Курсова робота на тему:

Стратегії інтелектуальних указників

Виконав: студент 3-го року навчання,
Спеціальності: 121 Інженерія програмного забезпечення,
Клепацький Олег Святославович

Керівник Бублик В.В.
доцент, кандидат фіз.-мат. наук

Завдання роботи

- Дослідити та описати прийоми метапрограмування, які використовує дизайн на основі стратегій.
- Дослідити та застосувати особливості метапрограмування в новому стандарті C++20.
- Порівняти стратегії інтелектуальних указників Александреску (на основі C++03) з STL (C++11).
- Виявити та виправити недоліки дизайну на основі стратегій в C++03.
- Визначити стратегії інтелектуальних указників на основі C++20.
- Протестувати реалізацію нових стратегій у демонстраційному проекті.

Проблеми сирих указників

- **Витоки пам’яті**
  - Неконтрольоване звільнення пам’яті
  - Ранній вихід із функції (early return)
  - Виняткова ситуація (exception)
- **Використання неініціалізованих указників**
- **Використання після звільнення пам’яті** («підвислі» указники)

Метод захоплення ресурсів (RAII)

Після отримання ресурсу, ми зобов’язані його звільнити («правила гарного тону»)

- malloc - free
- new – delete
- lock – unlock
- open – close

…

Метод захоплення ресурсу (RAII – Resource acquisition is initialization) – ідіома C++, яка визначає, що ресурсом має володіти деякий клас, відповідальний за його звільнення.

При отриманні ресурсу він передається у конструктор такого класу, а звільняється у деструкторі (виклик якого гарантовано відбудеться).

Інтелектуальний указник

Інтелектуальний указник – це клас-обгортка звичайного указнику, який реалізує ідіому RAII та відповідальний за звільнення пам’яті, на який указує.

```
template<typename T>
struct SmartPointer
{
	SmartPointer(T* p) : pointee_(p) {}

	~SmartPointer()
	{
		delete pointee_;
	}
private:
	T* pointee_;
};
```

Інтелектуальні указники STL (C++11)

unique_ptr (одноосібний указник)

- Одноосібне володіння
- Відсутній копіювальний конструктор
- Можна визначати власну стратегію видалення

shared_ptr (розподілений указник)

- Підрахунок відсилок
- Указник видаляється останнім, хто ним володіє
- Стратегія видалення є параметром конструктора

weak_ptr (слабкий указник)

- Не володіє об’єктом
- Для користування указника треба отримати shared_ptr
- Вирішує проблему циклічних указників

Дизайн на основі стратегій за Александреску

```
struct ConcreteStrategy
{
	void someMethod()
	{
		// implementation
	}
};

template<class Strategy>
struct MyClass : private Strategy
{
	void someMethod()
	{
		// delegating this method to strategy
		Strategy::someMethod();
	}
};

int main()
{
	MyClass<ConcreteStrategy> obj;
	obj.someMethod();
}
```

Основна ідея: поєднання шаблонів та множинного спадкування

Переваги:

- Гнучка зміна поведінки класу
- Відсутність віртуальних викликів
- Велика кількість комбінацій стратегій
- Перевикористання коду

Недоліки:

- Відсутність явних вимог до стратегій
- Необхідність великої кількості документації

Стратегії інтелектуальних указників

Зберігання	Володіння	Перетворення	Перевірки	Видалення
• Зберігає ресурс (указник) • Визначає оператори розіменування	• Визначає спосіб володіння ресурсом • Визначає, коли буде звільнено ресурс	• Визначає, чи існує неявна конвертація до типу ресурсу	• Визначає, які значення є допустимими • Перевірка у конструкторах • Перевірка перед розіменуванням	• Визначає, як ресурс буде звільнено • У Александреску – частина стратегії зберігання

Рефакторинг успадкованого коду Александреску

Бібліотека Loki надає реалізацію інтелектуальних указників на основі стратегій.

