Week 3 – Breaking dependencies

Advanced Programming Concepts

Dawid Zalewski, Robert de Groote

SAXION
UNIVERSITY OF
APPLIED SCIENCES

Advanced Programming Concepts

BREAKING DEPENDENCIES

SAXION UNIVERSITY OF APPLIED SCIENCES

Task

Design & program a simple logger class:

- Logs messages to an output stream
- By default, to `std::cout`
- Writes a timestamp with every message

SAXION UNIVERSITY OF APPLIED SCIENCES

The design

```
namespace lib {
  class logger {
  public:
    explicit logger(std::ostream& out);
    logger();
    void log(const std::string& msg) const;
  private:
    std::ostream& m_out;
  };
}
```

The implementation

```
logger::logger(std::ostream& m_out) : m_out(m_out) {}

logger::logger(): logger(std::cout) {}

void logger::log(const std::string& msg) const {
    auto time_point = std::time(nullptr);
    auto local_time = std::localtime(&time_point);
    char buffer[16];
    std::strftime(&buffer[0], sizeof(buffer),"%H:%M:%S.fff", local_time);
    m_out << '[' << buffer <<  "]: " << msg << '\n';
}
```

delegating constructor

SAXION UNIVERSITY OF APPLIED SCIENCES

Typical usage

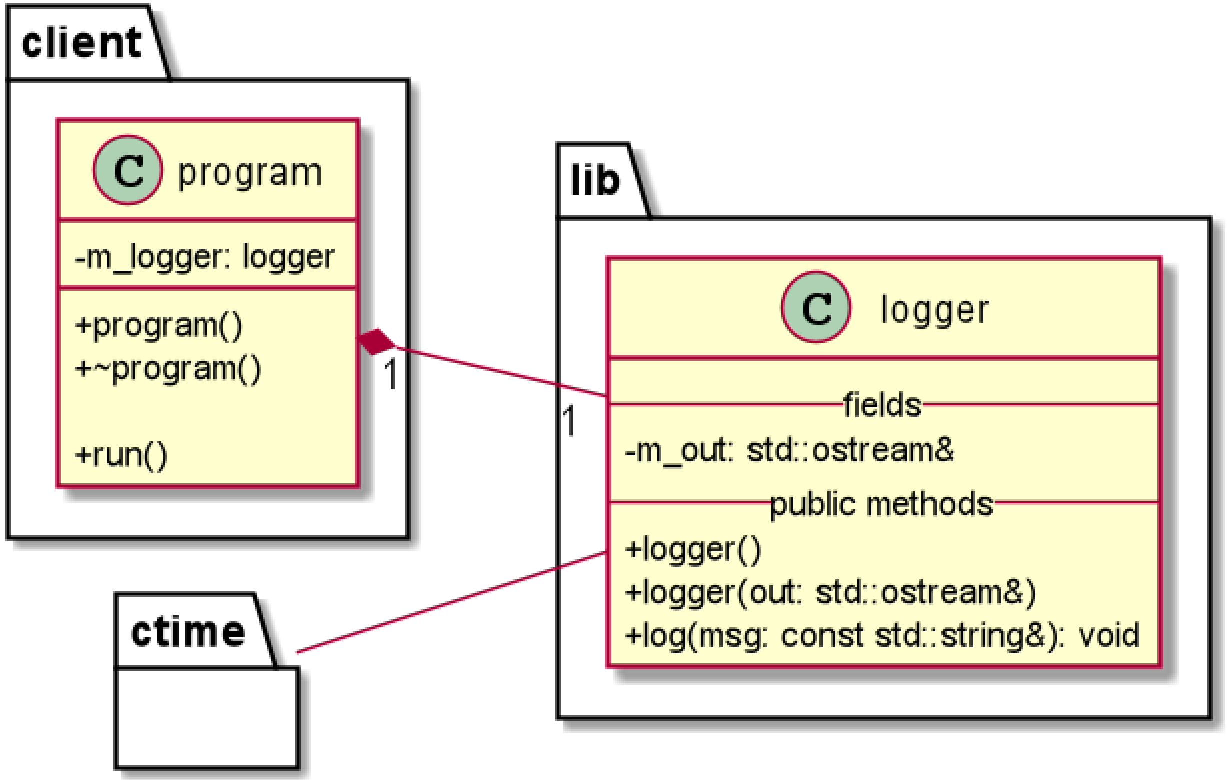
```
class program{
public:
  program(): m_logger{} { m_logger.log("Starting"); }
  ~program(){ m_logger.log("Quitting"); }

  void run(){
    using namespace std::literals;

    for(auto n{1}; n <= 5; ++n)
      m_logger.log("Running: "s + std::to_string(n++));
  }
private:
  lib::logger m_logger;
};
```

Typical usage

```
int main(){
  program prog{};
  prog.run();
}
```

```
> ./prog
[15:23:24]: Starting
[15:23:24]: Running: 1
[15:23:24]: Running: 2
[15:23:24]: Running: 3
[15:23:24]: Running: 4
[15:23:24]: Running: 5
[15:23:24]: Quitting
```

The design in UML



What can be improved?

