Annotated Design Patterns Selections

Design Patterns, Gamma et. al, Addison Wesley, 1995

Jim Fawcett

CSE776 – Design Patterns

Fall 2016

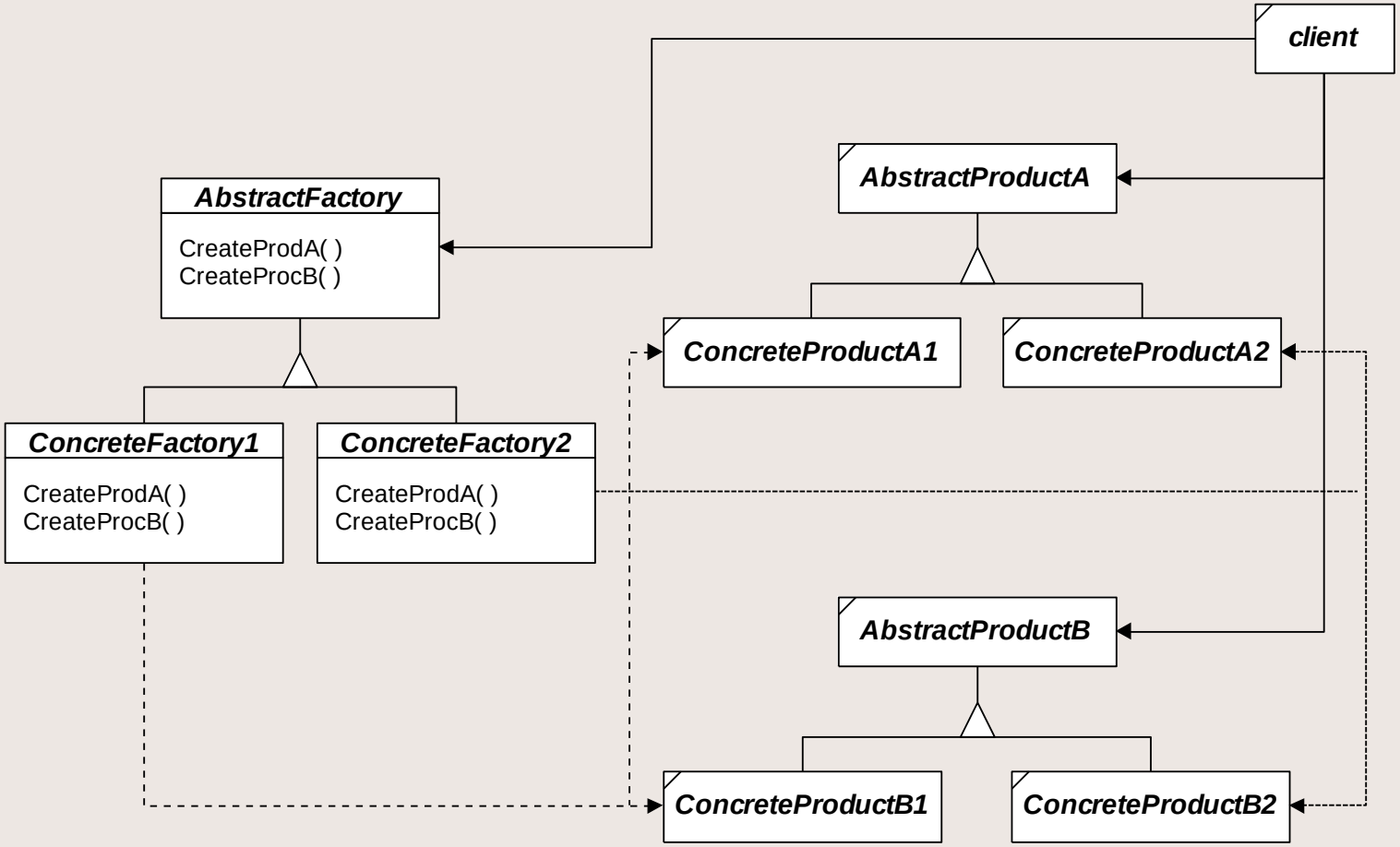
Contents

- Abstract Factory
- Builder — Parser::ConfigParser
- Factory Method
- Prototype
- Singleton — Parser::Repository
- Adapter
- Bridge
- Composite — XmlDocument::XElement
- Decorator
- Facade — XmlDocument
- Flyweight
- Proxy — WCF Proxy
- Chain of Responsibility — Parser::IRule
- Command — Parser::IRule->IAction
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State — Tokenizer::ConsumeState
- Strategy — Parser::IRule
- Template Method
- Visitor

Abstract Factory

- Intent:
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Applicability – Use this pattern when:
  - A system should be independent of how its products are created, composed, and represented.
  - A system should be configured with one of multiple families of products.
  - A family of related product objects is designed to be used together, and you need to enforce this constraint.
  - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

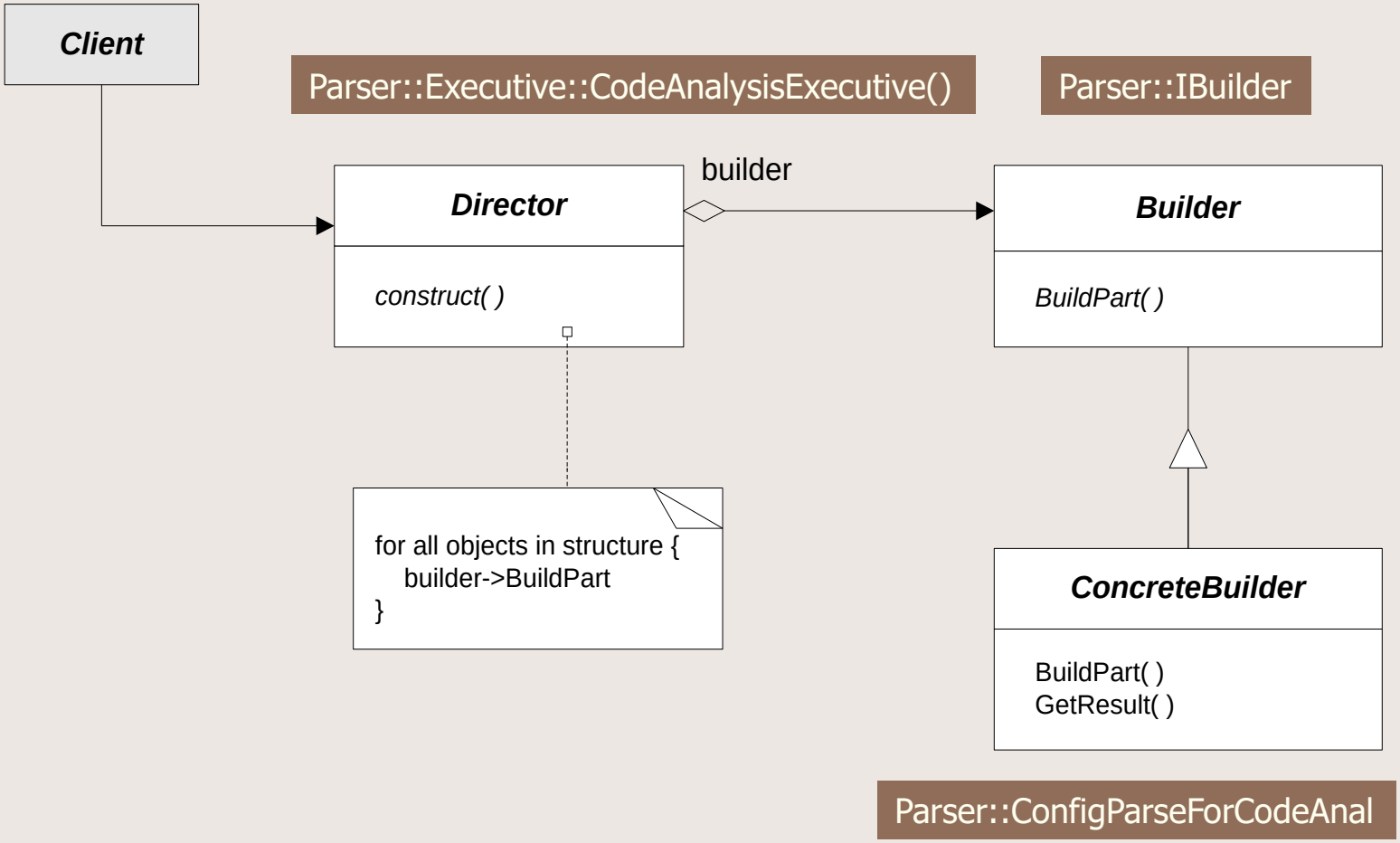
Abstract Factory Structure



Builder

- Intent
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Applicability – use this pattern when:
  - The algorithm for creating a complex object should be inde-pendent of the parts that make up the object and how they're assembled.
  - The construction process must allow different representations for the object that's constructed.

Builder Structure



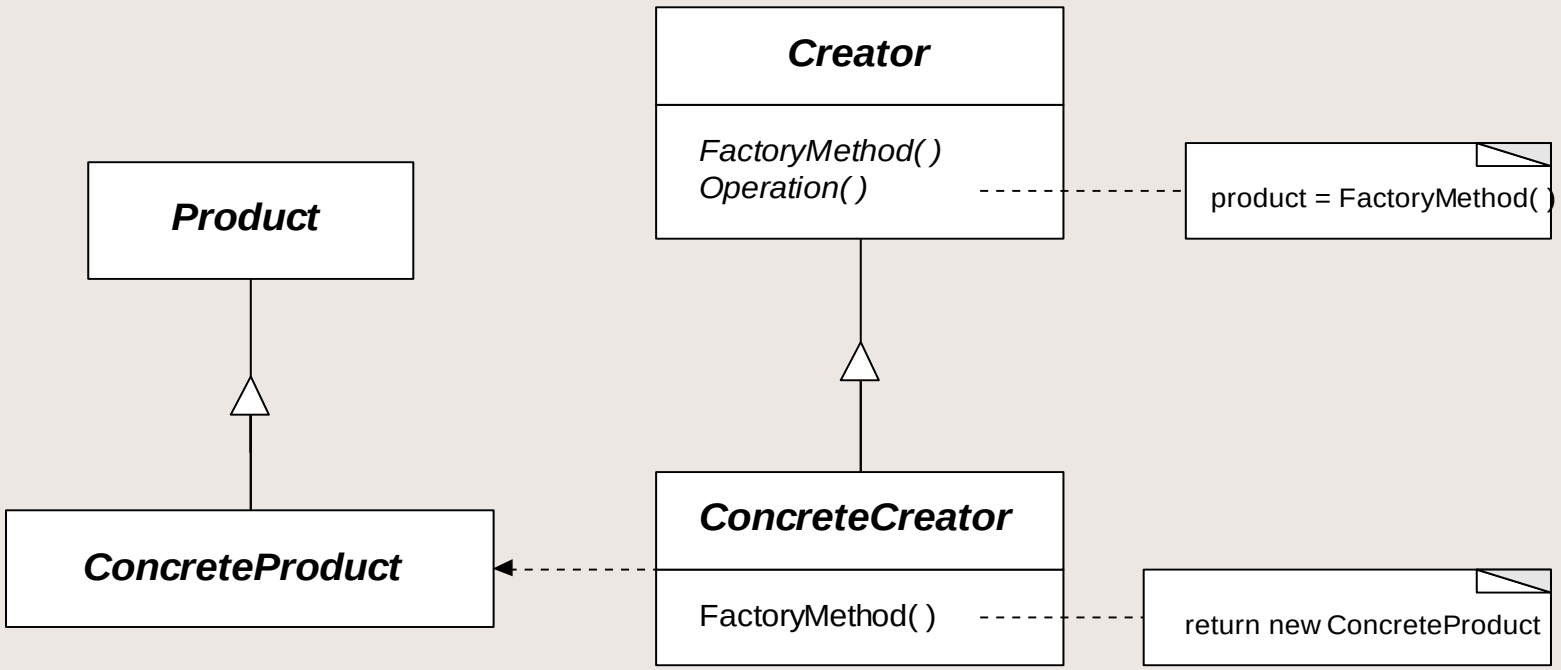
```
namespace CodeAnalysis
{
  ////////////////////////////////////////////////////////////
  // build parser that writes its output to console

  class ConfigParseForCodeAnal : IBuilder
  {
  public:
    ConfigParseForCodeAnal() : pIn(nullptr) {};
    ~ConfigParseForCodeAnal();
    bool Attach(const std::string& name, bool isFile = true);
    Parser* Build();
```

```
  CodeAnalysisExecutive::CodeAnalysisExecutive()
  {
    pParser_ = configure_.Build();
    if (pParser_ == nullptr)
    {
      throw std::exception("couldn't create parser");
    }
    pRepo_ = Repository::getInstance();
  }
```

