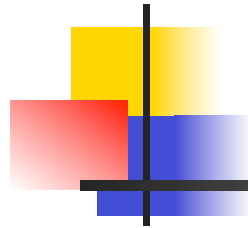
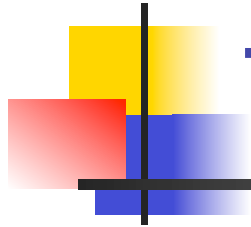


Behavioral Patterns



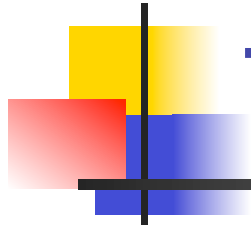
Behavioural Patterns

- Template Method (TM)
- Interpreter (IN)
- Mediator (ME)
- Chain of Responsibility (CoR)
- Observer (OB)
- Strategy (STG)
- Command (CMD)
- State (ST)
- Visitor (VST)
- Iterator (ITR)
- Memento (MMT)



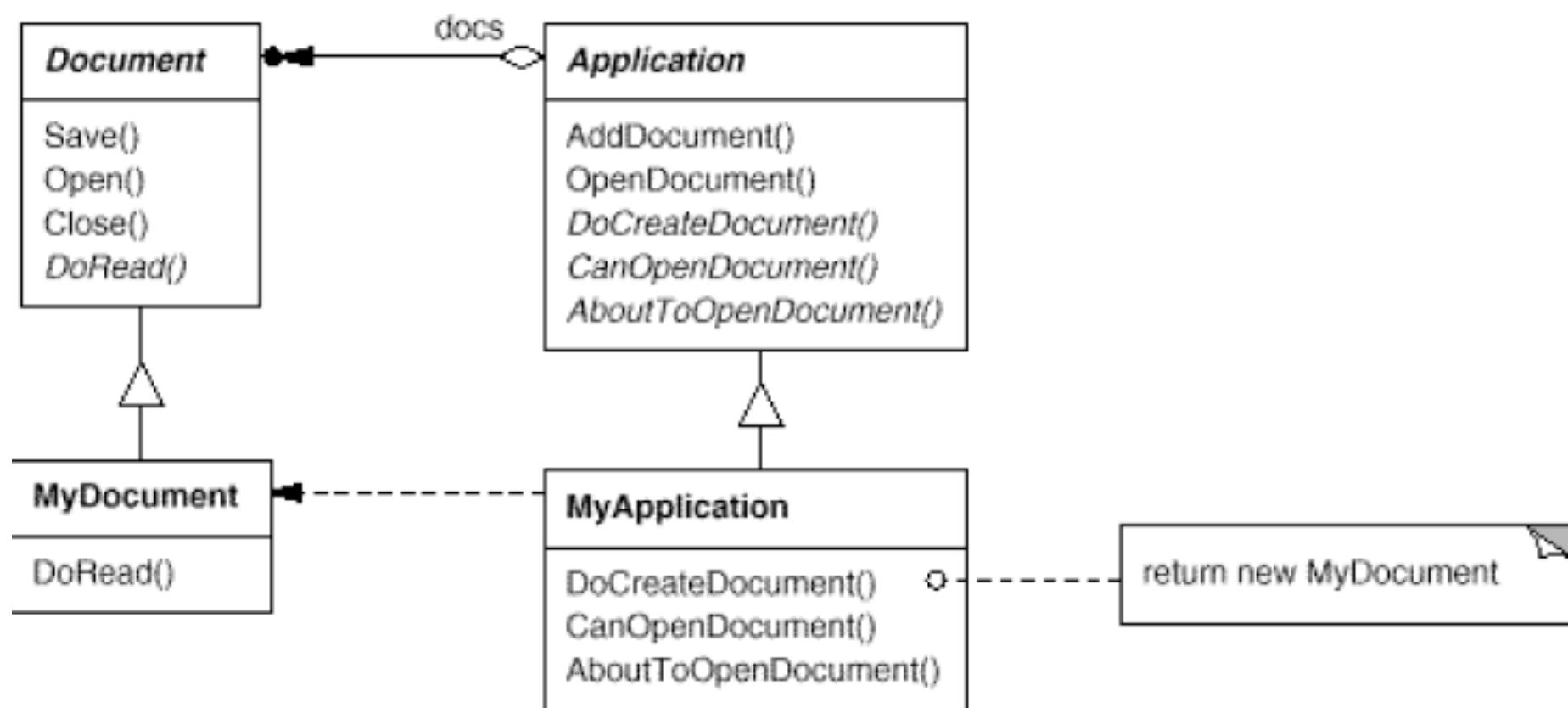
Template Method (TM)

