

Flip Model: a Design Pattern

Daniele Pallastrelli

C++ Day 2019
November 30, Parma

1994

computer Programming

Il primo mensile di programmazione

Spazio Q&A

- Costruttori e Thread in C++
- Tipo logico e fisico in C++
- Passare dal Turbo C++ 3.0 al C++ Builder

Recensioni Libri

- Inside Windows NT H. Custer
- Advanced Windows NT J. Ritcher
- Cross Platform W. Murray III, C. Pappa
- Windows NT Network Programming R. Davis

INTERFACCIA UTENTE

Installazione, utilizzo e funzionalità. Creare Toolbar e MFC. Coolbar e gli altri controlli di Memphis in 3. Progettare una GUI a oggetti.

Il linguaggio APL2
Programmare ad oggetti in Perl
Il Sistema Operativo Linux

Pattern in Java: Observer.
Sviluppo a SmallTalk: l'ambiente di programmazione

ASP per connettere Web Server e DBMS

Shell, hook e ToolHelp32 per monitorare il sistema

Le strutture dati Self-Organizing

Visual Basic

Controlli Data-aware con VB 5

Java

Utilizzo del JDBC per connettersi

Intervista a:
Bertrand Meyer

Autore del Design by Contract
e progettista di Eiffel

SOFTWARE E DIRITTO: L'ENTRATA IN VIGORE DELLA LEGGE SUI DATI PERSONALI ED IL DECRETO SULLE TARIFFE AGEVOLATE PER GLI UTENTI INTERNET

computer Programming

LUGLIO/AGOSTO 1997 N. 60

Il primo mensile di programmazione

GRAFICA E MULTIMEDIA

Grafica tridimensionale ed effetti speciali con OpenGL. Grafica veloce in Windows con DirectDraw. Un registratore di file WAV in Delphi. Le funzioni e le strutture dati dell'interfaccia MCI. Riconoscimento vocale nelle applicazioni. Audio e video su WEB.

C++

Gestire il ciclo di vita degli oggetti nella programmazione OO

Windows

Funzioni DrawDib per la grafica.
Realizzazione di una PrintScreen

Visual Basic

Creare controlli
X in VB5

Intervista a:
A. Stepanov
Uno degli autori di STL, la
libreria di template
del C++

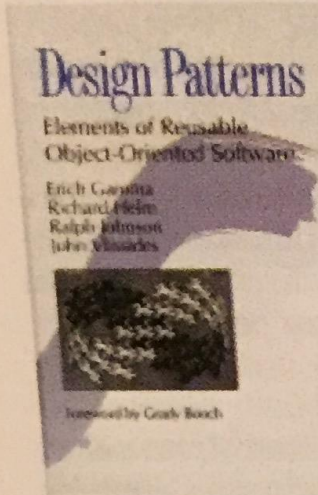
Recensioni Libri

- Design Patterns Gamma, Helm, et al.
- Pattern Languages of Program Design Coplien, Schmidt
- Pattern Languages of Program Design 2 Coplien, Schmidt
- Advanced C++, Programming and Idioms Coplien

Una Biblioteca di Pattern

di Graziano Lo Russo

Una definizione dei pattern è "un genere di letteratura". Per diventare esperti nell'uso dei pattern è essenziale munirsi di una buona biblioteca. Darò quindi alcuni suggerimenti bibliografici sui pattern



Titolo	Design Patterns
Autore	Gamma, Helm, Johnson, Vlissides
Editore	Addison Wesley
ISBN	0-201-63361-2
Pagine	395
Prezzo	109.000

Sulle pagine di Computer Programming mi sono occupato di Pattern. Nonostante l'argomento ci ha tenuto occupati per un po', non è stato possibile esaminare l'intera popolazione dei pattern. Per gli altri non mi resta che consigliarvi una buona bibliografia.

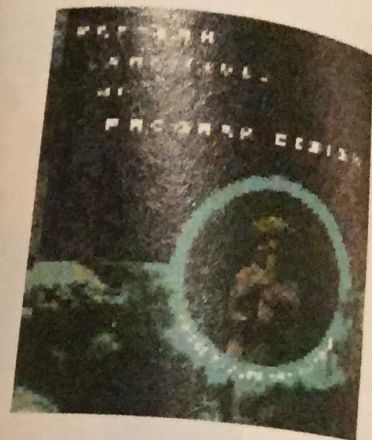
Testi di Base

Da dove incominciare lo studio dei Pattern? A costo di essere impopolare, sconsiglio di iniziare da *Design Patterns*. I motivi li esporrò fra breve, quando parlerò di questo libro.

Per iniziare la conoscenza dei Pattern consiglio di cominciare da *Patterns* di Coplien. Si tratta di un libro sottile, dal tono discorsivo, che si legge in un paio d'ore. Il prezzo è relativamente alto per il numero di pagine, ma è proporzionale alla ricchezza del contenuto. **Patterns** appartiene alla serie SIGS Management Briefings ed è davvero un libro per Manager nel migliore senso della parola: breve, facile da capire, trascura gli aspetti tecnicistici per convogliare in modo efficace il senso dell'approccio a pattern. Coplien dimostra con questo libro di essere un ottimo divulgatore oltre che quell'esperto mondiale di C++ che tutti conoscono. Sono esaminate tutte le specie di pattern, dai pattern di architettura di Alexander ai pattern del software fino ai pattern di organizzazione (di cui Coplien è esperto). *Patterns*, da cui ho attinto abbondanti citazioni, è stato il modello a cui mi sono ispirato per scrivere gli articoli sui pattern.

Pattern di Design

Design Patterns è una lettura indispensabile per chiunque voglia imparare a conoscere i pattern, anzi ormai è indispensabile per chiunque ambisca a diventare (o ad essere) un progettista C++ serio. In *Design Patterns* si trovano raccolti e spiegati magnificamente quasi tutti i pattern più importanti e di uso comune nel progetto del software. Singleton, Abstract Factory, Chain of Responsibility, Command, Iterator, Observer, Visitor, Adapter, Template Method, Composite, Bridge: sono solo alcuni dei 23 pattern di design raccontati dai quattro autori (Gamma, Helm, Johnson, Vlissides). *Design Patterns* è stato il libro che ha veramente diffuso i pattern nel mondo dell'OOD (Object Oriented Design). Quando venne presentato al primo convegno PLOP, ottenne il più alto numero di vendite mai raggiunto da un titolo Addison Wesley in un convegno specialistico. *Design Patterns* ha stabilito quelli che sono diventati gli standard nei pattern di design. La distinzione dei pattern in tre categorie: creazionali, strutturali e comportamentali è stata più volte criticata perché semplicistica ma nessuno ne ha saputo finora proporre una migliore. L'esposizione dei pattern in *Design Patterns* è diventata canonica: per ogni pattern vengono elencati, nell'ordine: il nome; la categoria cui appartiene; l'intento (problema); gli alias (Also Known As); le motivazioni (contesto); l'applicabilità; la struttura, sotto forma di diagrammi OMT; i partecipanti, cioè gli oggetti che prendono parte e i rispettivi ruoli; la collaborazione - con oggetti che non compongono il pattern stesso; le conseguenze; le [alcune delle] implementazioni possibili; esempi; pattern correlati; usi conosciuti. La parte più lunga è l'implementazione, che spesso è impostata come un pattern language in miniatura: viene esposta una implementazione, se ne esaminano i punti deboli, si espone un'altra implementazione (più complicata) che li risolve, si esaminano i punti deboli della seconda, eccetera. Nella parte di implementazione si trovano osservazioni acute sui vantaggi e svantaggi delle alternative; spesso le implementazioni usano tecniche di C++ evolute: per completezza.