Factory Method

- 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.
- Applicability – use Factory Method when:
  - A class can't anticipate the class of objects it must create.
  - A class wants its subclasses to specify the objects it creates.
  - Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

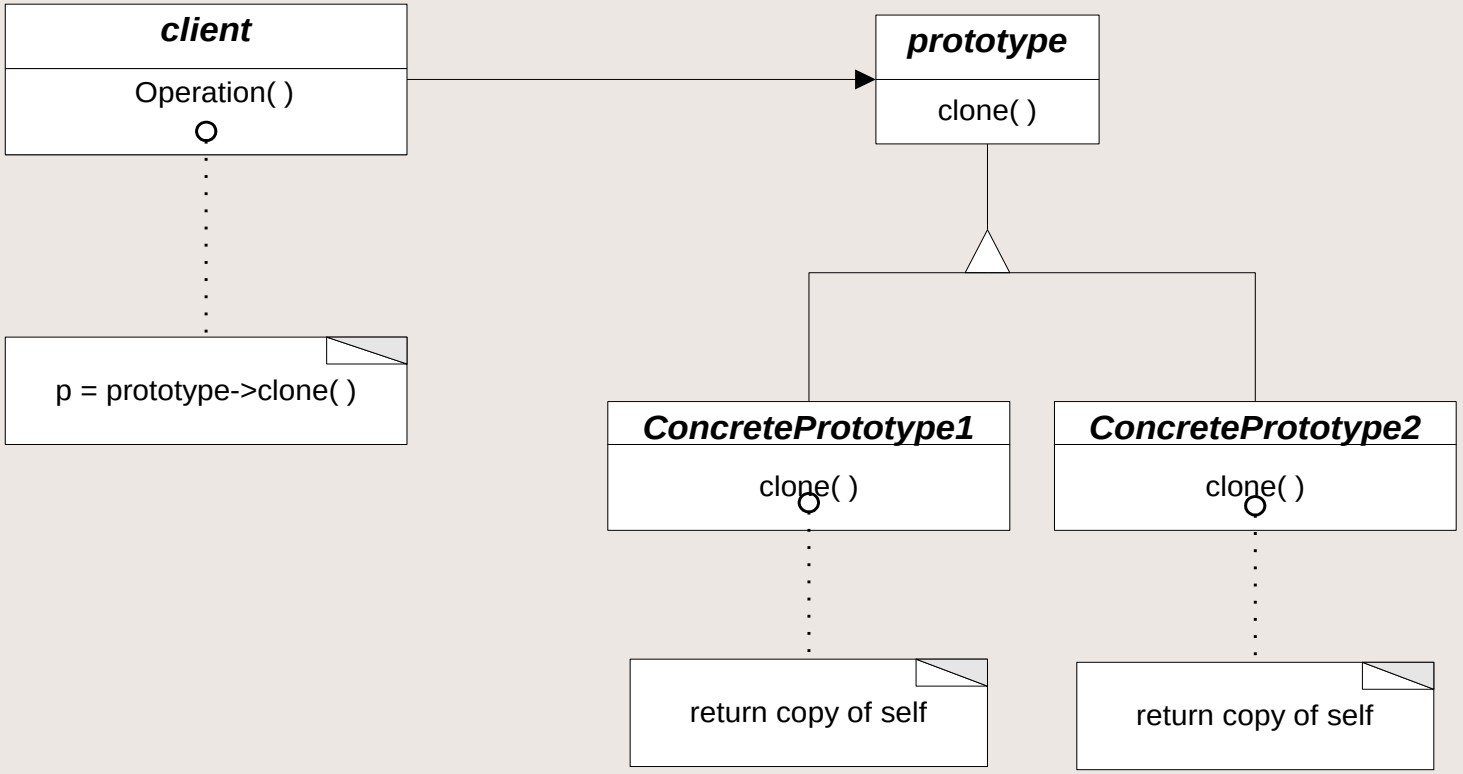
Factory Method Structure



Prototype

- Intent
  - Specify the kinds of objects to crate using a prototypical instance, and create new objects by copying this prototype.
- Applicability – Use this pattern when:
  - A system should be independent of how its products are created, composed, and represented; and:
    - When the classes to instantiate are specified at run-time; or
    - To avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
    - When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

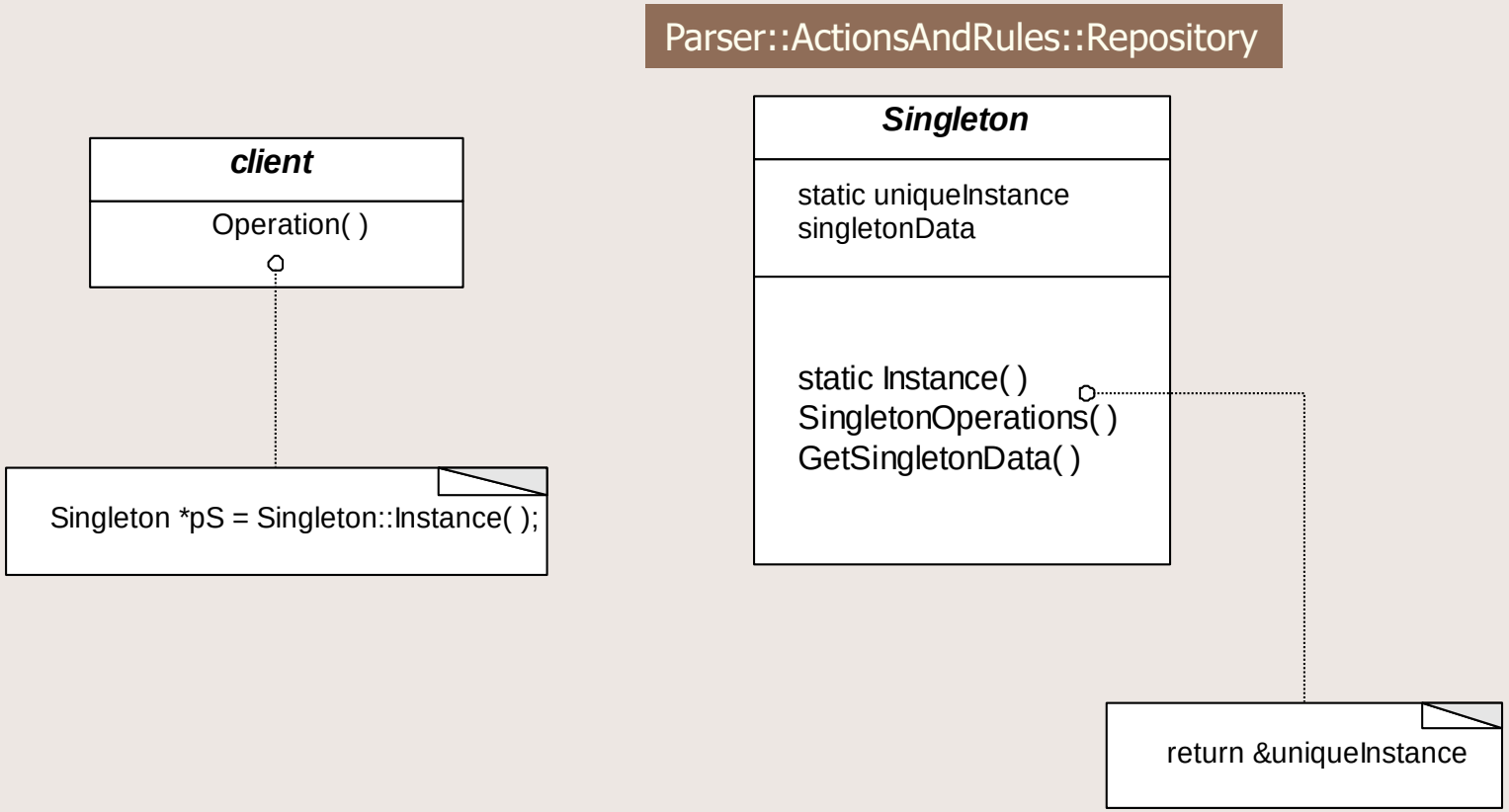
Prototype Structure



Singleton

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it.
- Applicability – use the Singleton pattern when:
  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Singleton Structure



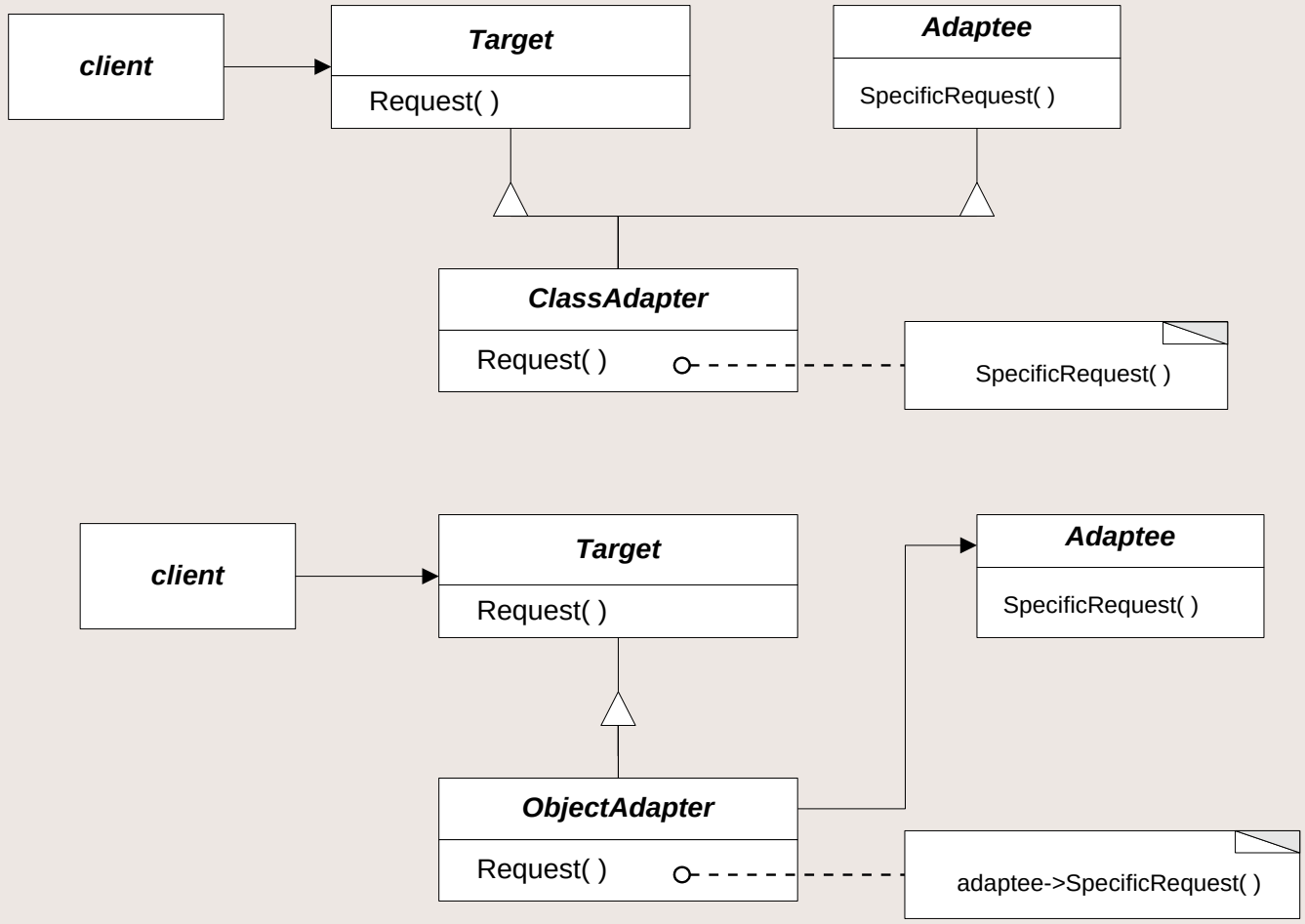
```
class Repository  // application specific
{
public:
  using Rslt = Logging::StaticLogger<0>;  // use for application results
  using Demo = Logging::StaticLogger<1>;  // use for demonstrations of processing
  using Dbug = Logging::StaticLogger<2>;  // use for debug output
  using Package = std::string;
  using Path = std::string;

private:
  Language language_ = Language::Cpp;
  Path path_;
  ScopeStack<ASTNode*> stack;
  AbstrSynTree ast;
  ASTNode* pGlobalScope;
  Package package_;
  Scanner::Toker* p_Toker;
  Access currentAccess_ = Access::publ;
  static Repository* instance;
```

```
  Access& currentAccess() { return currentAccess_; }

  static Repository* getInstance() { return instance; }

  ScopeStack<ASTNode*>& scopeStack() { return stack; }

  AbstrSynTree& AST() { return ast; }
```

