Object-Oriented Network Programming with C++

Design Patterns for Initializing Network Services in OO Communication Frameworks

Douglas C. Schmidt
schmidt@cs.wustl.edu

Washington University

Table of Contents

Introduction

- Design patterns capture successful solutions to recurring problems that arise during software development
- Useful design patterns and design pattern descriptions, like useful software, evolve over time
  - Likewise, patterns build upon other patterns to form families of related patterns
- The following material captures the evolution of design patterns that are useful for managing the lifecycle of services in OO communication software

Identifying Common Behavior and Structure

- "Pattern recognition" occurs by observing recurring solution strategies
- *e.g.*, network servers often look as follows:

```
void do_server (void) {
  listener_handle.initialize ();

  for (;;) {
    // Create service to handle connection.
    service = new Service;

    // Wait for and accept connection.
    client_handle = listener_handle.accept ();

    // Activate service.
    service->open (client_handle);

    // Perform service.
    service->run ();
  }
}
```

Identifying Limitations with the Existing Solution

- The solution above is limited by tightly coupling:
  - *Choice of IPC mechanisms*
  - *Connection and service handling strategies*
  - *Choice of demultiplexing strategy*
  - *Choice of advertisement strategy*
  - *Choice of endpoint listening strategy*
  - *Choice of service creation strategy*
  - *Choice of connection acceptance strategy*
  - *Choice of service concurrency strategy*
- *Strategic* and *tactical* design patterns remove these limitations and decrease coupling

Strategic and Tactical Patterns

- *Strategic* design patterns have an extensive impact on the software architecture
  - Typically oriented to solutions in a particular domain
  - *e.g.*, Reactor and Acceptor pattern in the domain of event-driven, connection-oriented communication software
- *Tactical* design patterns have a relatively localized impact on a software architecture
  - Typically domain-independent
  - *e.g.*, Wrapper, Adapter, Bridge, Factory Method, and Strategy
- It is important to understand both types of patterns

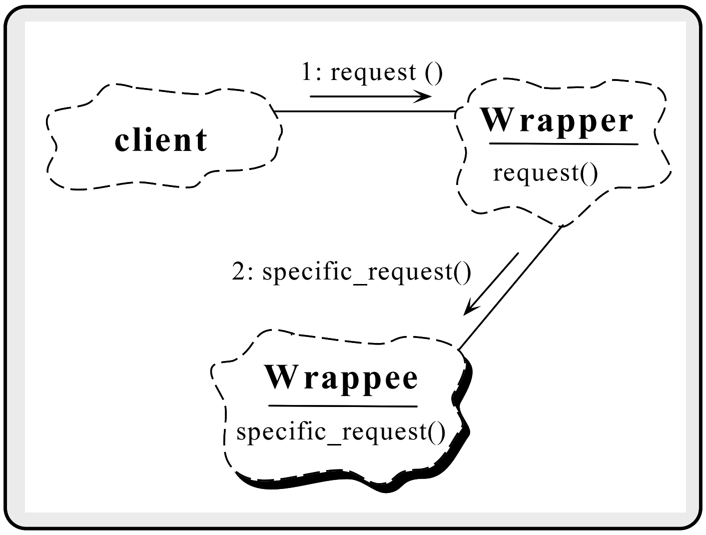
Decoupling IPC Mechanisms

- *Problem*
  - IPC mechanisms like sockets, TLI, and STREAM pipes have non-uniform interfaces, even though they behavior similarly
- *Key forces*
  - Many IPC mechanisms (or subsets of mechanisms) provide *logically* equivalent services even though their interfaces are *physically* different
    - ▹ *e.g.*, TLI vs. sockets
  - Many IPC mechanisms are written in low-level C code
- *Solution*
  - Use the *Wrapper* and *Adapter* patterns

The Wrapper Pattern

- *Intent*
  - Encapsulate lower-level functions within type-safe, modular, and portable class interfaces
- This pattern resolves the following force that arises when using native C-level OS APIs
  1. *How to avoid tedious, error-prone, and non-portable programming of low-level IPC mechanisms*
  2. *How to combine multiple related, but independent, functions into a single cohesive abstraction*

Structure of the Wrapper Pattern

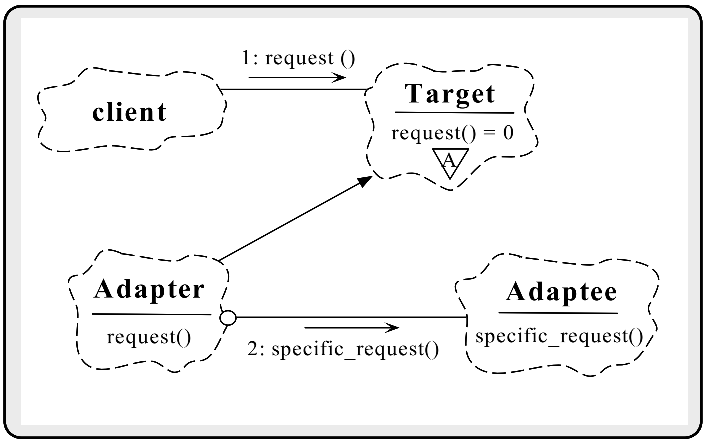


The Adapter Pattern

- *Intent*
  - Convert the interface of a class into another interface client expects
    - ▹ Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- This pattern resolves the following force that arises when using conventional OS interfaces

1. 'How to provide an interface that expresses the similarities of seemingly different OS mechanisms (such as networking or locking)

