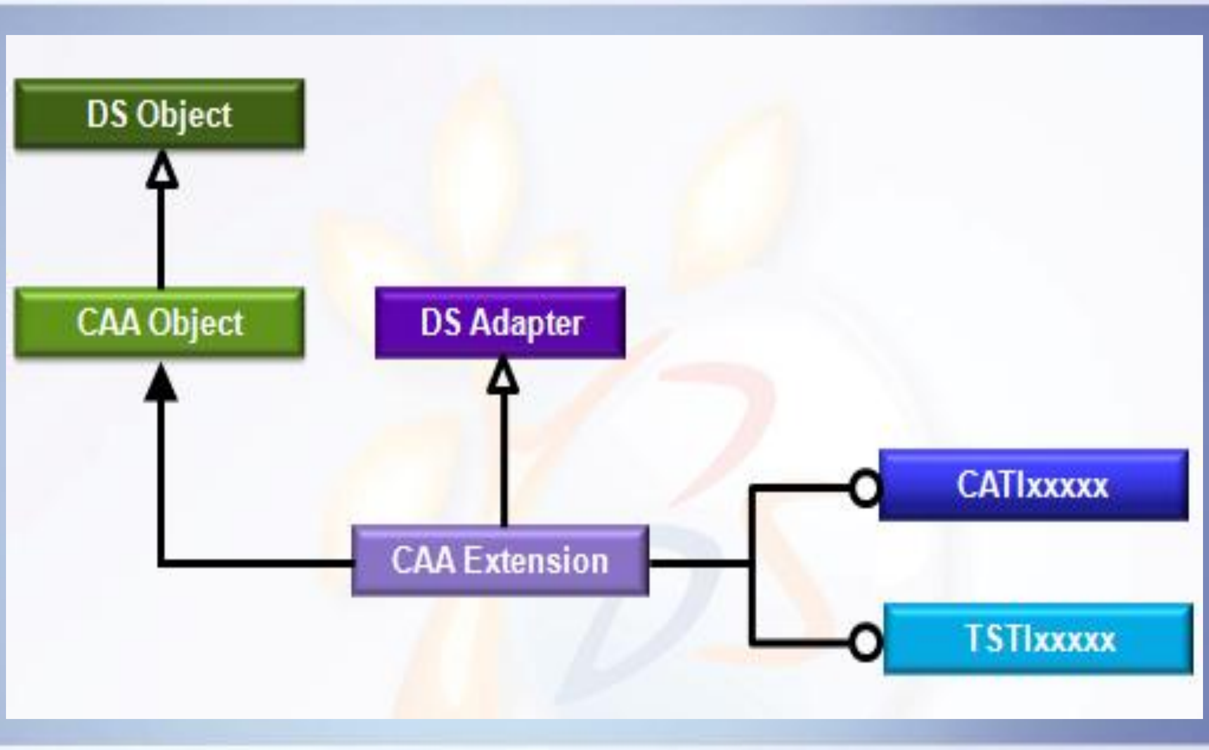
Object Modeler



V6
R2010x

Table Of Contents

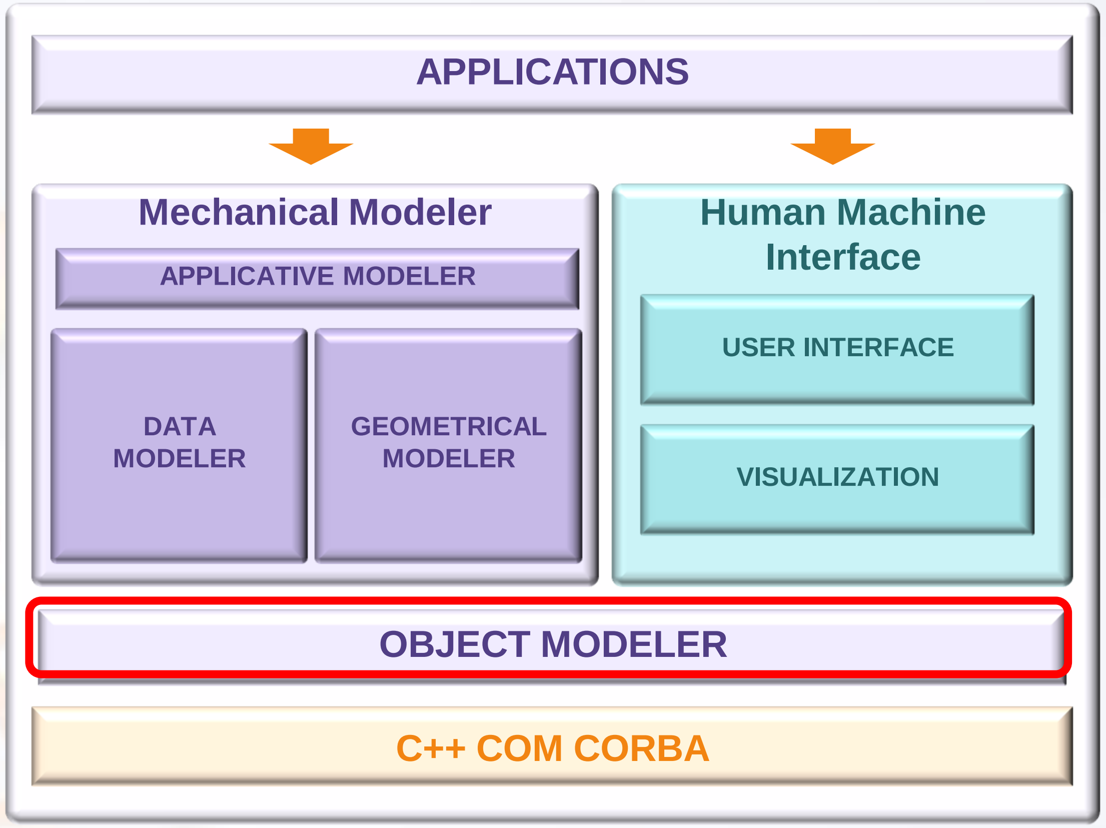
Overview (1/2)

- **The Object Modeler defines an Interface Mechanism to handle objects**

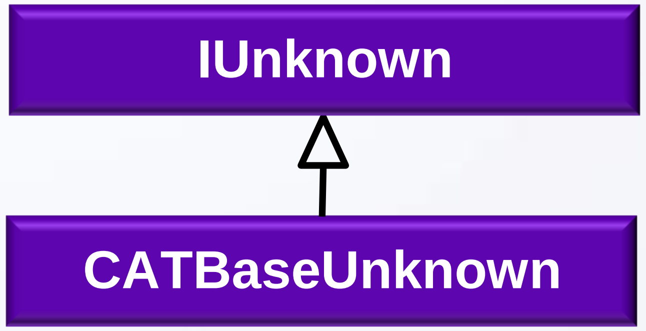
- **The Object Modeler enables:**
  - a build independence:
    - applications are built only on interfaces
    - application rebuild is not necessary for implementation modification
  - on demand dll loading
    - for example toolbar dlls are loaded when corresponding workbench is launched
  - an object behavior federation
    - it is possible to add the same behavior to different types of objects
  - an open architecture
    - A customer may:
      - → create it own object inheritance architecture
      - → implement Dassault Systèmes interfaces on its own objects
      - → implement its own interfaces on Dassault Systèmes objects and on its own objects

Overview (2/2)

- **The Object Modeler is a Dassault Systèmes modeler relying on:**
  - C++
    - Object Inheritance
    - Polymorphism
    - Introspection of objects type at runtime
      - → RTTI : **R**un-**T**ime **T**ype **I**nformation
  - COM
    - All DS objects inherit at least from the *IUnknown* COM interface enabling to:
      - → get a object behavior from another one (*QueryInterface()* method)
      - → get interface query return code (*HRESULT*)
      - → handle lifecycle management (*AddRef() & Release()* methods)