Adapter

- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Applicability – use the Adapter when:
  - You want to use an existing class, and its interface does not match the one you need.
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
  - (object adapter only) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

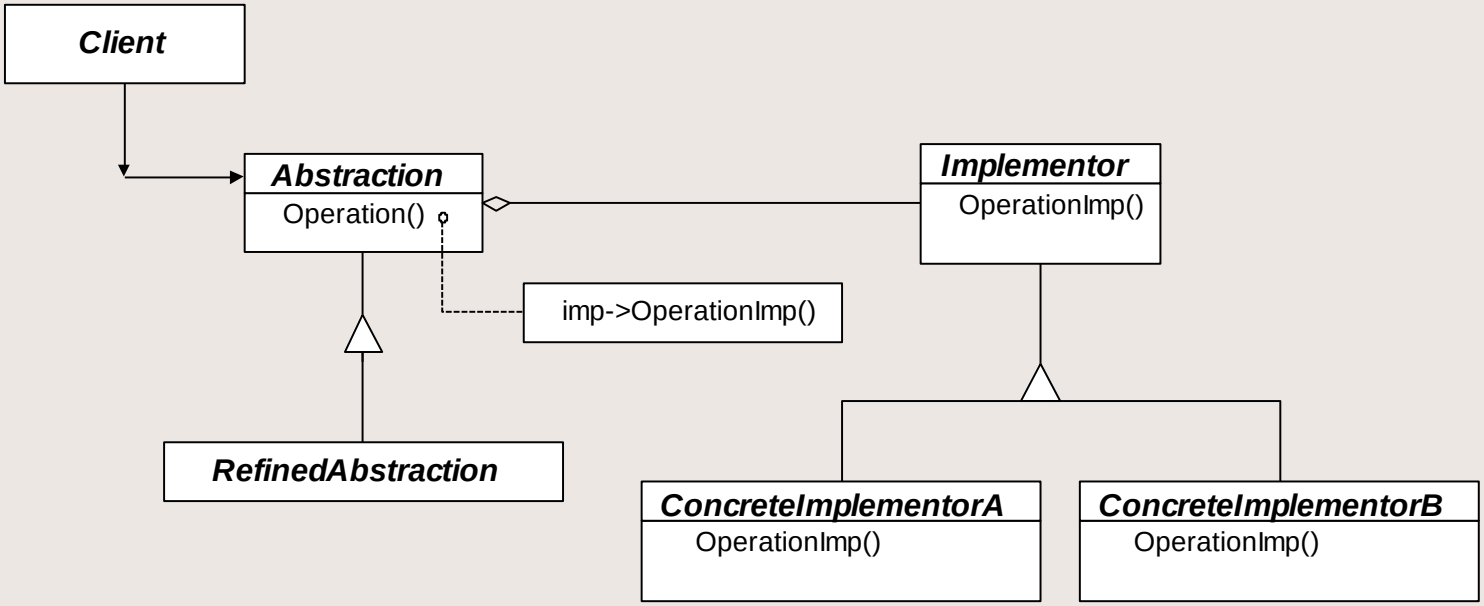
Adapter Structure



Bridge

- Intent
  - Decouple an abstraction from its implementation so that the two can vary independently.
- Applicability – use Bridge when:
  - You want to avoid a permanent binding between an abstraction and its implemen-tation. This might be the case, for example, when the implementation must be selected or switched at run-time.
  - Both the abstractions and their implementations should be extensible by subclass-ing. In the case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
  - Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
  - You want to hide the implementation of an abstraction completely from clients.
  - You have a proliferation of classes. Such a class hierarchy may indicate the need for splitting an object into two parts.
  - You want to share an implementation among multiple objects, perhaps using reference counting, and this fact should be hidden from the client.

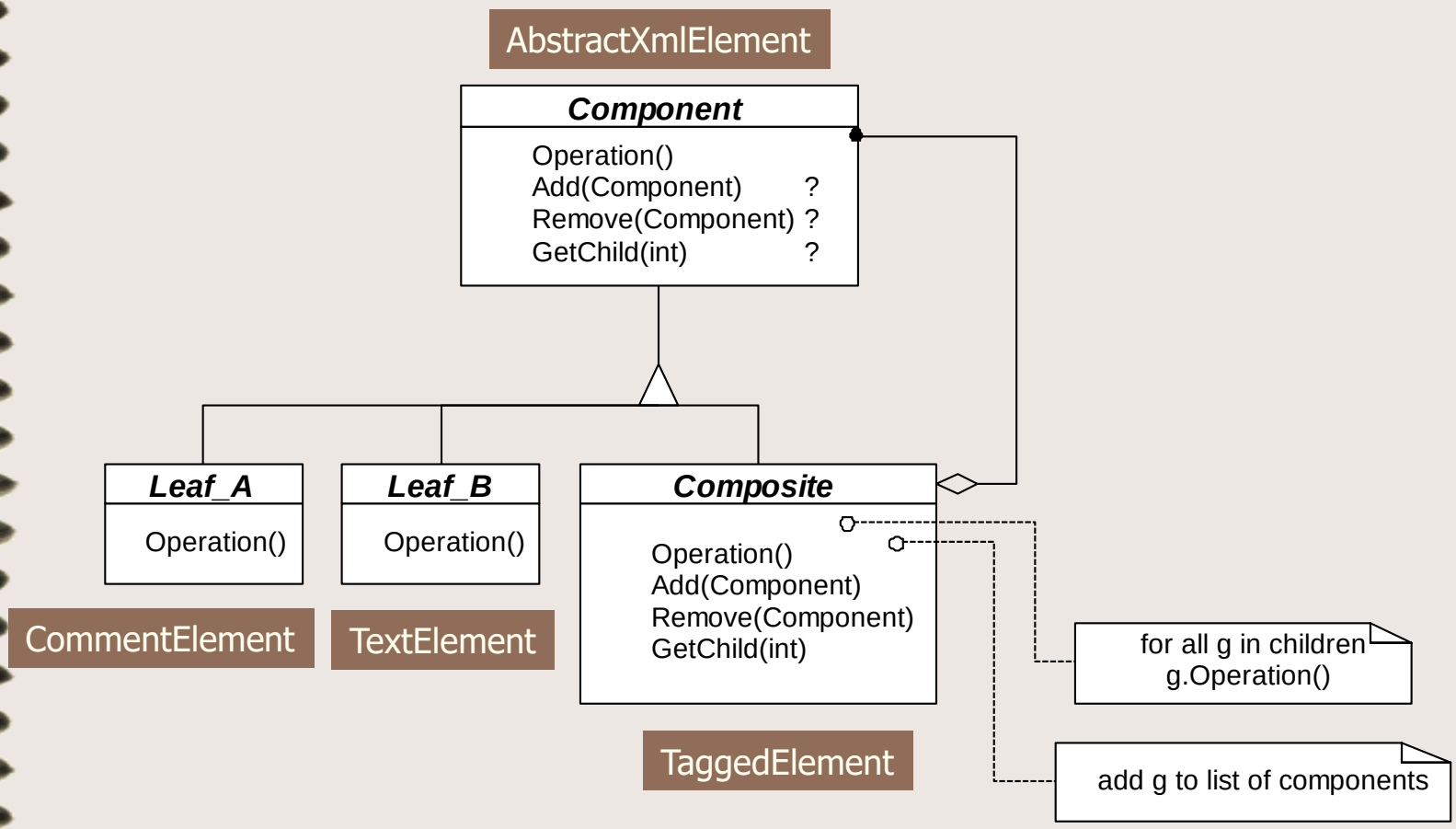
Bridge Structure



Composite

- Intent
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Applicability – use composite when:
  - You want to represent part-whole hierarchies of objects.
  - You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite Structure



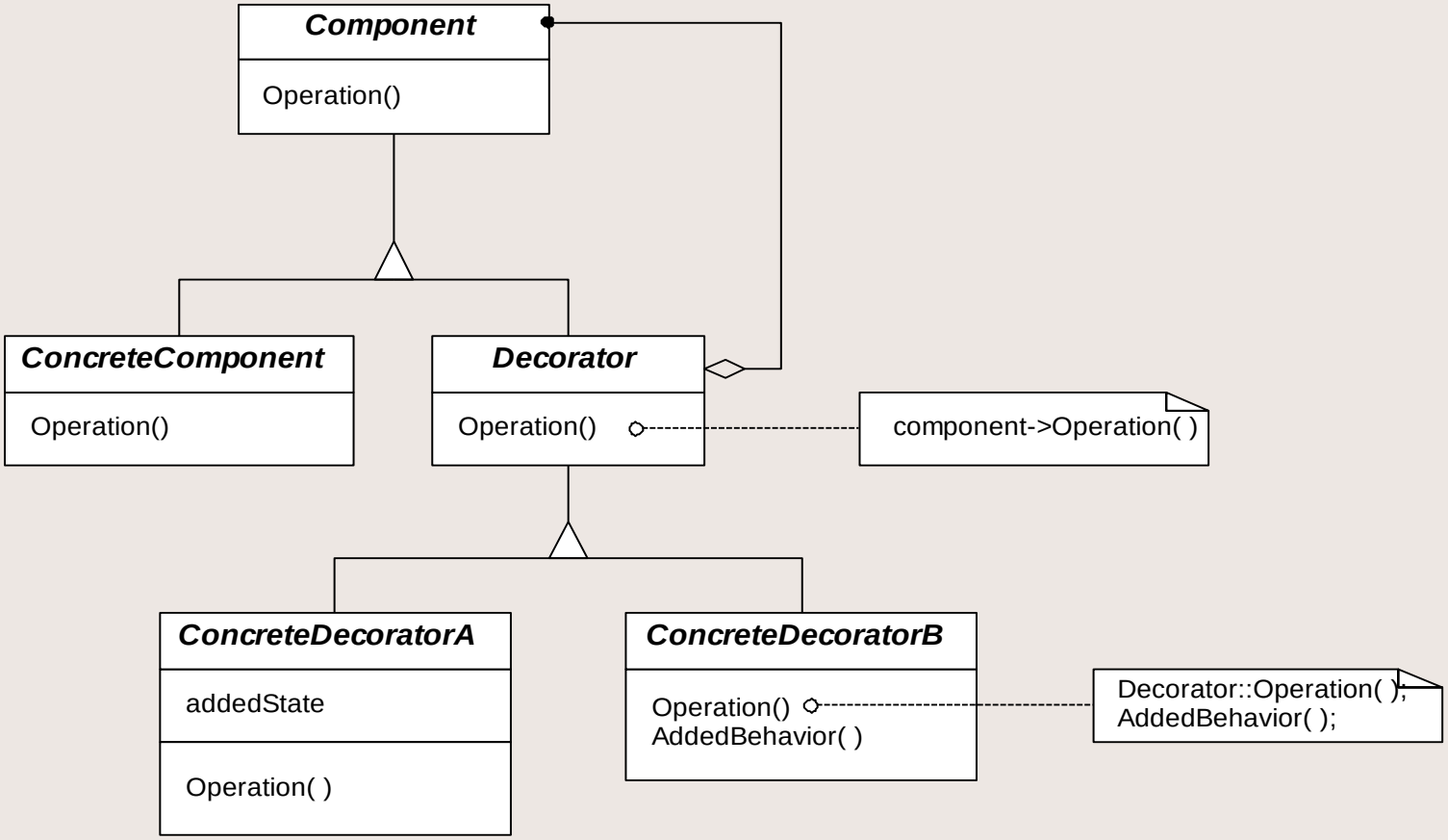
```
class AbstractXmlElement
{
public:
  using sPtr = std::shared_ptr < AbstractXmlElement > ;

  virtual bool addChild(std::shared_ptr<AbstractXmlElement> pChild);
  virtual bool removeChild(std::shared_ptr<AbstractXmlElement> pChild);
  virtual bool addAttrib(const std::string& name, const std::string& value);
  virtual bool removeAttrib(const std::string& name);
  virtual std::vector<sPtr> children();
  virtual std::string tag() { return ""; }
  virtual std::string value() = 0;
  virtual std::string toString() = 0;
  virtual
protected:
  static s
  static s
};
```

```
class TextElement : public AbstractXmlElement
{
public:
  TextElement(const std::string& text) : text_(text) {}
  virtual ~TextElement() {}
  TextElement(const TextElement& te) = delete;
  TextElement& operator=(const TextElement& te) = delete;
  virtual std::string value();
  virtual std::string toString();
private:
  std::string text_;
};
```