SAXION UNIVERSITY OF APPLIED SCIENCES

The obvious idea…

```
logger::logger(std::ostream& m_out) : m_out(m_out) {}

logger::logger(): logger(std::cout) {}

void logger::log(const std::string& msg) const {
    auto time_point = std::time(nullptr);
    auto local_time = std::localtime(&time_point);
    char buffer[16];
    std::strftime(&buffer[0], sizeof(buffer),"%H:%M:%S.fff", local_time);
    m_out << '[' << buffer <<  "]: " << msg << '\n';
}
```

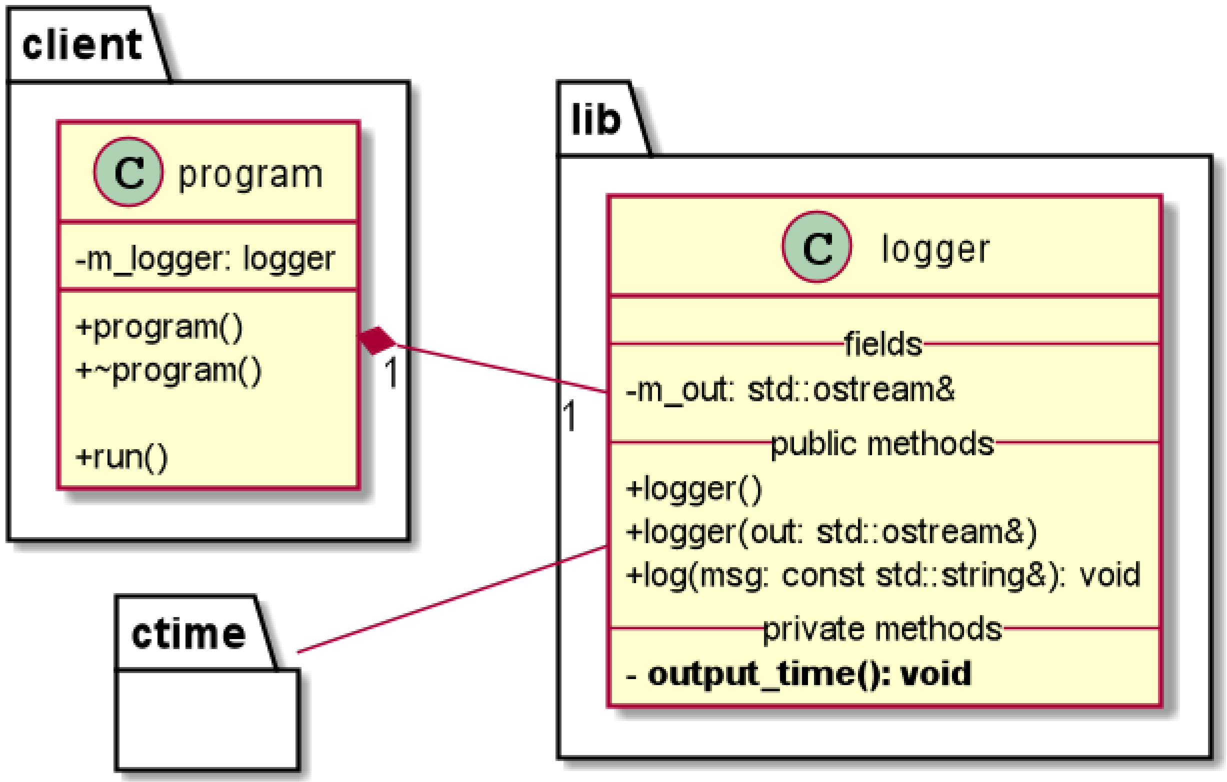
Extract

The obvious idea…

Extract timestamping to another function!

```
namespace lib{
  class logger{
  public:
    logger(std::ostream& out);
    logger();
    void log(const std::string& msg) const;
  private:
    std::ostream& m_out;
    void output_time() const;
  };
}
```

The obvious idea in UML



What can (still) be improved?

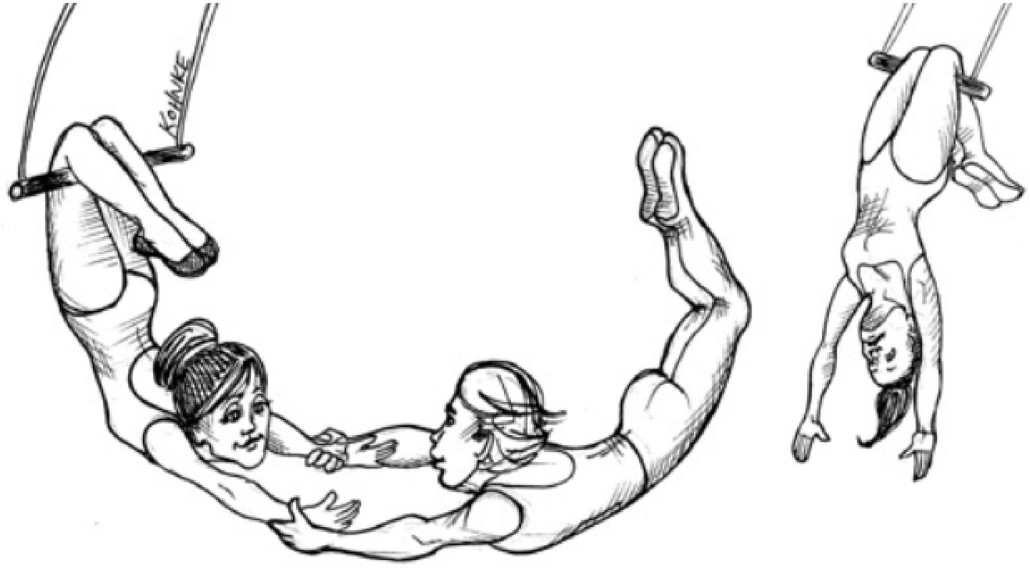
SAXION UNIVERSITY OF APPLIED SCIENCES

What will go wrong?

Very disappointed colleagues

11

DIP: The Dependency Inversion Principle



The Dependency Inversion Principle (DIP) tells us that the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.