- **Object Modeler has:**
  - deactivated RTTI
  - defined *CATBaseUnknown* as the mandatory base class inheriting from *IUnknown* interface
    - → get the class name (*ClassName()*)
    - → know if the object inherit from another (*IsAKindOf()*)

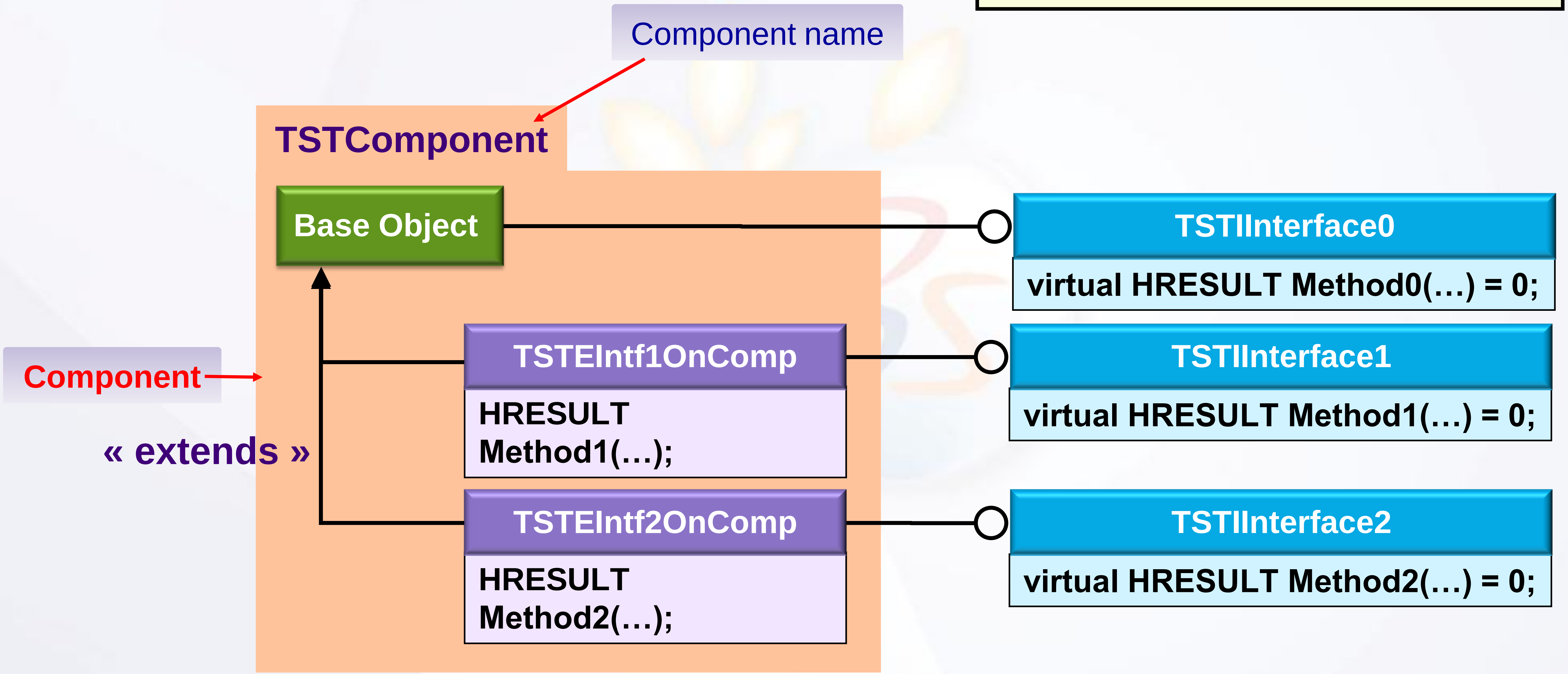


Component Definition

- A Component is composed of :
  - A Base Object
  - All its extensions

Dictionary

TSTComponent	TSTIInterface0	libTSTSharedLibrary0
TSTComponent	TSTIInterface1	libTSTSharedLibrary1
TSTComponent	TSTIInterface2	libTSTSharedLibrary2
…		



Interface Definition

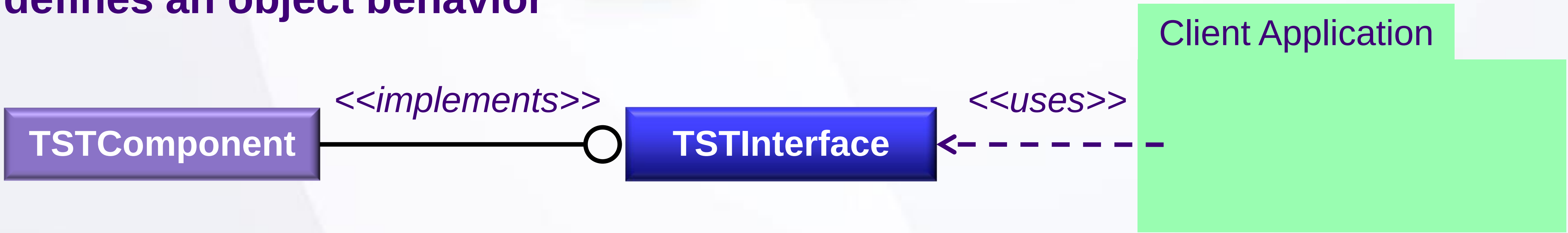
- **Interfaces are contracts between clients and implementations**
  - Interfaces should never change in order not to rebuild applications for each implementation change

- **The client application deals with components only through interfaces**

- **Interfaces shield the application code from the component implementation details**

- **A component can implement one or more interfaces**
  - these interfaces are the component external view

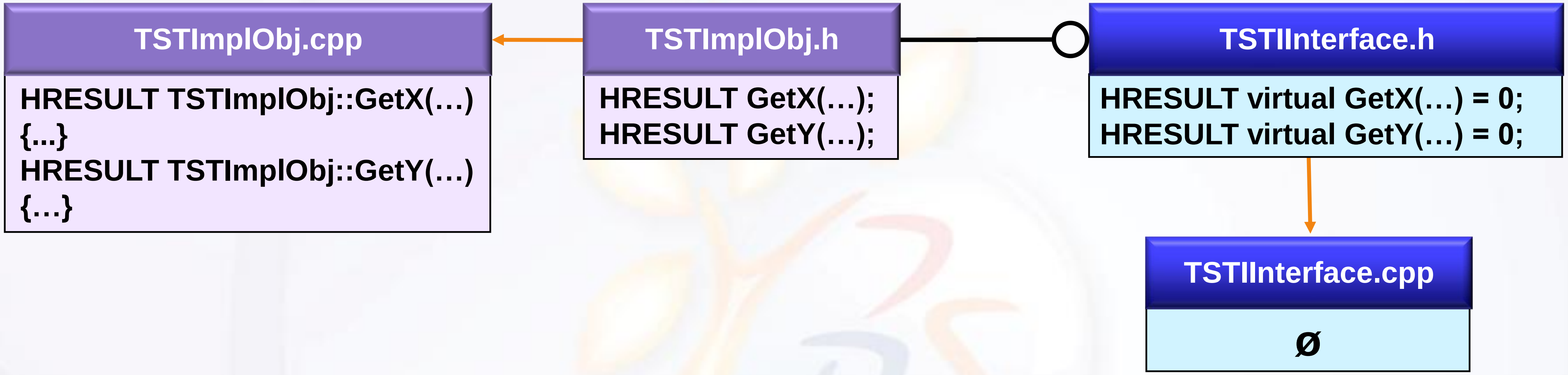
- **An interface is an abstract class with a set of pure virtual methods that defines an object behavior**



Convention: the 4th character of an interface name is an « I »

