symbian Academy

Compatibility

Compatibility

This lecture examines

- Compatibility at source level and at binary level

It be may regarded as a collection ...

- Of essential programming rules to ensure that code can be future-proof
- And behave as a good "compatibility citizen"

Compatibility

Fundamentally

- Any interface is a contract that has to be maintained with its users
- Breaking the contract breaks compatibility

A good Symbian developer should understand

- What can and what cannot be modified in an interface to extend a component
- Without breaking either source or binary compatibility

Levels of Compatibility

- Demonstrate an understanding of source, binary, library, semantic and forward/backward compatibility

Forward and Backward Compatibility

Compatibility works in two directions

- Forwards and backwards

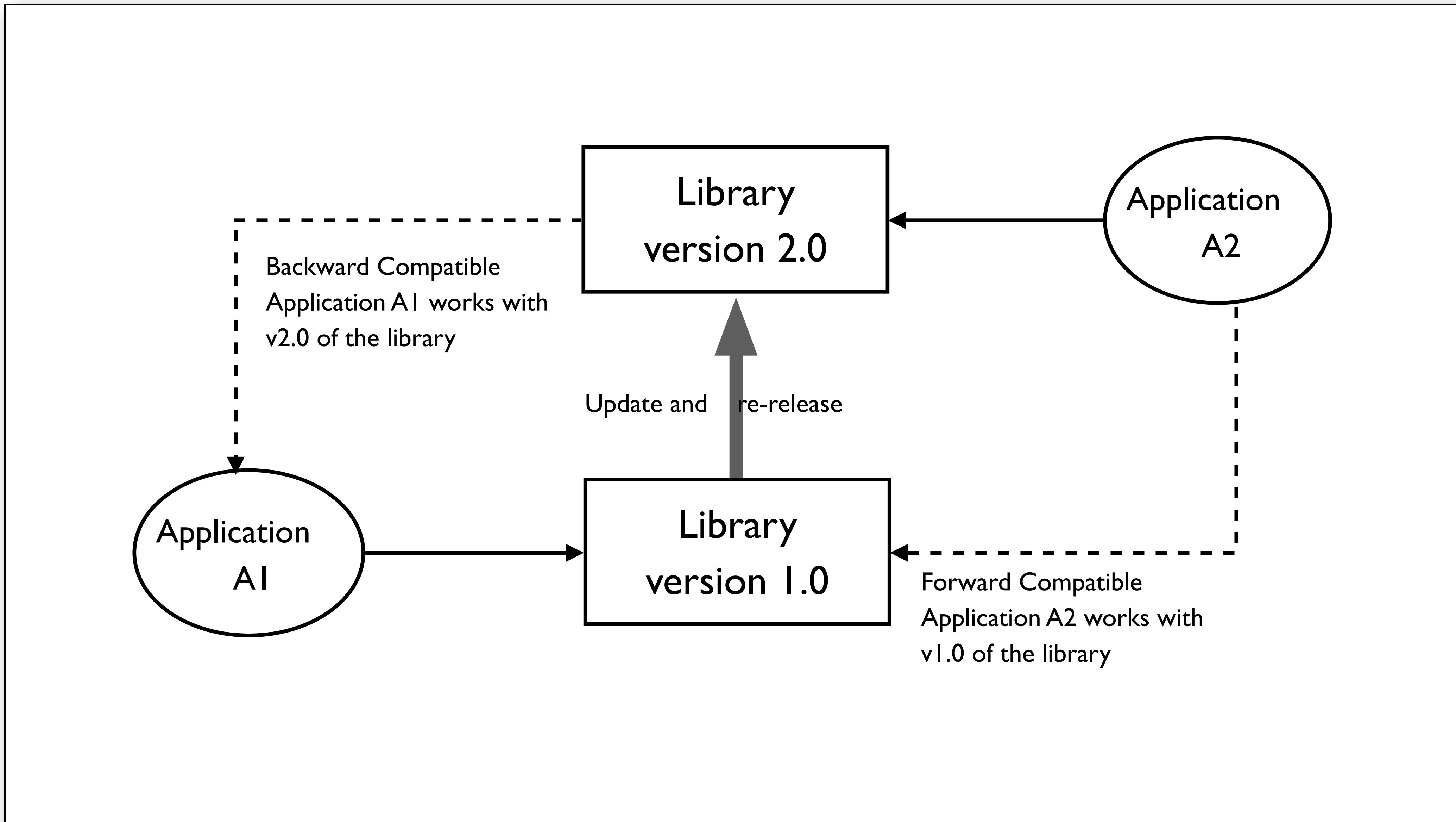
When a component is updated

- In such a way that other code that used the original version
- Can continue to work with the updated version
- That is a backward-compatible change