- Intent:
 - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
 - TM lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



TM Motivation (1)

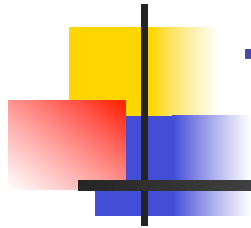
- Consider an application framework that provides Application and Document classes.
 - The Application is responsible for opening existing documents stored in an external format
 - Document represents the information in a document once it's read from the file.
- Application and Document might have subclasses to suit specific needs.
 - DrawApplication and DrawDocument subclasses;
 - Or, SpreadsheetApplication and SpreadsheetDocument subclasses.



```
void Application::OpenDocument (const char* name) {  
    if (!CanOpenDocument(name)) {  
        // cannot handle this document  
        return;  
    }  
  
    Document* doc = DoCreateDocument();  
  
    if (doc) {  
        _docs->AddDocument(doc);  
        _AboutToOpenDocument(doc);  
        doc->Open();  
        doc->DoRead();  
    }  
}
```

We call OpenDocument a **template method**.

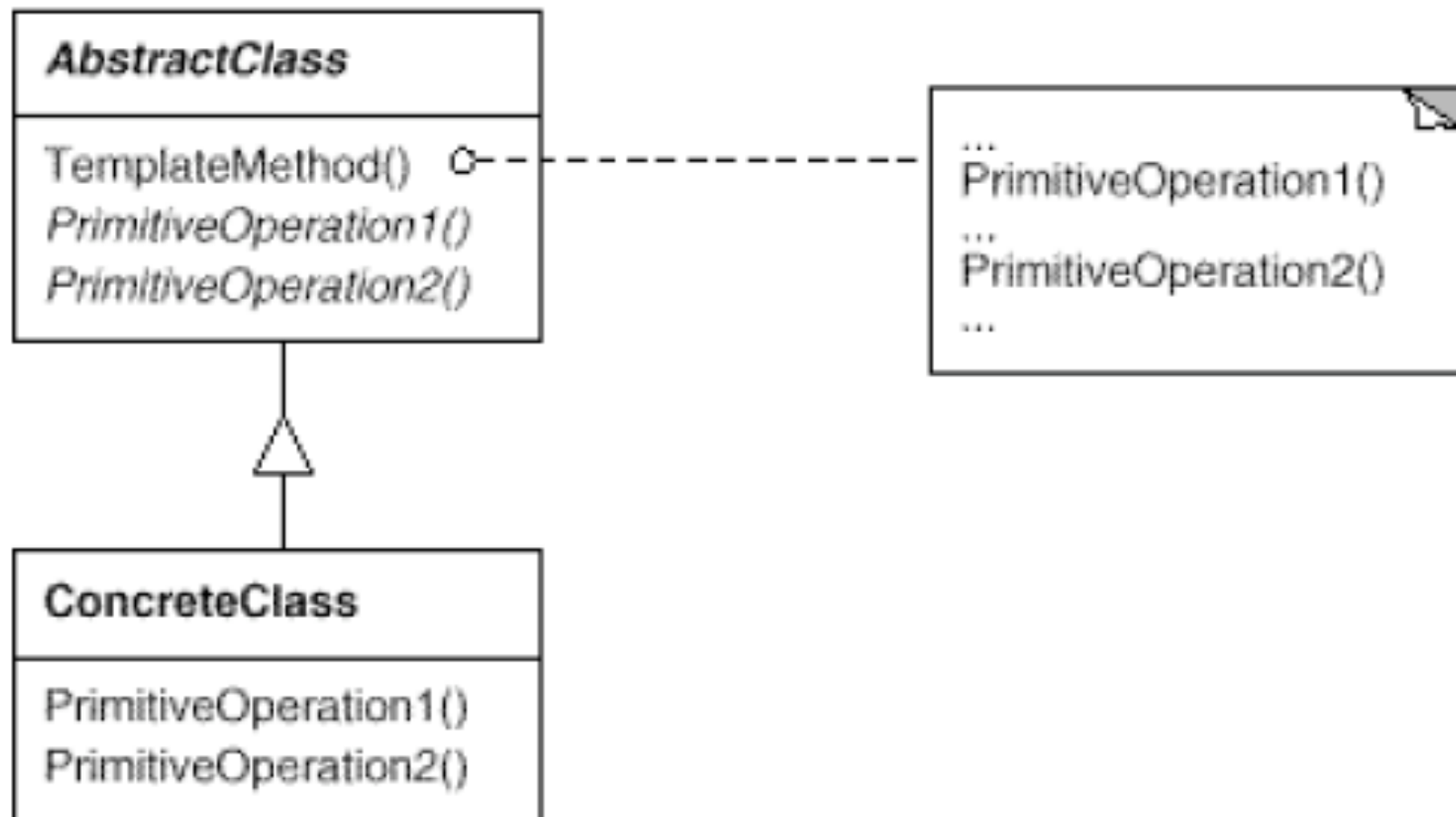
A template method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior.