Недоліки:

1. Застарілий код, несумісний з новими стандартами
2. Відсутність вимог до параметрів шаблону (стратегій)
3. Відсутність семантики переміщень
4. Використання застарілих ідіом
5. Використання застарілих технік метапрограмування
6. Відсутність стратегії видалення

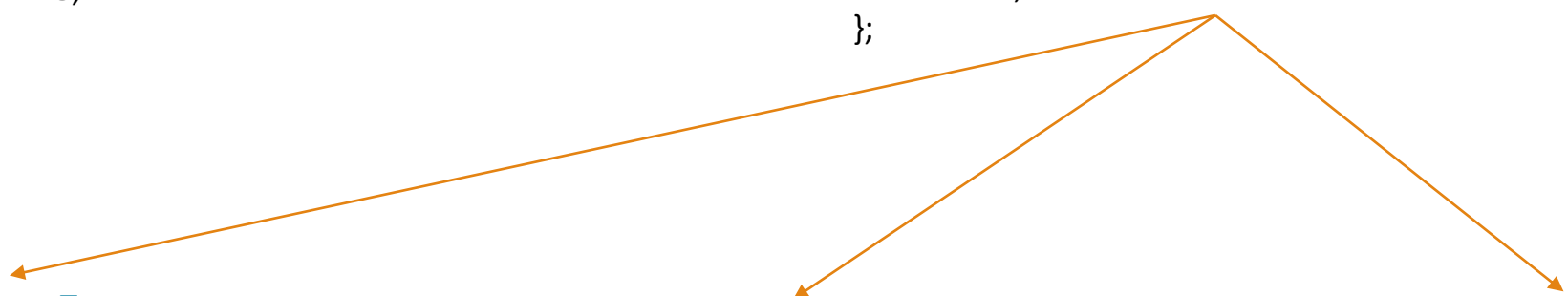
https://www.loveinartsz.com/loki-deceitful-god-of-norse-mythology/

Концепти (C++20)

Концепт (concept) – це предикат, який обчислюється на етапі компіляції, і визначає набір вимог до параметру шаблону.

```
template<typename T>
T add(T lhs, T rhs)
{
        return lhs + rhs;
};
```

```
// Concept definition
template<typename T>
concept Addable = requires (T a, T b) {
        a + b;
};
```



```
template<typename T>
T add(T lhs, T rhs)
requires Addable<T>
{
        return lhs + rhs;
};
```

```
template<Addable T>
T add(T lhs, T rhs)
{
        return lhs + rhs;
};
```

```
auto add(Addable auto lhs,
        Addable auto rhs)
{
        return lhs + rhs;
};
```

Переваги концептів

- Забезпечують типізацію «типових» параметрів шаблону.
- Вимоги до параметрів шаблону є частиною сигнатури.
- Недотримані вимоги виявляються вже на етапі виклику функції або інстанціювання шаблону.
  - А не глибоко всередині реалізації.
  - Це призводить до значно коротших повідомлень про помилки.
- Повідомлення про помилки часто є більш змістовними.
- Є можливість виявляти семантичні помилки.
- Функції можуть диспетчеризуватись за властивостями типу параметру.

Вимоги до стратегії зберігання

1. Визначає типи, які повертають оператори розіменування (operator* та operator->)
2. Визначає модифікатори
3. (опція) Визначає оператори розіменування

```
template<template <class> class T, typename P>
concept StoragePolicyConcept = requires(T<P> t, const T<P> ct) {
        typename T<P>::StoredType;
        typename T<P>::PointerType;
        typename T<P>::ReferenceType;
        {ct.getPointer() } -> std::same_as<typename T<P>::PointerType>;
        {t.getPointerRef() } -> std::same_as<typename T<P>::StoredType&>;
        {ct.getPointerRef() } -> std::same_as<const typename T<P>::StoredType&>;
        {ct.defaultValue()} -> std::same_as<typename T<P>::StoredType>;
};
```

```
template<template <class> class T, typename P>
concept Dereferencable = StoragePolicyConcept<T, P> and requires(T<P> t, const T<P> ct) {
        { t.operator->() } -> std::same_as<typename T<P>::PointerType>;
        { ct.operator->() } -> std::same_as<const typename T<P>::PointerType>;
        { t.operator*() } -> std::same_as<typename T<P>::ReferenceType>;
        { ct.operator*() } -> std::same_as<const typename T<P>::ReferenceType>;
};
```

