Core C++ 2024

Leveraging **Pure Interfaces** For Scalable C++ Applications

Udi Lavi

Who Am I?

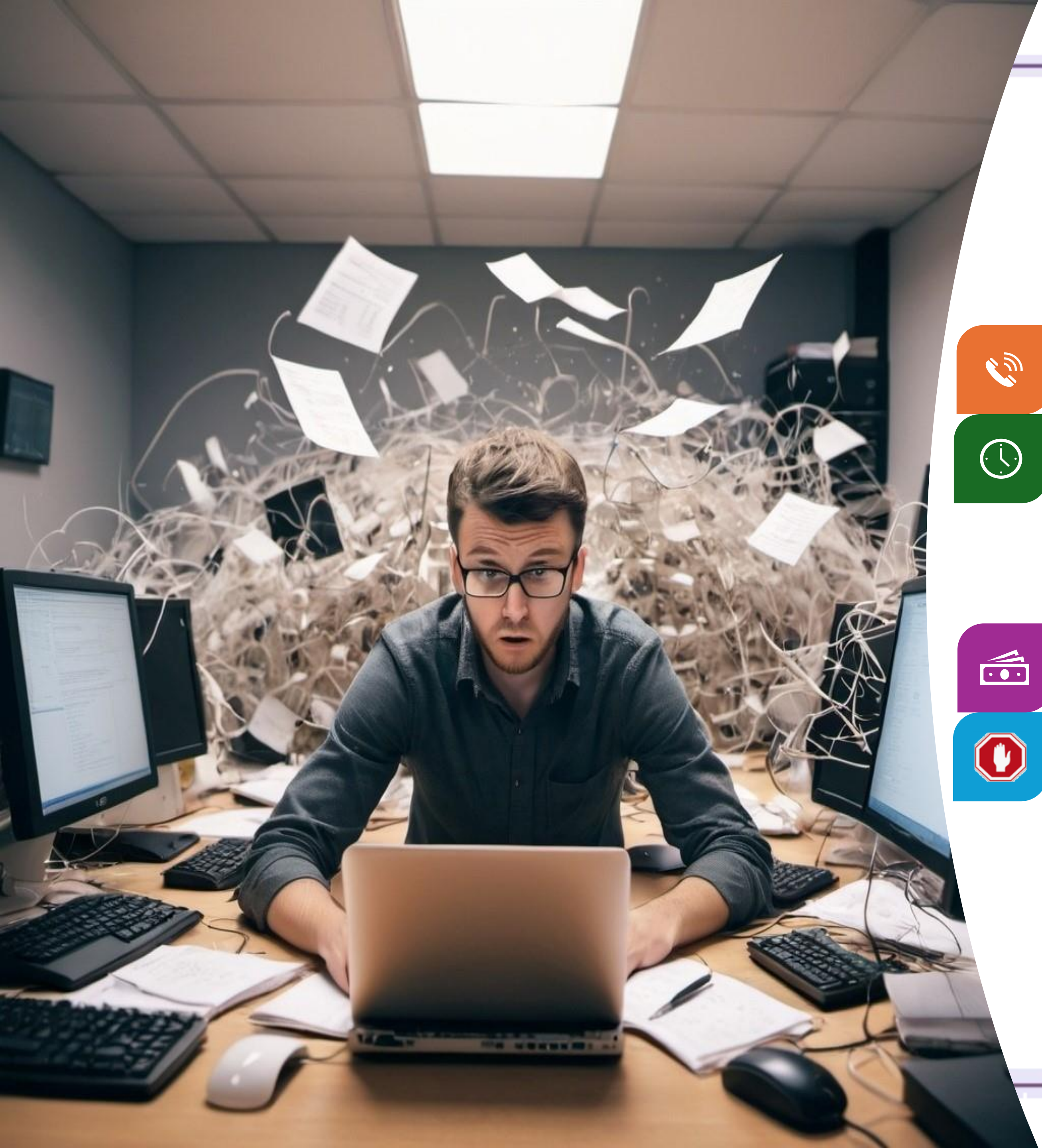
- Udi Lavi
- C++ developer since 2000 (mainly on Windows)
- Simulators infrastructure @ Elbit
- Technical code quality reviewer:
  - Standards / Portability / Performance / Maintainability / …