Decorator

- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Applicability – use Decorator:
  - To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
  - For responsibilities that can be withdrawn.
  - When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

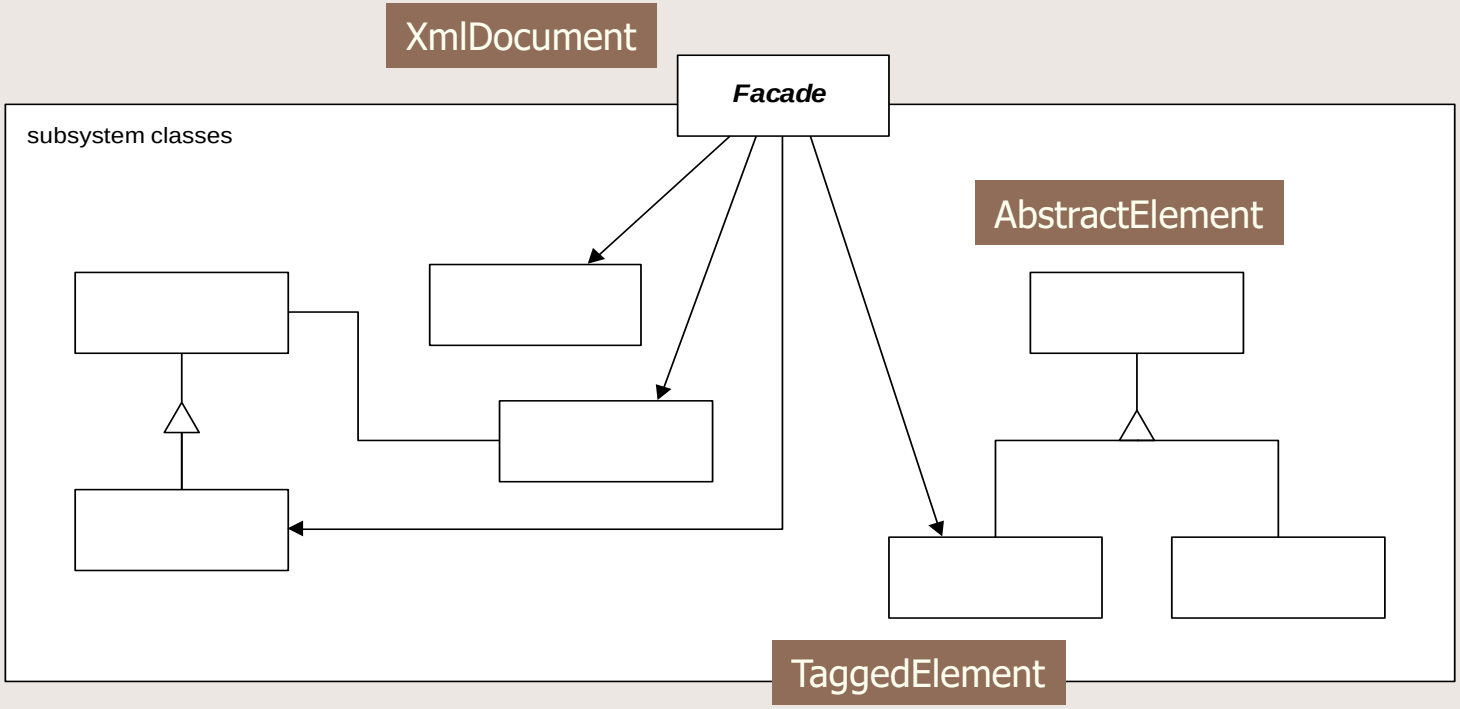
Decorator Structure



Facade

- Intent
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Applicability – use Facade when:
  - You want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it. A façade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.
  - There are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
  - You want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the depen-dencies between them by making them communicate with each other solely through their facades.

Facade Structure

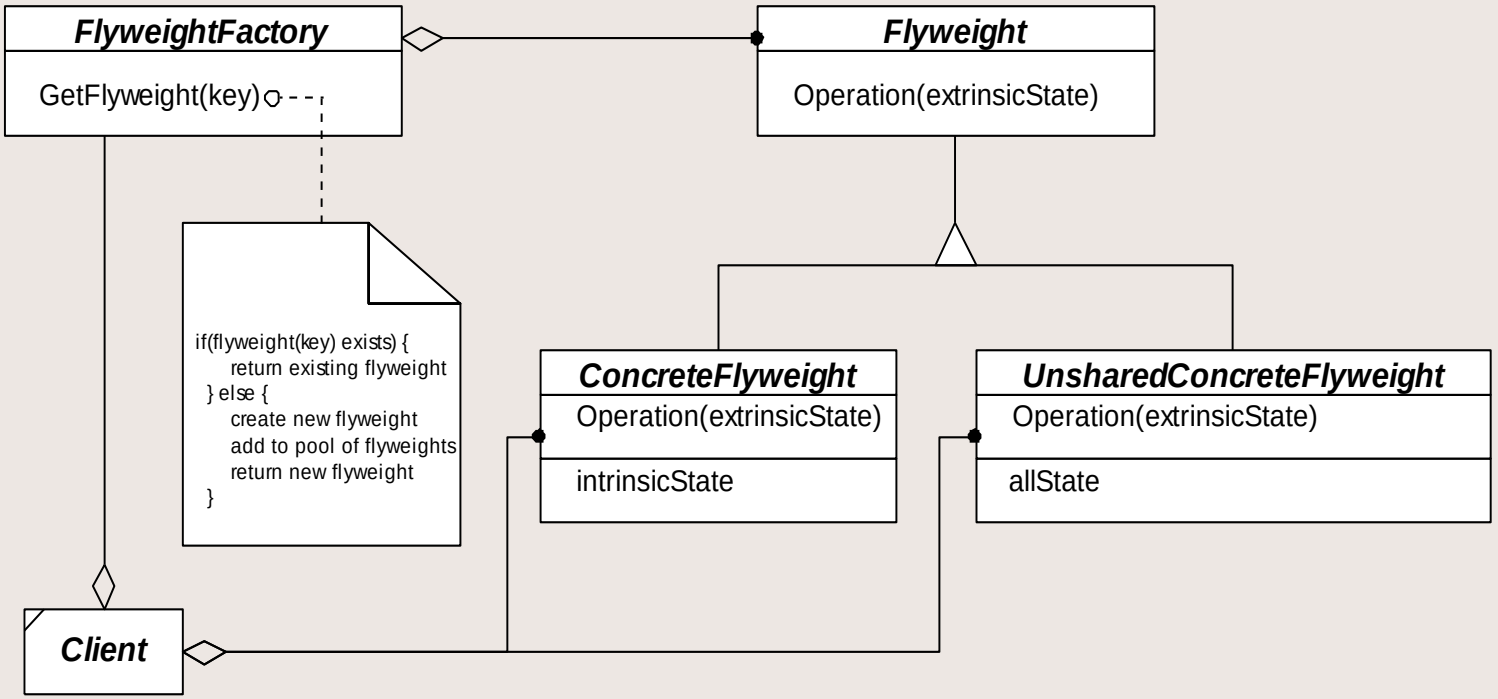


```
class XmlDocument
{
public:
  using sPtr = std::shared_ptr < AbstractXmlElement > ;
  enum sourceType { file, str };

  // construction and assignment

  XmlDocument(sPtr pRoot = nullptr) : pDocElement_(pRoot) {}
  XmlDocument(const std::string& src, sourceType srcType=str);
  XmlDocument(const XmlDocument& doc) = delete;
  XmlDocument(XmlDocument&& doc);
  XmlDocument& operator=(const XmlDocument& doc) = delete;
  XmlDocument& operator=(XmlDocument&& doc);

  // access to docElement and XML root

  std::shared_ptr<AbstractXmlElement>& docElement() { return pDocElement_;  }
  std::shared_ptr<AbstractXmlElement> xmlRoot();
  bool xmlRoot(sPtr pRoot);

  // queries return XmlDocument references so they can be chained, e.g., doc.element("foobar").descendents();

  XmlDocument& element(const std::string& tag);           // found_[0] contains first element (DFS order) with tag
  XmlDocument& elements(const std::string& tag);          // found_ contains all children of first element with tag
  XmlDocument& descendents(const std::string& tag = "");  // found_ contains descendents of prior found_[0]
  std::vector<sPtr> select();                             // returns found_.  Uses std::move(found_) to clear found_
  bool find(const std::string& tag, sPtr pElem, bool findall = true);

  size_t size();
  std::string toString();
  template<typename CallObj>
  void DFS(sPtr pElem, CallObj& co);
private:
  sPtr pDocElement_;         // AST that holds procInstr, comments, XML root, and more comments
  std::vector<sPtr> found_;  // query results
```

Flyweight

- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Applicability – apply the Flyweight pattern when *all* of the following are true:
  - An application uses a large number of objects.
  - Storage costs are high because of the sheer quantity of objects.
  - Most object sate can be made extrinsic.
  - Many groups o objects may be replaced by relatively few shared objects once extrinsic state is removed.
  - Application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for concept-ually distinct objects.