Base Object Definition

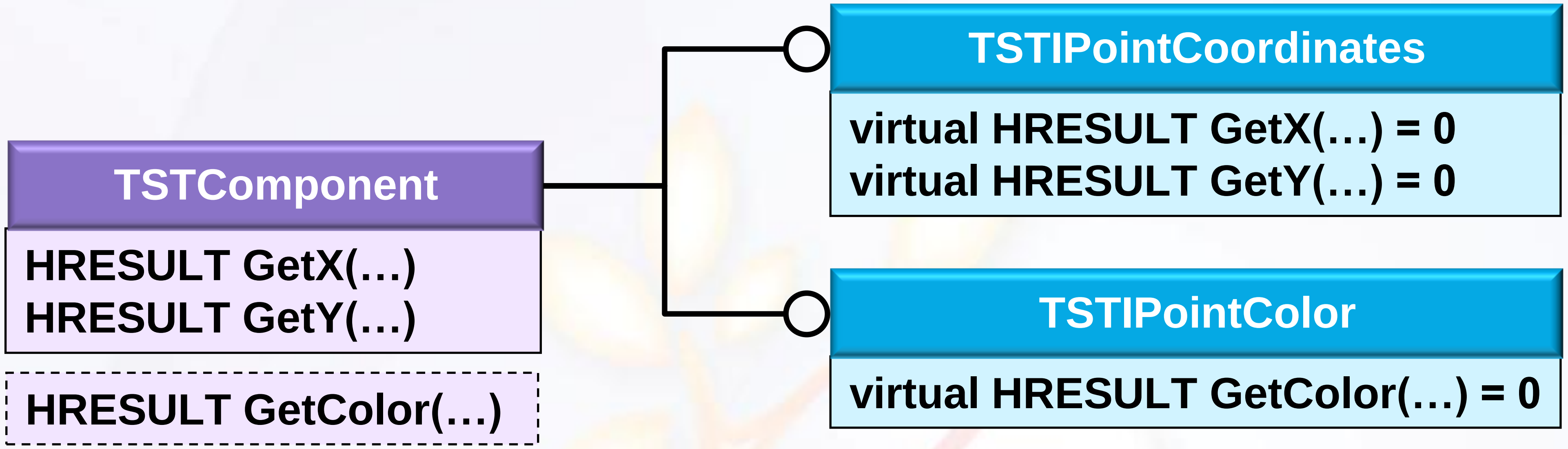
- A **Base Object** is:
  - an **Implementation Object** is a C++ class which can implement several interfaces


TSTImplObj.cpp
HRESULT TSTImplObj::GetX(…)
{...}
HRESULT TSTImplObj::GetY(…)
{…}
TSTImplObj.h
HRESULT GetX(…);
HRESULT GetY(…);
TSTIInterface.h
HRESULT virtual GetX(…) = 0;
HRESULT virtual GetY(…) = 0;
TSTIInterface.cpp
ø

  - a **Late Type** is a concept represented by a character string to which some behaviors can be added through the extension mechanism

Interface/Implementation

- **How to add new behaviors (methods) on a component without having access to its implementation?**



- **How to separate the implementation of interfaces on the same component in order to benefit from on demand dll loading?**

- **How to split semantically the implementation code?**
  - Coordinates does not mean the same thing than color

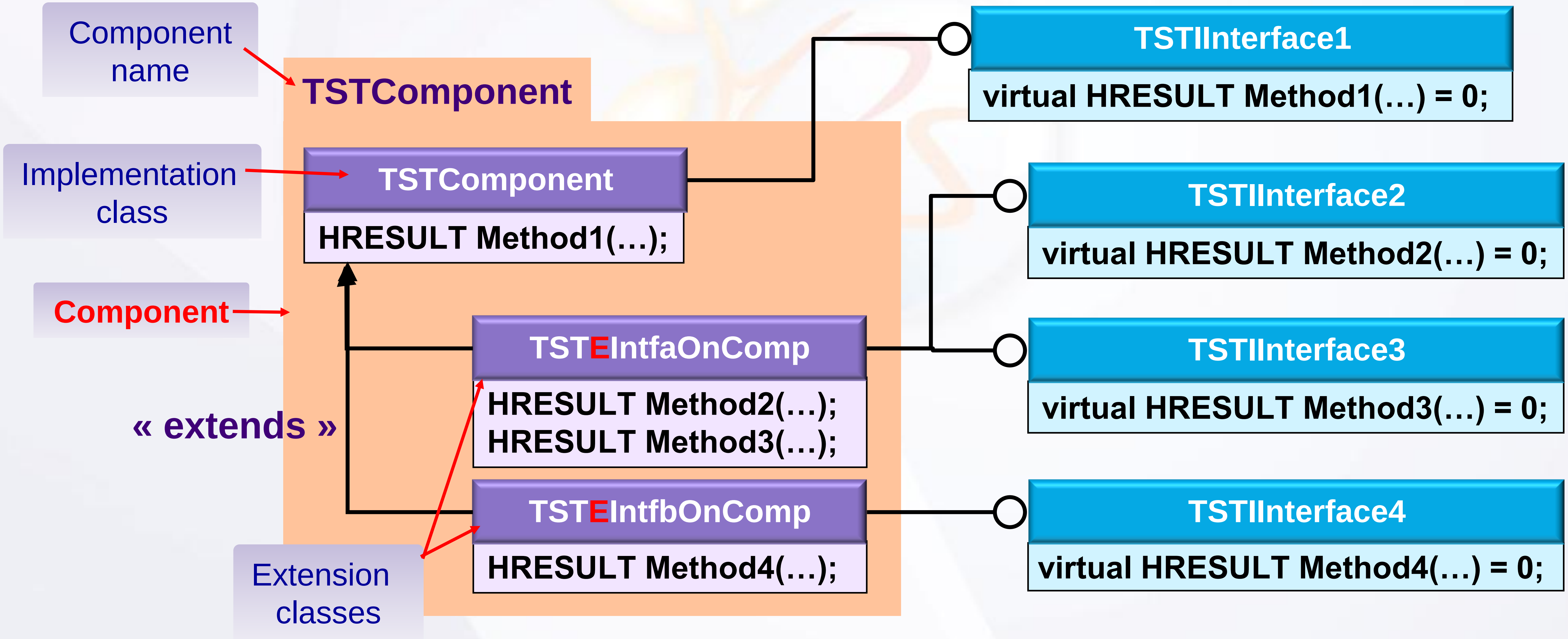
Component Extension Definition

- **A component extension is a C++ class that adds new capabilities through new interfaces to an existing component**
- **Hence the component definition is extended by:**
  - the implementation object
  - all its extensions

Convention: the 4th character of the extension name is an « E »
Wizard: the component name is the implementation class name

TSTFramework.dico

```
TSTComponent   TSTIInterface1   libTSTSharedLibrary1
TSTComponent   TSTIInterface2   libTSTSharedLibrary2
TSTComponent   TSTIInterface3   libTSTSharedLibrary2
TSTComponent   TSTIInterface4   libTSTSharedLibrary3
```



Extension Types

- **There two extension types:**
  - DataExtension
    - **Extension class can contains methods and data member**
    - **One single extension instance for each component instance**
    - **Data extensions are deleted when the component is deleted**