My "world"

- Simulation **SDK** – infrastructure & tools (integration / configuration)
- Mainly C++**98** (a constraint by some projects)
- **Variety** of projects requirements:
  - Scale & resources – CPU / memory / computers
  - Environments – Embedded / Cloud / PC
  - OS & Architecture – (x32 & x64) / Endian
  - Libraries linkage – Shared (.dll / .so) / Static (environment dependnent)
- Other considerations
  - **Security** standards
  - Backward **compatibility**

Outline

- Introduction to pure interface
    - What's a pure interface
    - Incentive
- Creating pure interfaces in C++
- Migration case study – cost & profit
- Performance
- Restrictions – API Evolution & Backward Compatibility



Before

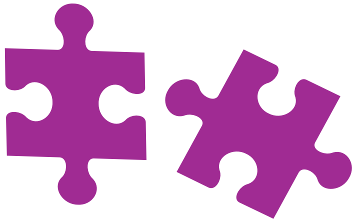
- **Many calls for support**
- **Long builds**
    - **Dependency prevented shortcuts**
        - **Shortcuts → extra support calls**
- **Long support time**
- **Waiting a few days for support**

Leveraging Code Quality

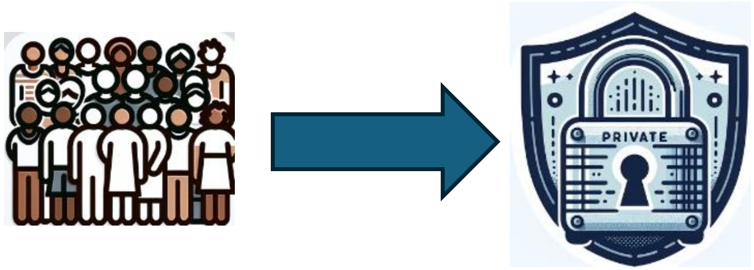
- **Standards enforcement** (Security / Style / Formatting)
- **Static / Dynamic analysis**
- **Warning as errors**
- **Unused code elimination** (+ code coverage)
- **Optimization & Performance** (+ profilers)
- **Portability** (OS / Endian / x32/x64)
- **CI/CD automation**

Leveraging Code Quality – cont.

- **Pure interface**



- Encapsulation
  - **No visible private data**



- Refactoring → **Automatic regression tests**

After

- Less calls for support
- Fast versioned hotfix release
  - ~1 hour after problem is resolved
  - No build of user's project
- Fast support / less regressions
- Fast response
  - Usually immediately / same day

Let’s Move To

Pure Interface

Let’s Move To Pure Interface

• Team Leader:	**Let’s move our API to pure interface!**	
• Me:	**Are you nuts?**	
	It’s a lot of work	(V)
	It has a performance tradeoff	(?)

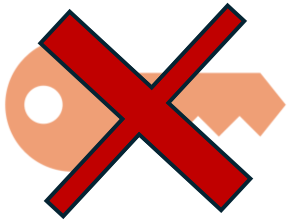
- **Why?**

Incentive – The Security Guard

Security guard:	**Good morning.**
	May I have your wallet please?
Visitor:	**Why do you need my wallet?**
Security guard:	**I need to identify who you are.**
Visitor:	**Oh, so you just need my ID card.**
Security guard:	**Give me your wallet, I’ll take what I need.**
Visitor:	**I don’t feel comfortable about it**

Moral

- We do not wish to expose more than we need to
- Some things are better kept **private**



- Some things better remain **unexposed**

Pure Interface
What's it all about

Definitions

- Abstract Class
  - A class that cannot be instantiated directly (only by derived)
  - Contains **at least one** pure virtual function
- Pure Interface
  - An abstract class containing **only** pure virtual functions
    - Implementation – only in derived class(es)

Incentive – Encapsulation

- Maintainability – Hiding implementation details
- Usability
  - Users see only what they need
  - No forbidden API
  - No cluttered huge API
- Improves Testability
  - Easier to make doubles