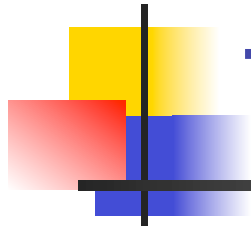


TM Applicability

- Use a TM:
 - to implement the invariant parts of an algorithm once and leave 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 ("refactoring to generalize")
 - to control subclasses extensions.
 - You can define a template method that calls "hook" operations permitting extensions only at those points.

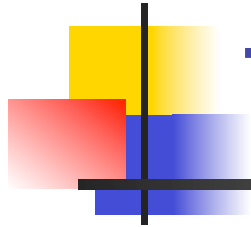
TM Structure





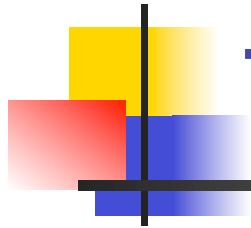
TM Participants

- AbstractClass (Application)
 - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
 - implements a template method defining the skeleton of an algorithm.
 - The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- ConcreteClass (MyApplication)
 - implements the primitive operations to carry out subclass-specific steps of the algorithm.



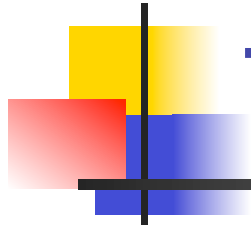
TM Collaboration

- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.



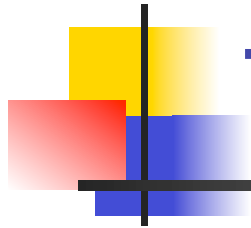
TM Consequences⁽¹⁾

- TM are fundamental for code reuse.
 - particularly important in class libraries
 - They factor out common behavior in library classes.
- Operations called by TM:
 - concrete operations
 - either on the ConcreteClass or on client classes
 - concrete AbstractClass operations
 - operations that are generally useful to subclasses
 - primitive operations or abstract operations
 - factory methods (FM)
 - hook operations
 - which provide default behavior that subclasses can extend if necessary; often does nothing by default.



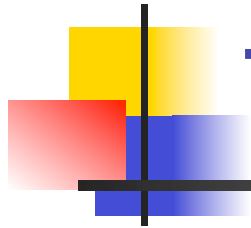
TM Consequences₍₂₎

- Writers must understand which operations are designed for overriding.
 - hooks $\leq \Rightarrow$ may be overridden
 - abstract operations $\leq \Rightarrow$ must be overridden
 - may use a Naming Convention



TM Implementation₍₁₎

- Using C++ access control.
 - The primitive operations that
 - a template method calls can be declared protected
 - must be overridden are declared pure virtual.
 - The TM itself should not be overridden;
 - make the template method a nonvirtual member function.
- Minimizing primitive operations.
 - Minimize the number of primitive operations to be overridden
- Naming conventions.
 - identify the operations that should be overridden by prefixing template method names with "Do-"



TM Sample Code

```
void View::Display () {  
    SetFocus();  
    DoDisplay();  
    ResetFocus();  
}
```

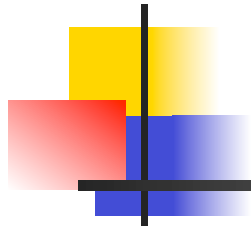
```
void View::DoDisplay () { }
```

```
void MyView::DoDisplay () {  
    // render the view's contents  
}
```