From *Clean Architecture* by Robert C. Martin

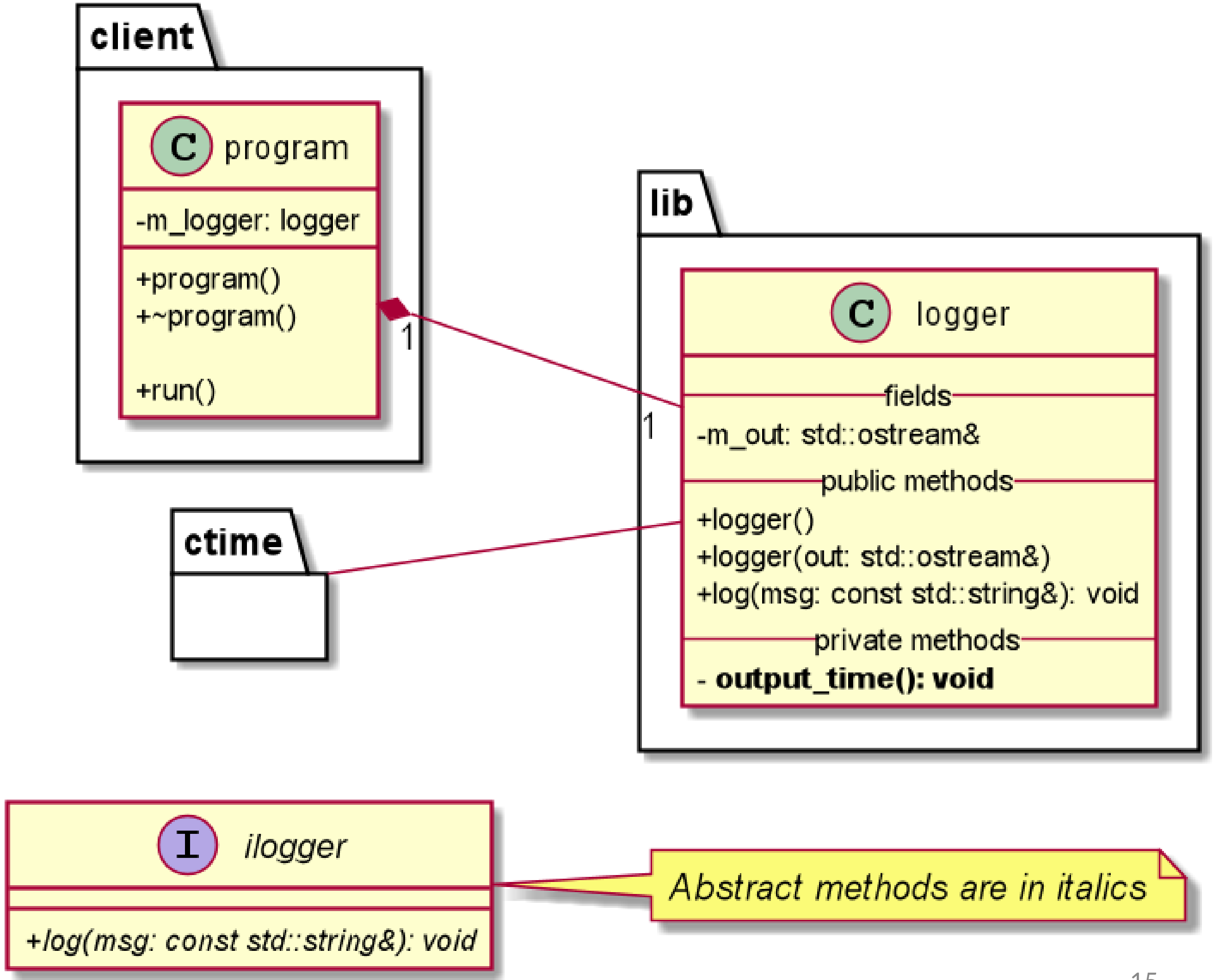
Dependency Inversion Principle

- **Don’t refer to volatile concrete classes**.
  Refer to ***abstract interfaces*** instead.
- Don’t derive from volatile concrete classes. This is a corollary to the previous rule(…)
- Don’t override concrete functions.
- Never mention the name of anything concrete and volatile. This is really just a restatement of the principle itself.
- ***Abstract interfaces*** **are stable.**