Structure of the Adapter Pattern



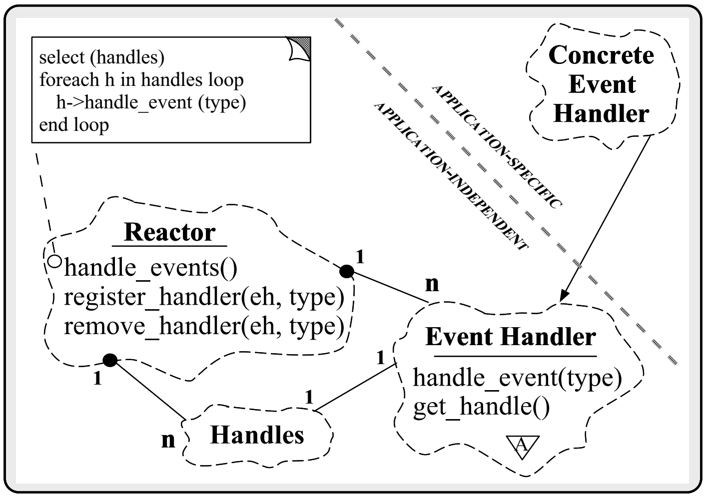
Decoupling Demultiplexing Strategy

- *Problem*
  - Servers often must handle events from more than one source
- *Key forces*
  - It is inefficient to repeatedly "poll" each event source and it is incorrect to block indefinitely on any one source of events
  - Extensibility is limited if the demultiplexing code is coupled with the application event handling code
- *Solution*
  - Use the *Reactor* pattern

The Reactor Pattern

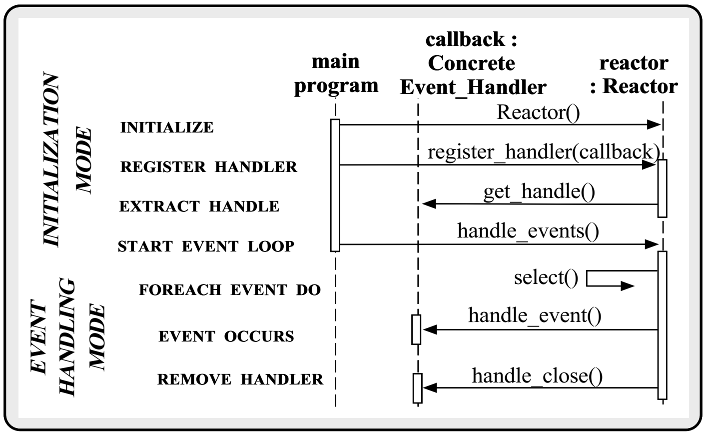
- *Intent*
  - Decouple event demultiplexing and event handler dispatching from the services performed in response to events
- This pattern resolves the following forces for event-driven software:
  1. *How to demultiplex multiple types of events from multiple sources of events efficiently within a single thread of control*
  2. *How to extend application behavior without requiring changes to the event demultiplexing/dispatching framework*

Structure



- Participants in the Reactor pattern

Collaborations



- Dynamic interaction among participants in the Reactor pattern

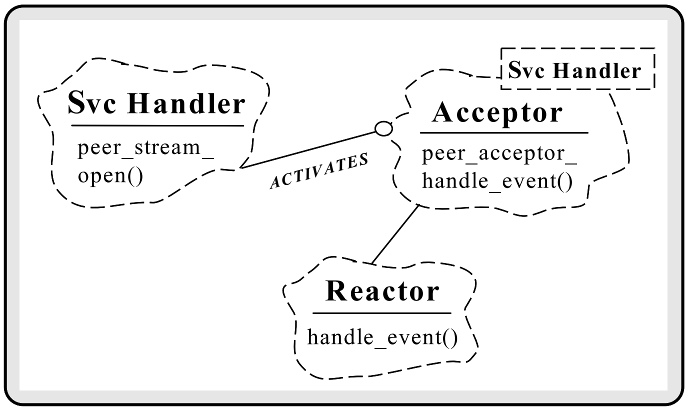
Decoupling Connection and Service Handling Strategies

- *Problem*
  - Coupling initialization strategies with the service handling makes it hard to change either one without affecting the other
- *Key forces*
  - Services tend to change more often than connection establishment strategies
- *Solution*
  - Use the *Acceptor* pattern

The Acceptor Pattern

- *Intent*
  - Decouple the passive initialization of a service from the tasks performed once the service is initialized
- This pattern resolves the following forces for network servers using interfaces like sockets or TLI:
  1. *How to reuse passive connection establishment code for each new service*
  2. *How to make the connection establishment code portable across platforms that may contain different IPC mechanisms*
  3. *How to ensure that a passive-mode descriptor is not accidentally used to read or write data*
  4. *How to enable flexible strategies for creation, connection establishment, and concurrency*

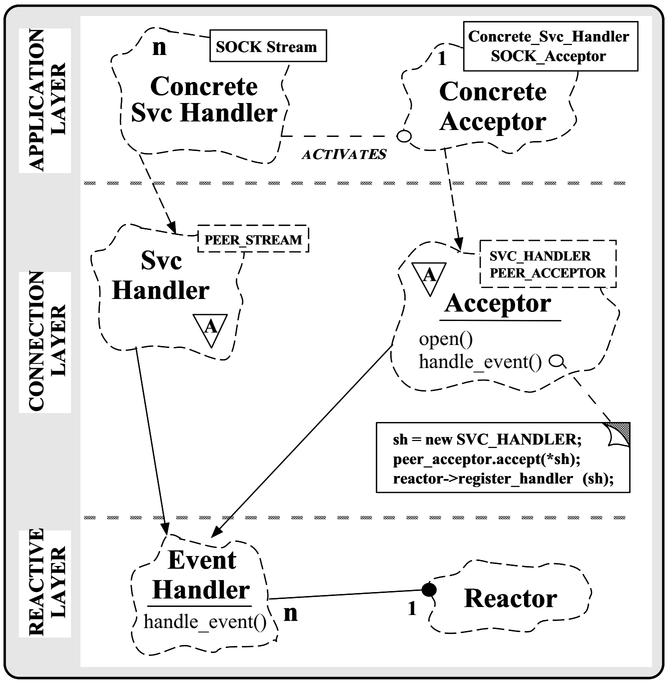
Structure of the Acceptor Pattern



First Implementation of the Acceptor Pattern

- The following slides illustrate an implementation of the Acceptor pattern
  - This solution relies upon an implementation of the Reactor pattern, as well as the Adapter pattern
- Subsequent slides will describe limitations with this first approach
- Other tactical patterns will be used to remove these limitations