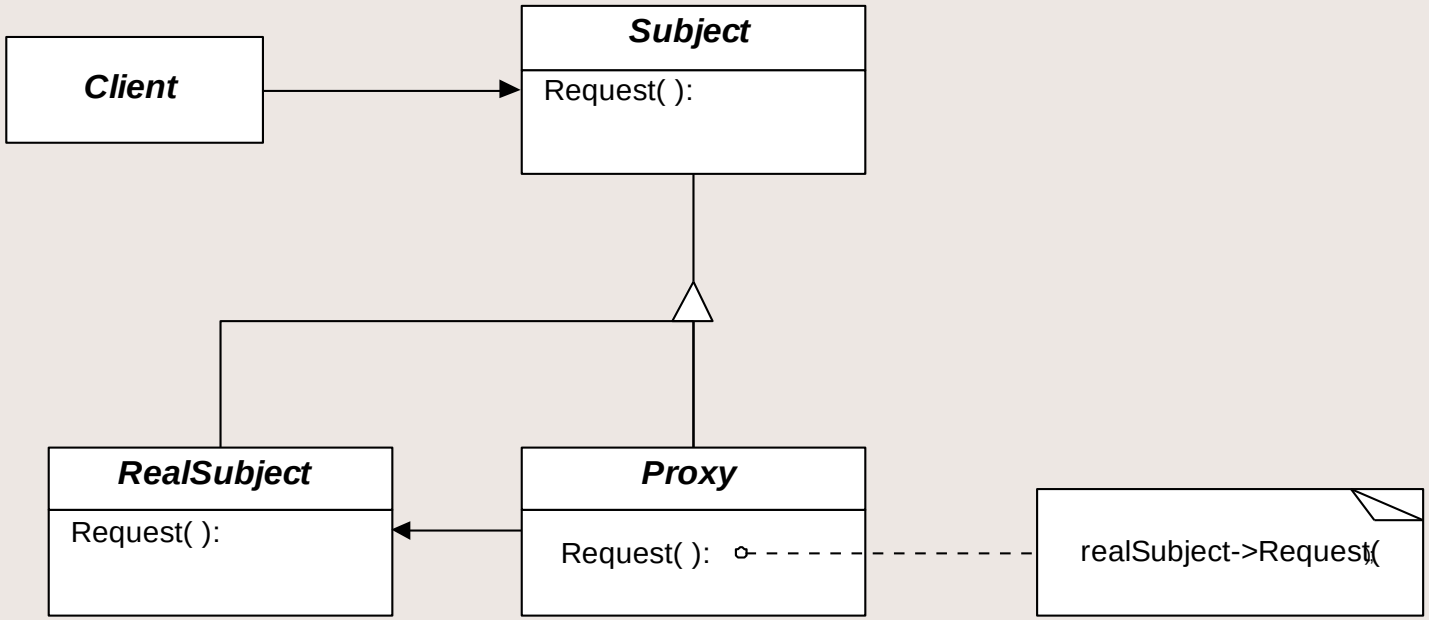
Flyweight Structure



Proxy

- Intent
  - Provide a surrogate or placeholder for another object to control access to it.
- Applicability
  - A remote proxy provides a local representative for an object in a different address space.
  - A virtual proxy creates expensive objects on demand.
  - A protection proxy controls access to the original object Protection proxies are useful when objects should have different access rights.
  - A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed, e.g., reference counting, loading persistent objects when referenced, and managing object locks when referencing the real object in a multi-threaded environment.

Proxy Structure



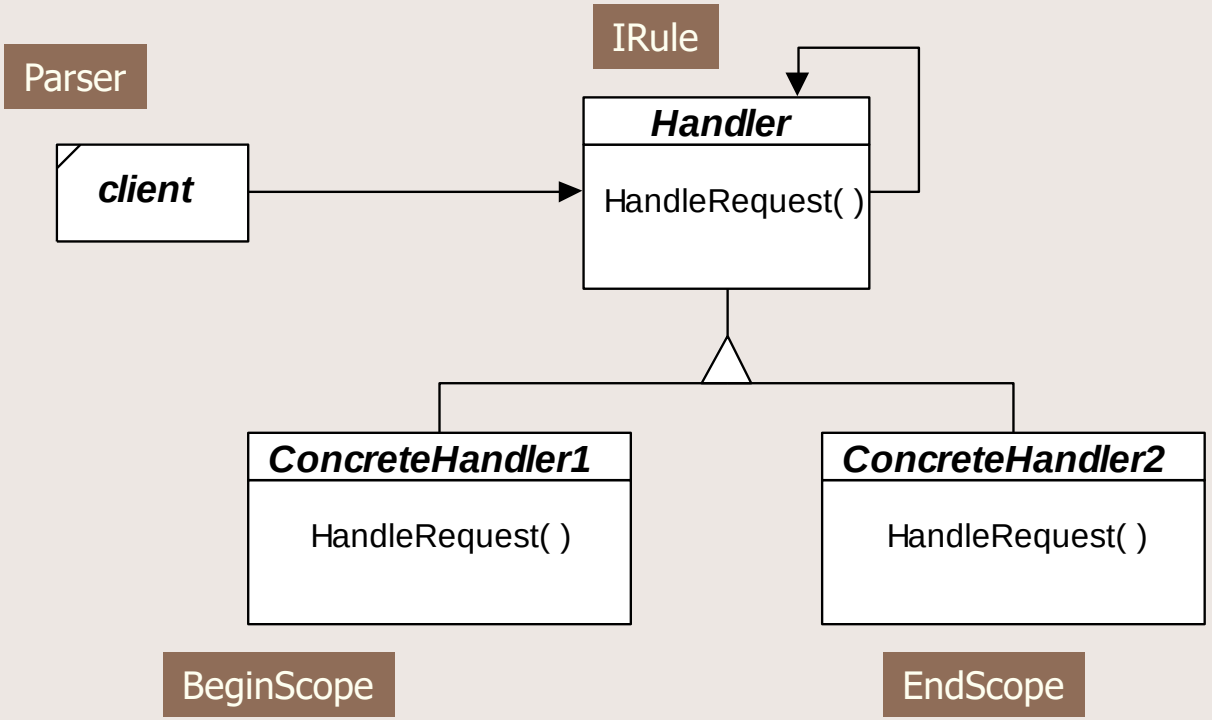
WCF BasicHttp Proxy

```
//----< returns a proxy of type T >----------------------------
   /*
    * Proxy is a local object that transparently uses
    * communication channel to converse with server
    * using contract C.
    *
    * Channel doesn't attempt to connect to server
    * until first service call.
    */
   static C CreateProxy<C>(string url)
   {
     BasicHttpBinding binding = new BasicHttpBinding();
     EndpointAddress address = new EndpointAddress(url);
     ChannelFactory<C> factory =
       new ChannelFactory<C>(binding, address);
     return factory.CreateChannel();
   }
```

Chain of Responsibility

- Intent
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Applicability – use Chain of Responsibility when:
  - More than one object may handle a request, and the handler isn't known a'priori. The handler should by ascertained automatically.
  - You want to issue a request to one of several objects without specifying the receiver explicitly.
  - The set of objects that can handle a request should be specified dynamically.

Chain of Responsibility Structure



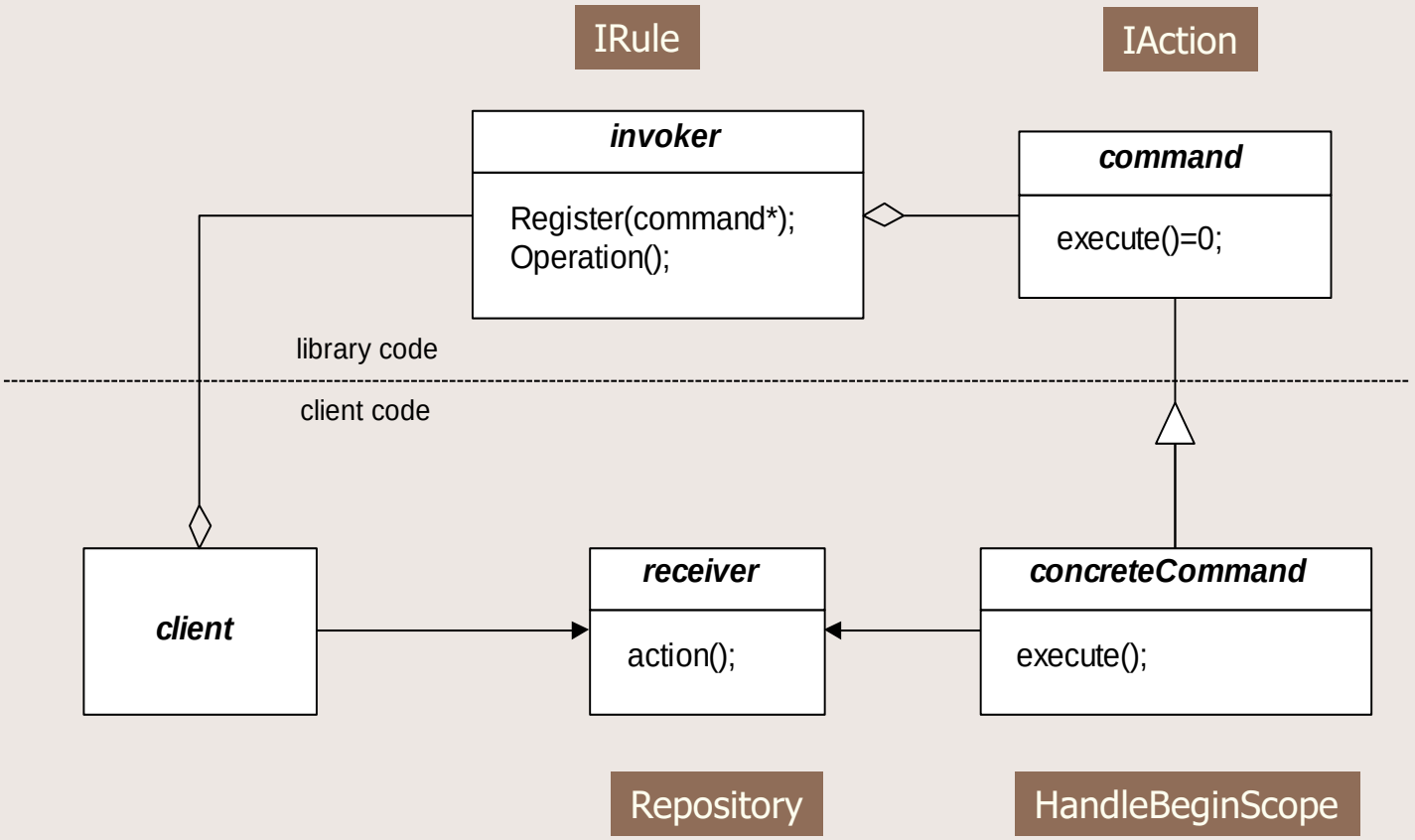
```
bool Parser::parse()
{
  for (size_t i = 0; i<rules.size(); ++i)
  {
    std::string debug = pTokColl->show();

    bool doWhat = rules[i]->doTest(pTokColl);
    if (doWhat == IRule::Stop)
      break;
  }
  return true;
}
```

```
class BeginScope : public IRule
{
public:
  bool doTest(const Scanner::ITokCollection* pTc) override
  {
    GrammarHelper::showParseDemo("Test begin scope", *pTc);

    // don't parse SemiExp with single semicolon token

    if (pTc->length() == 1 && (*pTc)[0] == ";")
      return IRule::Stop;

    if (pTc->find("{") < pTc->length())
    {
      doActions(pTc);
    }
    return IRule::Continue;
  }
};
```

Command

- Intent
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Applicability – use Command when you want to:
  - Parameterize objects by an action to perform. You can express such parameterization in a procedural language with a callback function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks.
  - Specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there.
  - Support undo. The Command's Execute operation can store state for reversing its effects in the command itself.
  - Support logging changes so that they can be reapplied in case of a system crash.
  - Structure a system around high-level operations built on primitive operations. Such a structure is common in information systems that support transactions.