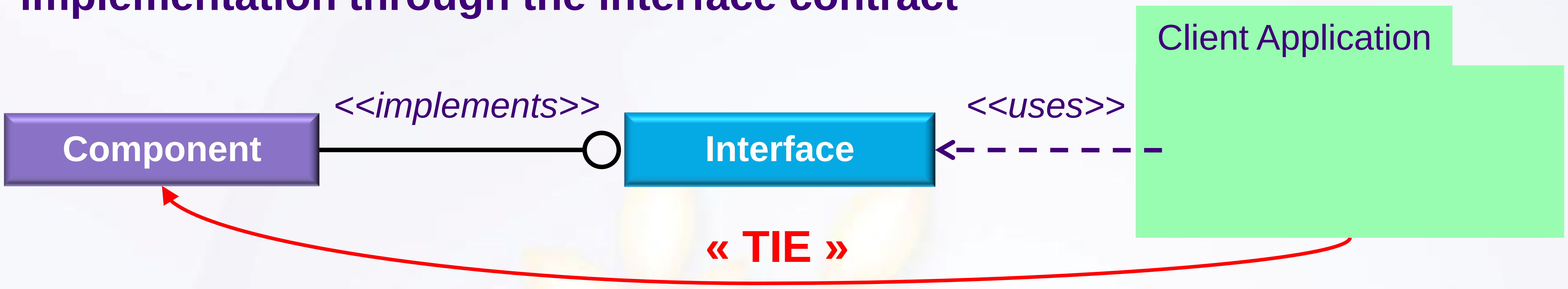
  - CodeExtension
    - **Extension class must not contain any data member**
    - **One single extension instance for all the component instances**
    - **To be carefully handled (“this” must not be used either implicitly or explicitly in the code extension class)**
    - **Code extensions are deleted at the end of the CATIA session**

- **If you don’t know which kind of type extension is the best for your scenario choose the DataExtension one**

```
CATImplementClass (<this ClassName>,
                   <its Extension Type>,
                   <its Inheritance>,
                   <what it extends>)
```

TIE Definition

- **A TIE is a C++ class allowing the client application to use the component implementation through the interface contract**



- **TIE header file will be generated if you create a file *TIE_TSTIxxx.tsrc* that just includes the interface header file *TSTIxxx.h***
  - generated by the Visual Studio wizard during the interface creation

TSTInterfaceFw/TSTModule.m/TSTIInterface.tsrc

```
// Code Generated by the CAA Wizard
// This source file insures the regeneration of the tie TIE_TSTIInterface.h
#include "TSTIInterface.h"
```

mkmk

TSTInterfaceFw/ProtectedGenerated/intel_a/TIE_TSTIInterface.h

```
…
```

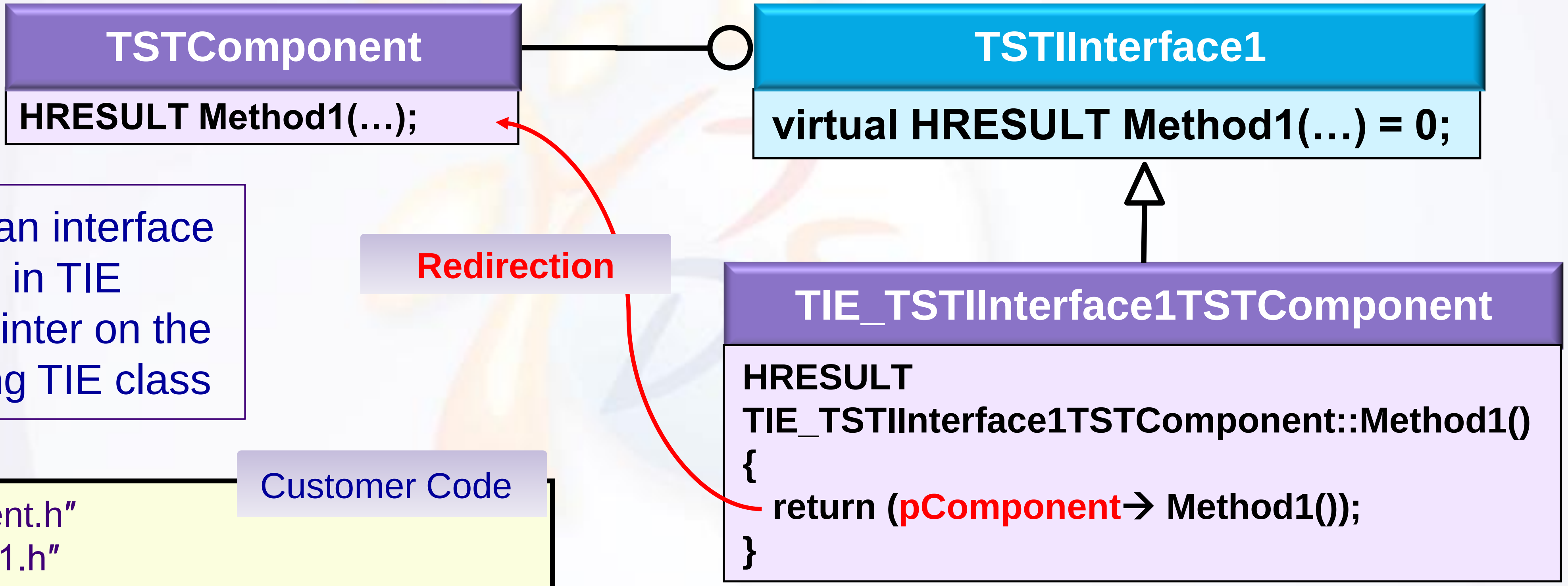
TIE Mechanism (1/2)

- **Every request to an interface method is redirected to the corresponding component method implementation through a TIE class instance**

TSTFramework.dico

TSTComponent TSTIInterface1 libTSTSharedLibrary1
…

- ***QueryInterface()* from component to interface**



TSTComponent
HRESULT Method1(…);
TSTIInterface1
virtual HRESULT Method1(…) = 0;
Redirection
TIE_TSTIInterface1TSTComponent

```
HRESULT
TIE_TSTIInterface1TSTComponent::Method1()
{
  return (pComponent→ Method1());
}
```

A pointer on an interface implemented in TIE mode is a pointer on the corresponding TIE class

Customer Code

```
# include "TSTComponent.h"
# include "TSTIInterface1.h"
...
TSTComponent* pComponent = new TSTComponent();
TSTIInterface1* pInterface1OnComponent = NULL;
pComponent →QueryInterface (IID_TSTIInterface1,
                         (void**)& pInterface1OnComponent);
pInterface1OnComponent → Method1(...);
```

Internal DS Code

```
pInterface1OnComponent =
      new TIE_TSTInterface1TSTComponent(pComponent);
```

TIE Mechanism (2/2)