Structure of the First Acceptor Pattern Implementation

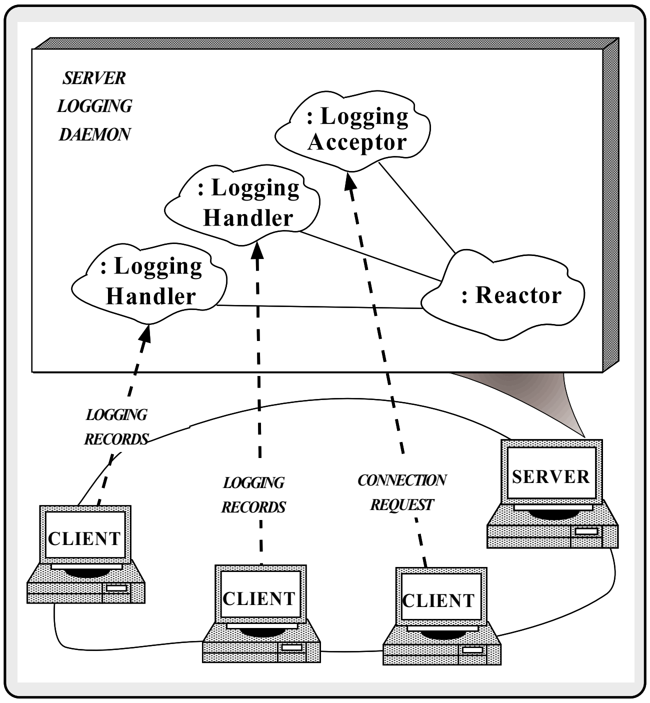


Acceptor Class Interface

- A factory for initializing network services passively

```
template <class SVC_HANDLER, // Type of service
          class PEER_ACCEPTOR> // Accepts connections
class Acceptor : public Event_Handler {
public:
  // Initialize and register with Reactor.
  virtual int open (const PEER_ACCEPTOR::PEER_ADDR &,
                    Reactor *);

  // Creates, accepts, and activates SVC_HANDLER's.
  virtual int handle_event (HANDLE);

  // Demultiplexing hooks.
  virtual HANDLE get_handle (void) const;

private:
  // Passive connection mechanism.
  PEER_ACCEPTOR peer_acceptor_;

  // Event demultiplexor.
  Reactor *reactor_;
};
```


Typical Acceptor Use-Case



Acceptor Use-case Code

- Distributed logging server

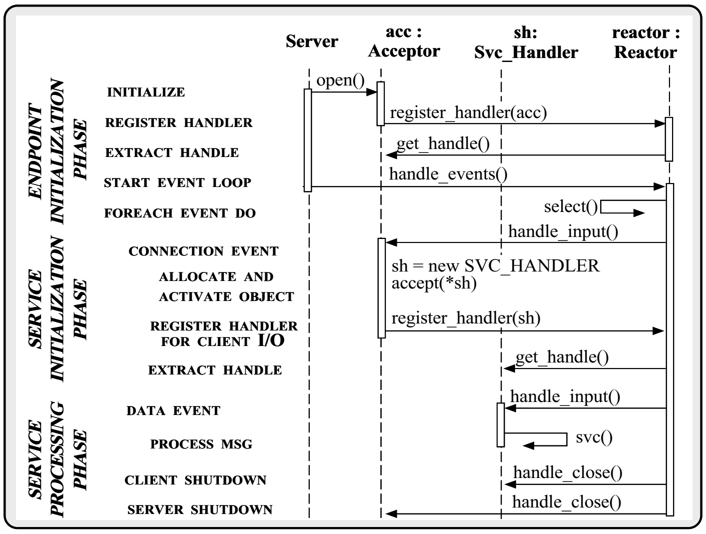
```
class Logging_Handler :
  public Svc_Handler<SOCK_Stream>
{
public:
  // Obtain HANDLE.
  virtual HANDLE get_handle (void) const;

  // Called back to process logging records.
  virtual int handle_event (HANDLE);
};

typedef Acceptor<Logging_Handler, SOCK_Acceptor>
        Logging_Acceptor;

int main (void)
{
  Reactor reactor;
  Logging_Acceptor acceptor;

  acceptor.open (LOGGER_PORT, &reactor);

  for (;;)
    reactor.handle_events ();
}
```


Acceptor Class Implementation

```
// Shorthand names.
#define SH SVC_HANDLER
#define PR_AC PEER_ACCEPTOR

// Initialize the Acceptor

template <class SH, class PR_AC> int
Acceptor<SH, PR_AC>::open (const PR_AC::ADDR &l_addr,
                           Reactor *reactor)
{
  this->reactor_ = reactor;

  // Forward to PEER_ACCEPTOR to advertise endpoint.

  this->peer_acceptor_.open (l_addr);

  // Register with Reactor to listen for connections.

  this->reactor_->register_handler
          (this, READ_MASK);
}
```

```
// Factory that create, connects, and activates new
// single-threaded SVC_HANDLER objects.

template <class SH, class PR_AC> int
Acceptor<SH, PR_AC>::handle_event (HANDLE)
{
  // Create a new SVC_HANDLER.

  SVC_HANDLER *svc_handler = new SVC_HANDLER;

  // Accept connections from clients

  this->peer_acceptor_.accept (*svc_handler);

  // Register SVC_HANDLER with Reactor.

  this->reactor_->register_handler
    (svc_handler, READ_MASK);
}
```


Collaboration in the Acceptor Pattern



- `Acceptor` factory creates, connects, and activates a `Svc_Handler`

Limitations with the First Solution

- Several problems with the solution above:
  1. Advertisement and listener strategies are hardcoded
  2. Service handler creation strategy is tightly coupled to dynamic allocation and constructor-based initialization
  3. Connection acceptance strategy is hard-coded
  4. Service handler activation is not extensible and concurrency strategy is overly restrictive
- This tight coupling makes it hard to extend the Acceptor to work in other contexts