Command Structure



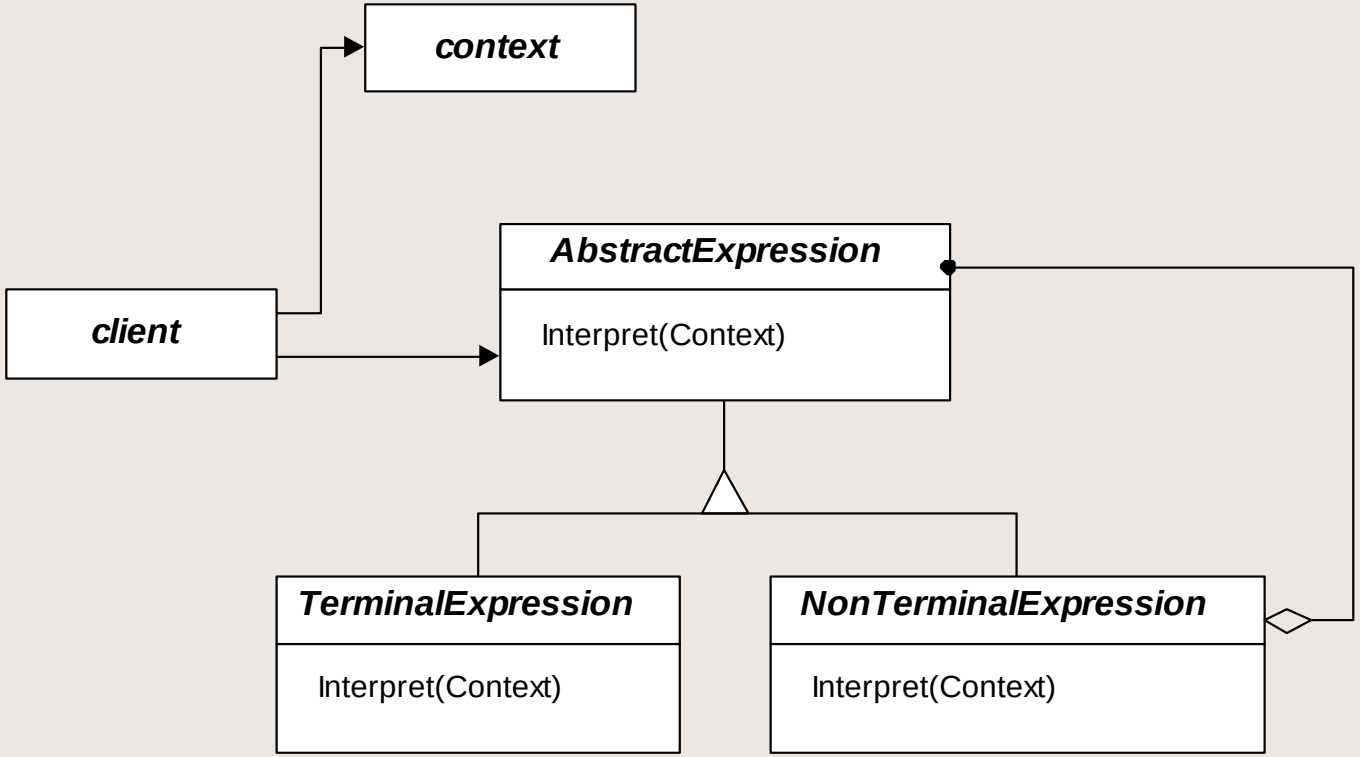
```
class IAction
{
public:
  virtual ~IAction() {}
  virtual void doAction(const Scanner::ITokCollection* pTc) = 0;
};

////////////////////////////////////////////////////////////
// abstract base class for parser lang
//   - rules are registered with the p

class IRule
{
public:
  static const bool Continue = true;
  static const bool Stop = false;
  virtual ~IRule() {}
  void addAction(IAction* pAction);
  void doActions(const Scanner::ITokCc
  virtual bool doTest(const Scanner::I
protected:
  std::vector<IAction*> actions;
};
```

```
class HandleBeginScope : public IAction
{
  Repository* p_Repos;
public:
  HandleBeginScope(Repository* pRepos)
  {
    p_Repos = pRepos;
  }
  void doAction(const Scanner::ITokCollection* pTc) override
  {
    GrammarHelper::showParseDemo("handle begin scope", *pTc);
    //if (p_Repos->scopeStack().size() == 0)
    //  Repository::Demo::write("\n--- empty stack ---");

    ASTNode* pElem = new ASTNode;
    pElem->type_ = "anonymous";
    pElem->name_ = "none";
    pElem->package_ = p_Repos->package();
    pElem->startLineCount_ = p_Repos->lineCount();
    pElem->endLineCount_ = 1;
    pElem->path_ = p_Repos->currentPath();
    /*
     * make this ASTNode child of ASTNode on stack top
     * then push onto stack
     */
    p_Repos->AST().add(pElem);
  }
};
```

Interpreter

- Intent
  - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret senten-ces in the language.
- Applicability – use the Interpreter pattern when:
  - The grammar is simple. For complex grammars, the class hier-archy for the grammar becomes large and unmanageable.
  - Efficiency is not a critical concern. The most efficient inter-preters are usually not implemented by interpreting parse trees directly but by first translating them into another form.

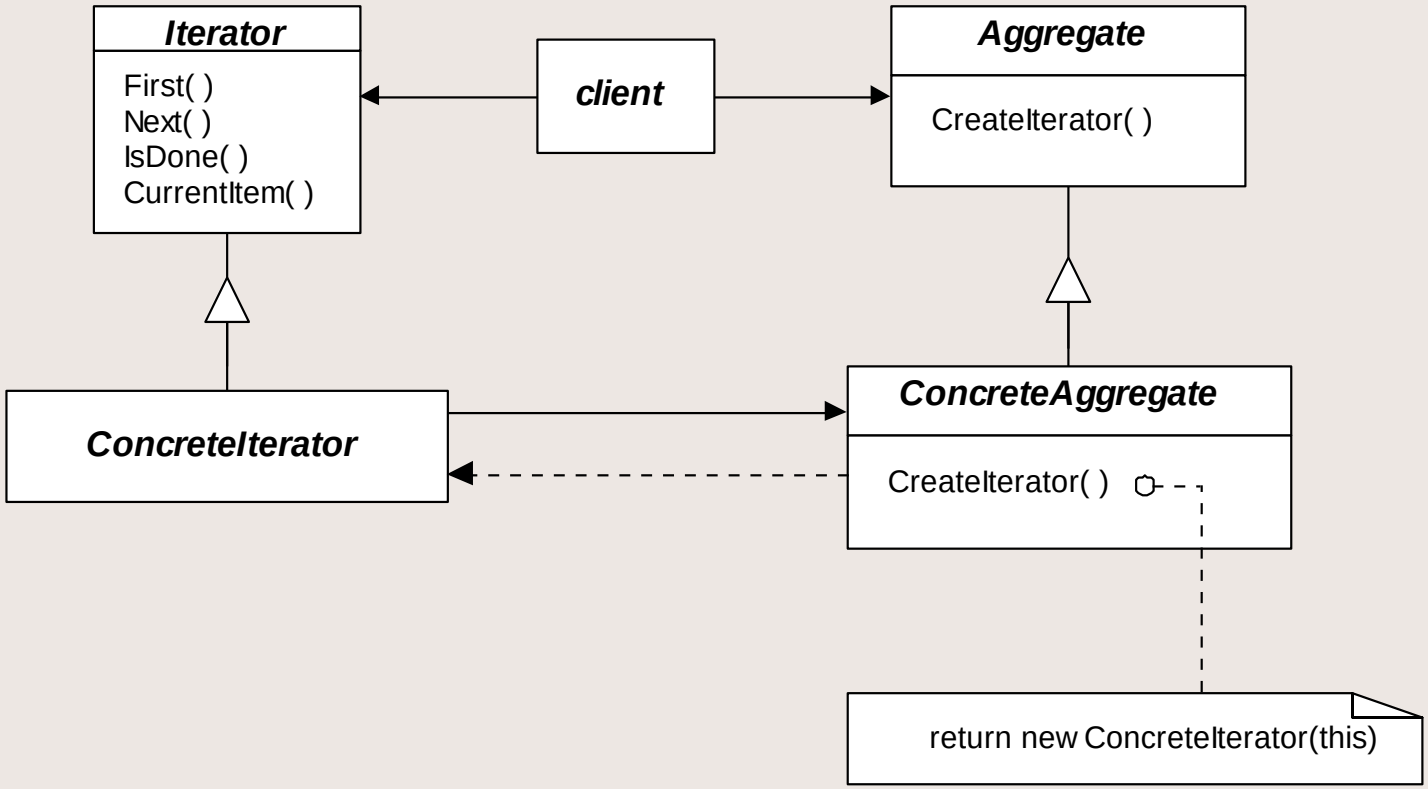
Interpreter Structure



Iterator

- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Applicability – use the Iterator pattern to:
  - Access an aggregate object's contents without exposing its internal representation.
  - Support multiple traversals of aggregate objects.
  - Provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

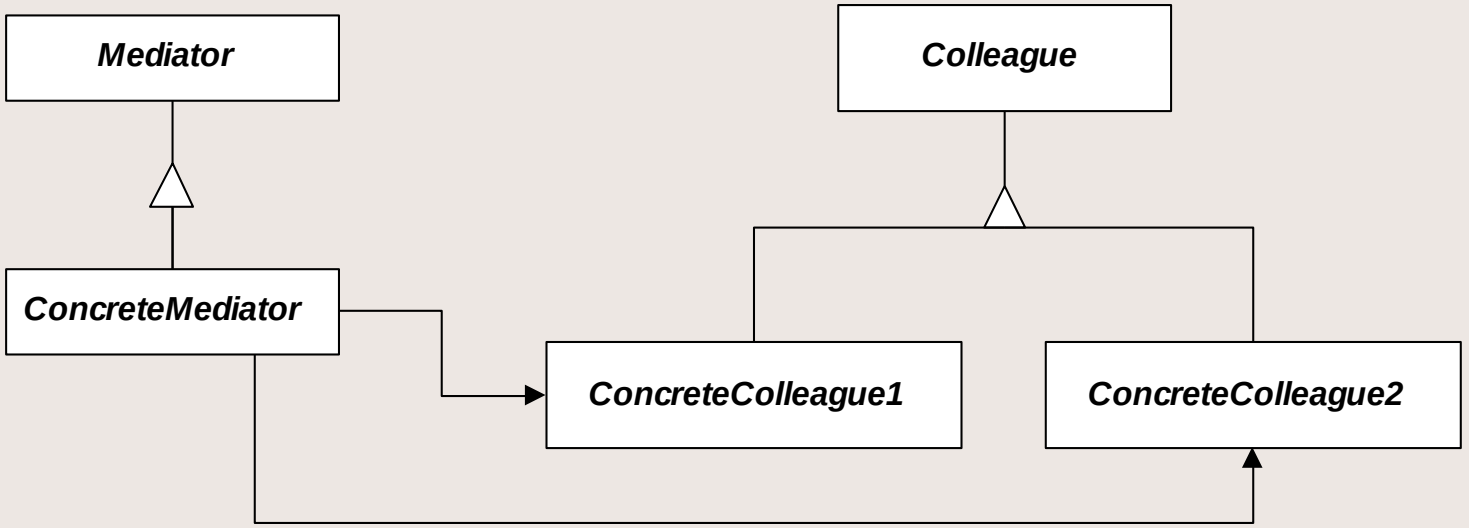
Iterator Structure



Mediator

- Intent
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Applicability – use Mediator when:
  - A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
  - Reusing an object is difficult because it refers to and communi-cates with many other objects.
  - A behavior that's distributed between several classes should be customizable without a lot of subclassing.