When software that works

- With the updated version of the component
- Also works with the original version
- The changes are said to be forward-compatible

Forward and Backward Compatibility



New Code works with Old Code

For example

- An application uses an existing library
- Which is updated to a newer version

If the application continues to behave

- In the same manner with new library
- Then the new library has maintained a backward-compatible relationship
- That is - new code works with old code

Old Code works with New Code

A forward-compatible relationship

- Is a little more tricky to achieve

If an earlier version of a library

- Replaces an existing library
- And the application continues to behave in the same manner

The earlier version of the library

- Is said to have a forward-compatible relationship
- That is - old code works with new code

Forward and Backward Compatibility

Backward compatibility

- Is typically the primary goal when making incremental releases of a component
- With forward compatibility a desirable extra

Some changes cannot be forward-compatible

- Such as bug fixes
- Which by their nature do not work "correctly" in releases prior to the fix

Source Compatibility

If a change is made to a component

- And its dependent components can recompile against it
- Without the need to make any changes
- It can be said to be a source-compatible change

An example of a source-compatible change

- Is a bug fix to the internals of an exported function
- Which does not require a change to the function declaration itself
- i.e. the component's interface

Source Compatibility

A typical source-incompatible change

- Involves modifying the internals of a member function
- To give it the potential to leave when previously it could not do so

To adhere strictly to the naming convention

- The name of the function must also be modified by the addition of a suffixed L

Source Compatibility

A source-compatible change

- Does not mean the dependent components do not need to be recompiled
- Just that they do not need to be modified to be compiled successfully.

Binary Compatibility

Binary compatibility is achieved when one component

- Dependent on another can continue to run
- Without recompilation or re-linking
- After the component on which it depends is modified
- The compatibility extends across compilation and link boundaries

Source and Binary Compatibility

One example of a binary-compatible change

- Is the addition to a class of a public non-virtual function
- Which is exported from the library with an ordinal
- That comes after the previous set of exported functions

A client component

- Which was dependent on the original version of the library
- Is not affected by the addition of a function to the end of the export list
- Thus the change is binary-compatible and backward-compatible

Source and Binary Compatibility

If the addition of a new function

- To the class causes the ordinals of the exported functions to be reordered
- The change is not binary-compatible
- Although it continues to be source-compatible i.e. recompiled not modified

Dependent code must be recompiled

- Otherwise it would use the original now invalid ordinal numbers
- To identify the exports

Class-Level and Library-Level Compatibility

Maintaining compatibility at a class level means:

- Ensuring methods continue to have the same semantics as were initially documented
- No publicly accessible data is moved
- Or made less accessible
- The size of an object of the class does not change

Maintaining library-level compatibility means ensuring:

- That the API functions exported by a DLL are at the same ordinal
- That the parameters and return values of each are still compatible.

Preventing Compatibility Breaks
What Cannot Be Changed?

- Recognize which attributes of a class are necessary for a change in the size of the class data not to break compatibility
- Understand which class-level changes will break source compatibility
- Understand which class-level changes will break binary compatibility
- Understand which library-level changes will break binary compatibility
- Understand which function-level changes will break binary and source compatibility
- Differentiate between derivable and non-derivable C++ classes in terms of what cannot be changed without breaking binary compatibility

The Size of a Class Object must not Change

Changing the size of a class object

- e.g. by adding or removing data
- Will cause a binary-compatibility break

Unless it can be guaranteed that:

The class is not externally derivable

- i.e. a constructor is not exported from the DLL which defines it

The only code that allocates an object

- Resides within the component/DLL being changed
- Or it has a non-public constructor that prevents it from being created on the stack

The class has a virtual destructor

The Size of a Class Object must not Change

The size of memory required for an object

- To be allocated on the stack is determined
- For each component at build time

To change the size of an object

- Would affect previously compiled client code
- Unless the client is guaranteed to instantiate the object
- Only on the heap
- For example by using a **`NewL()`** factory function

The Size of a Class Object must not Change

Additionally

- Access to data members within an object
- Occurs through an offset from the this pointer

If the class size is changed

- For example, by adding a data member
- The offsets of the data members of derived classes
- Are rendered invalid