Overcoming Limitations with Patterns

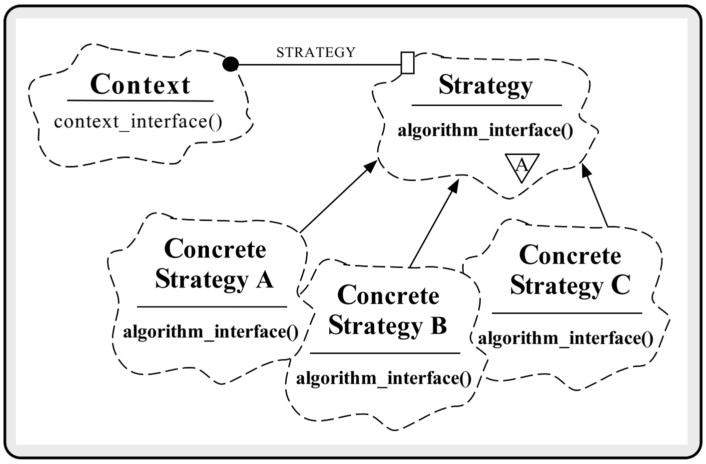
- To remove the limitations described above, we'll use various patterns to enhance our Acceptor
- Key patterns are *Strategy*, *Bridge*, *Factory Method*, and *Abstract Factory*
  - These are "tactical" patterns that are widely applicable across most application domains
- Note that the new solution does not break existing code
  - This is another important non-functional "force"...

The Strategy Pattern

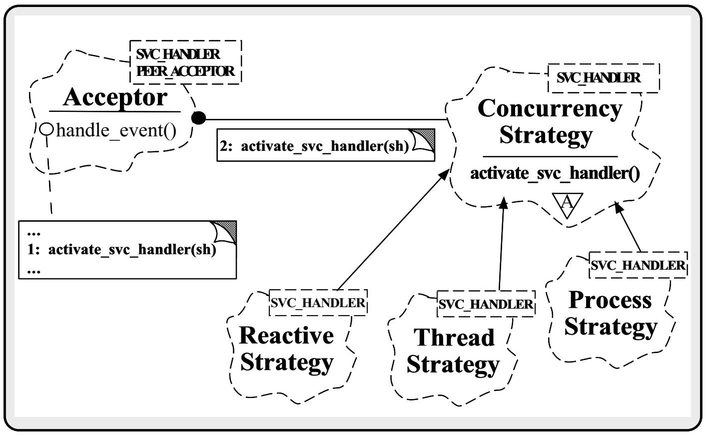
- *Intent*
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
    - ▹ Strategy lets the algorithm vary independently from clients that use it
- This pattern resolves the following force

1. *How to extend the policies for adversizing, listening, creating, accepting, and executing a service hander without modifying the core Acceptor algorithm*

Structure of the Strategy Pattern



Using the Strategy Pattern

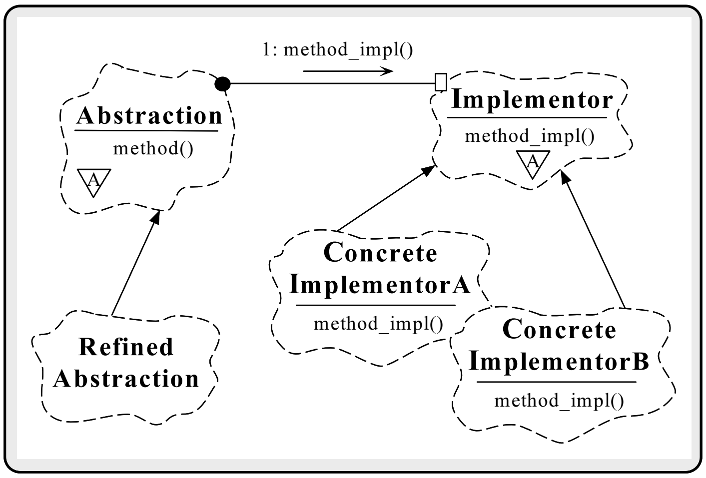


The Bridge Pattern

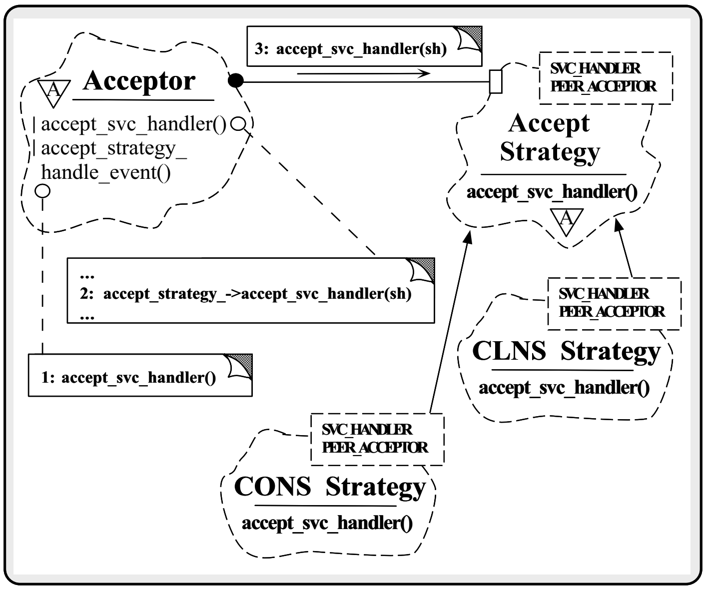
- *Intent*
  - Decouple an abstraction from its implementation so that the two can vary independently
- This pattern resolves the following force that arises when building extensible software

1. *How to provide a stable, uniform interface that is both closed and open, i.e.,*
   - *Closed* to prevent direct code changes
   - *Open* to allow extensibility