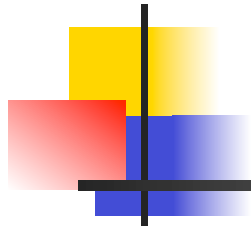
TM Known uses and Related Patterns

- Known Uses
 -many applications
- Related Patterns
 - Factory Methods
 - Strategy



Interpreter (IN)

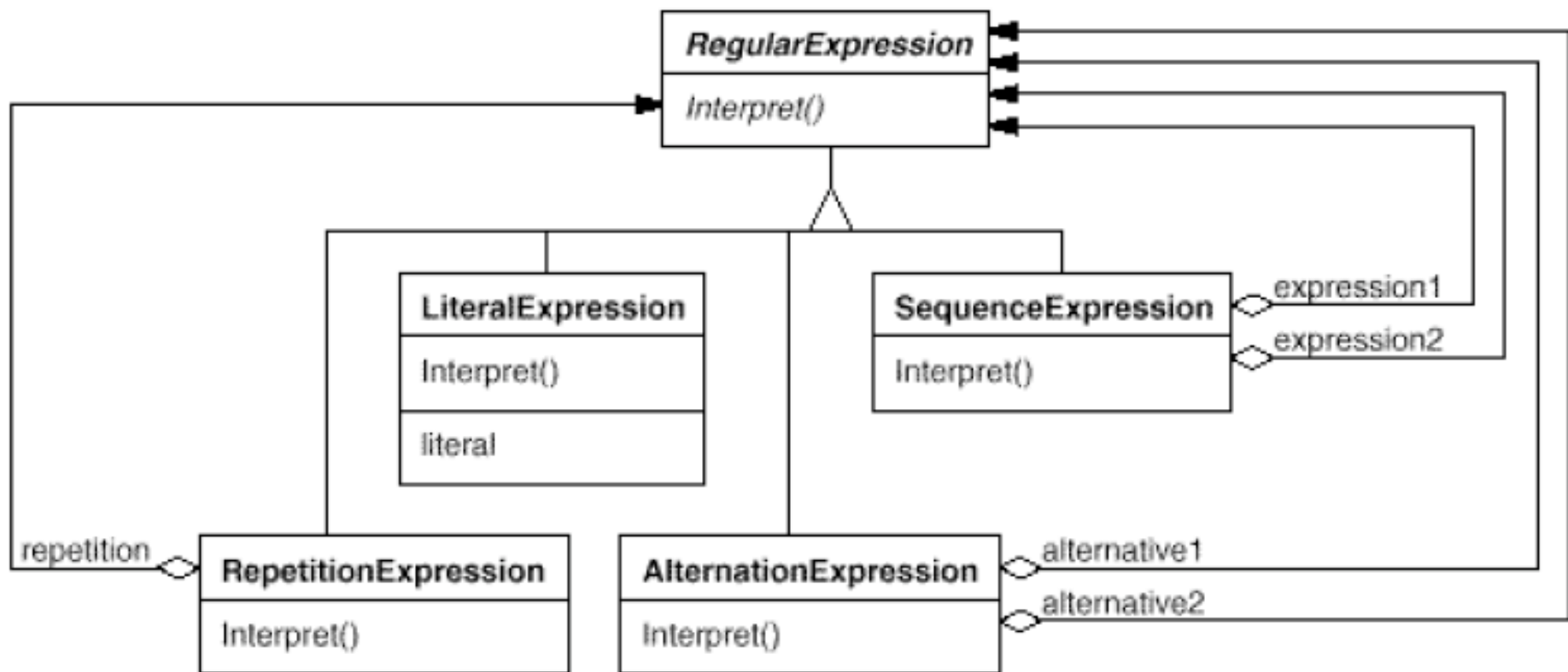
- Intent
 - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