The Size of a Class Object must not Change

To ensure that an object of a class

- Cannot be derived or instantiated
- Except by members (or friends) of the class
- It should have private non-inline and non-exported constructors

It is not sufficient

- Simply not to declare any constructors
- Since the compiler will then generate an implicit public default constructor

The Size of a Class Object must not Change

If a class needs a default constructor

- It should be defined as private
- And implemented in the source
- Or at least not inline where it is publicly accessible

All Symbian OS C classes derive from `CBase`

- Which defines a protected default constructor
- Preventing the compiler from generating an implicit version

If Something is Accessible
It must not be Removed

If something is removed from an API

- Which is used by an external component
- That component's code will no longer compile against the API
- i.e. a break in source compatibility
- Nor run against its implementation
- i.e. break in binary compatibility

If Something is Accessible ...

At an API level do not remove any:

- Externally visible classes
- Functions
- Enumerations
- Values within an enumeration
- Global data such as string literals or constants

At a class level do not remove any:

- Methods
- Member data

Private and protected member data

- Should not be removed
- As this will change the size of the resulting object

symbian Academy

Accessible Member Data must not be Rearranged

Rearranging the order of member data

- Can cause problems to client code that accesses that data directly
- As the offset of the member data
- From the object's this pointer will be changed

Do not change the position of member data if that data is:

- Public - or protected if the client can derive from the class
- Exposed through public or protected inline methods - which will have been compiled into client code

Accessible Member Data must not be Rearranged

This rule also means

- The order of the base classes from which a class multiply inherits
- Cannot be changed without breaking compatibility
- As the order affects the overall data layout of the derived object

Exported Functions must not be Reordered

Each exported API function

- Is associated with an ordinal number
- Used by the linker to identify the function
- The function ordinals are stored in the module definition (`.def`) file

If the `.def` file list is reordered

- For example, by adding a new export within the list
- The ordinal number values will change

Thus the previously compiled code

- Will be unable to locate the correct function

Exported Functions must not be Reordered

For example

- Adding a new function at the start of the list
- Will shunt all the ordinals up one
- Thus any component using ordinal 4
- Would be now be looking at what was previously ordinal 3

This change breaks binary compatibility

- To avoid this
- A new export should always be added to the end of the `.def` file
- Which assigns it a new previously unused ordinal value

Virtual Functions

Virtual functions of externally derivable classes must not be

- Added
- Removed
- Modified

As this will break binary compatibility

Virtual Functions

If a derived class

- Defines its own virtual functions
- The functions will be placed in the virtual function table
- Directly after those defined by the base class

If a virtual function

- Is added or removed in the base class
- There will be a change in the `vtable` position
- Of any virtual functions defined by a derived class

Thus any code

- That was compiled against the original version of the derived class
- Will now be using an incorrect `vtable` layout
- Breaking binary compatibility

Virtual Functions

The following modifications of virtual functions will also break compatibility:

- Changing the parameters
- Modifying the return type
- Changing the use of const

However

- Changes to the internal operation of the function
- For example bug fixes
- Do not affect backward compatibility

Virtual Functions must not be Reordered

Although not stated in the C++ standard

- The order in which virtual member functions
- Are specified in the class definition
- Can be assumed be the only factor which affects
- The order in which they appear in the virtual function table

This order should not be changed

- Since client code compiled against
- An earlier version of the virtual function table
- Will call what has become a completely different virtual function

Virtual Functions

Virtual functions that were previously inherited should not be overridden

- Overriding a virtual function that was previously inherited
- Alters the virtual function table of the base class
- As existing client code is compiled against the original vtable
- It will continue to access the inherited base-class function
- Rather than the new derived version

This leads to inconsistency

- Between callers compiled against the original version of the library
- And those compiled against the new version

Although it does not strictly result in incompatibility

- This is best avoided

Virtual Functions

For example

- A client of **CSiamese** version 1.0
- Calling **SleepL()** invokes **CCat::SleepL()**
- While clients of version 2.0
- Invoke **CSiamese::SleepL()...**

```
class CCat : public CBase // Abstract base class
  {
public:
  IMPORT_C virtual ~CCat() = 0;
public:
  IMPORT_C virtual void PlayL(); // Default implementation
  IMPORT_C virtual void SleepL(); // Default implementation
protected:
  CCat();
  };
```