Structure of the Bridge Pattern



Using the Bridge Pattern



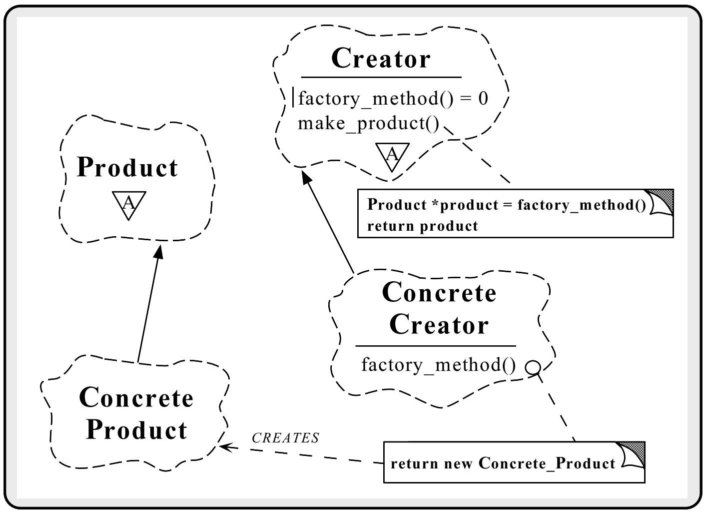
The Template Method

- *Intent*
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
    - ▹ Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- This pattern resolves the following force
  1. *How to extend the strategies for creating the check_sort algorithm*
- Template Method is an alternative for the Strategy/Bridge patterns

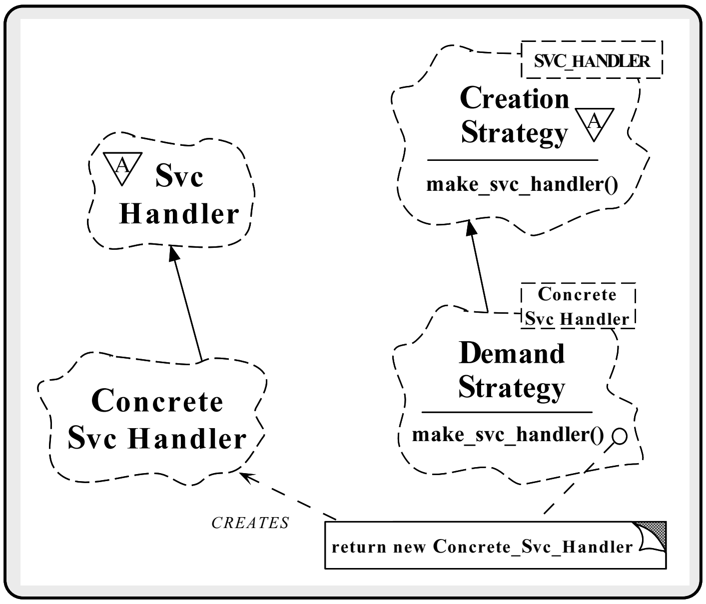
The Factory Method Pattern

- *Intent*
  - Define an interface for creating an object, but let subclasses decide which class to instantiate
    - ▹ Factory Method lets a class defer instantiation to subclasses
- This pattern resolves the following force in the Acceptor pattern:
  1. *How to extend each initialization strategy in the Acceptor pattern independently and transparently*

Structure of the Factory Method Pattern



Using the Factory Method Pattern



The Abstract Factory Pattern

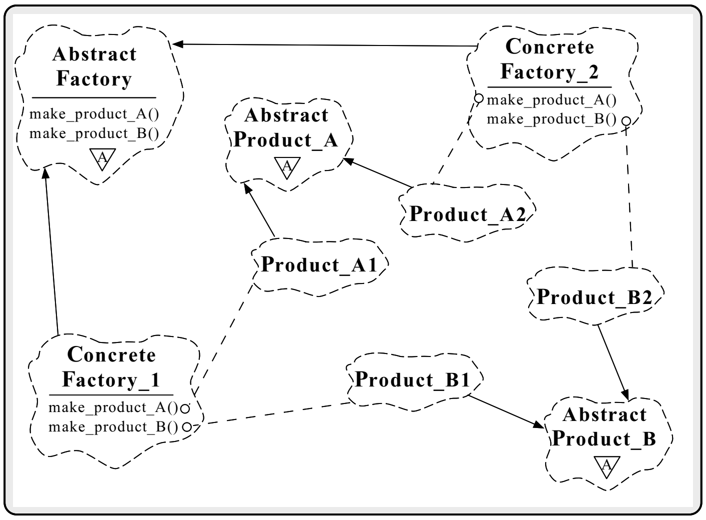
- *Intent*
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes

- This problem resolves the following forces in the Acceptor pattern:

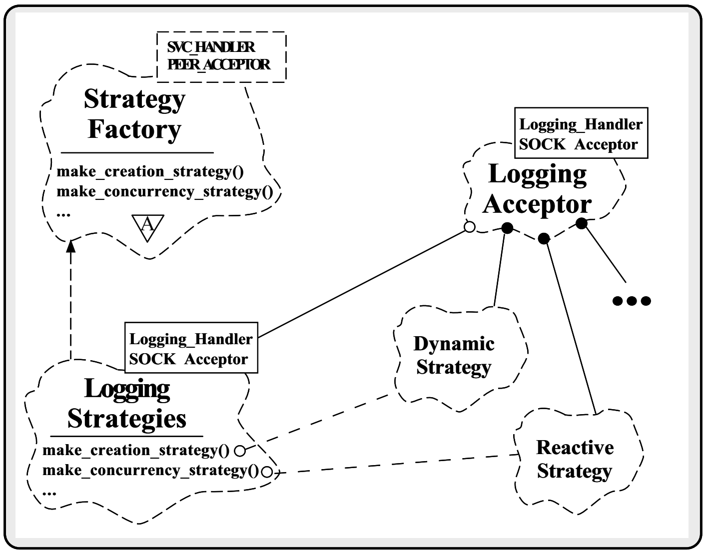
1. *How to simplify the interface to the Acceptor and keep it from having a large number of individual strategies*

2. *How to ensure that the selected strategies actually work together without conflict*