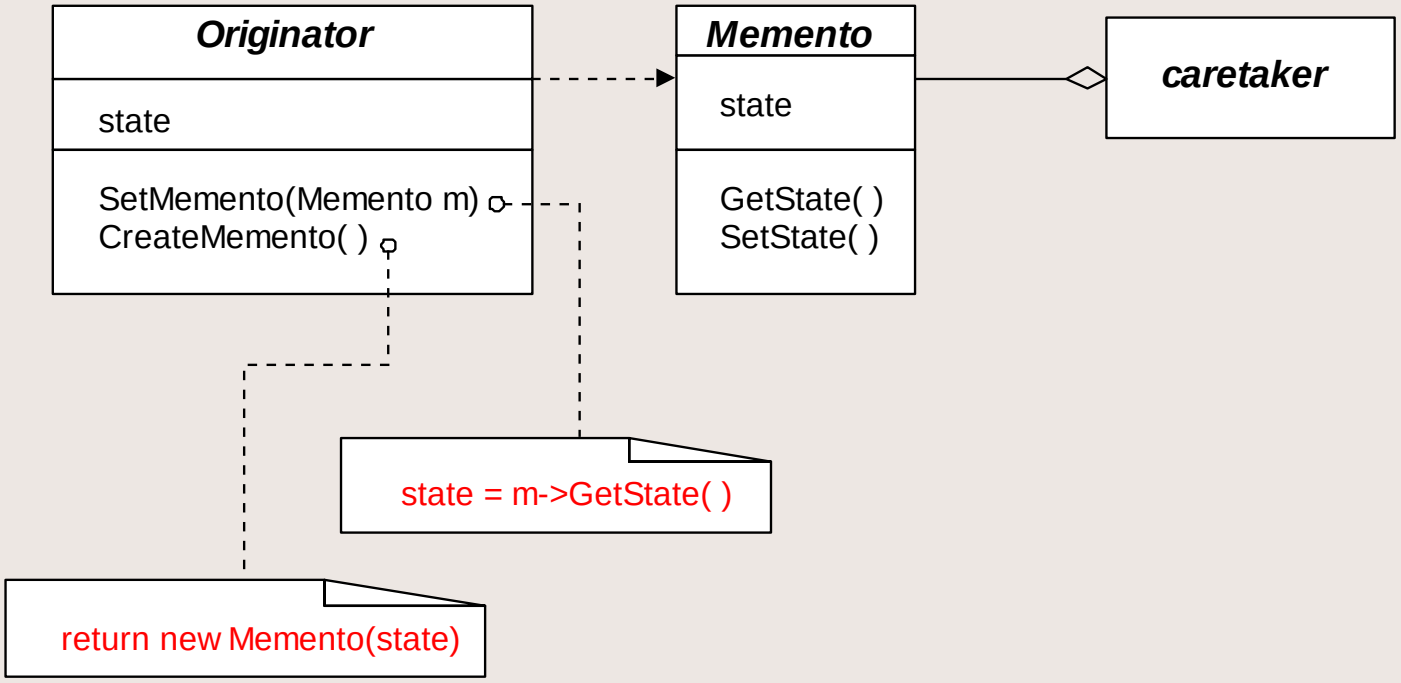
Mediator Structure



Memento

- Intent
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- Applicability – use Memento when:
  - A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
  - A direct interface to obtaining the state would expose implemen-tation details and break the object's encapsulation.

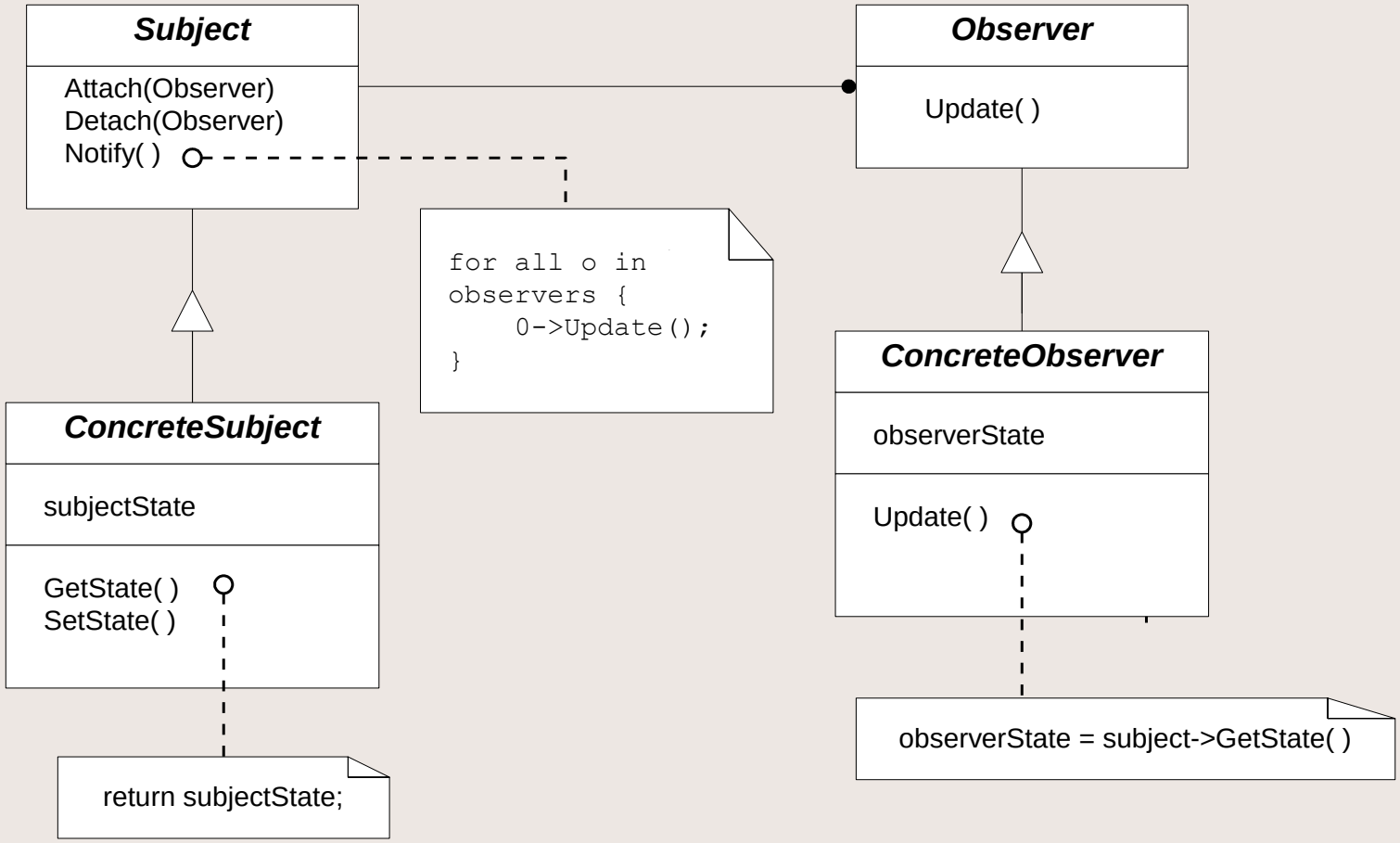
Memento Structure



Observer

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Applicability – use Observer in any of the following:
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others, and you don't know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

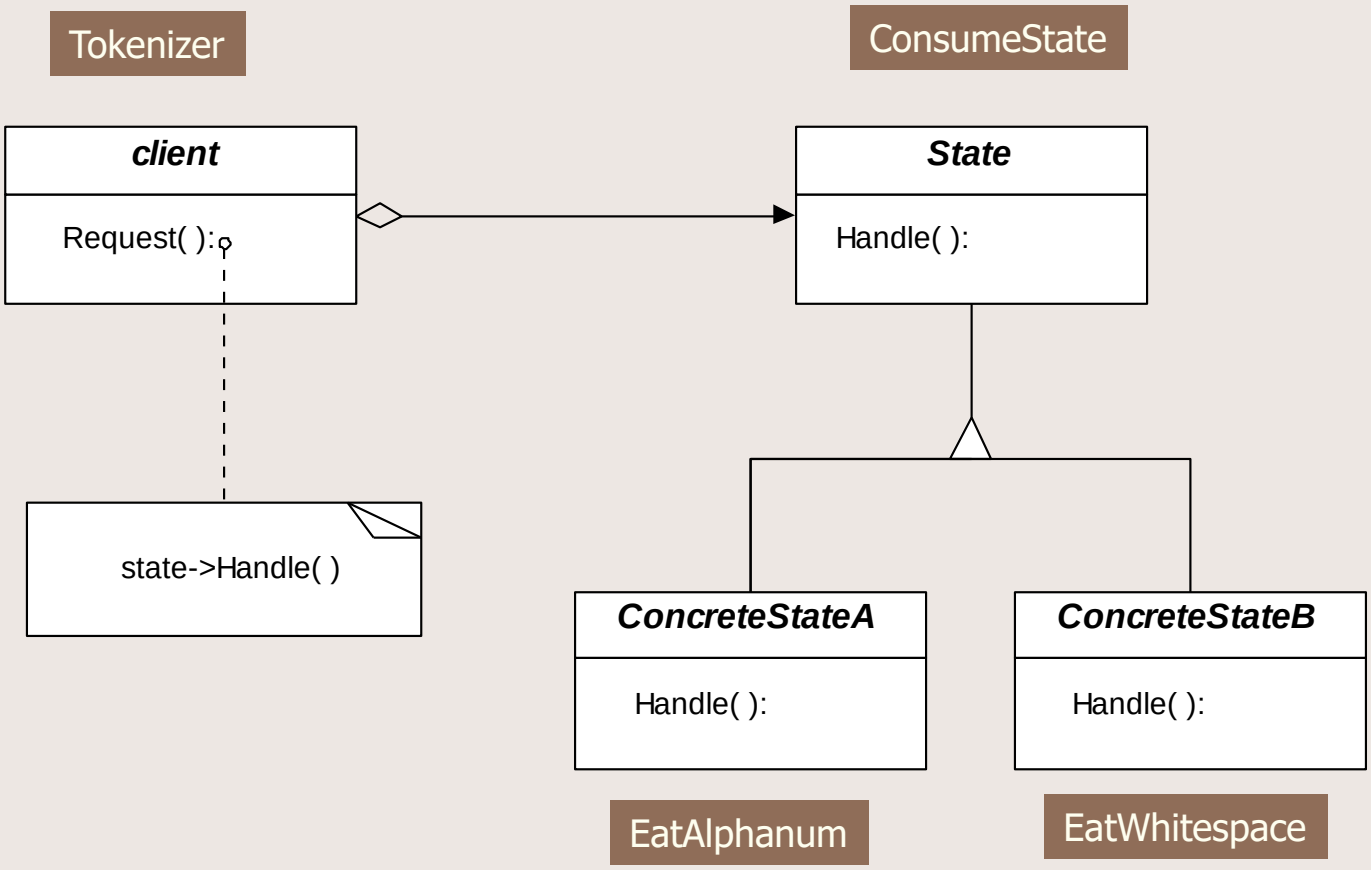
Observer Structure



State

- Intent
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Applicability – use in either of the following cases:
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
  - Operations have large, multipart conditional statements what depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