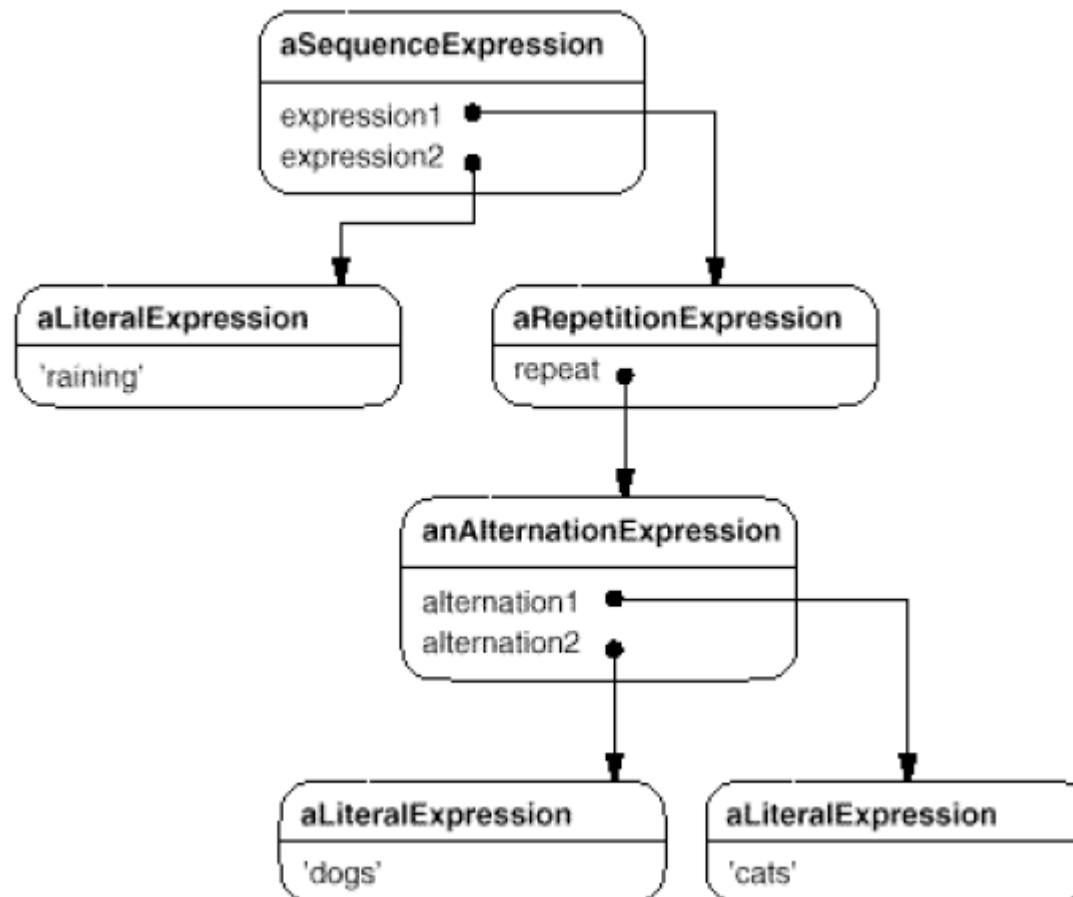
IN Motivation (1)

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- The Interpreter pattern uses a class to represent each grammar rule. Symbols on the right-hand side of the rule are instance variables of these classes.

IN Motivation (2)



IN Motivation (2)