di Kerth sul passaggio dall'OOD al pattern language e ad alta affidabilità, il pattern language



Program Design è un ottimo modo di

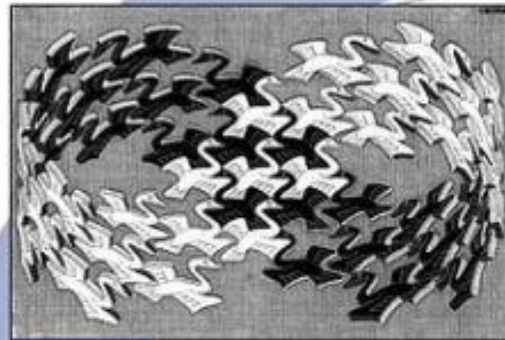
Pattern di Architettura

Il libro **Pattern di Architettura** *Pattern-Oriented Software Architecture* presenta pattern di tutti i generi esposti in *Patterns, a System*.

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Condon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



“ A **design pattern** is a general repeatable solution to a commonly occurring problem in software design ”

“ A design pattern is a **general repeatable solution** to a commonly occurring problem in software design ”

“ A design pattern is a general repeatable solution to a **commonly occurring problem** in software design ”

“ A design pattern is a general repeatable solution to a commonly occurring problem in **software design** ”

Pattern basic structure

Pattern name and classification

Intent

Also known as

Motivation (forces)

Applicability

Structure

Participants

Collaborations

Consequences

Implementation

Sample code

Known uses

Related patterns

Pattern basic structure

Pattern name

Intent

Also known as

Motivation (for)

Applicability

Structure

Participants

Operations

Sequences

Implementation

Code

Uses

Patterns

Basic Elements

Name

Problem

Solution

Consequences

Name: **Flip Model**

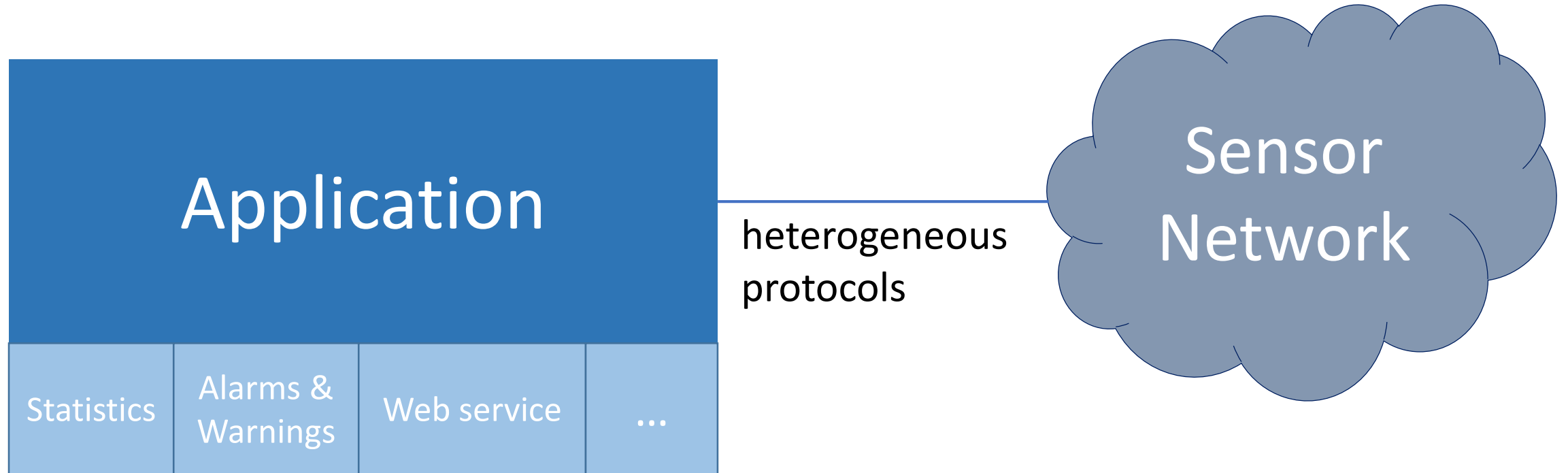
Model publisher

Aka: **Pressman**
Newsagent

Classification: **Behavioral**

The pattern allows multiple clients to read a complex data model that is continuously updated by a unique producer, in a thread-safe fashion.