State Structure



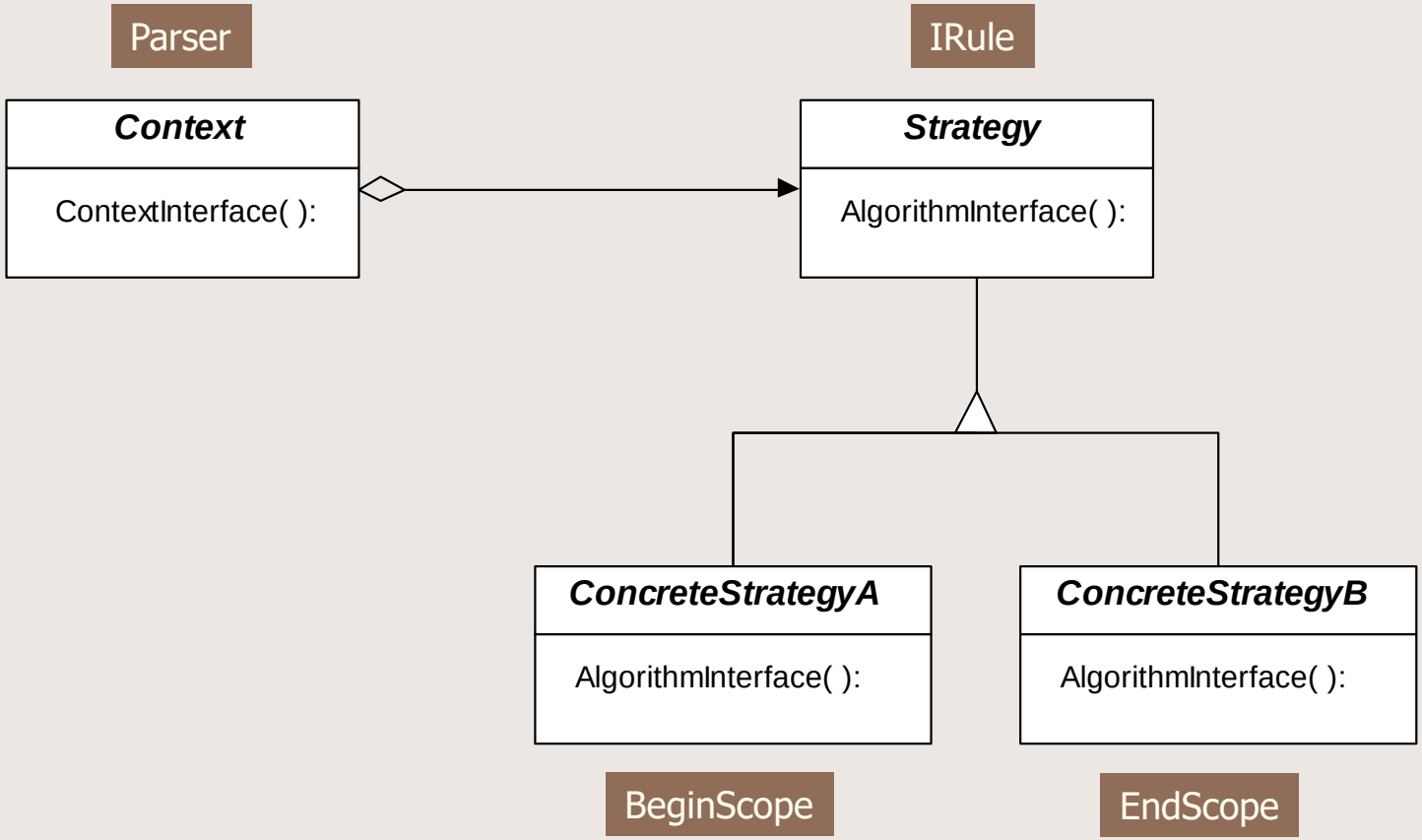
```
class ConsumeState
{
  friend class Toker;
public:
  using Token = std::string;
  ConsumeState();
  ConsumeState(const ConsumeState&) = delete;
  ConsumeState& operator=(const ConsumeState&) = delete;
  virtual ~ConsumeState();
  void attach(std::istream* pIn);
  virtual void eatChars() = 0;
  void consumeChars() {
    _pContext->_pState->eatChars();
    _pContext->_pState = nextState();
  }
  bool canRead() { return _pContext->_pIn->good(); }
  std::string getTok() { return  pContext->token; }
  bool hasTok(
  ConsumeState
  void returnC
  size_t curre
  void setSpec
  void setCont
protected:
  Context* _pC
  bool collect
  bool isOneCh
  bool isTwoCh
  Token makeSt
};
}
```

```
class EatWhitespace : public ConsumeState
{
public:
  EatWhitespace(Context* pContext)
  {
    _pContext = pContext;
  }
  virtual void eatChars()
  {
    _pContext->token.clear();
    do {
      if (!collectChar())
        return;
    } while (std::isspace(_pContext->currChar) && _pContext->currChar != '\n');
  }
};
```

Strategy

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Applicability – use strategy when:
  - Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
  - An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move relate conditional branches into their own Strategy class.

Strategy Structure

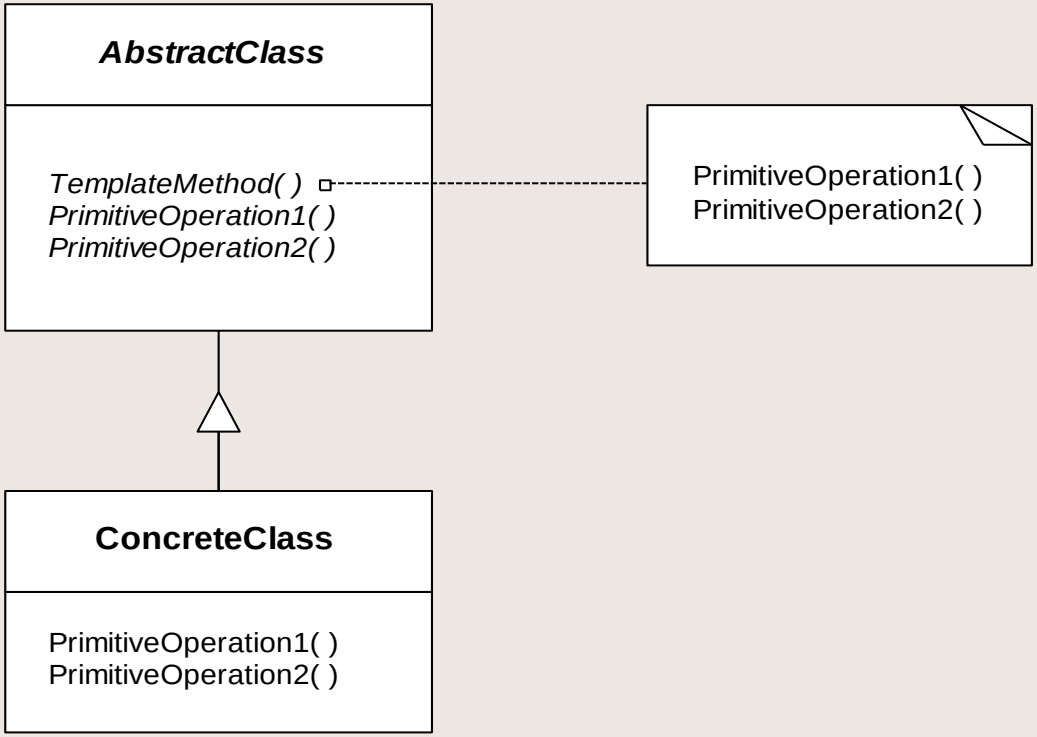


```
class IRule
{
public:
  static const bool Continue = true;
  static const bool Stop = false;
  virtual ~IRule() {}
  void addAction(IAction* pAction);
  void doActions(const Scanner::ITokCollection* pTc);
  virtual bool doTest(const Scanner::ITokCollection* pTc) = 0;
protected:
  std::vector<IAction*> actions;
};

class Parser
{
public:
  Parser(Scanner::ITokCollection* pTokCollection);
  ~Parser();
  void addRule(IRule* pRule);
  bool parse();
  bool next();
private:
  Scanner::ITokCollection* pTokColl;
  std::vector<IRule*> rules;
};
```

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.
- Applicability – use Template Method:
  - To implement the invariant parts of an algorithm once and leav3 it up to subclasses to implement the behavior that can vary.
  - When common behavior among subclasses should be factored and localized in a common class to avoid code duplication. You first identify the differences in existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.
  - To control subclasses extensions. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.

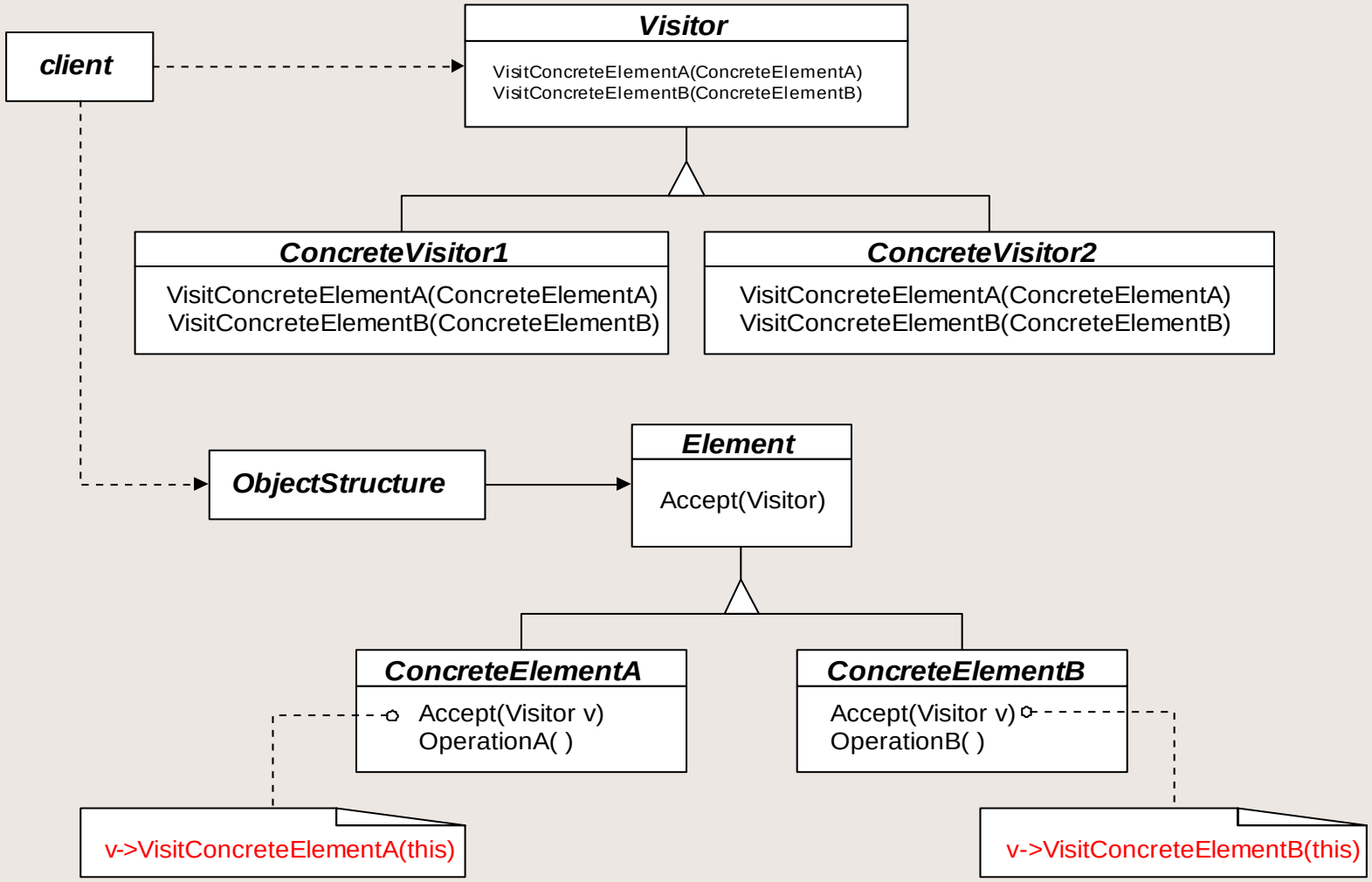
Template Method Structure



Visitor

- Intent
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Applicability – use Visitor pattern when:
  - An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
  - Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
  - The classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly.

Visitor Structure



Pattern Relationships

- Control creation of objects
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton, Flyweight
- Establish object connection structure
  - Composite, Decorator, Facade, Chain of Responsibility, Mediator, Observer
- Construct loosely coupled systems
  - Abstract Factory, Factory Method, Prototype, Bridge, Command, Iterator
- Support change
  - Adapter, Bridge, Facade, Command, Strategy, Visitor
- Simplify communication
  - Facade, Chain of Responsibility, Mediator
- Organize behavior of a family of classes
  - Interpreter, State, Strategy, Template Method, Visitor
- Manage the state of object(s)
  - Memento, Proxy, State
- Manage access to objects
  - Singleton, Adapter, Facade, Flyweight, Proxy, Iterator, Mediator, Visitor

Basic Techniques

- Defer definition
  - Provide protocol with abstract base, interpret protocol with derived (specialized) classes at some later time (Command, Visitor).
- Support interchangeability
  - Provide base class as a proxy definition for a set of interchangeable derived classes (Composite, State, Strategy).
- Promote loose coupling between components
  - Use abstract interfaces.
  - delegate creation (Factory Method).
  - hide implementation behind an opaque pointer (Bridge).
- Support heterogeneous collections
  - Provide reference to base class objects from other base or derived objects (Composite, Decorator, …).