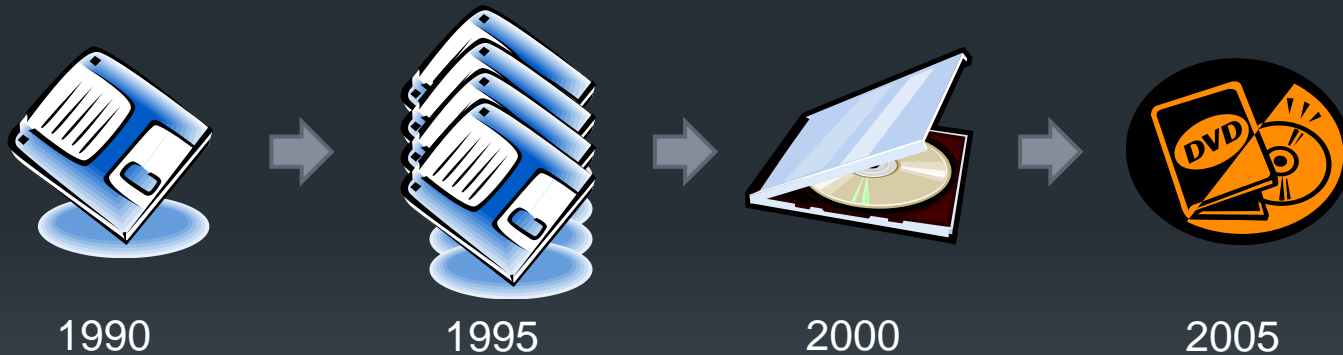


Labor-Saving Architecture: An Object-Oriented Framework for Network Software

William R. Otte and Douglas C. Schmidt

Modern Software is Complex



“What Andy gives, Bill takes away.”



Programming Paradigms

- Concurrent
- Event-driven
- Declarative
 - Functional
- Imperative
 - Non-structured
 - Structured
 - Procedural
 - Object-oriented
-



Principles in Object-Oriented Software Design

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle



Single Responsibility Principle

Every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class. (a.k.a. **cohesion principle**)



Open/closed Principle

“Software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.”



Liskov Substitution Principle

Derived types must be completely substitutable for their base types.

Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be true for objects y of type S where S is a subtype of T .



Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they don't use.

Many client-specific interfaces are better than one general purpose interface.