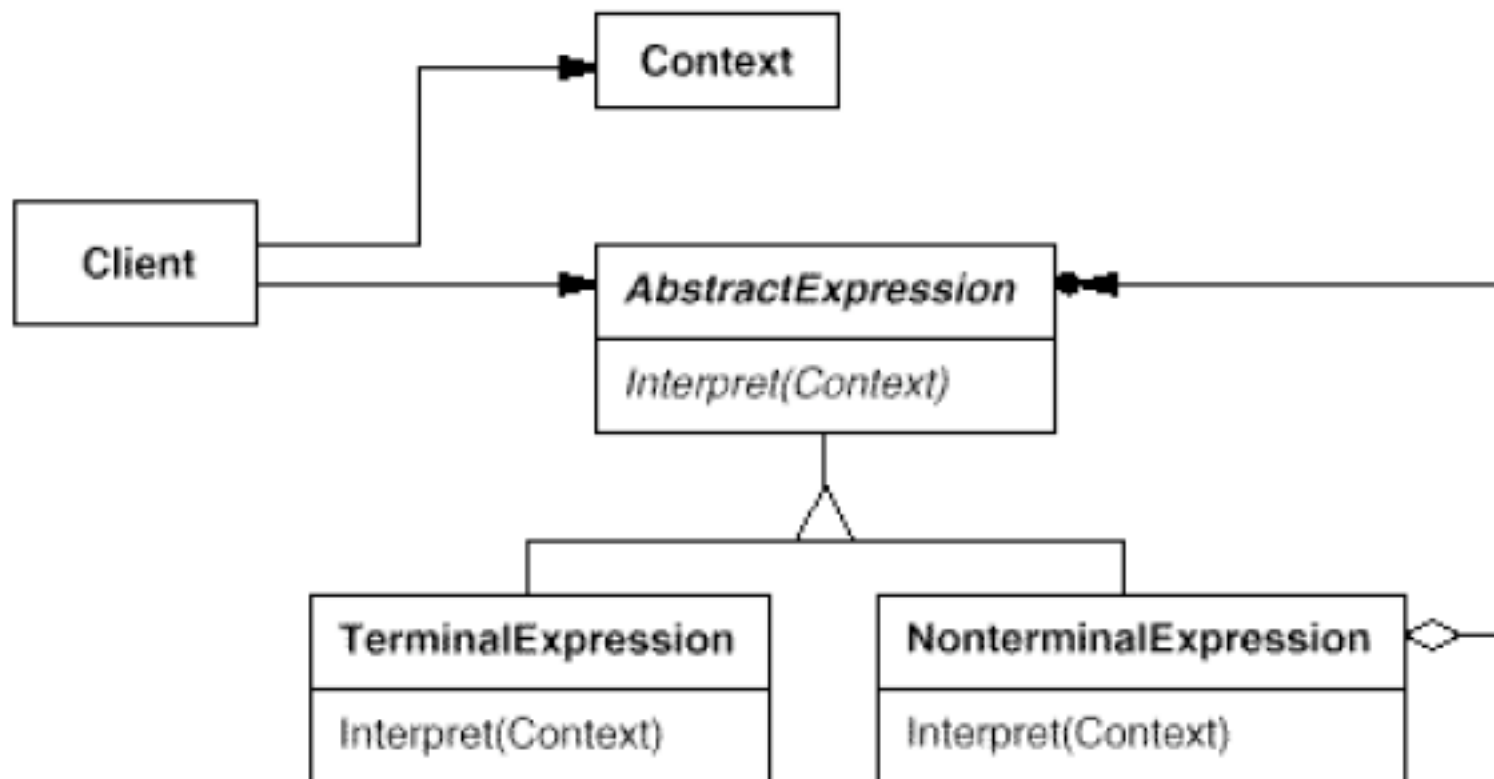


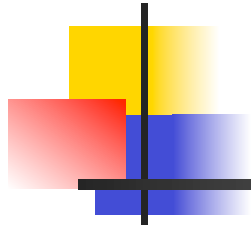
IN Applicability

- Use the IN when:
 - there is a language to interpret, and you can represent statements in the language as abstract syntax trees.
 - IN works best when the grammar is simple.
 - For complex grammars, the class hierarchy for the grammar becomes large and unmanageable.
 - Use parser generators in such cases.
 - efficiency is not a critical concern



IN Structure





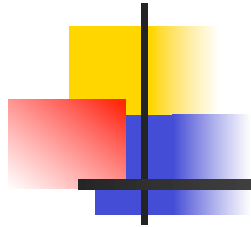
IN Participants (1)

- **AbstractExpression (RegularExpression)**
 - declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.
- **TerminalExpression (LiteralExpression)**
 - implements an Interpret operation associated with terminal symbols in the grammar.
 - an instance is required for every terminal symbol in a sentence.



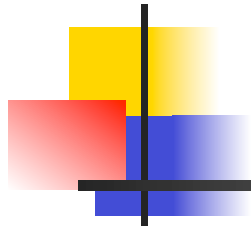
IN Participants (1)

- **NonterminalExpression** (AlternationExpression, ...)
 - one such class is required for every rule in the grammar.
 - maintains instance variables of type AbstractExpression
 - implements an Interpret operation for nonterminal symbols in the grammar.
 - IN typically calls itself recursively
- **Context**
 - contains information that's global to the interpreter.
- **Client**
 - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines.
 - invokes the Interpret operation.