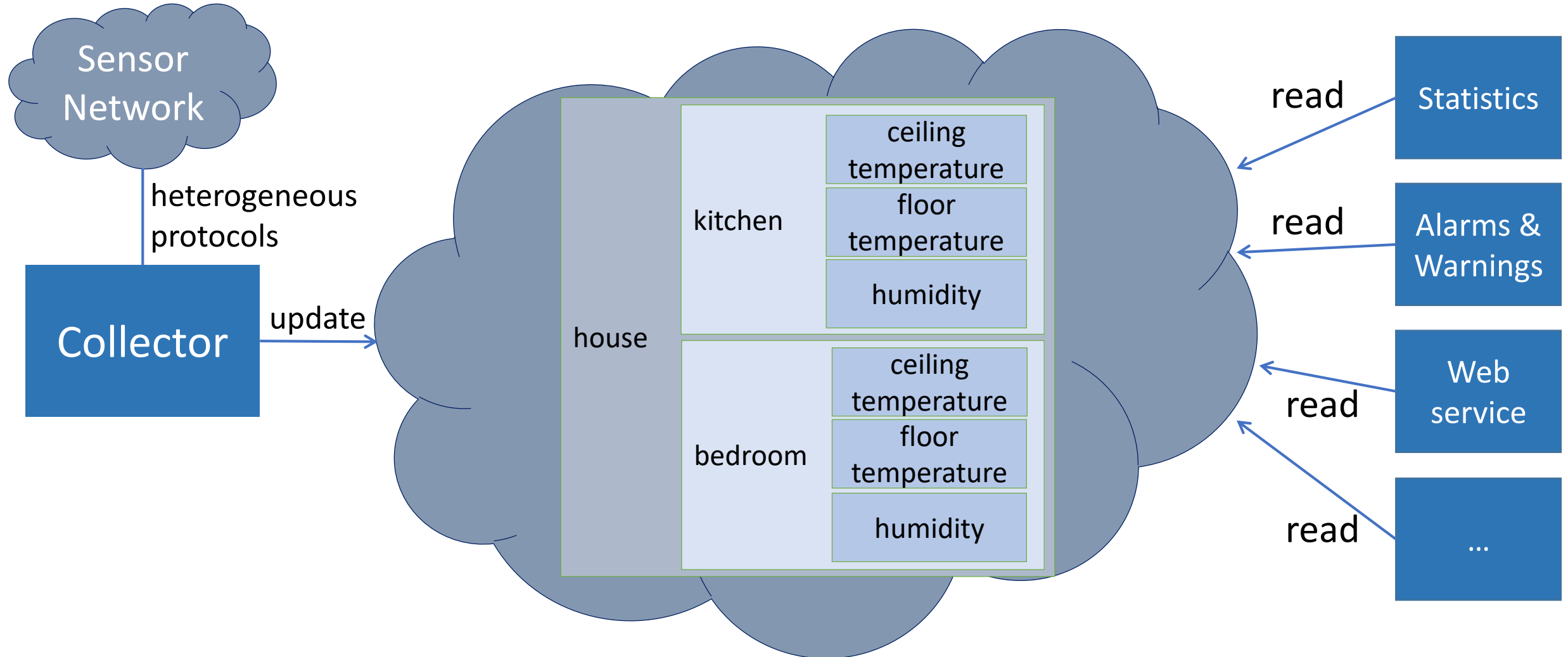
Motivation/Forces (i.e., the Problem)



The problem



The problem



Relevant issues

How can all the modules of the application work together on the same data structure?

Relevant issues

How can all the modules of the application work together **on the same** data structure?

How can all the clients use **the most updated data** available in a consistent fashion?

Relevant issues

How can all the modules of the application work together **on the same** data structure?

How can all the clients use **the most updated data** available in a consistent fashion?

How can the application **get rid of the old data** when it is no longer needed?

```

void SensorNetwork::OnTimerExpired()
{
    filling = make_shared<SensorAcquisition>();
    filling->Scan( // start an async operation
        [this]() { OnScanCompleted(); }
    );
}

```

```

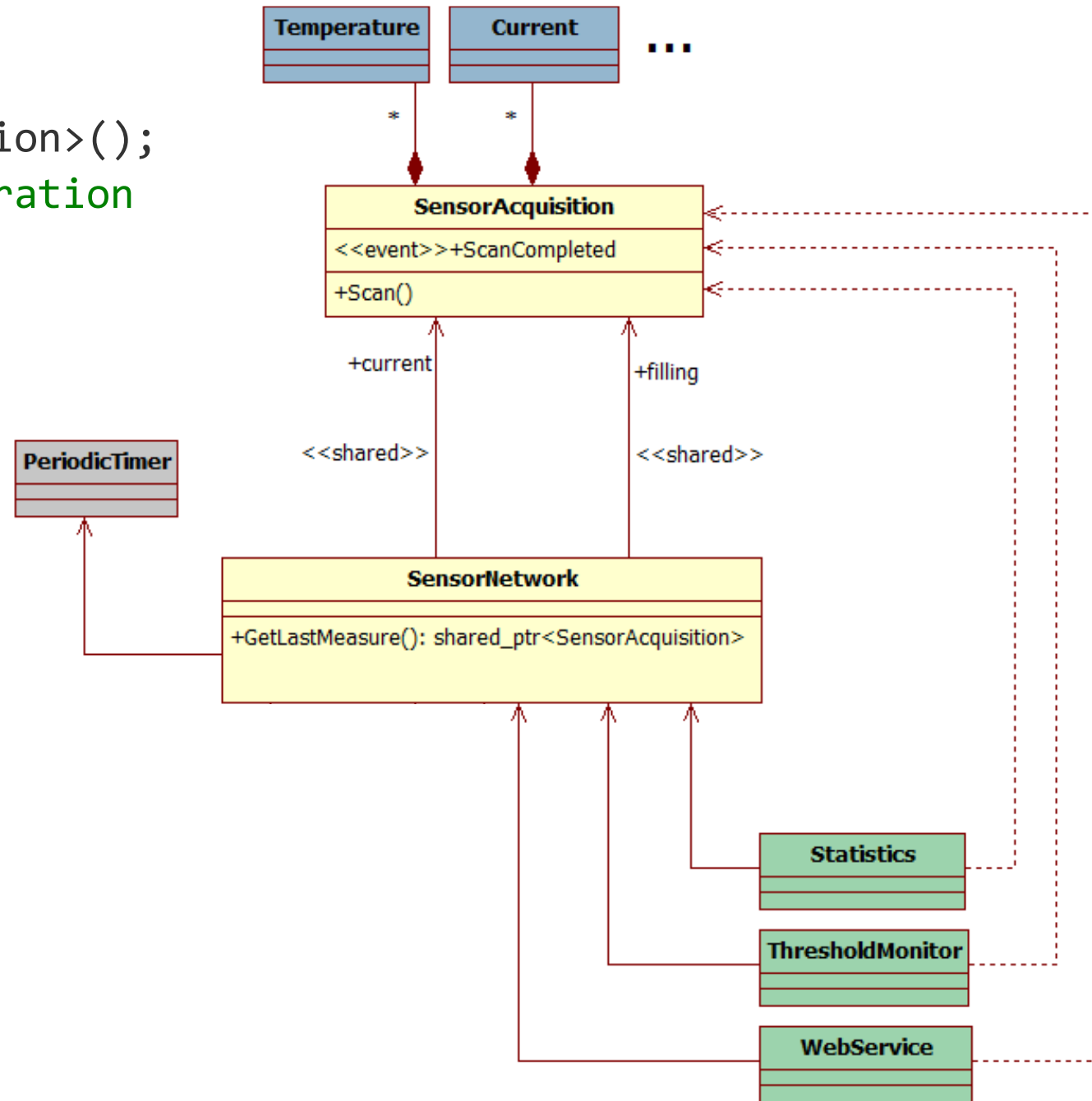
void SensorNetwork::OnScanCompleted()
{
    scoped_lock<mutex> lock(mtx);
    current = filling;
}