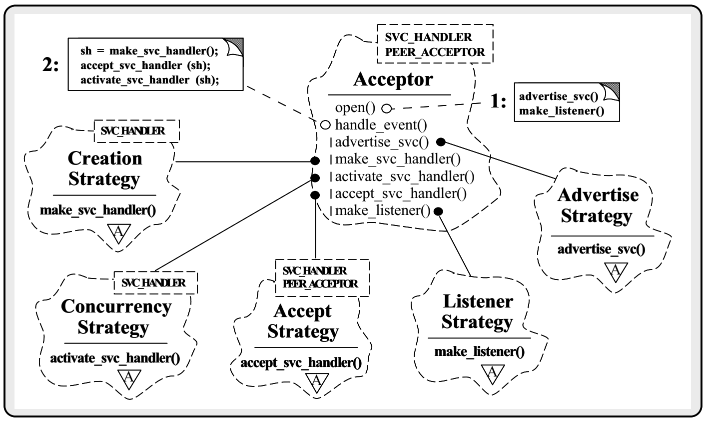
Structure of the Abstract Factory Pattern



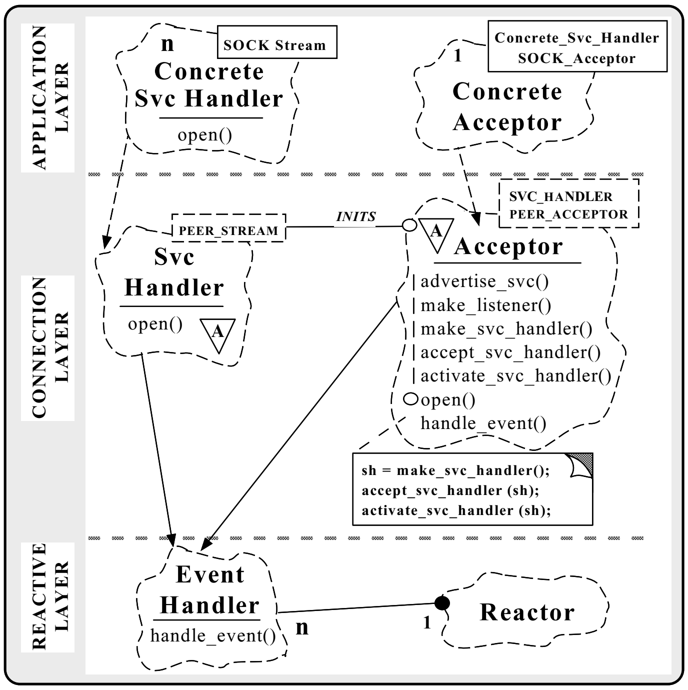
Using the Abstract Factory Pattern



Revised Acceptor Class Model



Structure of the Revised Acceptor Pattern



Advantages of New Design

- *Decouple advertisement strategy*
  - *e.g.*, use well-known ports or name server
- *Decouple listening strategy*
  - *e.g.*, using Reactor or some other demultiplexor
- *Decouple service creation strategy*
  - *e.g.*, dynamic allocation vs singleton or dynamic linking vs. static linking
- *Decouple service connection acceptance strategy*
  - *e.g.*, connection-oriented vs. connectionless
- *Decouple service concurrency strategy*
  - *e.g.*, single-threaded reactive vs. multi-threaded vs. multi-process

Revised Acceptor Class Public Interface

```
template <class SVC_HANDLER, // Type of service
          class PEER_ACCEPTOR> // Accepts connections
class Acceptor : public Event_Handler {
public:
  // = Initialization.
  virtual int open
    (const PEER_ACCEPTOR::PEER_ADDR &,
     Strategy_Factory<SVC_HANDLER, PEER_ACCEPTOR> *);

  // = Factory that creates, connects,
  // and activates SVC_HANDLER's.
  virtual int handle_event (HANDLE);

  // = Demultiplexing hooks.
  virtual HANDLE get_handle (void) const;
  virtual int handle_close (HANDLE, Reactor_Mask);
```


Revised Acceptor Class Protected and Private Interface

```
protected:
  // = Bridge methods (default behavior delegates
  // to the Strategy objects...)
  virtual int advertise_svc (const PEER_ACCEPTOR::PEER_ADDR &);
  virtual int make_listener (Event_Handler *);
  virtual SVC_HANDLER *make_svc_handler (void);
  virtual int accept_svc_handler (SVC_HANDLER *);
  virtual int activate_svc_handler (SVC_HANDLER *);

private:
  // = Strategy objects.
  Advertise_Strategy<PEER_ACCEPTOR::PEER_ADDR>
    *advertise_strat_;
  Listener_Strategy<PEER_ACCEPTOR> *listen_strat_;
  Creation_Strategy<SVC_HANDLER> *create_strat_;
  Accept_Strategy<SVC_HANDLER, PEER_ACCEPTOR>
    *accept_strat_;
  Concurrency_Strategy<SVC_HANDLER>
    *concurrency_strat_;
};
```


Strategy Factory Interface

```
template <class SVC_HANDLER, // Type of service
          class PEER_ACCEPTOR> // Accepts connections
class Strategy_Factory {
public:
  Strategy_Factory
   (Advertise_Strategy<PEER_ACCEPTOR::PEER_ADDR> *,
    Listener_Strategy<PEER_ACCEPTOR> *,
    Creation_Strategy<SVC_HANDLER> *,
    Accept_Strategy<SVC_HANDLER, PEER_ACCEPTOR> *,
    Concurrency_Strategy<SVC_HANDLER> *);

  // Factory methods called by Acceptor::open().
  Advertise_Strategy *make_advertise_strategy (void);
  Listener_Strategy *make_listener_strategy (void);
  Creation_Strategy<SVC_HANDLER>
    *make_create_strategy (void);
  Accept_Strategy<SVC_HANDLER, PEER_ACCEPTOR>
    *make_accept_strategy (void);
  Concurrency_Strategy<SVC_HANDLER>
    *make_concurrency_strategy (void);
```