Стратегія зберігання указника

```
template <class T>
class PointerStorage
{
public:
        typedef T* StoredType;
        typedef T* PointerType;
        typedef T& ReferenceType;

        PointerStorage() : pointee_(defaultValue()) {}
        PointerStorage(const StoredType& p) : pointee_(p) {}

        PointerType operator->() const { return pointee_;}
        ReferenceType operator*() const {       return *pointee_;  }
        inline PointerType getPointer() const {return pointee_;}
        inline const StoredType& getPointerRef() const { return pointee_;       }
        inline StoredType& getPointerRef() { return pointee_; }
        inline StoredType&& getPointerRRef() { return std::move(pointee_); }
        constexpr static StoredType defaultValue()    { return nullptr; }
private:
        StoredType pointee_;
};
```

- Зберігає указник
- Визначає тривіальні оператори розіменування
- nullptr за замовчуванням

Вимоги до стратегії володіння

1. Визначає метод release, який визначає, чи необхідно звільняти ресурс
2. (опція) Визначає методи clone чи moveClone для копіювання чи переміщення

```
template <template <class> class T, typename P>
concept OwnershipPolicyConcept = requires(T<P>& t, const P& p, const T<M>& t1) {
        { t.release(p) } -> std::same_as<bool>;
};
```

```
template<template <class> class OP, typename T>
concept CopyCloneable = requires (OP<T> op, const T& t) {
        {op.clone(t)} -> std::same_as<T>;
};
```

```
template<template <class> class OP, typename T>
concept MoveCloneable = requires (OP<T> op, T&& t) {
        {op.moveClone(std::move(t))} -> std::same_as<T>;
};
```

Стратегії володіння

Unique

- Одноосібне володіння
- Забороняє копіювання

RefCounted

- Розподіленне володіння
- Підрахунок відсилок

Observer

- Стратегія без володіння
- Дозволяє конвертацію з інших стратегій

Вимоги до стратегії перетворення

1. Визначає статичну змінну allow, яка означає, чи можна виконати неявне перетворення до типу указника.

```
template <typename T>
concept ConversionPolicyConcept = requires {
      { T::allow } -> std::same_as<const bool&>;
};
```

Стратегія перетворення

```
struct AllowConversion
{
	static constexpr bool allow = true;
	void swap(AllowConversion&) {}
};
```

Існують лише дві стратегії перетворення, які дозволяють чи забороняють неявне перетворення

```
struct DisallowConversion
{
	static constexpr bool allow = false;
	DisallowConversion() {}
	DisallowConversion(const AllowConversion&) {}
	void swap(DisallowConversion&) {}
};
```

Застосування

```
class SmartPtr
{
// ...
	operator StoredType()
	requires CP::allow
	{
		return SP::getPointer();
	}
}
```

Вимоги до стратегії перевірки

Визначає методи, які перевірятимуть на коректність значення указника

- onDefault – у конструкторі за замовчуванням
- onInit – у конструкторі з параметром – указником
- onDereference – у операторах розіменування

```
template<template <typename> class T, typename P>
concept CheckingPolicyConcept = requires(T<P>&t, const P & p, const T<M>& t1) {
        { T<P>::onDefault(p)} -> std::same_as<void>;
        { T<P>::onInit(p)} -> std::same_as<void>;
        { T<P>::onDereference(p)} -> std::same_as<void>;
};
```