```

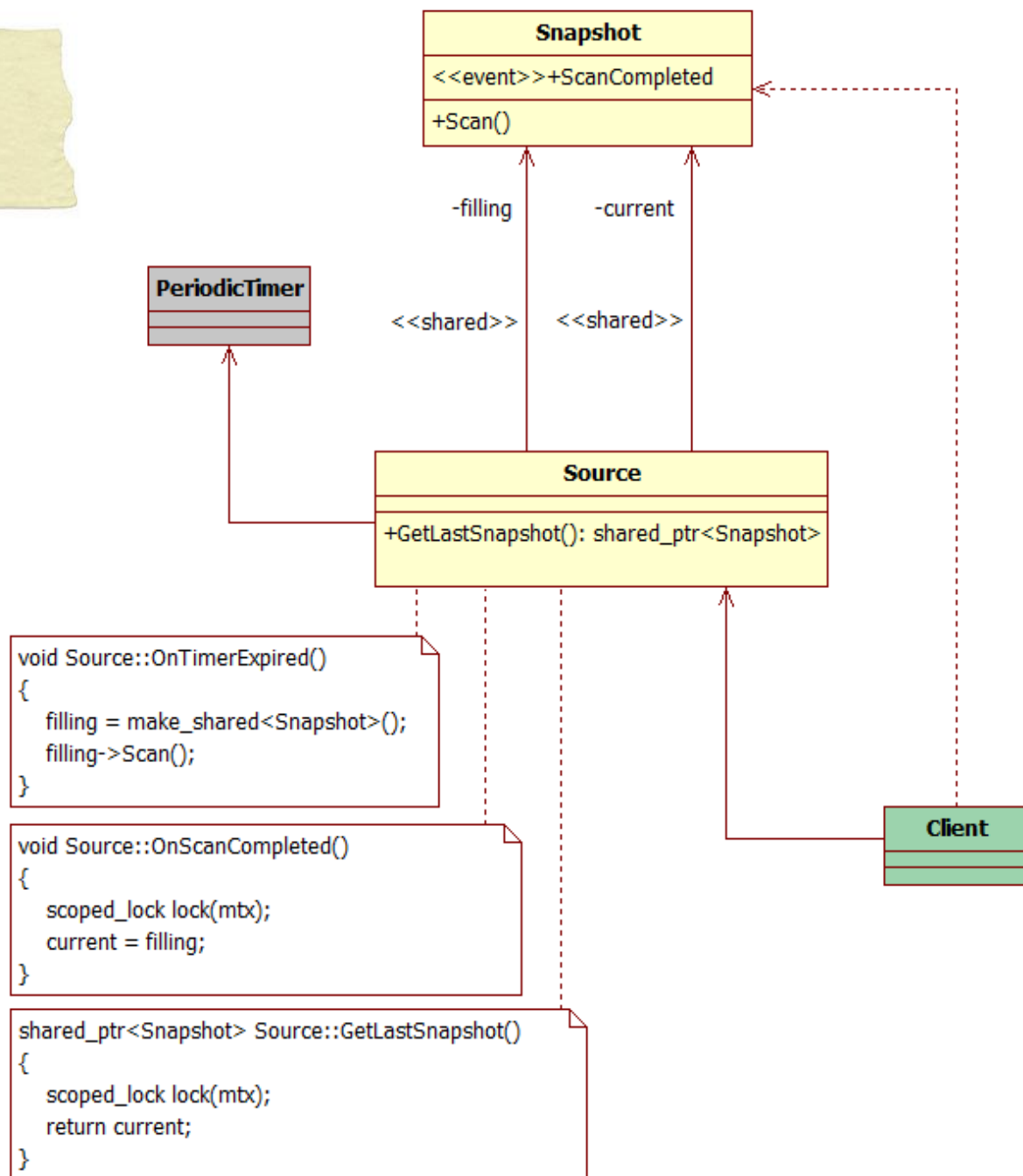
```

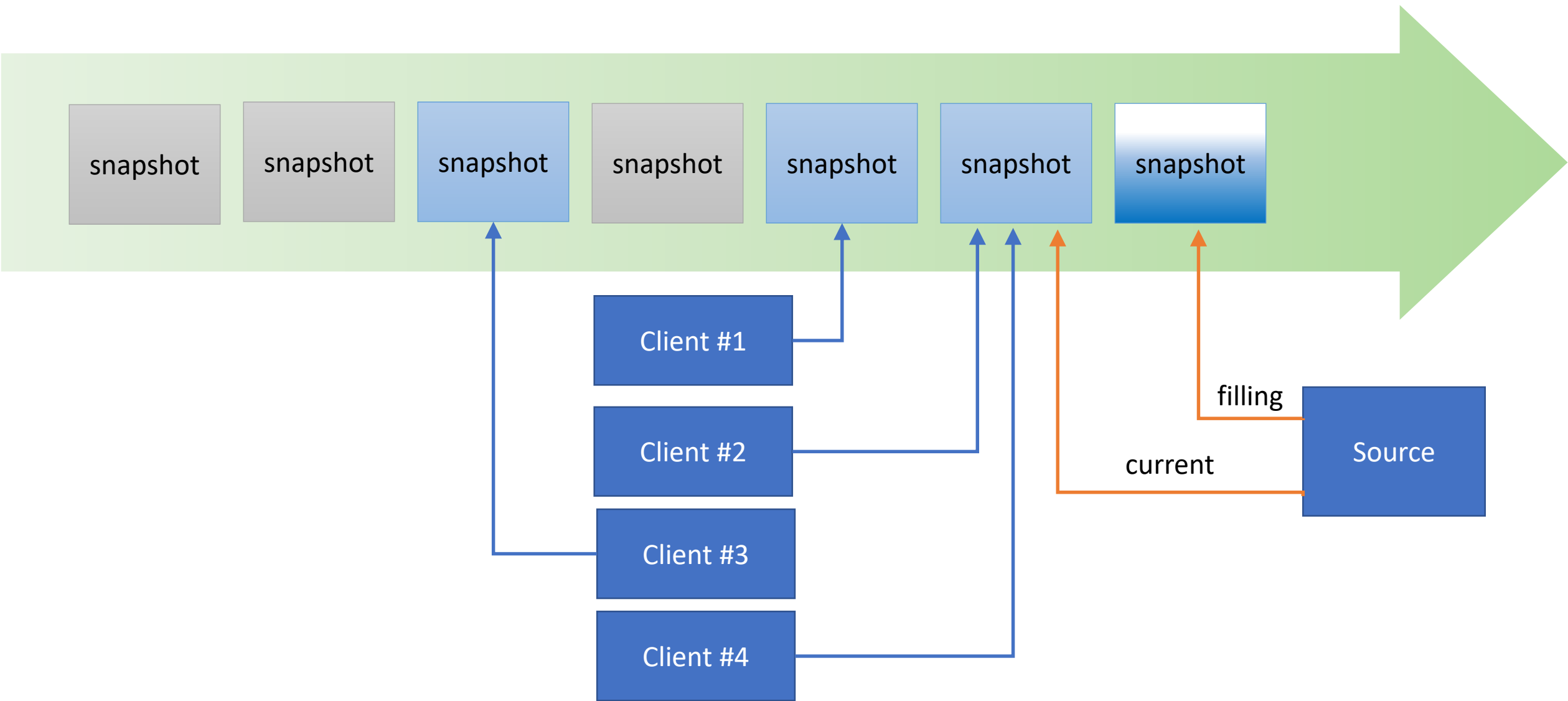
shared_ptr<const SensorAcquisition>
SensorNetwork::GetLastMeasure() const
{
    scoped_lock<mutex> lock(mtx);
    return current;
}

```



Structure






Applicability

Use Flip Model when:

- you have a complex data structure slow to update.