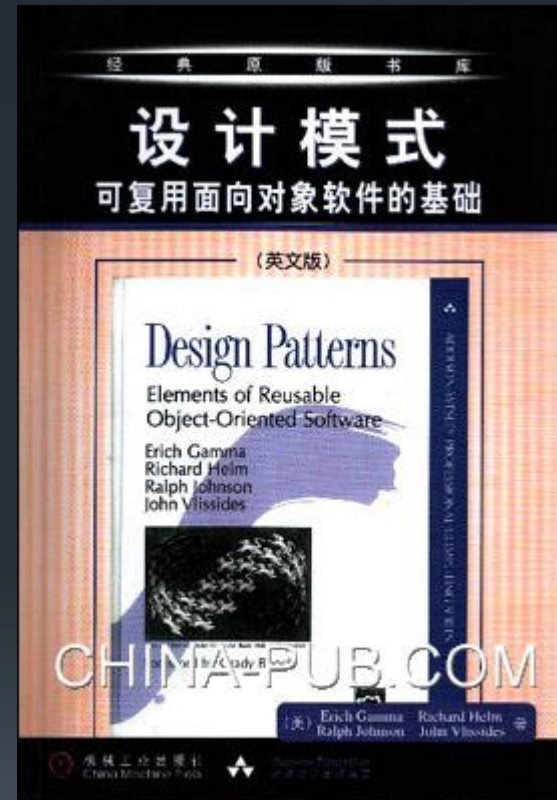


Dependency Inversion Principle

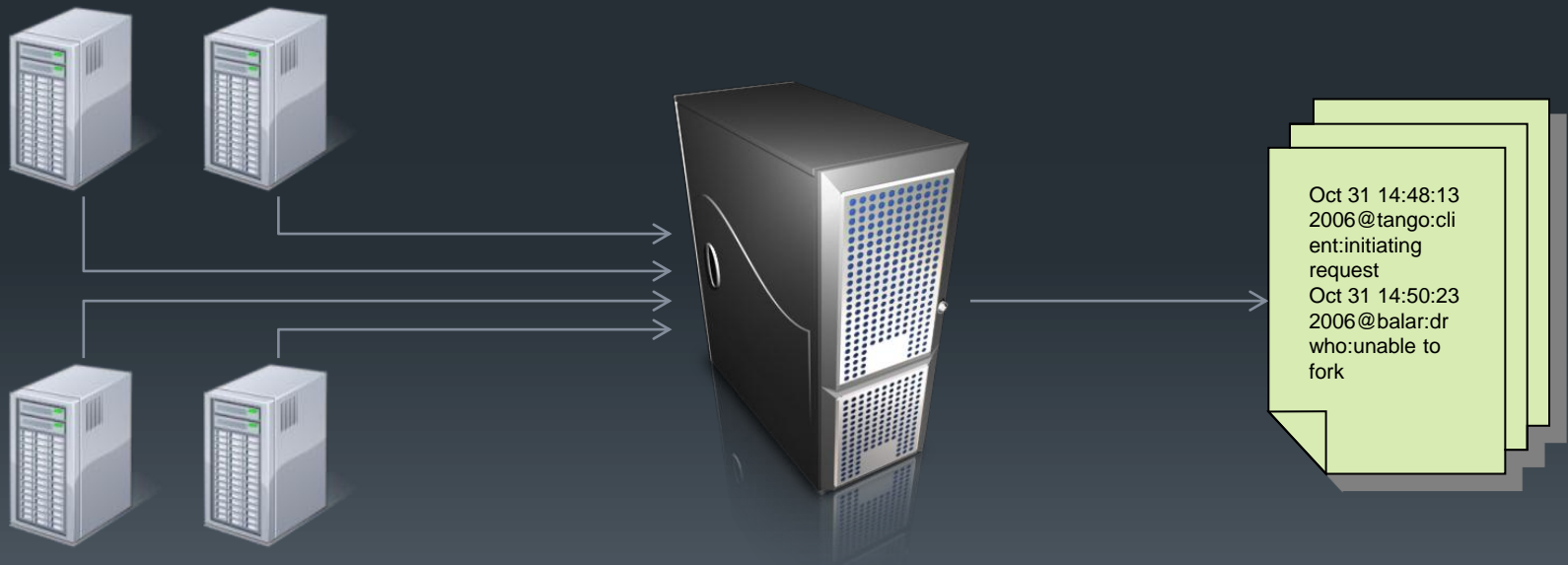
1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Design Patterns