- ***QueryInterface()* from interface to another one:**

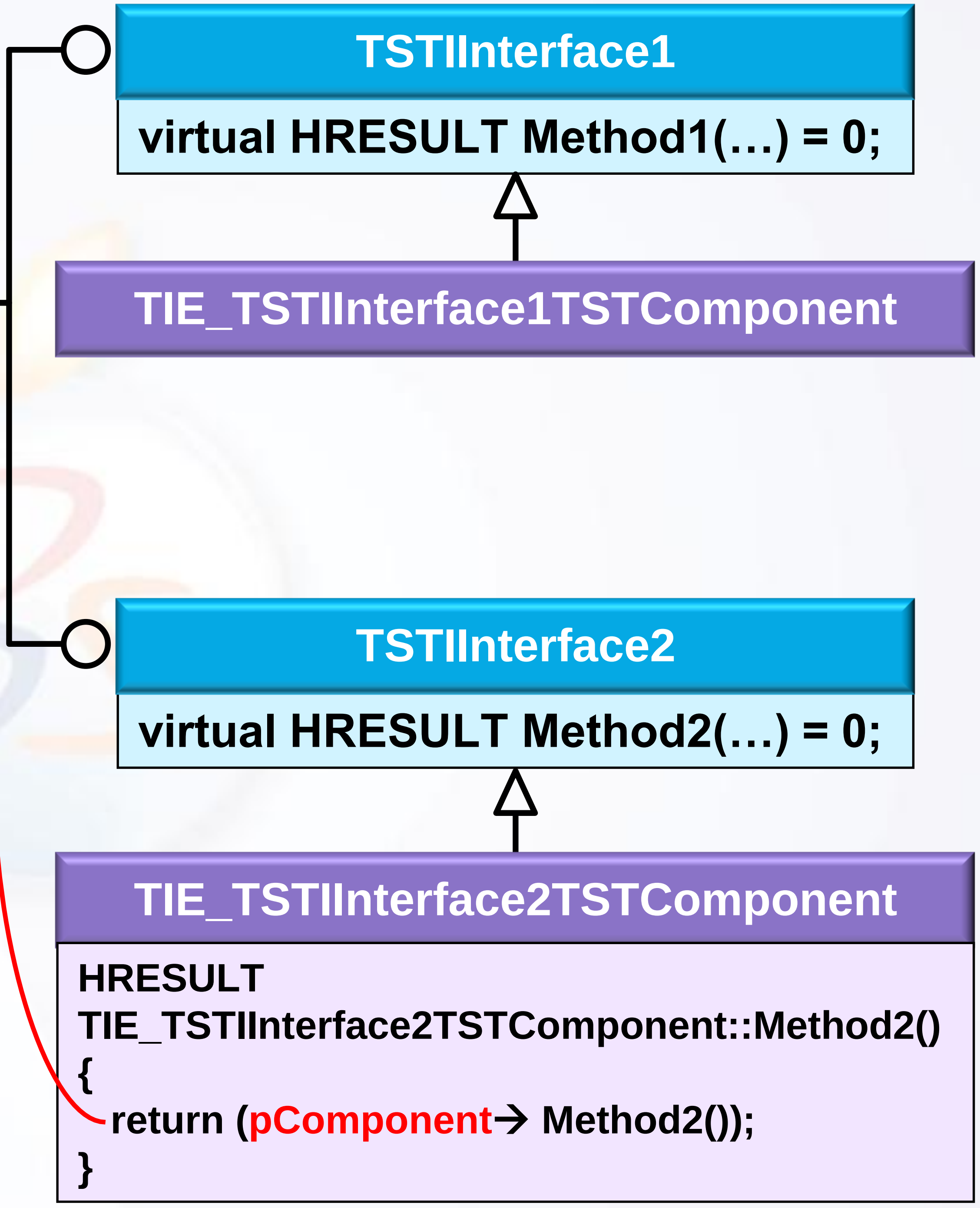
TSTFramework.dico

```
TSTComponent TSTIInterface1 libTSTSharedLibrary1
TSTComponent TSTIInterface2 libTSTSharedLibrary2
…
```

TSTComponent

```
HRESULT Method1(…);
HRESULT Method2(…);
```

Redirection



TSTIInterface1

```
virtual HRESULT Method1(…) = 0;
```

TIE_TSTIInterface1TSTComponent

TSTIInterface2

```
virtual HRESULT Method2(…) = 0;
```

TIE_TSTIInterface2TSTComponent

```
HRESULT
TIE_TSTIInterface2TSTComponent::Method2()
{
 return (pComponent→ Method2());
}
```

Customer Code

```
# include ″TSTComponent.h″
# include ″TSTIInterface1.h″
# include ″TSTIInterface2.h″
...
TSTIInterface2* pInterface2OnComponent = NULL;
pInterface1OnComponent →QueryInterface (IID_TSTIInterface2,
                          (void**)& pInterface2OnComponent );
pInterface2OnComponent → Method2(...);
```

Internal DS Code

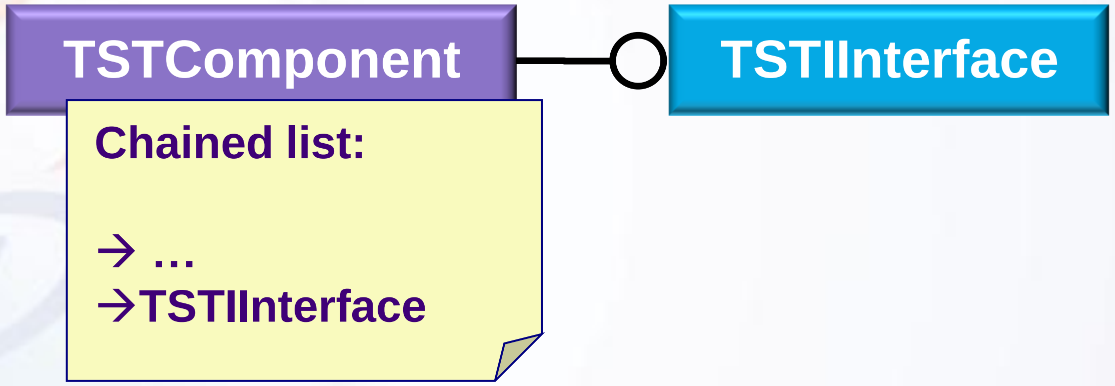
```
pInterface1OnComponent → GetComponent (pComponent);
pInterface2OnComponent =
      new TIE_TSTInterface2TSTComponent(pComponent);
```

Standard TIE And Chained TIE Introduction

- **There are two kinds of TIE**
  - Standard TIE
    - always create an instance of the TIE class when *QueryInterface()* is used
  - Chained TIE
    - **only one TIE instance** created when several *QueryInterface()* are done on the **same component instance** to get several pointers of the **same interface**
      - → Each Component has an interface chained list filled with a new interface at each *QueryInterface()* only if it is not already in the list

Customer Code

```
TSTComponent* pComponent = …;
…
TSTIInterface* pi1InterfaceOnComponent = NULL;
pComponent →QueryInterface (IID_CATIInterface,
                       (void**)& pi1InterfaceOnComponent);
…
TSTIInterface* pi2InterfaceOnComponent = NULL;
pComponent →QueryInterface (IID_CATIInterface,
                        (void**)& pi2InterfaceOnComponent);
```



TSTComponent

TSTIInterface

Chained list:

→ …

→TSTIInterface

Internal DS Code
Standard TIE

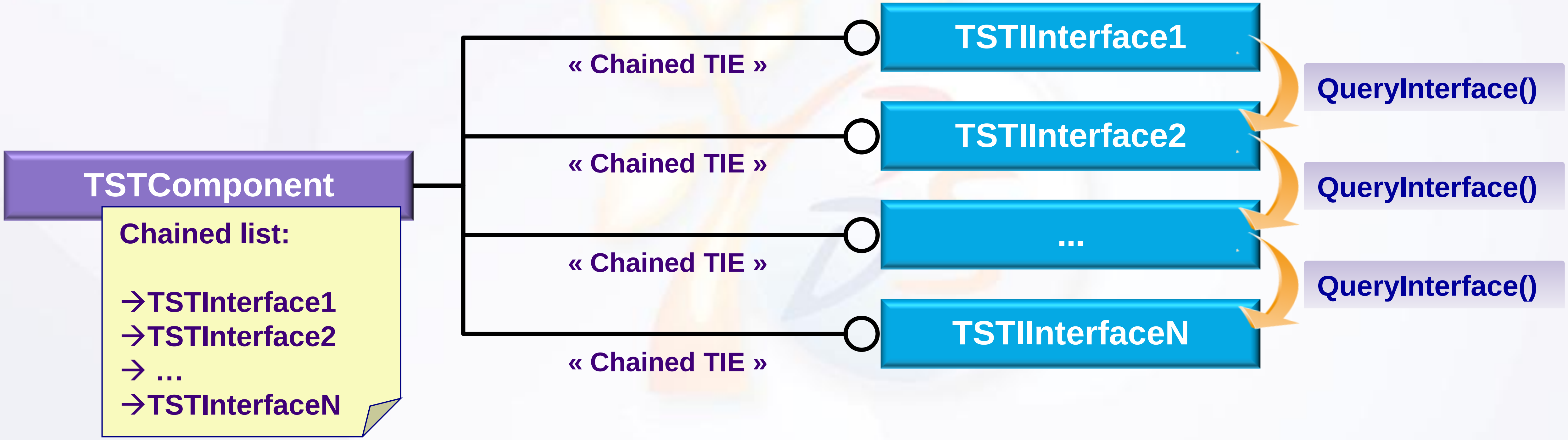
```
TIE_ TSTIInterfaceTSTComponent*
       pi1InterfaceOnComponent =
       new TIE_ TSTIInterfaceTSTComponent();
…
TIE_ TSTIInterfaceTSTComponent*
       pi2InterfaceOnComponent =
       new TIE_ TSTIInterfaceTSTComponent();
```