Acceptor Class Implementation

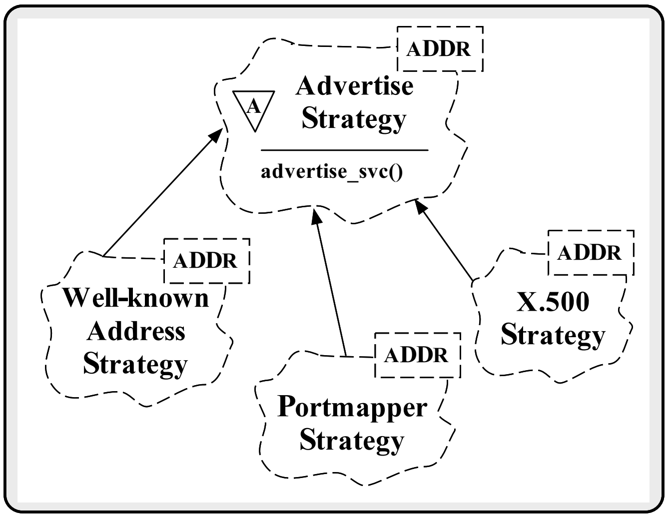
```
// Shorthand names.
#define SH SVC_HANDLER
#define PA_AC PEER_ACCEPTOR
#define PA_AD PEER_ACCEPTOR::PEER_ADDR
// Initialize the Acceptor.

template <class SH, class PR_AC> int
Acceptor<SH, PR_AC>::open
  (const PR_AC::PEER_ADDR &local_addr,
   Strategy_Factory<SH, PR_AC> *strat_fact)
{
  // Initialize the strategies.
  this->create_strat_ =
    strat_fact->make_creation_strategy ();
  this->accept_strat_ =
    strat_fact->make_accept_strategy ();
  this->concurrency_strat_ =
    strat_fact->make_concurrency_strategy ();
  this->listen_strat_ =
    strat_fact->make_listener_strategy ();
  this->advertise_strat_ =
    strat_fact->make_advertise_strategy ();

  // Advertise the service.
  this->advertise_svc (local_addr);

  // Make a listener.
  this->make_listener (this);
}
```

```
// Implements the strategy for creating, connecting,
// and activating new SVC_HANDLER objects.

template <class SH, class PR_AC> int
Acceptor<SH, PR_AC>::handle_event (HANDLE)
{
  // Create a new SVC_HANDLER.

  SH *svc_handler = this->make_svc_handler ();

  // Accept connection from client.

  this->accept_svc_handler (svc_handler);

  // Activate SVC_HANDLER.

  this->activate_svc_handler (svc_handler);
}
```



```
// Bridge method for creating a service handler.

template <class SH, class PA> SH *
Acceptor<SH, PA>::make_svc_handler (void)
{
  return this->creation_strategy_->make_svc_handler ();
}

// Accept connections from clients (can be overridden).

template <class SH, class PA> int
Acceptor<SH, PA>::accept_svc_handler (SH *svc_handler)
{
  return this->accept_strategy_->
    accept_svc_handler (svc_handler);
}

// Activate the service handler (can be overridden).

template <class SH, class PA> int
Acceptor<SH, PA>::activate_svc_handler (SH *svc_handler)
{
  return this->concurrency_strategy_->
    activate_svc_handler (svc_handler);
}
```

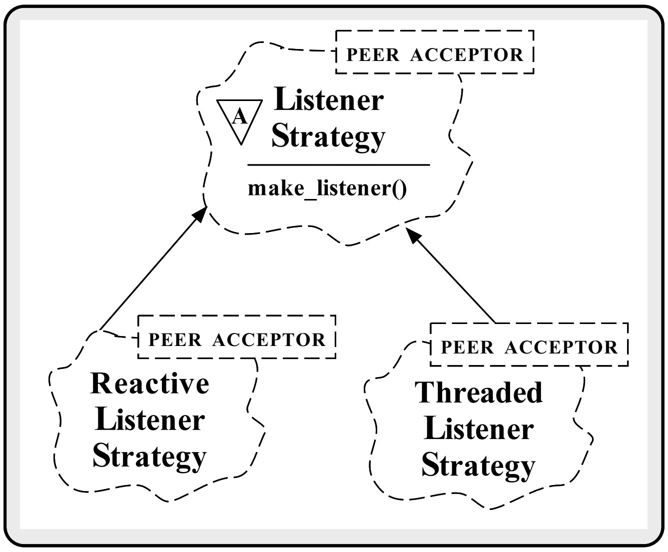

Advertisement Strategies



Advertisement Strategy Implementations

```
template <class PA_AD>
Well_Known_Address<PA_AD>::advertise_svc
  (const PA_AD &local_addr)
{
  // Advertise the IP port and IP address.

  this->advertise_endpoint (local_addr);
}

template <class PA_AD>
Port_Mapper<PA_AD>::advertise_svc
  (const PA_AD &local_addr)
{
  // Advertise using the portmapper

  // ...
}
```

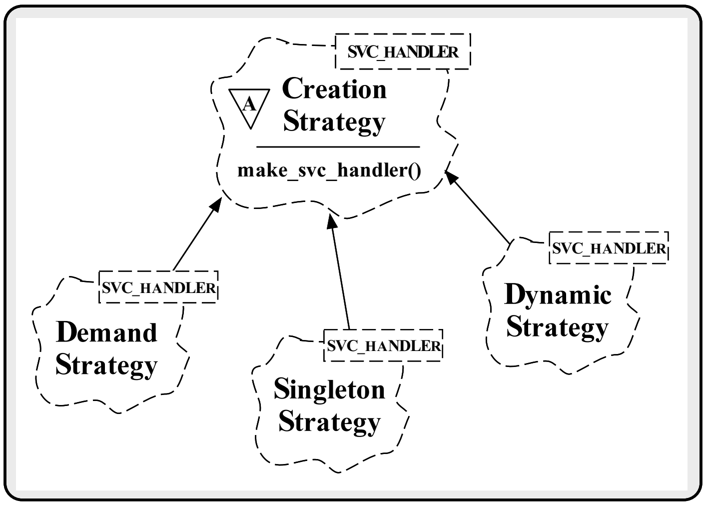
Listener Strategies



Listener Strategy Implementations

```
// Cache a Reactor.
template <class PA_AC>
Reactive_Listener_Strategy<PA_AC>::Listener_Strategy
  (Reactor *r)
  : reactor_ (r)
{
}

// Register with a Reactor
int
Reactive_Listener_Strategy::make_listener
  (Event_Handler *h)
{
  this->handle_ = h->get_handle ();
  this->reactor_->register_handler (h, READ_MASK);
}

// Remove from Reactor
Reactive_Listener_Strategy::~Reactive_Listener_Strategy
  (void)
{
  this->reactor_->remove_handler (this->handle_);
}
```