- Singleton
- Wrapper Façade
- Template Method
- Abstract Factory
- Adapter
-



Sample: A Logging Service

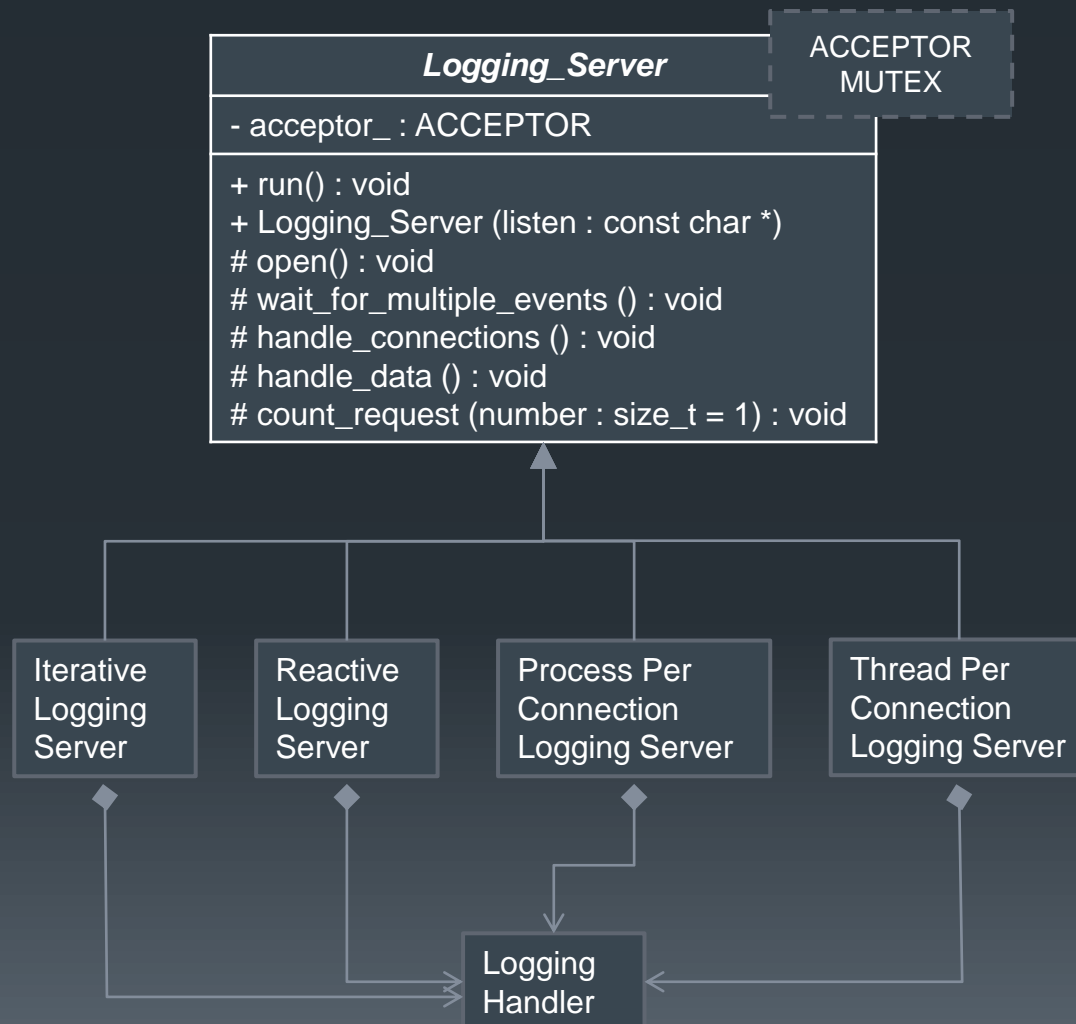




Dimensions of Variability

- Different inter-process communication mechanisms
 - Sockets, SSL, shared memory, named pipes, ...
- Different concurrency models
 - Iterative, reactive, thread-per-connection, process-per-connection, ...
- Different locking strategies
 - Thread-level or process-level recursive mutex, non-recursive mutex, r/w lock, ...
- Different log record formats
- Different transmission formats

An Extensible Solution





Frameworks vs. Class Libraries

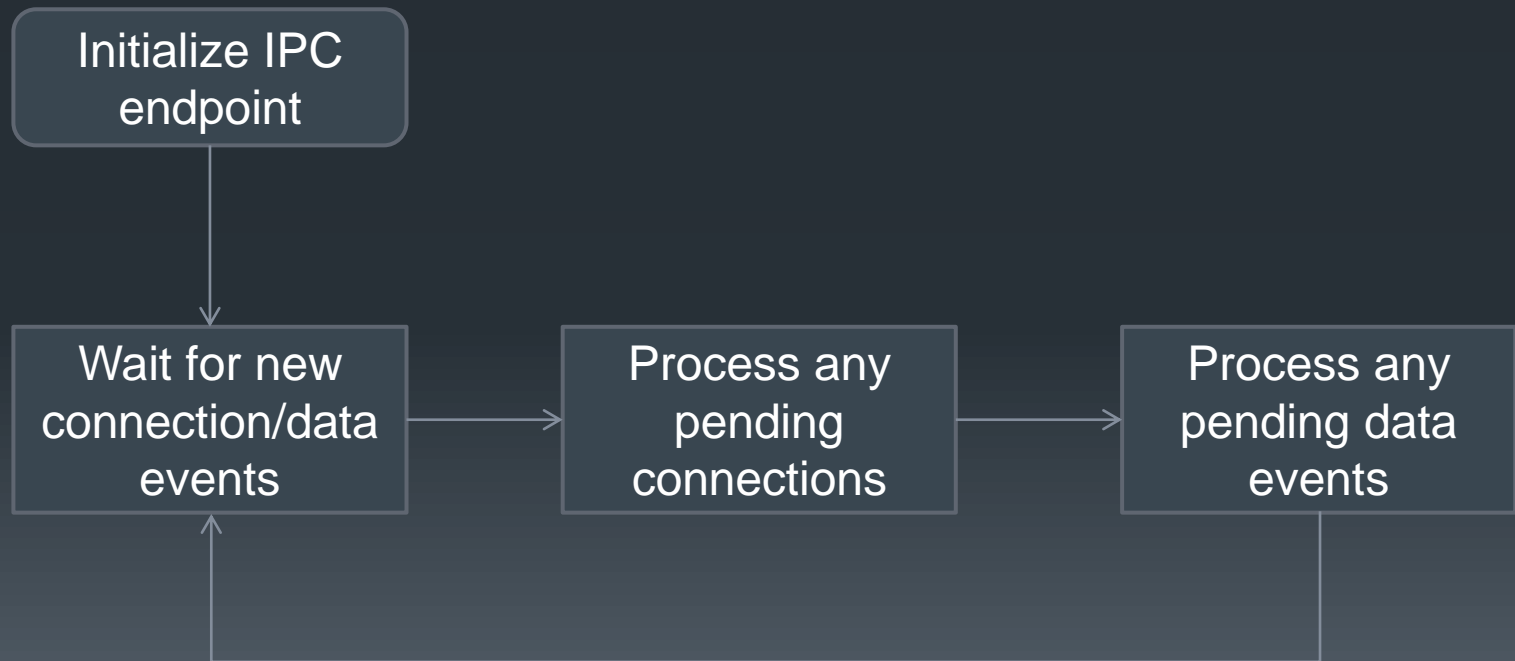
- Frameworks are “semi-complete” applications.
 - Class libraries are low-level components.
- Frameworks are active. “Don’t call us, we’ll call you.”
 - Class libraries are passive.