Internal DS Code
Chained TIE

```
TSTInterface* pChainedInterface =
        pComponent → ChainedList (IID_TSTInterface)
TIEchain_ TSTIInterfaceTSTComponent*
       pi1InterfaceOnComponent =
       new TIEchain_ TSTIInterfaceTSTComponent();
…
TSTInterface* pChainedInterface =
        pComponent → ChainedList (IID_TSTInterface)
pi2InterfaceOnComponent = pChainedInterface;
   pi2InterfaceOnComponent → AddRef();
```

TIE Recommendation

- **If you don’t know which kind of TIE is the best on your scenario choose the standard one**
- **Effectively in some cases chained TIE may lead to:**
  - CPU use increasing
    - for instance if a component has several interfaces implemented with a Chained TIE the chained list could be huge if we use a lot of behaviors

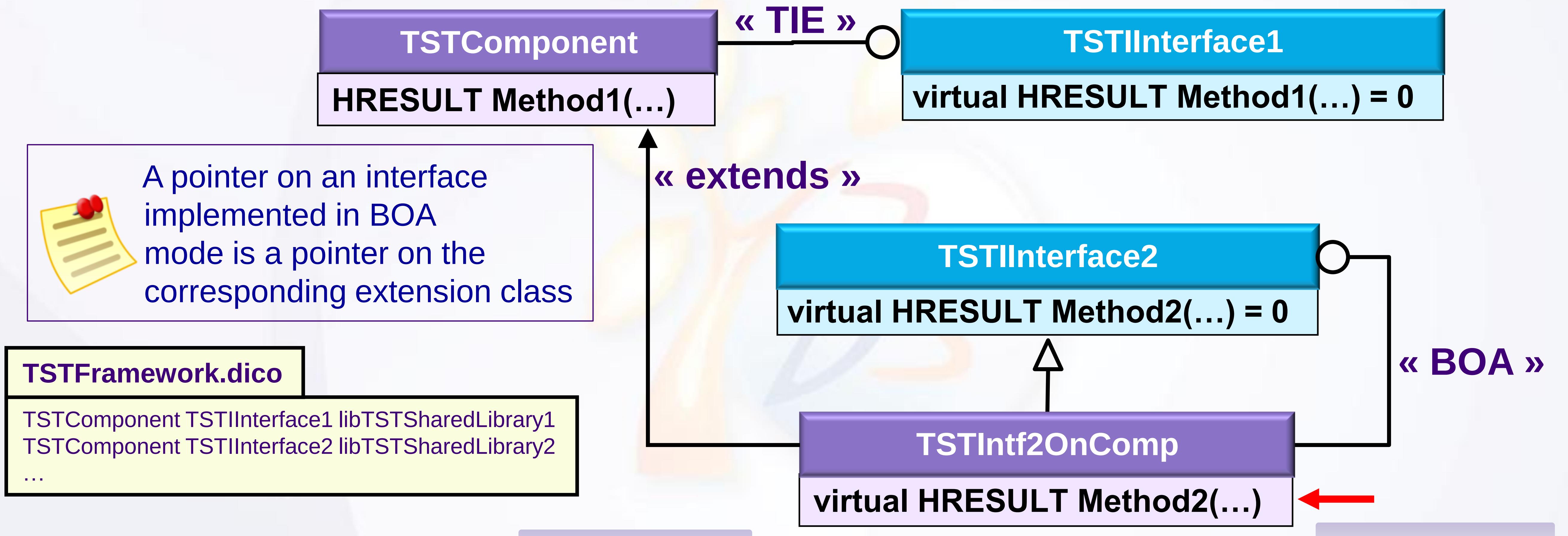


  - memory consumption increasing
    - for instance if there are a lot of component instances and if we use several of their behaviors the memory will increase strongly

      → TIE objects are not deleted until the component is deleted
  - tricky debugging to manage the interface pointer lifecycle (see lifecycle chapter)

BOA (Basic Object Adapter)

- **Component extension can be implemented in TIE mode but also in BOA**
- **In BOA mode the component extension class inherits from the interface**
  - there is no intermediate object unlike TIE



A pointer on an interface implemented in BOA mode is a pointer on the corresponding extension class

TSTFramework.dico

```
TSTComponent TSTIInterface1 libTSTSharedLibrary1
TSTComponent TSTIInterface2 libTSTSharedLibrary2
…
```

Customer Code

```
TSTInterface1* piInterface1OnComponent = …;
TSTIInterface2* piInterface2OnComponent = NULL;
piInterface1OnComponent →QueryInterface
   (IID_TSTIInterface2, (void**)& piInterface2OnComponent);
piInterface2OnComponent → Method2(...);
```

Internal DS Code

```
piInterface1OnComponent → GetComponent (pComponent);
piInterface2OnComponent =
       pComponent → GetExtension (“TSTEIntf2OnComp”);
If (NULL == piInterface2OnComponent )
       piInterface2OnComponent =  new TSTEIntf2OnComp ();
```

BOA Recommendations

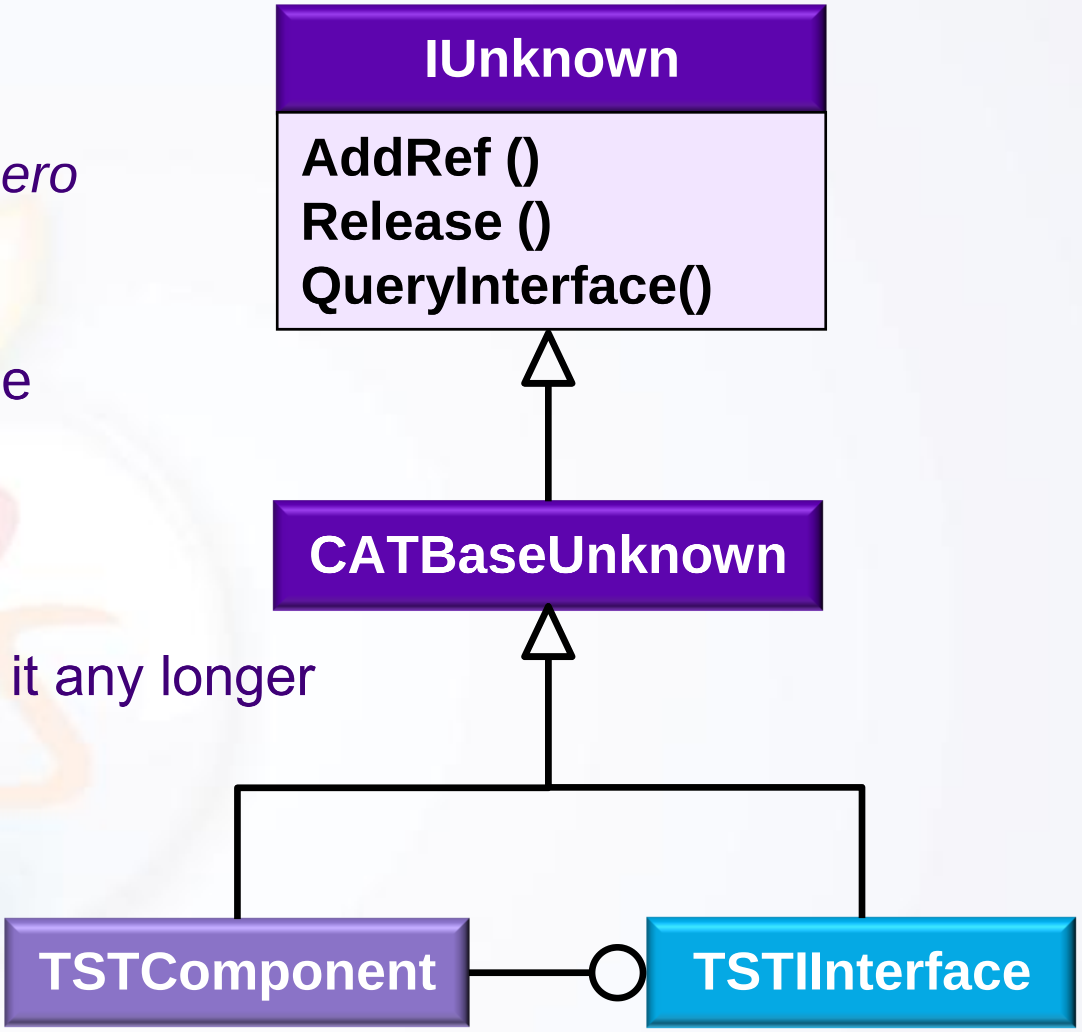
- **Advantages:**
  - Don’t create TIE object (save memory)
  - Direct access to component (better CPU performances)