Стратегія перевірки

```
template <class P>
struct AssertCheck
{
	AssertCheck() {}

	template <class P1>
	AssertCheck(const AssertCheck<P1>&) {}

	template <class P1>
	AssertCheck(const NoCheck<P1>&) {}

	static void onDefault(const P&) {}

	static void onInit(const P&) {}

	static void onDereference(P val)
	{
		assert(val);
	}

	static void swap(AssertCheck&) {}
};
```

AssertCheck – перевірка перед розіменуванням за допомогою **assert**, чи вказівник не є nullptr

Існують й інші стратегії:

- **NoCheck** – жодної перевірки
- **AssertCheckStrict** – забороняє nullptr

Вимоги до стратегії видалення

1. Має визначати operator() типу, який визначає стратегія зберігання

```
template<typename T>
struct StandardDeleter {
        void operator()(T* ptr) {
                delete ptr;
        }
};
```

Стратегія за замовчуванням

```
template<typename T>
struct ArrayDeleter {
        void operator()(T* ptr) {
                delete[] ptr;
        }
};
```

Стратегія видалення масивів

```
template<typename T>
struct NoDeletion {
        void operator()(T*) {}
};
```

Пуста стратегія видалення (неволодіючий указник)

Поєднання стратегій

UniquePtr – одноосібний указник

```
template <typename T, std::invocable<T*> DeletePolicy = StandardDeleter<T>>
using UniquePtr = SmartPtr<T,DeletePolicy, Unique, AllowConversion, AssertCheck, PointerStorage>;
```

SharedPtr – розподілений указник, підрахунок відсилок

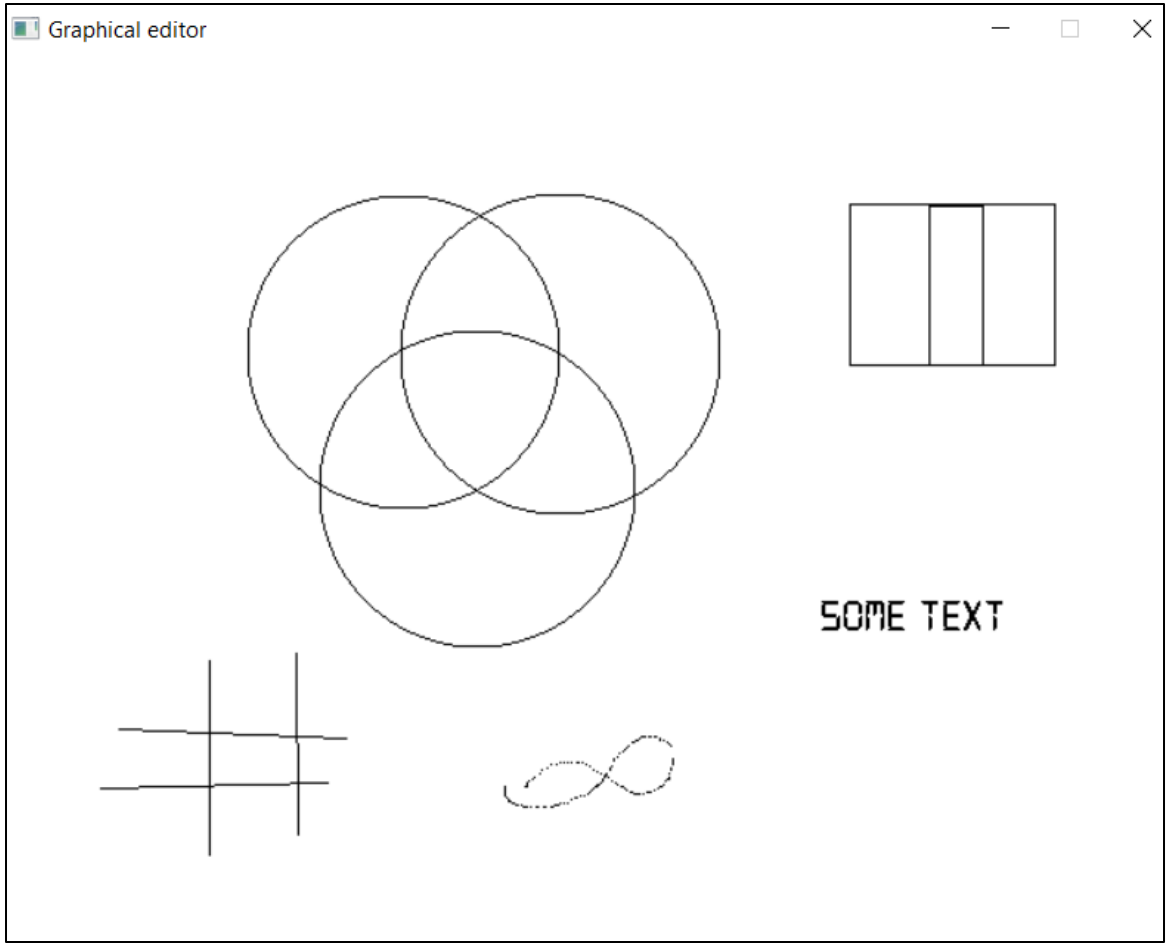
```
template <typename T, std::invocable<T*> DeletePolicy = StandardDeleter<T>>
using SharedPtr = SmartPtr<T, DeletePolicy, RefCounted, AllowConversion, AssertCheck, PointerStorage>;
```