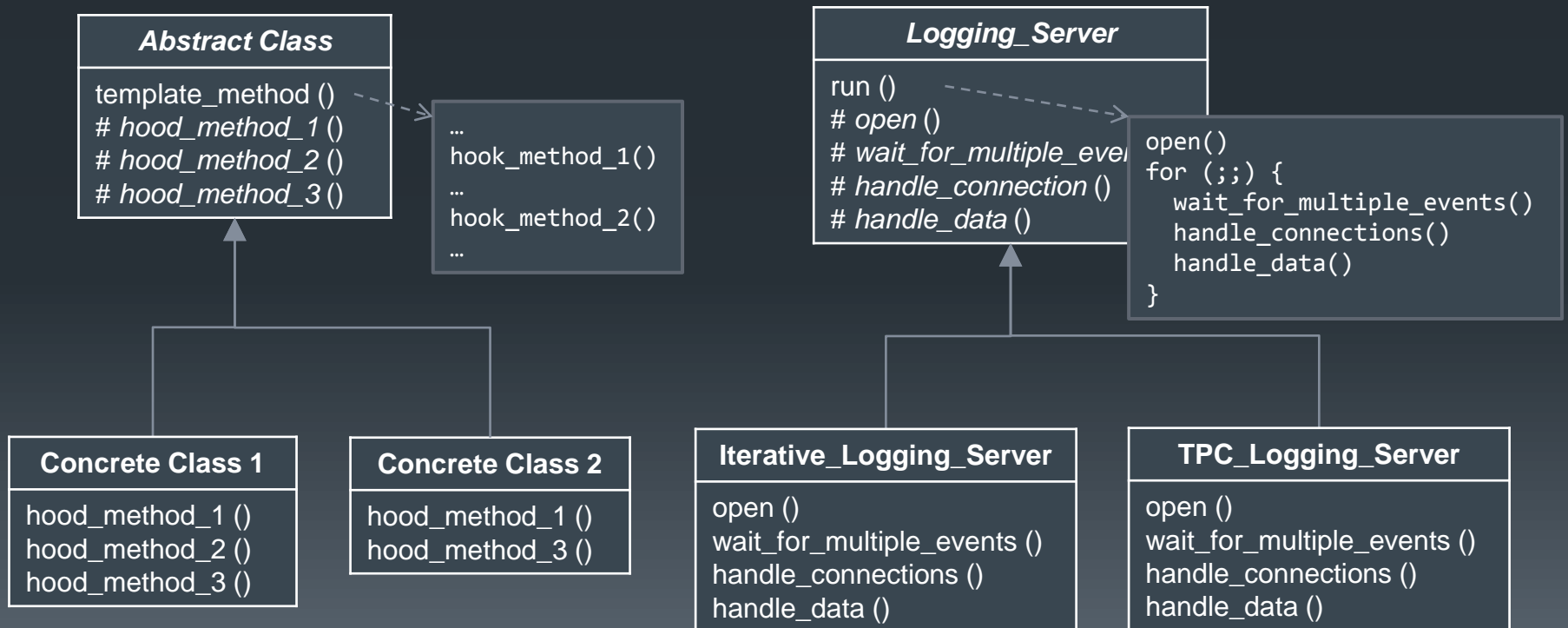
Considerations in Framework Design

- Scope
- Commonalities
- Variabilities

Commonalities



Template Method Pattern

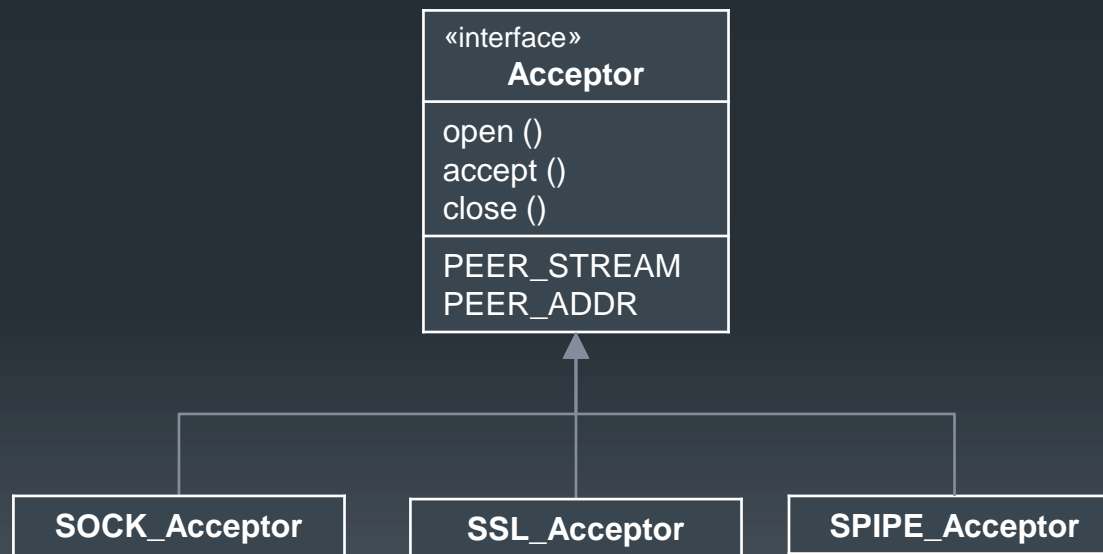




Accommodating Variabilities

- Different concurrency models
 - Addressed with **Template Method** pattern
- Different inter-process communication mechanisms
 - Same interface: **open/accept**
- Different locking strategies
 - Same interface: **acquire/release**
- Different log record formats
- Different transmission formats
 - Addressed in Logging Handler class.

Wrapper Façade Pattern



ACE has already done this for us.



Tying it All Together

- **Strategy** pattern could do this.
 - But we don't need dynamic binding!
- C++ template mechanism will do the trick.

The Base Class

```
template <typename ACCEPTOR, typename MUTEX>
class Logging_Server
{
public:
    typedef Log_Handler<typename ACCEPTOR::PEER_STREAM> HANDLER;

    Logging_Server(const char *listen);

    // Template method that runs each step in the main event loop.
    virtual void run();
};
```



protected:

```
// Hook methods that enable each step to be varied.  
virtual void open();  
virtual void wait_for_multiple_events() = 0;  
virtual void handle_connections() = 0;  
virtual void handle_data() = 0;  
  
// Increment the request count, protected by the mutex.  
virtual void count_request(size_t number = 1);
```



protected:


```
// Instance of template parameter that accepts connections.  
ACCEPTOR acceptor_;
```

```