Incentive – Decoupling

- Reduced dependency
  - Internal changes are invisible
- Easy hotfixes / versions releases
  - Reduced user rebuild (depend only on API changes)
- Potential Debug / Release mix
  - Requires allocation / deallocation boundaries isolation
    - Registering creation & destruction function (for plugins)

Incentive – ABI (Application Binary Interface)

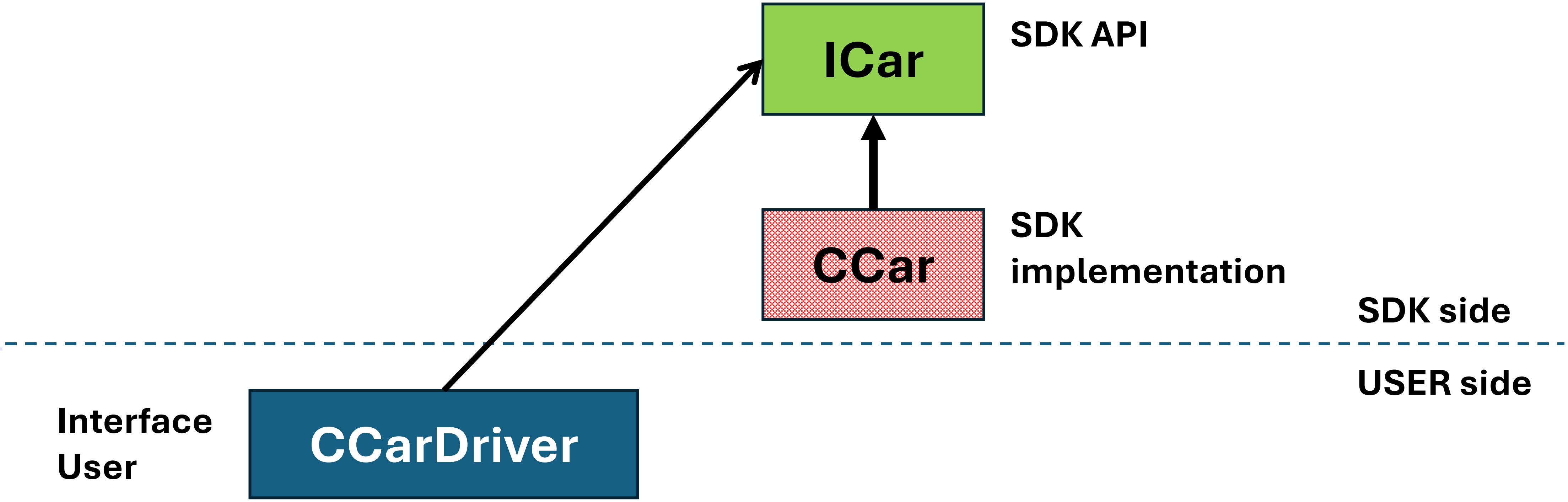
- Pure interface keeps ABI compatibility*
  - At least in practice
- Potential breakers
  - Cross boundary heap management (e.g. std::string)
  - Some optimization flags

Maybe we don’t need it?

- Pure interfaces **<u>might</u>** be an overkill
  - Too simple product
  - Internal API
  - Short life-span product

Creating
Pure Interface

“Simple” Interfaces



Interface Example – **Car_Interface**.h

```
class ICar
{
protected:
        // prevent direct destruction (and construction ???)
        virtual ~ICar() = 0;


public:
        virtual void Drive() = 0;
};
```

Interface Example – .h (new instances)

```
class ICar
{
protected:
	virtual ~ICar() = 0;
public:
	// Heap allocation (instead of new / delete) – static functions
	static ICar &CreateInstance(optional parameters);
	static void ReleaseInstance(ICar *&rpIntstance);
	static void ReleaseInstance(ICar &rIntstance);

	virtual void Drive() = 0;
};
```

Interface Example – **Car_Interface**.cpp

```
ICar &ICar::CreateInstance(optional parameters) {
	CCar * const pObj = new CCar(optional parameters);	// DERIVED
	return *pObj;
}
void ICar::ReleaseInstance(ICar *&rpIntstance) {
	delete rpIntstance;				// ICar may delete ICar (BASE)
	rpIntstance = NULL;
}
void ICar::ReleaseInstance(ICar &rIntstance) {
	delete &rIntstance;
}
```