ObserverPtr – невладний указник

```
template <typename T>
using ObserverPtr = SmartPtr<T, NoDeletion<T>, Observe, AllowConversion, AssertCheck, PointerStorage>;
```

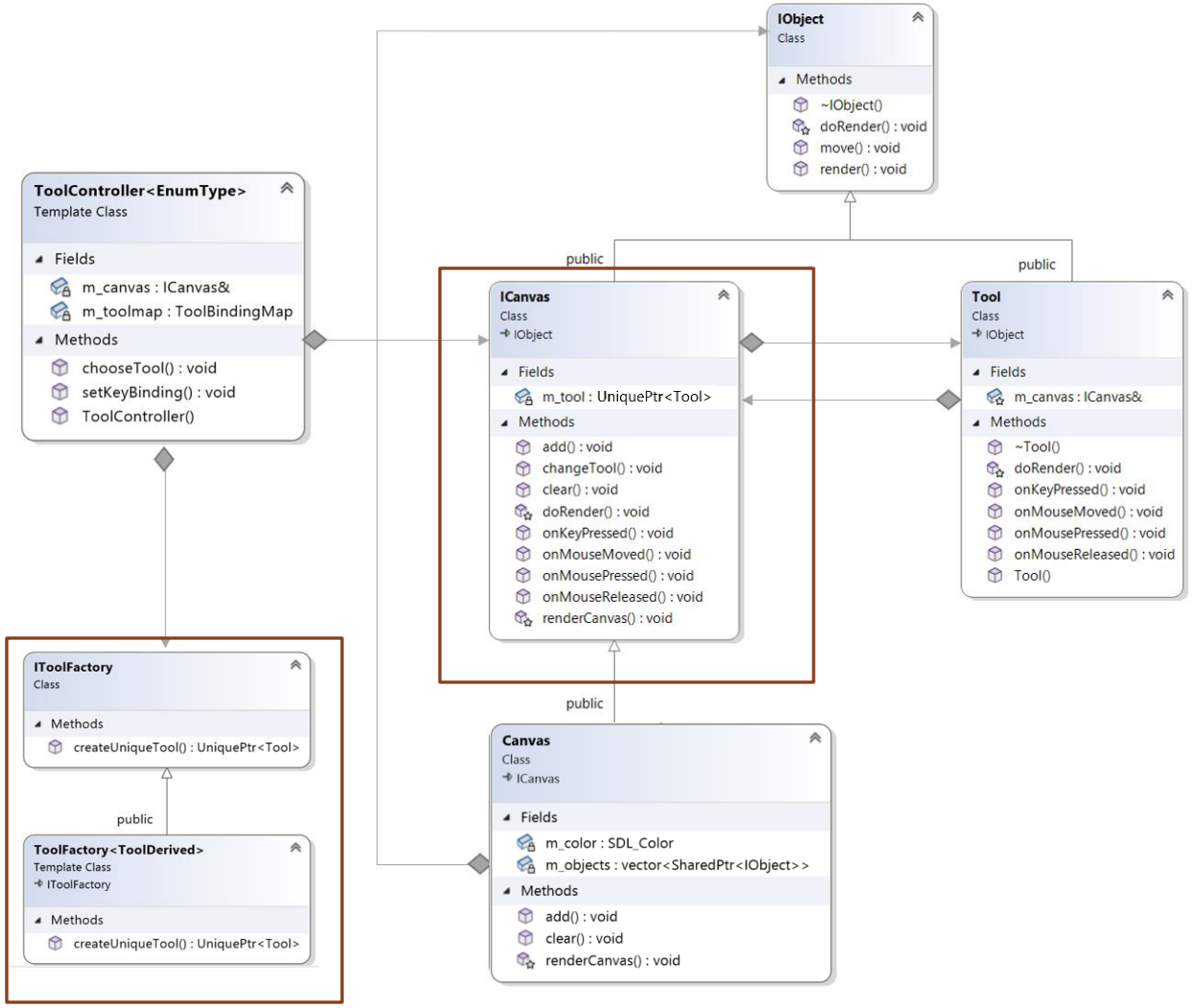
Демонстраційний проект

- Графічний редактор
- Використана графічна бібліотека SDL
- У проекті застосовані три типи інтелектуальних указників з різними стратегіями володіння та видалення



Одноосібний указник (UniquePtr)

- Зберігання класів з бібліотеки SDL: текстури, поверхні, шрифти тощо (власні стратегії видалення)
- Зберігання поточного інструменту у класі ICanvas
- Створення інструментів у фабриці