// Keeps a count of the number of log records received.  
size_t request_count_;
```


```
// Instance of template parameter that serializes access to  
// the request_count_.  
MUTEX mutex_;
```

```
// Address that the server will listen on for connections.  
ACE_INET_Addr server_address_;
```

```
};
```



```
template <typename ACCEPTOR, typename MUTEX>
void Logging_Server<ACCEPTOR, MUTEX>::run() {
    try {
        // Step 1: initialize an IPC factory endpoint to listen for
        // new connections on the server address.
        open();
        // Step 2: Go into an event loop
        for (;;) {
            // Step 2a: wait for new connections or log records
            // to arrive.
            wait_for_multiple_events();
            // Step 2b: accept a new connection (if available)
            handle_connections();
            // Step 2c: process received log record (if available)
            handle_data();
        }
    } catch (...) { /* ... Handle the exception ... */ }
}
```

```
template <typename ACCEPTOR, typename MUTEX>
Logging_Server<ACCEPTOR, MUTEX>::Logging_Server(const char *listen) :
    request_count_ (0),
    server_address_(listen, PF_INET) {
}
```


```
template <typename ACCEPTOR, typename MUTEX>
void Logging_Server<ACCEPTOR, MUTEX>::open() {
    acceptor_.open(server_address_);
}
```

```
template <typename ACCEPTOR, typename MUTEX>
void Logging_Server<ACCEPTOR, MUTEX>::count_request(size_t number) {
    mutex_.acquire();
    request_count_ += number;
    mutex_.release();
}
```