From *Clean Architecture* by Robert C. Martin

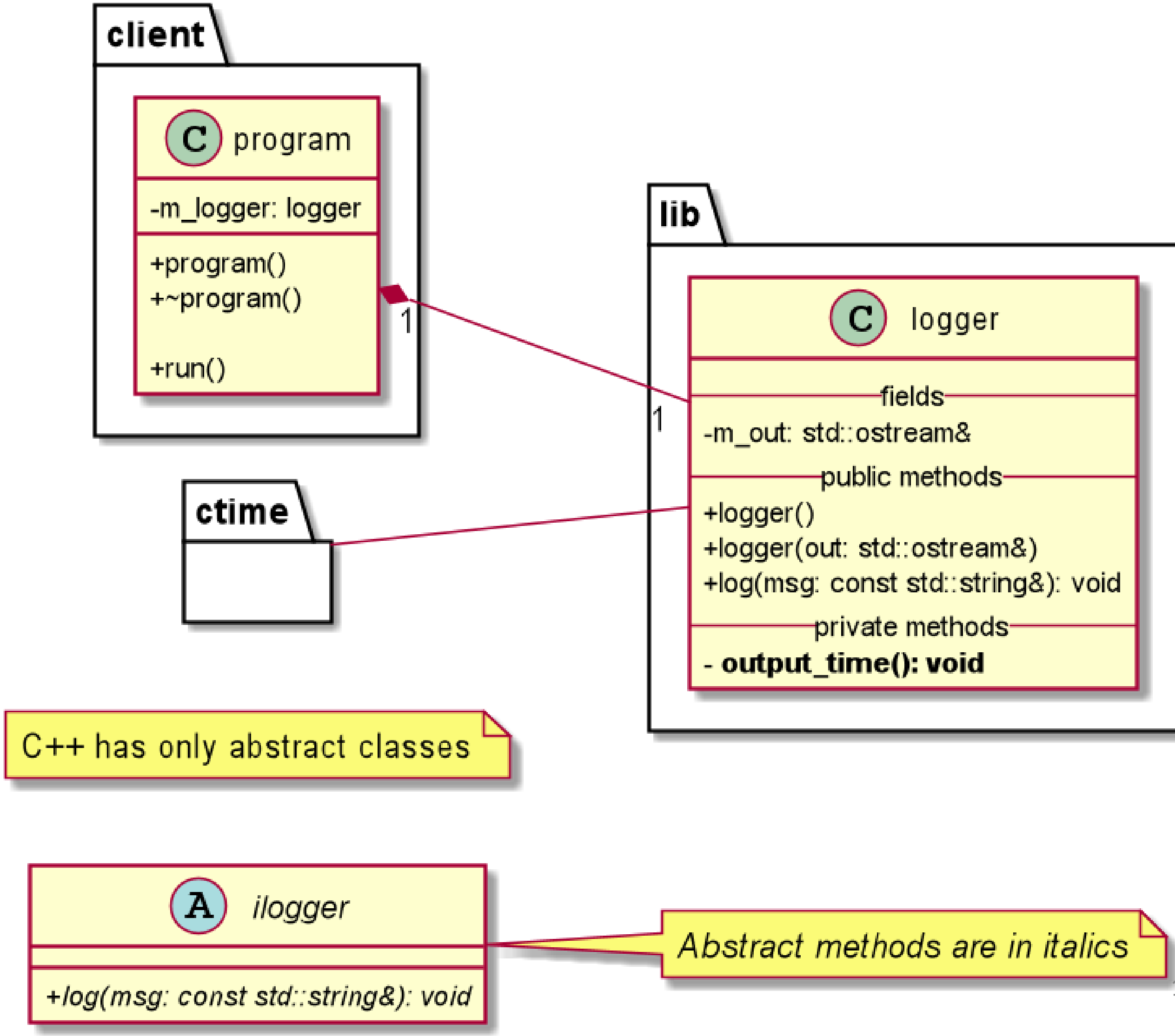
Dependency Inversion Principle

We need an **abstract interface**



Dependency Inversion Principle

We need an **abstract** ~~interface~~ **class**

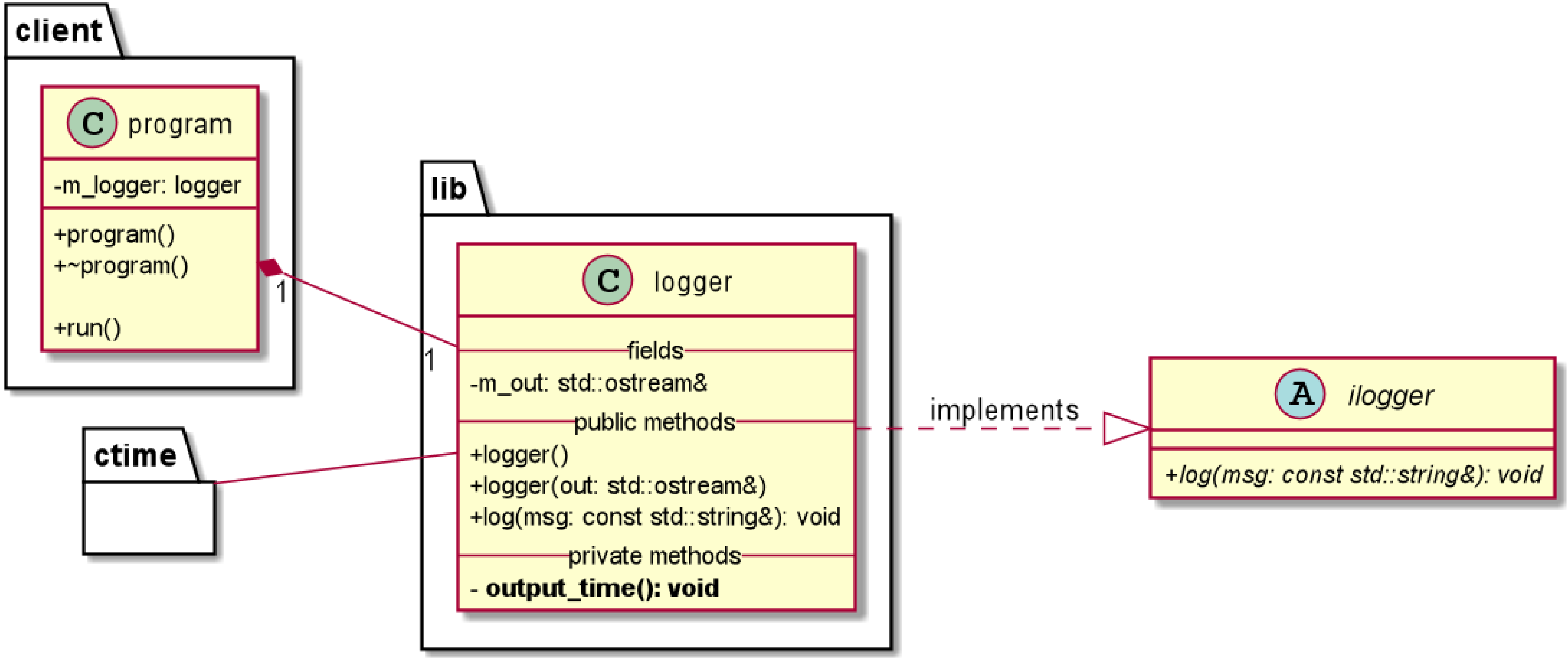


Dependency Inversion Principle

```
class ilogger {
public:
  virtual void log(const std::string& msg) const = 0;
};
```