- **Restrictions:**
  - A component implement / extension class can implement only one interface in BOA mode
    - multiple inheritance is forbidden in CAA
  - If the interface has an adapter class you can only implement it in BOA mode if the adapter inherits from the interface
  - Code extension and BOA are not compatible
  - Implementing some DS interfaces with BOA may not be authorized
    - Refer to the CAA documentation (ex : *CATIModelEvents*)

- **Recommendations:**
  - In term of architecture it is better to implement interfaces in TIE mode for component implementation classes
    - Otherwise the interface pointer will be a direct cast of the component
  - If you don’t know which mode is the best for your scenario
    - Use BOA with extensions and implement a single interface per extension
    - Use TIE otherwise

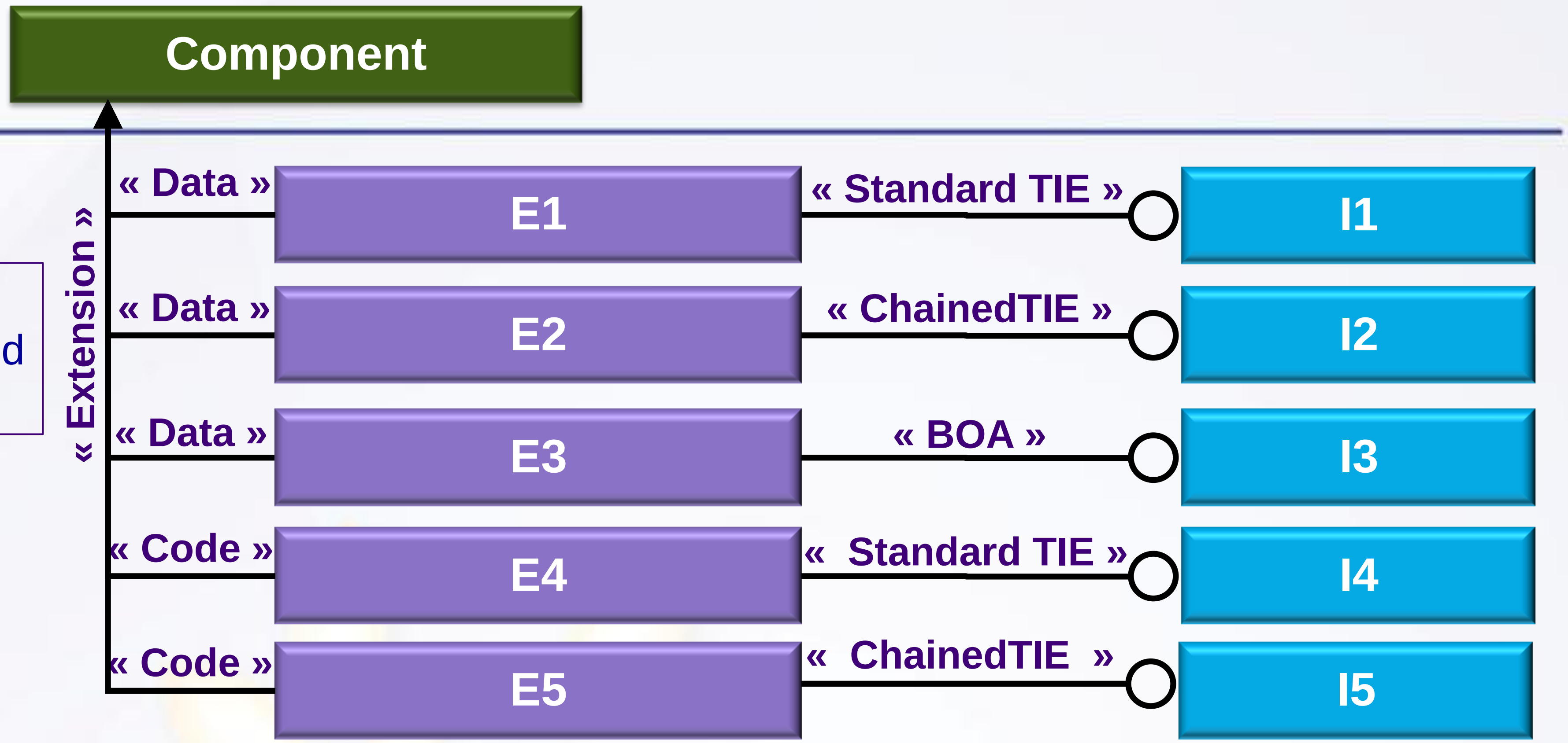
Life Cycle (1/2)

- **A component must be deleted when its behaviors are no longer used**
  - Lifecycle is managed by a COM counter mechanism
    - *AddRef() to increment the counter*
    - *Release() to decrement the counter*
    - *Object will be deleted if the counter equals zero*