An Iterative Logging Server

```
template <typename ACCEPTOR>
class Iterative_Logging_Server :
    public Logging_Server<ACCEPTOR, ACE_Null_Mutex>
{
public:
    typedef Logging_Server<ACCEPTOR, ACE_Null_Mutex>::HANDLER HANDLER;
    Iterative_Logging_Server(const char *listen, HANDLER *handler) :
        Logging_Server<ACCEPTOR, ACE_Null_Mutex>(listen),
        log_handler_(handler) {}

protected:
    virtual void wait_for_multiple_events() {}
    virtual void handle_connections();
    virtual void handle_data();
    HANDLER *log_handler_;
};
```



```
template <typename ACCEPTOR>
void Iterative_Logging_Server<ACCEPTOR>::handle_connections() {
    acceptor_.accept(log_handler_->peer());
}
```

```
template <typename ACCEPTOR>
void Iterative_Logging_Server<ACCEPTOR>::handle_data() {
    while (log_handler_->log_record())
        count_request();
}
```



Problem of Iterative Implementation

- Only one client could be served at a time.
- Stuck at `handle_data()` most of the time.
- Should leverage `select()` or `WaitForMultipleObjects()` to handle multiple clients.

A Reactive Logging Server

```
template <typename ACCEPTOR>
class Reactive_Logging_Server :
    public Iterative_Logging_Server<ACCEPTOR>
{
public:
    Reactive_Logging_Server(const char *listen, HANDLER *handler) :
        Iterative_Logging_Server<ACCEPTOR>(listen, handler) {}

protected:
    virtual void open();
    virtual void wait_for_multiple_events();
    virtual void handle_connections();
    virtual void handle_data();

private:
    ACE_Handle_Set master_set_, active_handles_;
};
```



Acceptor
----------	-------

```
template <typename ACCEPTOR>
void Reactive_Logging_Server<ACCEPTOR>::open() {
    // Delegate to base class.
    Iterative_Logging_Server<ACCEPTOR>::open();
    // Mark the handle associated with the acceptor as active.
    master_set_.set_bit(acceptor_.get_handle());
    // Set the acceptor's handle into non-blocking mode.
    acceptor_.enable(ACE_NONBLOCK);
}
```



```
template <typename ACCEPTOR>
void Reactive_Logging_Server<ACCEPTOR>::wait_for_multiple_events() {
    active_handles_ = master_set_;
    int width = (int)active_handles_.max_set() + 1;
    if (ACE::select(width, active_handles_) == -1)
        throw 1;
}
```



Acceptor	Peer1	Peer2
----------	-------	-------	--------

```
template <typename ACCEPTOR>
void Reactive_Logging_Server<ACCEPTOR>::handle_connections () {
    if (active_handles_.is_set(acceptor_.get_handle())) {
        while (acceptor_.accept(log_handler_->peer()) == 0)
            master_set_.set_bit(log_handler_->current_peer().get_handle());
        active_handles_.clr_bit(acceptor_.get_handle());
    }
}
```




```
template <typename ACCEPTOR>
void Reactive_Logging_Server<ACCEPTOR>::handle_data() {
    ACE_Handle_Set_Iterator i(active_handles_);
    for (ACE_HANDLE hdl; (hdl = i()) != ACE_INVALID_HANDLE;) {
        // Select active peer
        log_handler_->peer (hdl);
        try {
            log_handler_->log_record();
            this->count_request();
        }
        catch (...) {
            // connection shutdown/comm fail
            master_set_.clr_bit(hdl);
        }
    }
}
```



Evaluation

- Enough for small number of clients.
- Cannot utilize multiple processors;
- Cannot overlap communication and computation.



Concurrent Solutions

- Thread per connection:
 - Workflow is simple
 - Locking is important
- Process per connection:
 - ... Why do we need log servers at the first place?
- Other solutions?
 - One thread per CPU + one handler thread



Conclusion

- Paradigms
 - Concurrency, Object-Oriented, Event-driven
- Principles
 - Single Responsibility Principle, Open/closed principle
- Patterns
 - Template Method, Wrapper Façade, (Strategy)

Q & A