- Its clients must asynchronously read the most updated data available in a consistent fashion.


- Older information must be discarded when is no longer needed.



Consequences

Decoupling


Flip Model **decouples** the producer from the readers: the producer can go on with the update of the data (slow) and each reader gets each time the most updated version.



Consequences

Decoupling
Synchronization


Producer and readers can run in different threads.



Consequences

Decoupling
Synchronization
Coherence

Flip Model grants the **coherence** of all the data structures that are read in a given instant from a reader, without locks taken for long times.



Consequences

Decoupling
Synchronization
Coherence

Memory

Memory consumption to the bare minimum
to grant that every reader has a coherent
access to the most recent snapshot.

Sample code

```
class SensorAcquisition
{
public:
    // interface for clients
    const SomeComplexDataStructure& Data() const;

    // interface for SensorNetwork
    template <typename Handler>
    void Scan(Handler h);
};
```

```

class SensorNetwork
{
public:

    SensorNetwork() :
        timer([this]() { OnTimerExpired(); })
    {
        timer.Start(10s);
    }

    shared_ptr<const SensorAcquisition>
    GetLastMeasure() const
    {
        scoped_lock<mutex> lock(mtx);
        return current;
    }
}

```

Sample code

```

private:

    void OnTimerExpired()
    {
        filling = make_shared<SensorAcquisition>();
        filling->Scan( // start an async operation
            [this]() { OnScanCompleted(); });
    }

    void OnScanCompleted()
    {
        scoped_lock<mutex> lock(mtx);
        current = filling;
    }

    PeriodicTimer timer;
    shared_ptr<SensorAcquisition> filling;
    shared_ptr<SensorAcquisition> current;
    mutable mutex mtx; // protect "current"
};

```

```
class Client
{
public:
    Client(const SensorNetwork& sn) : sensors(sn) {}

    // possibly runs in another thread
    void DoSomeWork()
    {
        auto measure = sensors.GetLastMeasure();
        // do something with measure
        // ...
    }
private:
    const SensorNetwork& sensors;
};
```

Sample code

Concurrency

The pattern supports every concurrency model:

from the **single thread** (in a fully asynchronous application)
to the **maximum parallelization possible** (when the acquisition has its own threads, as well as each client).

When the acquisition and the usage of the snapshots run in different threads, a synchronization mechanism must be put in place to protect the `shared_ptr`

The (thread-)safety is granted by:

A SensorAcquisition is modified just in one thread, and never changes after it becomes public

```
SensorNetwork() :  
    timer([this]() { OnTimerExpired(); })  
{  
    timer.Start(10s);  
}
```

```
shared_ptr<const SensorAcquisition>  
GetLastMeasure() const  
{  
    scoped_lock<mutex> lock(mtx);  
    return current;  
}
```

```
private:  
void OnTimerExpired()  
{  
    filling = make_shared<SensorAcquisition>();  
    filling->Scan( // start an async operation  
        [this]() { OnScanCompleted(); });  
}
```

```
void OnScanCompleted()  
{  
    scoped_lock<mutex> lock(mtx);  
    current = filling;  
}
```

```
PeriodicTimer timer;  
shared_ptr<SensorAcquisition> filling;  
shared_ptr<SensorAcquisition> current;  
mutable mutex mtx; // protect "current"  
};
```

The (thread-)safety is granted by:

```
class SensorNetwork
{
public:
```

```
    SensorNetwork() :
        timer([this]() { OnTimerExpired(); })
    {
```

The smart pointer exchange is protected by a mutex

```
    getLastMeasure() const
```

```
    {
        scoped_lock<mutex> lock(mtx);
        return current;
    }
```

```
private:
```

```
    void OnTimerExpired()
```

```
    {
        filling = make_shared<SensorAcquisition>();
        filling->Scan( // start an async operation
            [this]() { OnScanCompleted(); });
    }
```

```
    void OnScanCompleted()
```

```
    {
        scoped_lock<mutex> lock(mtx);
        current = filling;
    }
```

```
    PeriodicTimer timer;
```

```
    shared_ptr<SensorAcquisition> filling;
```

```
    shared_ptr<SensorAcquisition> current;
```

```
    mutable mutex mtx; // protect "current"
```

```
};
```

Why the mutex?!

Only the control-block is protected, not the **shared_ptr** instance

Thread 1

```
void SensorNetwork::OnScanCompleted()  
{  
    scoped_lock<mutex> lock(mtx);  
    current = filling;  
}
```

Thread 2

```
shared_ptr<const SensorAcquisition>  
GetLastMeasure() const  
{  
    scoped_lock<mutex> lock(mtx);  
    return current;  
}
```

What if I don't like mutexes?

You can use instead:

`std::atomic_...<std::shared_ptr>` C++11

`std::atomic<std::shared_ptr>` C++20

The implementation of the atomic functions **might not be lock-free**

Better solutions next

Snapshot

Get with `Source::GetLastSnapshot()`

Snapshot

Get with Source::GetLastSnapshot()

Immutable

Snapshot

Get with Source::GetLastSnapshot()

Immutable

Pool

Snapshot

Get with `Source::GetLastSnapshot()`

Immutable

Pool

No stupid classes!

Snapshot

Get with Source::GetLastSnapshot()

Immutable

Pool

No stupid classes!

Async VS Sync

Asynchronous VS Synchronous

```
template <typename Handler>
void Snapshot::Scan(Handler h)
{ /* ... */ }

void Source::OnTimerExpired()
{
    filling=make_shared<Snapshot>();
    filling->Scan( // start an async operation
        [this]() { OnScanCompleted(); });
}

void Source::OnScanCompleted()
{
    scoped_lock<mutex> lock(mtx);
    current = filling;
}
```

```
void Snapshot::Scan()
{ /* ... */ }

void Source::Run()
{
    while (!shutdown)
    {
        filling = make_shared<Snapshot>();
        filling->Scan();
        scoped_lock<mutex> lock(mtx);
        current = filling;
    };
}
```