Usage Example

<u>Option #1</u>

```
IUIntVector &rMyIds = IUIntVector::CreateInstance();
...
IUIntVector::ReleaseInstance(rMyIds);
```

<u>Option #2</u> **(for containers / data members not in MIL)**

```
m_pMyIds = &IUIntVector::CreateInstance();
...
IUIntVector::ReleaseInstance(m_pMyIds);
```

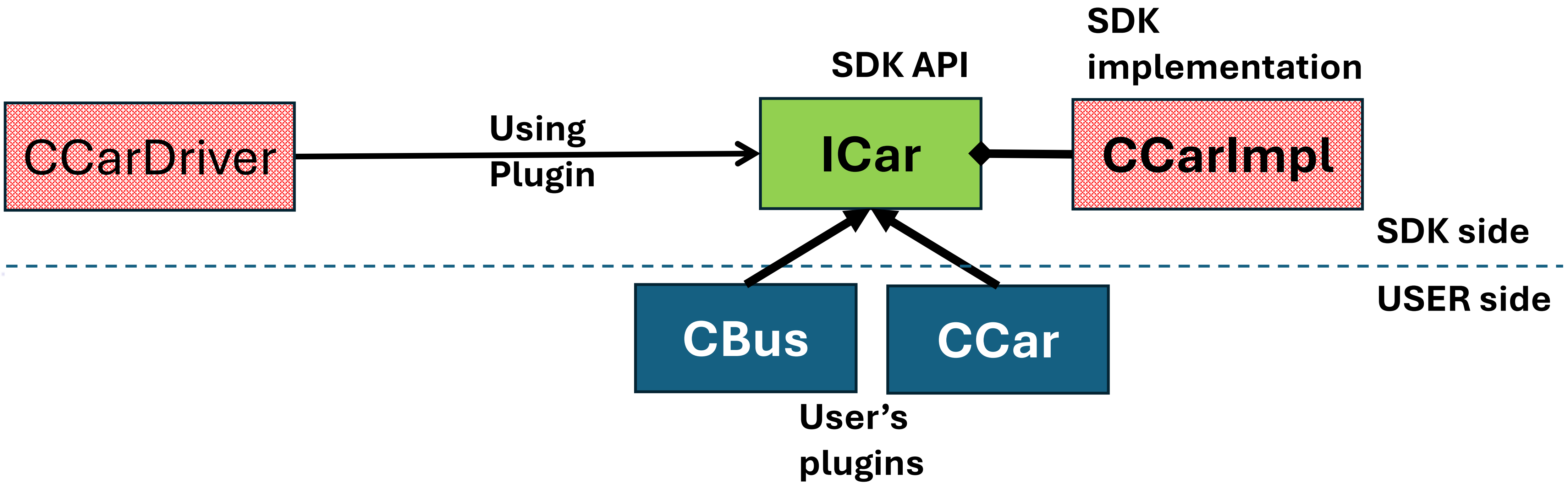
Singleton Example – .h

```
class ICarManager
{
public:
        static ICarManager &Instance();         // SINGLETON

        // Implemented in CCarManager
        virtual ICar *GetFreeCar() = 0;         // ICar


protected:
        virtual ~ICarManager() = 0;
};
```

Singleton Example – .cpp

```
ICarManager &ICarManager::Instance()
{
    // standard singleton
    static CCarManager s_cTheInstance;
    return s_cTheInstance;
}
```

Plugins Interface ("inverse" inheritance)



Plugins – “Inverse” Inheritance

- User’s plugins – usually polymorphic
- Pimpl – Pointer to Implementation
  - Pimpl hides SDK side implementation
  - Interface holds a pointer to a **forward declared** class
- Interface
  - pure-virtual – implemented by derived
  - Non-virtual – delegates to pImpl