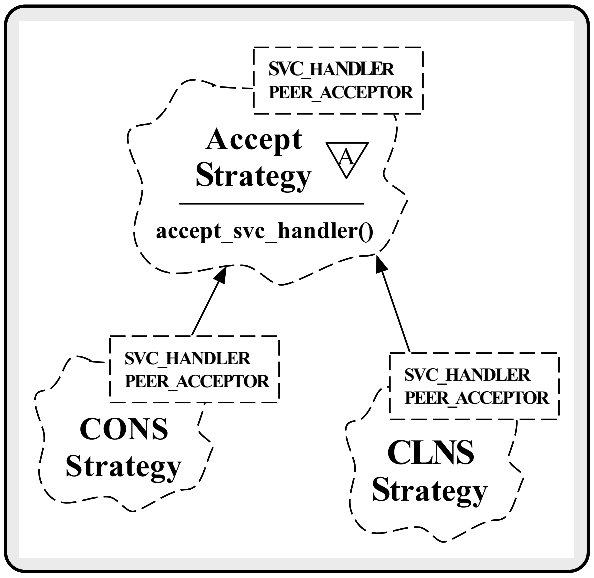

SVC_HANDLER Creation Strategies



Creation Strategy Implementations

```
// Make a Singleton SVC_HANDLER.
template <class SH> SH *
Singleton_Strategy<SH>::make_svc_handler (void) {
  if (this->svc_handler_ == 0)
    this->svc_handler_ = new SH;
  return this->svc_handler_; // Pre-cached...
}

// Make a new SVC_HANDLER on-demand.
template <class SH> SH *
Demand_Strategy<SH>::make_svc_handler (void) {
  return new SH;
}

// Make a SVC_HANDLER by dynamically linking it.
template <class SH> SH *
Dynamic_Strategy<SH>::make_svc_handler (void) {
  // Open the shared library.
  void *handle = (void *) dlopen (this->shared_library_);

  // Extract the factory function.
  SH *(*factory)(void) = (SH *(*)(void))
    dlsym (handle, this->factory_function_);

  // Call factory function to get new SVC_Handler.
  return (*factory)();
}
```

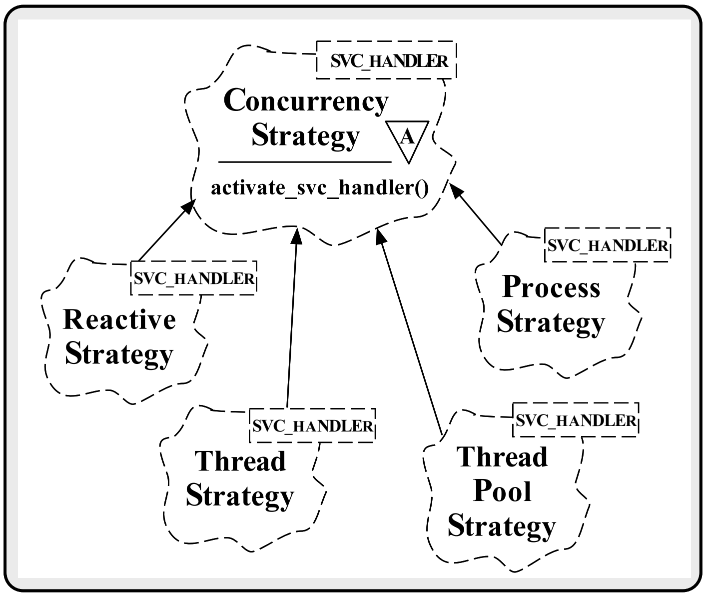
SVC_HANDLER Connection Acceptance Strategies



Connection Acceptance Strategy Implementation

```
// Delegate to the accept() method of the PEER_ACCEPTOR.

template <class SH, class PR_AC> int
Accept_Strategy<SH, PR_AC>::accept_svc_handler
  (SH *svc_handler)
{
  return this->peer_acceptor_.accept (*svc_handler);
}

// Implement "stateless connection" strategy

template <class SH, class PR_AC> int
CLNS_Strategy<SH, PR_AC>::accept_svc_handler
  (SH *svc_handler)
{
  // ...
}
```


SVC_HANDLER Concurrency Strategies



Concurrency Strategy Implementations

```
// Activate the SVC_HANDLER by calling it's open() method.

template <class SH> int
Reactive_Strategy<SH>::activate_svc_handler
  (SH *svc_handler, void *arg)
{
  // Delegate control to the application-specific service
  // handler.

  return svc_handler->open (arg);
}

// Activate the SVC_HANDLER by first calling its open()
// method and then calling its activate() method to turn
// it into an active object.

template <class SH> int
Thread_Strategy<SH>::activate_svc_handler
  (SH *svc_handler, void *arg)
{
  // Initialize SVC_HANDLER.
  svc_handler->open (arg);

  // Turn the svc_handler into an active object (if it isn't
  // already one as a result of the first activation...)
  return svc_handler->activate ();
}
```

New Acceptor Use-Case

```
class Logging_Handler :
  public Svc_Handler<SOCK_Stream>
{
public:
  // Initialize handler.
  virtual int open (void *);

  // Obtain HANDLE.
  virtual HANDLE get_handle (void) const;

  // Called back to process logging records.
  virtual int handle_event (HANDLE);
};

typedef Acceptor<Logging_Handler, SOCK_Acceptor>
        Logging_Acceptor;

int main (void)
{
  Reactor reactor;
  Logging_Acceptor acceptor;
  Logging_Strategy strategy (/* ... */, &reactor);

  acceptor.open (LOGGER_PORT, &strategy);

  for (;;)
    reactor.handle_events ();
}
```


Concluding Remarks

- Design patterns can alleviate coupling and unnecessary complexity in communication software
- Patterns do not exist in isolation
  - Instead, they form “families of patterns” that build upon each other
- Both *strategic* and *tactical* patterns are necessary to build flexible and extensible solutions