Periodic **VS** Continuous acquisition

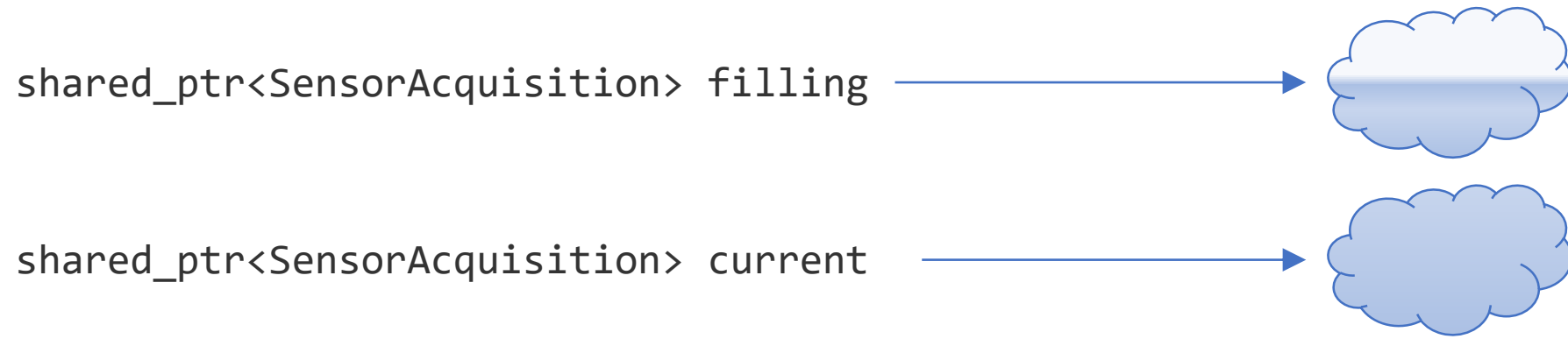
A new acquisition can be started:

periodically (example)

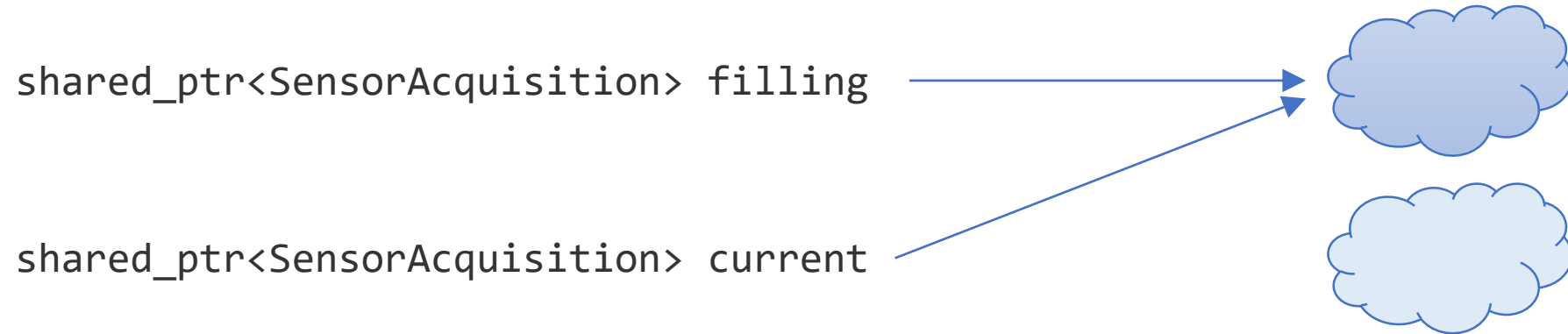
continuously (immediately after the previous one is completed).

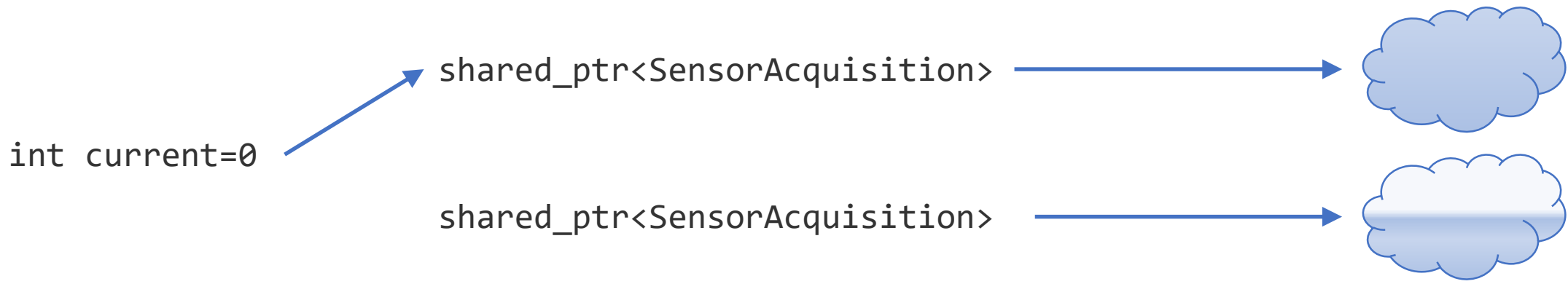
Also implementable in GC languages

Alternative lock-free #1

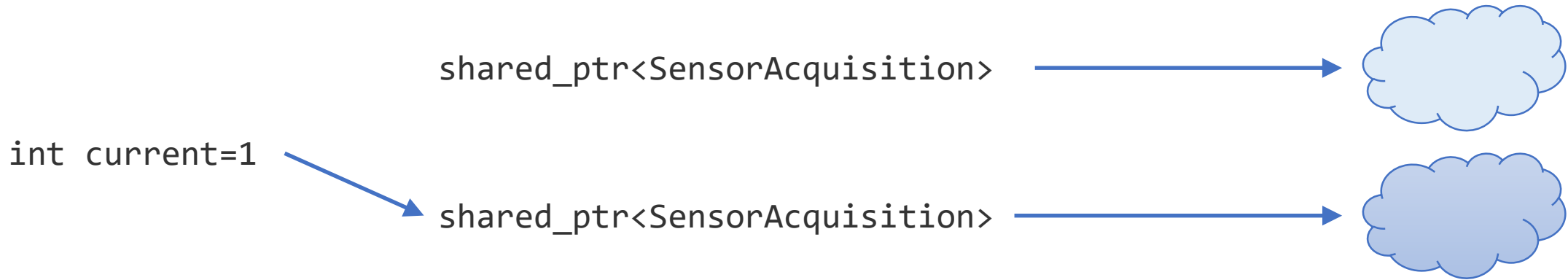


↓ `current = filling`





current = 1-current



```

class SensorNetwork
{
public:
    SensorNetwork() :
        timer( [this]() { OnTimerExpired(); })
    {
        // just to be sure :-)
        static_assert(
            current.is_always_lock_free,
            "No lock free"
        );

        timer.Start(10s);
    }

    shared_ptr<const SensorAcquisition>
    GetLastMeasure() const
    {
        assert(current < 2);
        return measures[current];
    }
}