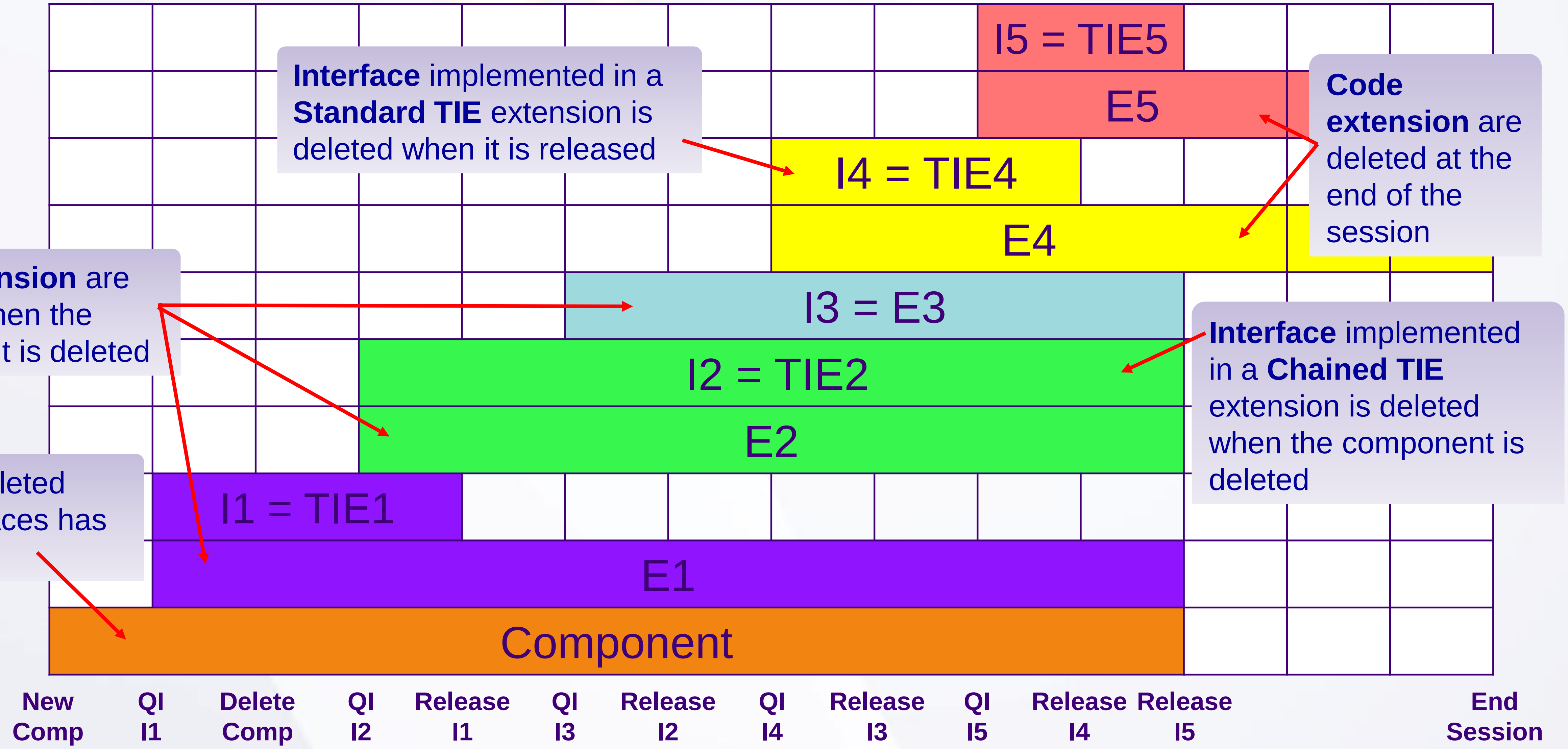


- **The TIE / Extension counter**
  - is already incremented when you get an interface pointer from any methods as *QueryInterface()*
  - must be incremented when you get an interface pointer thanks to an affectation
  - must be decremented each time you don’t need it any longer
- **The Component counter**
  - is managed automatically when one of its interfaces is “AddRef()/Release()”
- **Components are deleted when their counters are equal to zero**
  - Standard TIE are deleted once their counter is equal to zero
  - Chained TIEs and Data extensions are deleted when the component is deleted

Life Cycle (2/2)



a Chained TIE behave like a Standard TIE if it is implemented with a Code extension



Interface implemented in a **Standard TIE** extension is deleted when it is released

Code extension are deleted at the end of the session

Data extension are deleted when the component is deleted

Interface implemented in a **Chained TIE** extension is deleted when the component is deleted

Component is deleted when all its interfaces has been released

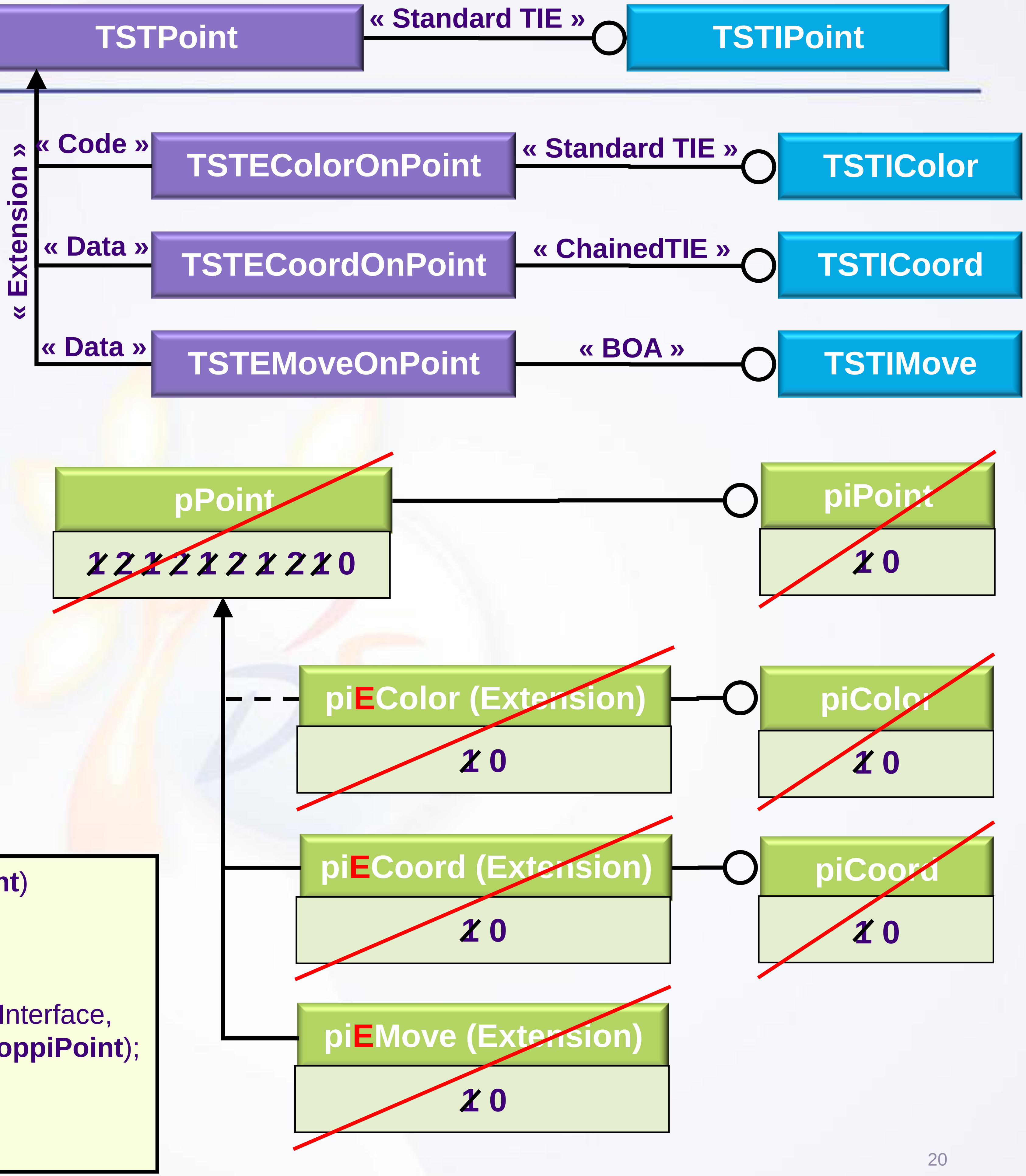
Example

TSTMain.cpp

```
TSTIFactory * piFactory = …;
TSTIPoint * piPoint = NULL;
piFactory → CreatePoint (&piPoint );
piPoint → QueryInterface(IID_ TSTIColor,
                         (void**)& piColor);
piPoint → Release(); piPoint = NULL;
…
piColor → QueryInterface(IID_ TSTICoord,
                         (void**)& piCoord);
piColor → Release(); piColor = NULL;
…
piCoord→ QueryInterface(IID_ TSTIMove,
                         (void**)& piMove);
piCoord → Release(); piCoord = NULL;
…
piMove → Release(); piMove = NULL;
…
Delete_Session (“TestSession”);
```

TSTEFactory.cpp

```
HRESULT CreatePoint (TSTIPoint** oppiPoint)
{
        …
        TSTPoint* pPoint = new TSTPoint();
         rc = pPoint → QueryInterface(IID_ TSTIInterface,
                                      (void**) oppiPoint);
         pPoint → Release(); pPoint = NULL;
        …
}
```



Conclusion

- **Best practices**
  - Use BOA when it is possible
    - Code extension and BOA are not compatible
    - Implementing some DS interfaces with BOA may not be authorized
    - …
  - Use TIE otherwise
  - Use mainly standard TIE, chained TIE might be useful only for DS implementations
  - Use mainly data extension

www.3ds.com

Thank you

DASSAULT SYSTEMES