- **`ilogger`**`::log` is an ***abstract*** method (***pure virtual*** in C++ lingo)
- Classes with at least one ***pure virtual*** method are ***abstract***
- All the **`ilogger`** methods are ***pure virtual*** ⇨ **`ilogger`** is effectively an ***interface***

Implementing an interface



Implementing an interface == inheriting from an abstract class

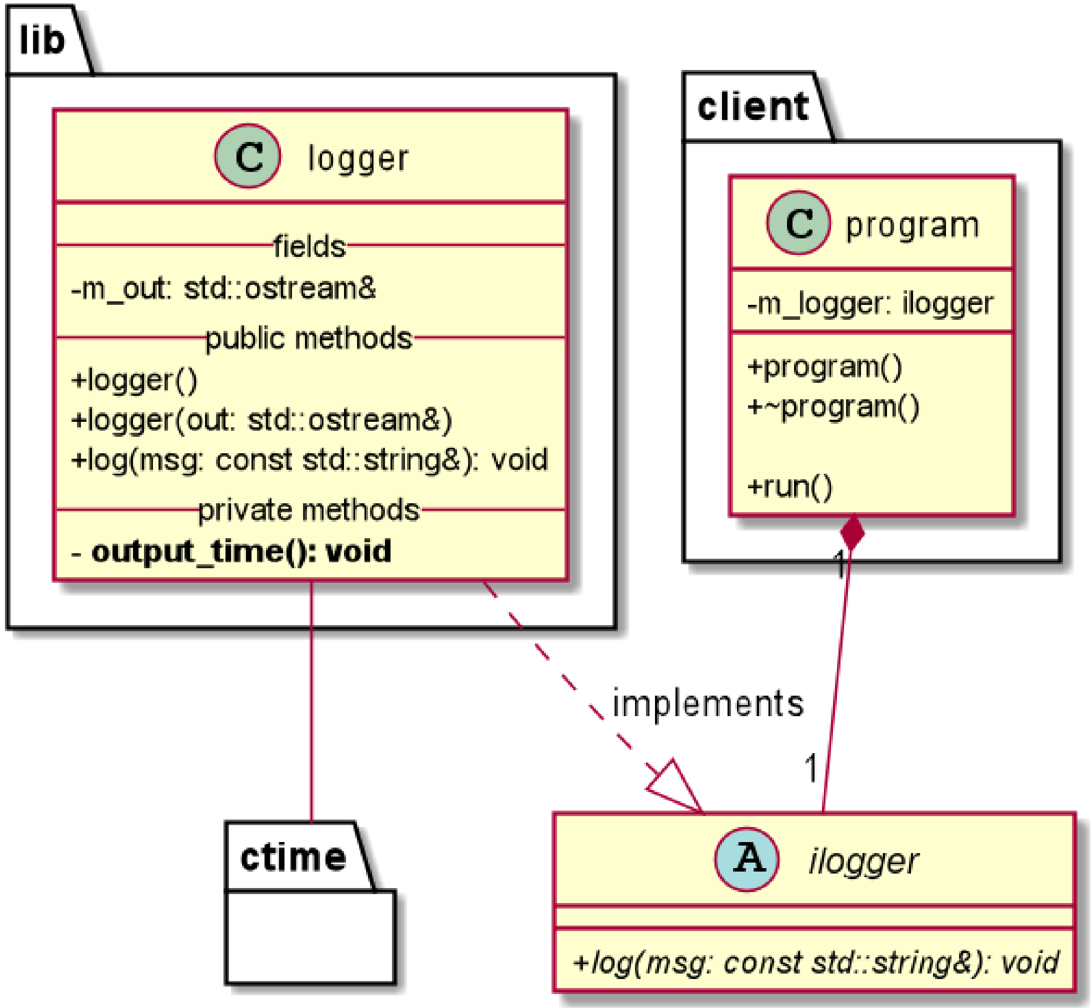
```
namespace lib {
  class logger : public ilogger {
  public:
    logger(std::ostream& out);
    logger();

    void log(const std::string& msg) const override;
  private:
    std::ostream& m_out;
    void output_time() const;
  };
}
```

logger inherits from ilogger

Communicates the intent of re-implementing ilogger::log

Breaking the dependency on the concrete class

- `program` depends only on `ilogger` interface
- `logger` provides the implementation



Abstract classes cannot be instantiated...

```
class program{
public:
  program():
    m_logger{}
  {
      m_logger.log("Starting");
  }

  /* ~~~ */

private:
  ilogger m_logger;
};      ^
      error: field type 'ilogger' is an abstract class
```

Abstract classes cannot be instantiated…

```
int main(){
  ilogger m_logger;                          ❌
  ilogger m_logger =  lib::logger{};         ❌
  ilogger m_logger = new  lib::logger{};     ❌
}
```

Abstract classes cannot be instantiated…

```
int main(){

  ilogger* m_logger = new  lib::logger{}; ✓

}
```

Pointers to *abstract classes* can be used to refer to the child classes.