Virtual Functions

```
class CSiamese : public CCat // Version 1.0
  {
public:
  IMPORT_C virtual ~CSiamese();
public:
  // Overrides PlayL() but defaults to CCat::SleepL()
  IMPORT_C virtual void PlayL();
  // ...
  };

class CSiamese : public CCat // Version 2.0
  {
public:
  IMPORT_C virtual ~CSiamese();
public:
  // Now overrides PlayL() and SleepL()
  IMPORT_C virtual void PlayL();
  IMPORT_C virtual void SleepL();
  // ...
  };
```

The Semantics of an API should not be Modified

Changing the documented behavior

- Of a class or global function or the meaning of a constant
- May break compatibility
- Regardless of whether source and binary compatibility are maintained

The Semantics of an API should not be Modified

As a very simple example

- Consider a class which when supplied with a data set
- Returns the average value of that data

If the first release of the **`Average()`** function

- Returned the arithmetic mean value

The second release of **`Average()`**

- Should continue to do so

<u>Not return</u>

- A median value or some other interpretation of an average

The Semantics of an API should not be Modified

Default arguments

- Specified in header files are compiled into client code

Although a change made to a default argument

- Does not break binary or source compatibility
- The client must be recompiled to pick up the change

A client using the old default argument

- Could get an unexpected return value
- Which would be a problem because the behavior of a function
- Also forms part of its interface.

Use of Const should not be Removed

The semantics of "const" should not be removed

- Since this will be a source-incompatible change

This means That the "constness" of a

- Parameter
- Return type
- Or method

Should not be removed

Parameters Passed by Value

Parameters passed by value

- Must not be changed to pass them by reference
- Or vice versa
- As this breaks binary compatibility

When a parameter is passed by value

- The compiler generates a stack copy and passes it to the function

However if the function signature is changed

- To accept the parameter by reference
- A word-sized reference to the original object
- Is passed to the function instead

Parameters Passed by Value

The stack frame usage

- For a pass-by-reference function call is significantly different
- From that for a pass-by-value function call
- Causing binary incompatibility

Parameters Passed by Value

```
class TColor
  {
  ...
private:
  TInt iRed;
  TInt iGreen;
  TInt iBlue;
  };

// version 1.0
// Pass in TColor by value (12 bytes)

  IMPORT_C void Fill(TColor aBackground);

// version 2.0 – binary compatibility is broken
// Pass in TColor by reference (4 bytes)

  IMPORT_C void Fill(TColor& aBackground);
```

What Can Be Changed Without Breaking Compatibility?

- Understand which class-level changes will not break source compatibility
- Understand which class-level changes will not break binary compatibility
- Understand which library-level changes will not break binary compatibility
- Understand which function-level changes will not break binary and source compatibility
- Differentiate between derivable and non-derivable C++ classes in terms of what can be changed without breaking binary compatibility

An API may be Extended

Classes, constants, global data or functions

- May be added without breaking compatibility

A class can be extended by the addition of

- static member functions
- Or non-virtual member functions
- But not virtual member functions

The ordinals for exported functions

- Must be added to the bottom of the module definition file (`.def`) export list
- To avoid re-ordering the existing functions

The Private Internals of a Class may be Modified

Changes to private and protected methods

- That are neither exported nor virtual do not break client compatibility

However the functions must not be called

- By externally-accessible inline methods

Since the call inside the inline method

- Would be compiled into external calling code
- And be broken by an incompatible change to the internals of the class

The Private Internals of a Class may be Modified

Changes to private member data

- Are also permissible

Unless

- They result in a change to the size of the object
- Or move the position of public or protected data in the object
- That are exposed directly through inheritance
- Or through public inline accessor methods

Access Specification may be Relaxed

The C++ access specifier

- `public, protected, private`
- Does not affect the layout of a class
- Can be relaxed without affecting the data order of the object

The position of member data

- In an object is determined solely by the order of specification
- In the class definition

Access Specification may be Relaxed

Changing the access specification

- To a more restricted form
- for example - from **`public`** to **`private`**

Means that the member data

- Becomes invisible to external clients when previously it was visible
- This breaks source compatibility
- but not binary compatibility

Pointers may be Replaced with References and Vice Versa

Changing from a pointer to a reference parameter

- Or return type
- Or vice versa
- Does not break binary compatibility
- But does break source compatibility

This is because references and pointers

- Can be considered to be represented in the same way by the C++ compiler
- That is one machine word.