“Inverse” Inheritance Example – .h

```
class CCarImpl;             // forward declaration (only SDK can include its .h)
class ICar
{
private:
        CCarImpl  *m_pImpl;
protected:
        ICar();             // Called by user derived CTOR. Constructs m_pImpl
public:
        virtual ~ICar();                  // public* SDK may* delete (protected otherwise)
        void Drive();                     // SDK delegates to pImpl->Drive();
        virtual void OnDrive() = 0;       // User must implement (called by SDK)
        CCarImpl &GetCarImpl();           // Possibly also 'const' version
};
```

“Inverse” Inheritance Example – .h

```
...
class ICar
{
        ...
protected:
        ...
        virtual ~ICar();                       // protected
public:
        virtual void DeleteSelf () = 0;        // public – delete this (EFFORT)
        ...
};
```

Case Study

SDK – Legacy Code Migration

Migration Cost

- Refactoring SDK API – Approx. 1 human year (~75% capacity)
  - Minor part of API is not pure interface per-se
    - E.g. templates (base class is pure interface)
- Migrating 1st big-scale project – Approx. 1 month:
  - API stabilization
  - Fixing project’s bugs / misuses
- **Partial** automation could reduce migration effort

Migration result

- Usability – over 50% cut in SDK's .h files (originally over 300)
  - Reduced .h files size
- Compatibility – **ABI** compatibility
  - VS 2010 → VS 2017
  - VS 2017 → VS 2022
- **Encapsulation** – users are not exposed to "internals"
  - No mistakes / No abuse / No surprises
- Maintainability (user) – **hotfixes** w/o project rebuild
- Maintainability (SDK) – detection & deletion of **unreachable** code
- **Support** – less calls & fast response

Performance

Virtual Function – Performance Penalties

- The penalties (might be irrelevant – next slide):
  - Indirect Call – function's address lookup in virtual table at runtime [~0]
  - No build time determination of function address [+]
  - **Cache Miss** – potential for additional cache miss (vtable not in cache) [+++]
- Penalties are generally small
- Often benefits outweigh penalties
- Performance usually goes unnoticed in other places w/o benefits:
  - Allocations / chattiness in loops / CTOR & copy

Optimized Performance

- Plugins – methods are virtual anyway
- **Spatial locality** – eliminates cache miss impact for consecutive calls
  - Loops – SDK – mostly **NON-**polymorphic interfaces
  - Chattiness – calls on same object
- Devirtualization (+ optional inline)
  - Internal use of CCar (instead of ICar)
    - ‘final’ keyword
  - Optimization
    - PGO – Profile Guided Optimization
    - WPO / LTO – Whole Program Optimization / Link Time Optimization

Restrictions

API Evolution & Compatibility

API Changes Effects – No User Build

- Indirect new & delete – hiding DTOR
- Adding member functions
  - Virtual methods – at end of class [**vtable order**]
  - Static member functions – anywhere
- Using only interfaces & primitive types (**no templates / STL**)
- No data members (optional pImpl)
- No change to plugins interfaces [**vtable order**]
- No methods deletion (can use **DEPRECATED** keyword in VS)
- Can’t have:
  - postfix iteration / copy-CTOR

API Changes Effects – User Modifications

- Cases requiring **user's code modifications**
  - Usage of new methods / classes
  - Renamed methods / classes
  - API deletion / modifications (e.g. new parameters w/o default)

API Changes Effects – User Build (only)

- Cases requiring **user recompilation**
  - Change in methods order (changed vtable)
  - Adding parameters with default (changes signature)
  - Adding overloading for existing methods (may eliminate previous cast)
  - Modified API templates (changed implementation)

Summary

- A way to hide implementation details
  - Makes code more robust & maintainable
  - Improves productivity
- Suitable for “products” between development Groups
- Fits design principles of encapsulation & decoupling

- Effort may pay off (It did in our case)

Summary – API constraints

- Methods – pure virtual / static (otherwise delegate to m_pImpl)
- Types – primitives & interfaces
- Plugins – m_pImpl
- Object Lifetime
  - CreateInstance() / ReleaseInstance()
  - Instance() + some kind of "getter" – e.g. **GetFreeCar()**
  - Prevent access to DTOR
- Order (private / protected / public)

Q & A

?

END