Using abstract classes (dynamic polymorphism)

```
class program {
public:
  program():
    m_logger{ new lib::logger }
  {
      m_logger->log("Starting");
  }
  /* ~~~ */
  ~program(){
    m_logger->log("Quitting");
    delete m_logger;
  }
private:
  ilogger* m_logger;
};
```

m_logger *is initialized with a (pointer to)* **lib**::logger *object*

program *owns* m_logger *and needs to dispose of it*

Dynamic polymorphism and object destruction

```
int main() {

  ilogger* m_logger = new lib::logger{};

  delete m_logger;
}
```

There is a problem:

- `delete` calls the destructor on an `ilogger` object
- But `m_logger` is not an `ilogger` object (it points to `lib::logger`)
- This won't work 😭
- … unless we help the runtime a bit

C++ inheritance rule #0

A class with a *virtual function* must provide a *virtual destructor*.

```
class ilogger {
public:
  virtual void log(const std::string& msg) const = 0;

  virtual ~ilogger() = default;
};
```

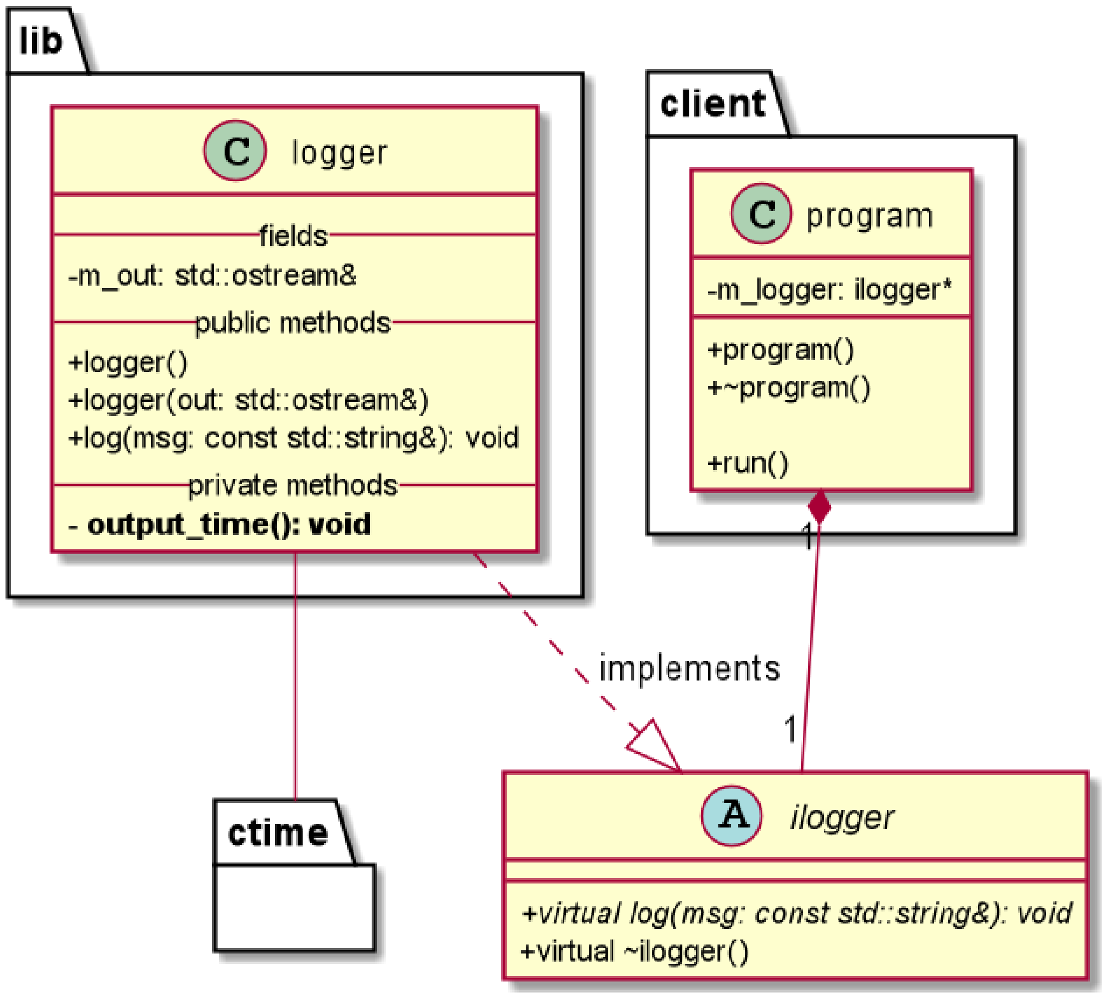
We could also do:

```
virtual ~ilogger() {}
```