```

```

private:
    void OnTimerExpired()
    {
        auto sa=make_shared<SensorAcquisition>();

        // start an async operation
        sa->Scan([this]() { OnScanCompleted(); } );

        // filling = 1-current
        assert(current < 2);
        measures[1-current] = sa;
    }
    void OnScanCompleted()
    {
        current.fetch_xor(1); // current = 1-current
    }

    PeriodicTimer timer;
    array<shared_ptr<SensorAcquisition>,2> measures;
    atomic_uint current=0; // filling = 1-current
};

```

Do we really need an atomic integer?

```
// thread #1
```

```
shared_ptr<const SensorAcquisition> GetLastMeasure() const  
{  
    assert(current < 2);  
    return measures[current];  
}
```

```
// thread #2
```

```
void OnScanCompleted()  
{  
    current.fetch_xor(1); // current = 1-current  
}
```

```
atomic_uint current=0; // filling = 1-current
```

ISO C++ standard

- The memory available to a C++ program is one or more contiguous sequences of *bytes*. Each byte in memory has a unique *address*.
- A *memory location* is:
 - an object of [scalar type](#) (arithmetic type, pointer type, enumeration type, or `std::nullptr_t`)
 - or the largest contiguous sequence of [bit fields](#) of non-zero length
- Two expression evaluations [conflict](#) if one of them modifies a memory location ([\[intro.memory\]](#)) and the other one reads or modifies the same memory location.
- Two actions are [potentially concurrent](#) if [\(21.1\)](#):
 - they are performed by different threads, or
 - they are unsequenced, at least one is performed by a signal handler, and they are not both performed by the same signal handler invocation.
- The execution of a program contains a [data race](#) if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below.
- Any such data race results in [undefined behavior](#).

Thread #1

```
shared_ptr<const SensorAcquisition>  
GetLastMeasure() const  
{  
    assert(current < 2);  
    return measures[current];  
}
```

Thread #2

```
void OnTimerExpired()  
{  
    auto sa = make_shared<SensorAcquisition>();  
  
    // start an async operation  
    sa->Scan([this]() { OnScanCompleted(); });  
  
    // filling=1-current  
    assert(current < 2);  
    measures[1-current] = sa;  
}  
  
void OnScanCompleted()  
{  
    current.fetch_xor(1); //current = 1-current  
}
```


Thread #1

```
// GetLastMeasure()  
    return measures[current];
```

Thread #2

```
void OnTimerExpired()  
{  
    auto sa = make_shared<SensorAcquisition>();  
  
    // start an async operation  
    sa->Scan([this]() { OnScanCompleted(); });  
  
    // filling=1-current  
    assert(current < 2);  
    measures[1-current] = sa;  
}  
  
void OnScanCompleted()  
{  
    current.fetch_xor(1); //current = 1-current  
}
```

Thread #1

```
// GetLastMeasure()  
    return measures[current];
```

Thread #2

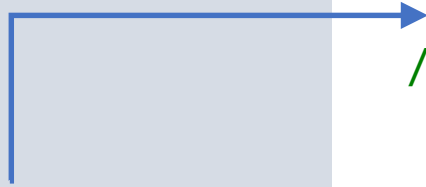
```
    current = 0  
    ...  
// OnTimerExpired()  
    auto sa = make_shared<SensorAcquisition>()  
    sa->Scan(...); // start async operation...  
    measures[1] = sa;  
    ...  
// OnScanCompleted()  
    current = 1;  
    ...  
// OnTimerExpired()  
    auto sa = make_shared<SensorAcquisition>()  
    sa->Scan(...); // start async operation...  
    measures[0] = sa;  
    ...  
// OnScanCompleted()  
    current = 0;  
    ...
```

Thread #1

```
// GetLastMeasure()  
    return measures[current];
```

Thread #2

```
    current = 0  
    ...  
    // OnTimerExpired()  
    auto sa = make_shared<SensorAcquisition>()  
    sa->Scan(...); // start async operation...  
    measures[1] = sa;  
    ...  
    // OnScanCompleted()  
    current = 1;  
    ...  
    // OnTimerExpired()  
    auto sa = make_shared<SensorAcquisition>()  
    sa->Scan(...); // start async operation...  
    measures[0] = sa;  
    ...  
    // OnScanCompleted()  
    current = 0;  
    ...
```



Thread #1

Thread #2

```
// GetLastMeasure()  
return measures[current];
```

```
current = 0  
...  
// OnTimerExpired()  
auto sa = make_shared<SensorAcquisition>()  
sa->Scan(...); // start async operation...  
measures[1] = sa;  
...  
// OnScanCompleted()  
current = 1;  
...  
// OnTimerExpired()  
auto sa = make_shared<SensorAcquisition>()  
sa->Scan(...); // start async operation...  
measures[0] = sa;  
...  
// OnScanCompleted()  
current = 0;  
...
```

Alternative lock-free #2

```
std::enable_shared_from_this<T>
```

```
std::shared_ptr<Widget> self(this);
```


std::enable_shared_from_this<T>

```
class Widget : public std::enable_shared_from_this<Widget>
{
    void DoSomething()
    {
        std::shared_ptr<Widget> self = shared_from_this();
        subject->Register(self);
    }
};
```

```
auto sa1 = std::make_shared<Widget>();
auto sa2 = sa1->shared_from_this();
assert(sa1 == sa2);
```

```
class SensorAcquisition : public enable_shared_from_this<SensorAcquisition>
{
public:
    // interface for clients
    const SomeComplexDataStructure& Data() const { /* ... */ }

    // interface for SensorNetwork
    template <typename Handler>
    void Scan(Handler h) { /* ... */ }
};
```

```

class SensorNetwork
{
public:
    SensorNetwork() :
        timer([this]() { OnTimerExpired(); })
    {
        // just to be sure :-)
        static_assert(
            current.is_always_lock_free,
            "No lock free"
        );

        timer.Start(10s);
    }

    shared_ptr<const SensorAcquisition>
    GetLastMeasure() const
    {
        return current.load()->shared_from_this();
    }
}

```

```

private:
    void OnTimerExpired()
    {
        auto sa = make_shared<SensorAcquisition>();

        finalizationQueue.push(sa);
        if (finalizationQueue.size() > QUEUE_SIZE)
            finalizationQueue.pop();

        filling = sa.get();

        // start an async operation
        sa->Scan([this]() { OnScanCompleted(); } );
    }

    void OnScanCompleted()
    {
        current = filling;
    }

    static constexpr size_t QUEUE_SIZE = 2;
    PeriodicTimer timer;
    SensorAcquisition* filling = nullptr;
    atomic<SensorAcquisition*> current = nullptr;

    queue<shared_ptr<SensorAcquisition>> finalizationQueue;
};

```

Thread #1

```
shared_ptr<const SensorAcquisition>  
GetLastMeasure() const  
{  
    return  
        current.load()->shared_from_this();  
}
```