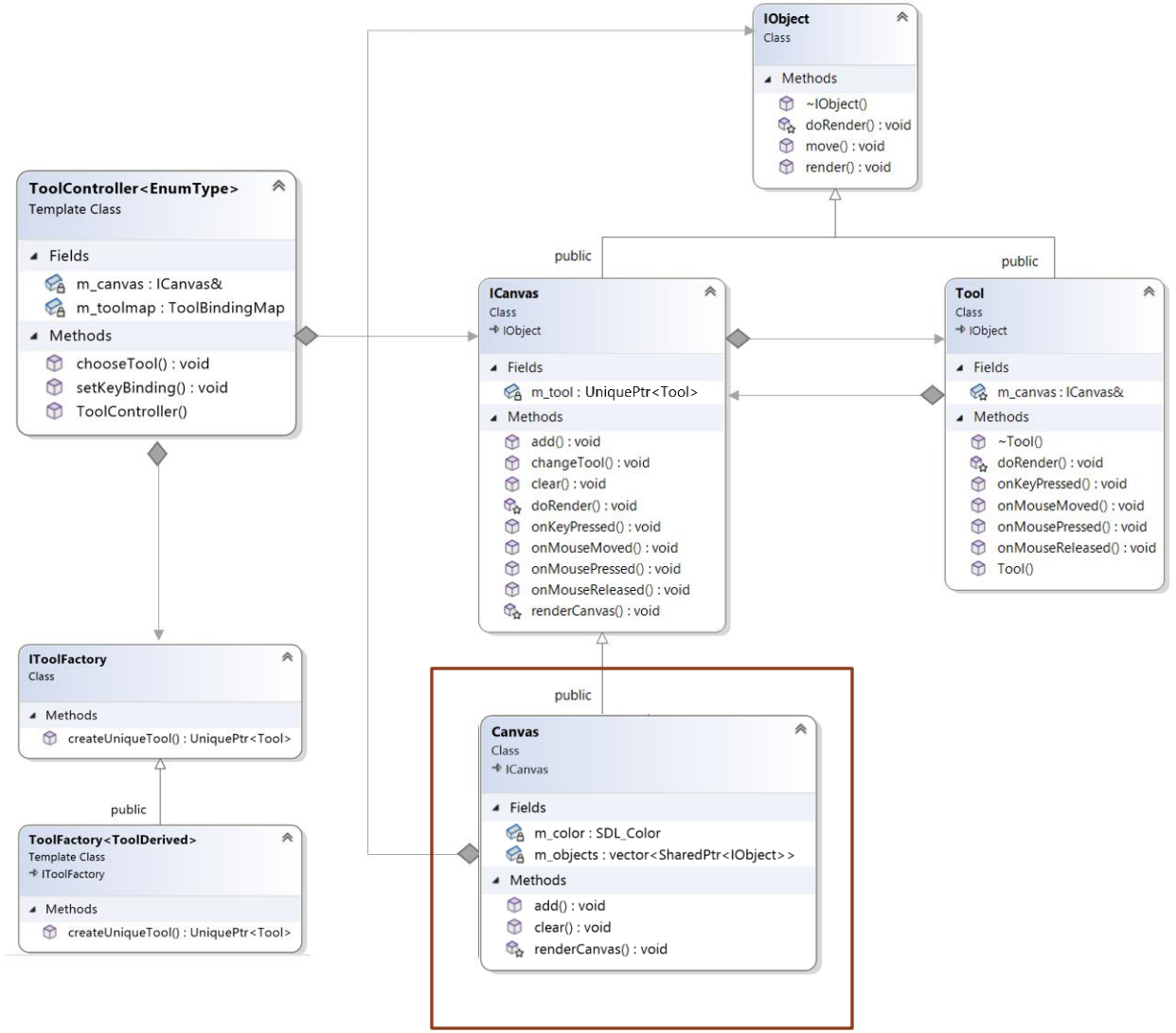
Стратегії видалення для UniquePtr

```
struct TextureDeleter {
	void operator()(SDL_Texture* texture) {
		if (texture) {
			SDL_DestroyTexture(texture);
		}
	}
};

using TextureUniquePtr =
	UniquePtr<SDL_Texture, TextureDeleter>;
```

```
class TextureWrapper {
public:
	TextureWrapper(SDL_Texture* texture = nullptr)
	: m_texture(texture) {}

	~TextureWrapper()
	{
		if (m_surface != nullptr) {
			SDL_DestroyTexture(m_surface);
		}
	}

	operator SDL_Texture*()
	{
		return m_surface;
	}

private:
	SDL_Texture* m_texture;
};
```

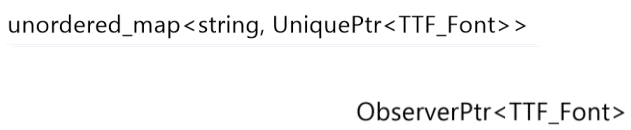
Розподілений указник (SharedPtr)

- Зберігання усіх графічних об’єктів



Невладний указник (ObserverPtr)

- Отримання указника TTF_Font з класу Font, який зберігає їх як мапу UniquePtr
- Передача об’єкта SDL_Renderer в метод render() класу IObject



Висновок

- Продемонстровано мотивацію для використання інтелектуальних указників
- Визначені стратегії інтелектуальних указників на основі C++20
  - Зберігання
  - Володіння
  - Перетворення
  - Перевірки
  - Видалення
- Стратегії застосовано до різних типів інтелектуальних указників
- Виконано рефакторинг стратегій Александреску до C++20

Дякую за увагу!

Буду радий відповісти на Ваші питання