But since it's trivial it's better to `default` *it.*

SAXION UNIVERSITY OF APPLIED SCIENCES

Breaking dependencies: the Dependency Inversion Principle



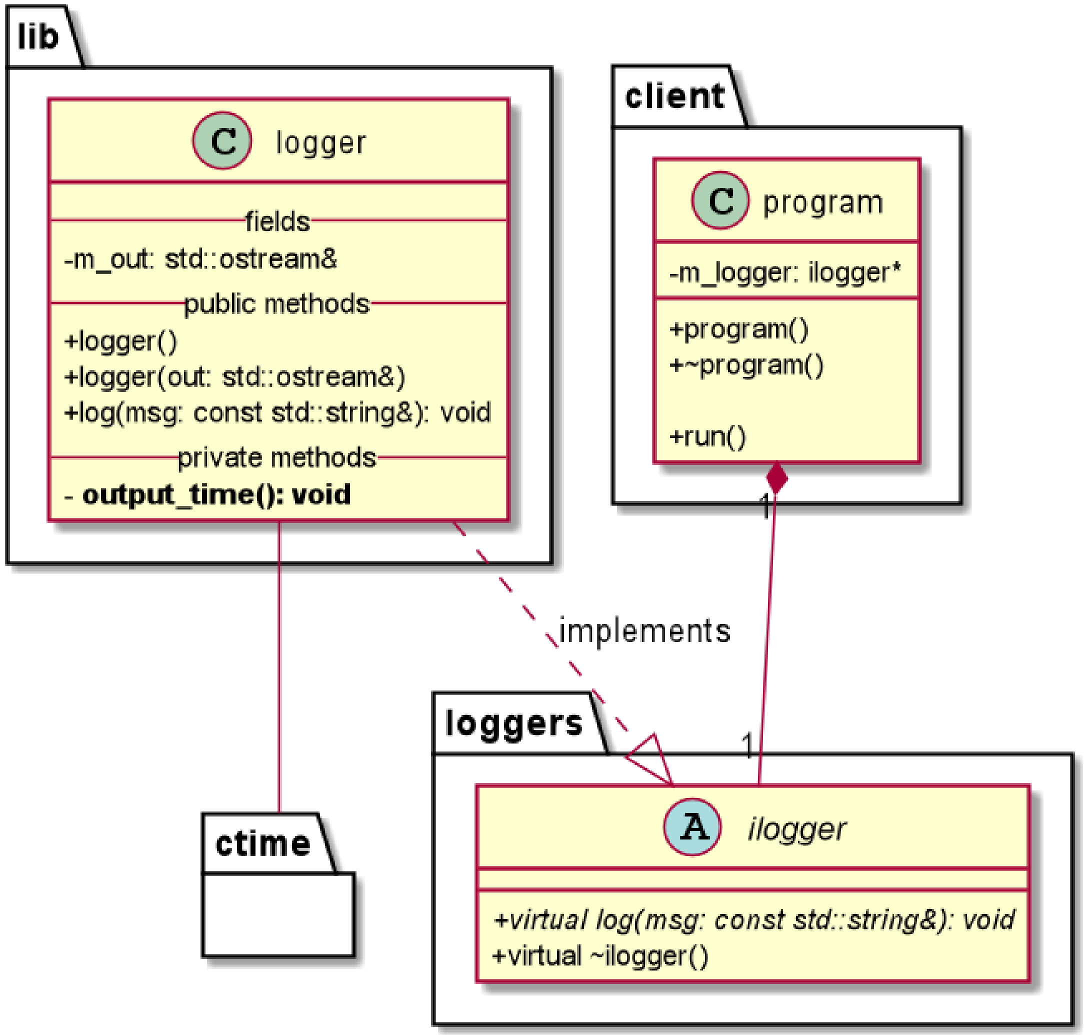
Side note: `logger` doesn't have a destructor defined. A compiler will generate one and it will be also `virtual`.

What about namespaces / modules

Q: Shouldn't `ilogger` reside in a namespace?

A: Yes, but not the one where its implementation lives.

Breaking dependencies: the Dependency Inversion Principle



Dependency Inversion Principle

- **Don't refer to volatile concrete classes.**
  Refer to ***abstract interfaces*** instead.

- Never mention the name of anything concrete and volatile. This is really just a restatement of the principle itself.

- Program to interfaces not to implementations.

- ***Interfaces*** **should be stable.**

From *Clean Architecture*
by Robert C. Martin

Advanced Programming Concepts

INJECTING DEPENDENCIES

Dependency Inversion Principle

- **Don't refer to volatile concrete classes.**
  Refer to ***abstract interfaces*** instead.
- Never mention the name of anything concrete and volatile. This is really just a restatement of the principle itself.
- Program to interfaces not to implementations.
- *Interfaces* should be stable.

From *Clean Architecture*
by Robert C. Martin

SAXION UNIVERSITY OF APPLIED SCIENCES

Off with dependencies

```
class program{
public:

  program():
    m_logger{ new lib::logger }
  {
    m_logger->log("Starting");
  }

  /* ~~~ */

private:
  loggers::ilogger* m_logger;
};
```

*Precisely what we are doing here: mentioning a name of **something concrete***

Off with dependencies

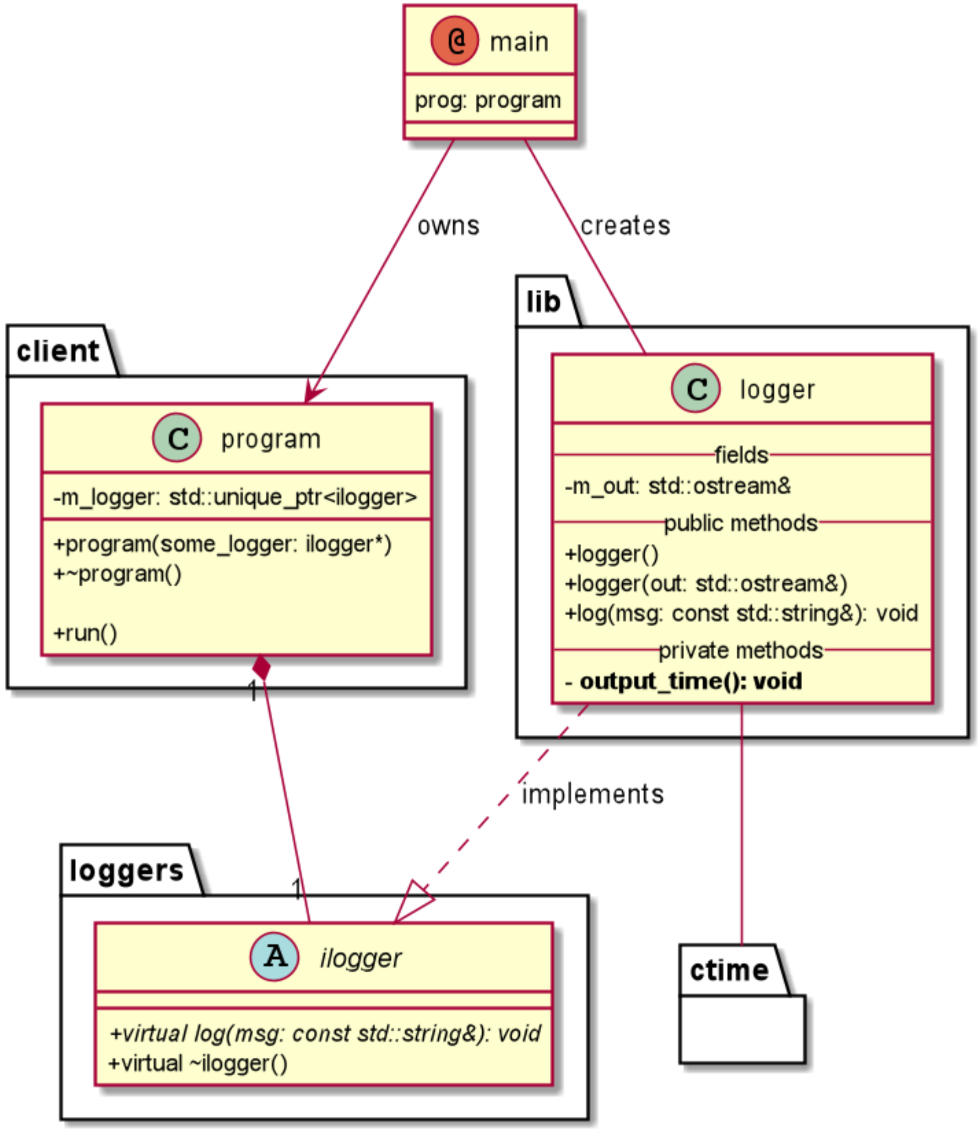
```
class program{
public:
  program(loggers::ilogger* some_logger):
    m_logger{ some_logger }
  {
    m_logger->log("Starting");
  }
  /* ~~~ */
};

int main(){
  auto log{ new lib::logger{} };
  program prog{ log };
  prog.run();
}
```