The Names of exported Non-Virtual Functions may be Changed

Symbian OS is linked purely by ordinal

- Not by name and signature

This means

- It is possible to make changes to the name of exported functions
- Retaining binary compatibility
- But not source compatibility

The Input may be Widened and Output Narrowed

Input can be made more generic or widened

- As long as input that is currently valid retains the same interpretation

For example

- A function can be modified to accept a less derived pointer
- And extra values can be added to an enumeration
- As long as it is extended rather than re-ordered
- Which would change the original values

The Input may be Widened and Output Narrowed

Output can be made less generic ("narrowed")

- As long as any current output values are preserved

For example

- The return pointer of a function can be made more derived
- As long as the new return type applies to the original return value

For multiple inheritance

- A pointer to a class is unchanged
- When it is converted to a pointer to the
- First base class in the inheritance declaration order

That is

- The layout of the object follows the inheritance order specified

The `const` Specifier may be Applied

It is acceptable to change

- Non-`const` parameters
- Return types
- Or the `this` pointer
- To be `const` in a non-virtual function,

Provided the parameter

- Is no more complicated than a reference or pointer

This is because

- It is possible to pass non-`const` parameters
- To `const` functions or those that take `const` parameters

Best Practice
Designing to Ensure Future Compatibility

- Recognize best practice for maintaining source and binary compatibility
- Recognize the coupling arising from the use of inline functions and differentiate between cases where it will make maintaining binary compatibility more difficult and where it will be less significant

Functions should not be Inline

An inline function is compiled into the client's code

- Which means that a client must recompile its code
- In order to pick up a change to an inline function

When using private inline methods

- They must not be accessible externally
- And should be implemented in a file that is accessible
- Only to the code module using it

Using an inline function

- Increases the coupling between a component and its dependents
- Which should generally be avoided.

No Public or Protected Member Data should be Exposed

The position of data is fixed

- For the lifetime of the object if it is externally accessible
- Either directly or through derivation

More flexibility is achieved

- By encapsulating member data privately
- And providing non-inline accessor functions where necessary

Derived Virtual Functions should be Stubbed

This is a defensive programming technique

- Where a derived class overrides all the base-class virtual functions
- Regardless of whether they are needed in the first release
- i.e. there is no need to modify the functions beyond what the base class supplies the override should call the base-class function
- This allows the functions to be extended in future releases

To clarify ...

- Consider the earlier example of **`CSiamese`** deriving from **`CCat`**
- Inherited the default implementation of the **`CCat::SleepL()`** virtual method in version 1.0
- But overrode it in version 2.0 ...

symbian Academy

Derived Virtual Functions should be Stubbed

The sample code below shows how to avoid this

- By overriding both virtual functions in version 1.0
- **CSiamese::SleepL()** calls through to **CCat::SleepL()**

```
Class CCat : public CBase // Abstract base class
  {
public:
  IMPORT_C virtual ~CCat() = 0;
public:
  IMPORT_C virtual void EatL(); // Default implementation
  IMPORT_C virtual void SleepL();// Default implementation
  // ...
  };
```

Derived Virtual Functions should be Stubbed

```
class CSiamese : public CCat // Version 1.0
  {
  IMPORT_C virtual ~CSiamese();
public:
  IMPORT_C virtual void EatL(); // Overrides base class functions
  IMPORT_C virtual void SleepL();
  // ...
  };

void CSiamese::EatL()
  {// Overrides base class implementation
  ... // Omitted for clarity
  }

void CSiamese::SleepL()
  {// Calls base class implementation
  CCat::SleepL();
  }
```


symbian Academy

"Spare" Member Data and Virtual Functions

"Spare" member data and virtual functions

- Should be provided from the outset

Another practical defensive programming technique

- Is to add at least one reserve exported virtual function
- Where there is the possibility for future expansion

This provides the mean To extend the class

- Without disrupting the `vtable` layout of classes

"Spare" Member Data and Virtual Functions

Further more - Reserving at least four extra bytes

- Of private member data in classes
- Is a good future-proofing technique

This reserved data can be used as a pointer

- To extra data as it is required

If the class is unlikely to require later modification

- Extra memory should not be reserved
- Avoiding wasting limited memory resources

Compatibility

- ✓ Levels of Compatibility
- ✓ Preventing Compatibility Breaks - What Cannot Be Changed?
- ✓ What Can Be Changed Without Breaking Compatibility?
- ✓ Best Practice - Designing to Ensure Future Compatibility