Thread #2

```
void OnTimerExpired()  
{  
    auto sa = make_shared<SensorAcquisition>();  
  
    finalizationQueue.push(sa);  
    if (finalizationQueue.size() > QUEUE_SIZE)  
        finalizationQueue.pop();  
  
    filling = sa.get();  
  
    // start an async operation  
    sa->Scan([this]() { OnScanCompleted(); });  
}  
  
void OnScanCompleted()  
{  
    current = filling;  
}
```

Thread #1

```
// GetLastMeasure
```

```
current.load()
```

```
->shared_from_this());
```

Thread #2

```
void OnTimerExpired()
{
    auto sa = make_shared<SensorAcquisition>();

    finalizationQueue.push(sa);
    if (finalizationQueue.size() > QUEUE_SIZE)
        finalizationQueue.pop();

    filling = sa.get();

    // start an async operation
    sa->Scan([this]() { OnScanCompleted(); });
}

void OnScanCompleted()
{
    current = filling;
}
```

Thread #1

```
// GetLastMeasure
```

```
current.load() SA3
```

```
->shared_from_this();
```

Thread #2

```
current = filling SA3
```

```
...
```

```
// OnTimerExpired()
```

```
auto sa = make_shared<SensorAcquisition>() SA4
```

```
finalizationQueue.push(sa);
```

```
if (finalizationQueue.size() > QUEUE_SIZE)
```

```
    finalizationQueue.pop(); |SA3|SA4| delete SA2
```

```
filling = sa.get(); SA4
```

```
sa->Scan(...); // start async operation...
```

```
...
```

```
// OnScanCompleted()
```

```
current = filling SA4
```

```
...
```

```
// OnTimerExpired()
```

```
auto sa = make_shared<SensorAcquisition>() SA5
```

```
finalizationQueue.push(sa);
```

```
if (finalizationQueue.size() > QUEUE_SIZE)
```

```
    finalizationQueue.pop(); |SA4|SA5| delete SA3
```

```
filling = sa.get(); SA5
```

```
sa->Scan(...); // start async operation...
```

```
...
```

```
// OnScanCompleted()
```

```
current = filling SA5
```

Solution

Increase the **size** of the queue

Intrusive

```
class SensorAcquisition : public enable_shared_from_this<SensorAcquisition>
{
    ...
};
```


Related Patterns

Ping Pong Buffer (AKA Double Buffer in computer games)

Related Patterns

Ping Pong Buffer (AKA Double Buffer in computer games)

Snapshot can/should be a **Façade**

Related Patterns

Ping Pong Buffer (AKA Double Buffer in computer games)

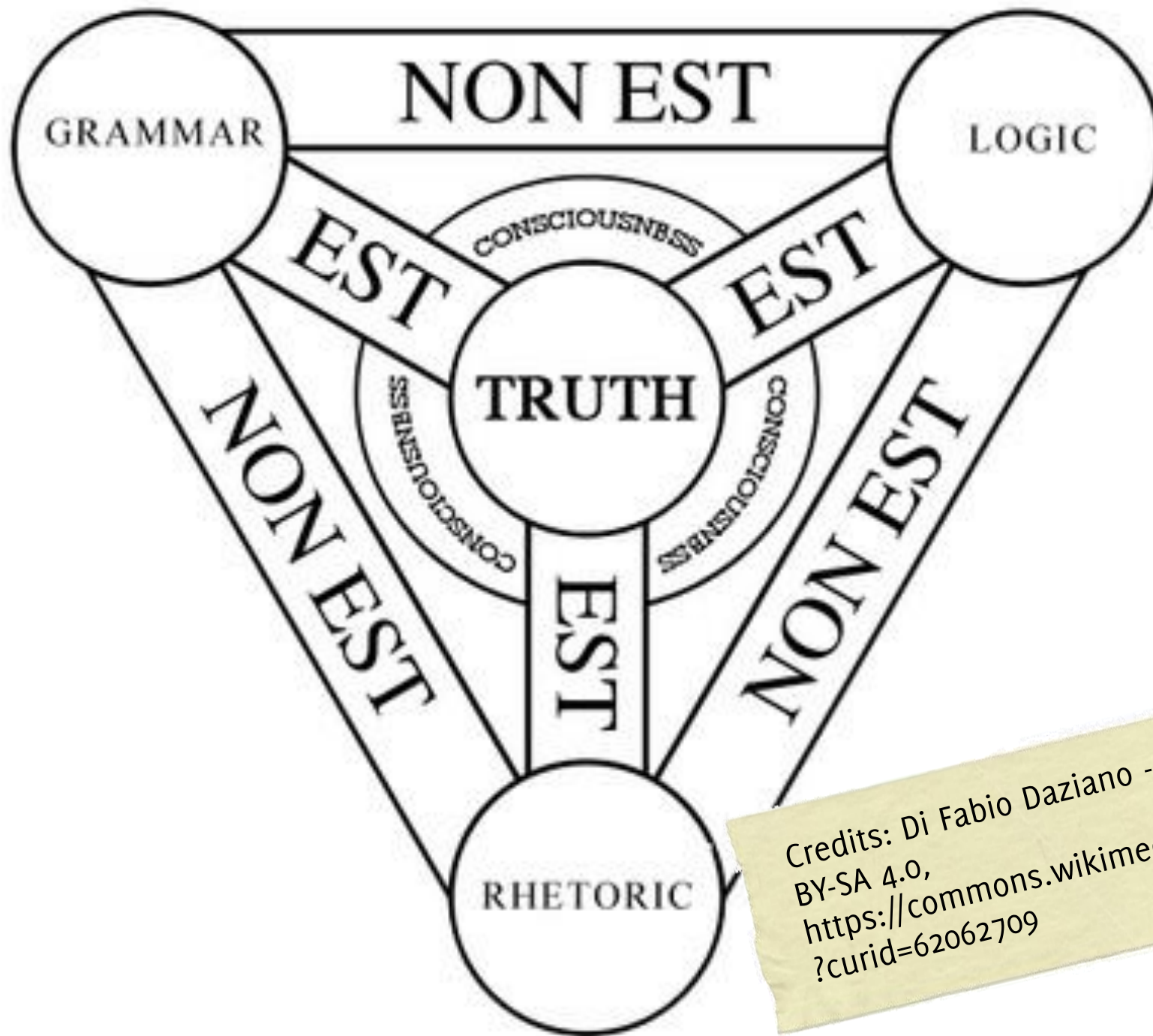
Snapshot can/should be a **Façade**

Source can use a **Strategy** to change the policy of update
(e.g., periodic VS continuous)

Design is a balance of forces


Mutex **VS** Lock-Free

Intrusive **VS** Non-Intrusive



Credits: Di Fabio Daziano - Opera propria, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=62062709>

Coding **vs** Design



Language syntax
Language tools
Design

References

Me: @DPallastrelli

Github: <http://github.com/daniele77>

Web: daniele77.github.io