Constructor takes an argument through an interface pointer

Concrete object created…

…and passed to a constructor

Dependency injection in UML



SAXION UNIVERSITY OF APPLIED SCIENCES

Dependency injection 101

Dependency injection:

- Concrete objects are created outside objects that use them
- Concrete objects are passed to objects that use them

Dependency injection through interfaces

- Standard dependency injection and:
  - Concrete objects implement abstract interfaces
  - Objects that use concrete objects refer to them through those interfaces

Resetting dependencies (setting later)

```
class program{
public:
  program(loggers::ilogger* some_logger):
    m_logger{ some_logger }
  { }

  void set_logger(loggers::ilogger* some_logger){
    delete m_logger;
    m_logger = some_logger;
  }

  ~program(){ delete m_logger; }

private:
  loggers::ilogger* m_logger;
};
```

Dependency injection 102

A dependency can be injected

- In a constructor
- Through a function (setter)

The latter allows for changing a dependency dynamically.

Dependency injection: ownership transfer (ambiguous)

```
int main(){
  auto log{ new lib::logger{} };
  program prog{ log };
  prog.run();
}
```

Transfer of ownership?

```
class program{
public:
  program(loggers::ilogger* some_logger):
    m_logger{ some_logger }
  { }

  ~program(){
    delete m_logger;
  }

private:
   loggers:: ilogger* m_logger;
};
```

`program` *owns* `m_logger` *and must dispose of it**

**The same is achieved with* `m_logger` *as* `std::unique_ptr<ilogger>`

Dependency injection: no ownership transfer (ambiguous)

```
int main(){
  auto log{ new lib::logger{} };
  program prog{ log };
  prog.run();
  delete log;
}
```

No ownership transfer?

```
class program{
public:
  program(loggers::ilogger* some_logger):
    m_logger{ some_logger }
  { }

  ~program(){

  }

private:
   loggers::ilogger* m_logger;
};
```

`main` *owns*
`log` *and must dispose of it*

Dependency injection 103

- Dependencies are usually injected through pointers
- Ownership can be unclear
- Document the ownership

- Or use the language to resolve ambiguities

Interlude end

WHAT DOES IT ALL MEAN FOR US?

Modern dependency injection ownership semantics

Rules:

1. Pass a **raw pointer** for **no ownership** transfer

   (unless you can use a reference instead)
2. Pass a **smart pointer** for **ownership** transfer

Modern dependency injection: 1. No transfer of ownership

```
class program{
public:
  program(loggers::ilogger* some_logger):
    m_logger{ some_logger };

  void set_logger(loggers::ilogger* some_logger){
    m_logger = some_logger;
  }

  ~program(){ }

private:
  loggers:: ilogger* m_logger;
};
```

Passing by pointer == no ownership transfer

No ownership == no cleanup

SAXION UNIVERSITY OF APPLIED SCIENCES

Modern dependency injection: 2. Transfer of ownership

```
class program{
public:
  program(std::unique_ptr<loggers::ilogger> some_logger):
    m_logger{ std::move(some_logger) } {};

  void set_logger(std::unique_ptr<loggers::ilogger> some_logger){
    m_logger = std::move(some_logger);
  }

private:
  std::unique_ptr<loggers::ilogger> m_logger;
};
```

- `std::unique_ptr` *is passed by value*
- *Ownership transfer with* `std::move`

Modern dependency injection ownership semantics

```
int main(){
    auto log = std::make_unique<lib::logger>();
```

```
program prog{ log.get() };
```

main *still owns* log

```
program prog{ std::move(log) };
```

main *transfers the ownership of* log

```
    prog.run();
}
```

Dependency inversion & injecting dependencies

- Never mention the name of anything concrete and volatile.
- Construct concrete objects in one place.
- Build dependencies by **injecting** them.
- Inject using interfaces (**Dependency Inversion Principle**)
- Use **clear ownership semantics** and document it.