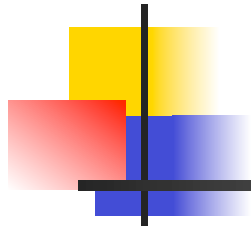
IN Collaborations

- The client builds the sentence as an abstract syntax tree of NonterminalExpression and TerminalExpression instances. Then the client initializes the context and invokes the Interpret operation.
- Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression.
- The Interpret operations at each node use the context to store and access the state of the interpreter.



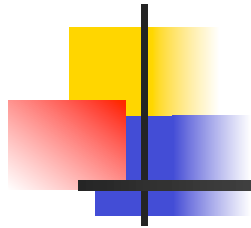
IN Consequences (1)

- It's easy to change and extend the grammar.
 - use classes to represent grammar
 - use inheritance to change or extend the grammar.
 - Modify existing expressions incrementally
 - define new expr. as variations on old ones.
- Implementing the grammar is easy, too.
 - Classes defining nodes in the abstract syntax tree have similar implementations.
 - The generation of classes defining nodes often can be automated with a compiler or parser generator.



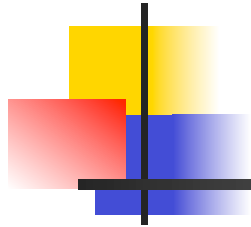
IN Consequences (2)

- Complex grammars are hard to maintain.
 - IN defines at least one class for every rule in the grammar => grammars containing many rules can be hard to manage and maintain.
 - If grammar is very complex than parser or compiler generators are more appropriate.
- Adding new ways to interpret expressions.
 - The Interpreter pattern makes it easier to evaluate an expression in a new way.
 - you can support pretty printing or type-checking an expression by defining a new operation on the expression classes.



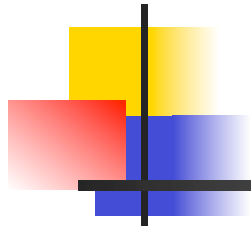
IN implementation (1)

- Creating the abstract syntax tree.
 - IN doesn't explain how to create an abstract syntax tree.
 - can be created by table-driven parser, by a hand-crafted parser, or directly by the client.
 - **IN doesn't address parsing.**
- Defining the Interpret operation.
 - Do not necessarily define the Interpret operation in the expression classes (Visitor can help)



IN implementation (2)

- Sharing terminal symbols with the Flyweight pattern.
 - Grammars whose sentences contain many occurrences of a terminal symbol might benefit from sharing a single copy of that symbol.



IN Sample Code

- Consider a system for manipulating and evaluating Boolean expressions implemented in C++.
 - The terminal symbols in this language are Boolean variables (constants true and false).
 - Nonterminal symbols represent expressions containing the operators and, or, and not.

```
BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp |  
            '(' BooleanExp ')'
```

```
AndExp ::= BooleanExp 'and' BooleanExp
```

```
OrExp ::= BooleanExp 'or' BooleanExp
```

```
NotExp ::= 'not' BooleanExp
```

```
Constant ::= 'true' | 'false'
```

```
VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'
```

```
class BooleanExp {
```

```
public:
```

```
    BooleanExp();
```

```
    virtual ~BooleanExp();
```

```
    virtual bool Evaluate(Context&) = 0;
```

```
    virtual BooleanExp* Replace(const char*, BooleanExp&) = 0;
```

```
    virtual BooleanExp* Copy() const = 0;
```

```
};
```

```
class Context {
```

```
public:
```

```
    bool Lookup(const char*) const;
```

```
    void Assign(VariableExp*, bool);
```

```
};
```

```
class VariableExp : public BooleanExp {
public:
    VariableExp(const char*);
    virtual ~VariableExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    char* _name;
};

bool VariableExp::Evaluate (Context& aContext) {
    return aContext.Lookup(_name);
}
```

```

class AndExp : public BooleanExp {
public:
    AndExp(BooleanExp*, BooleanExp*);
    virtual ~ AndExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    BooleanExp* _operand1;
    BooleanExp* _operand2;
};

AndExp::AndExp (BooleanExp* op1, BooleanExp* op2) {
    _operand1 = op1;
    _operand2 = op2;
}

bool AndExp::Evaluate (Context& aContext) {
    return
        _operand1->Evaluate(aContext) &&
        _operand2->Evaluate(aContext);
}

```



```
BooleanExp* expression;
Context context;
VariableExp* x = new VariableExp("X");
VariableExp* y = new VariableExp("Y");

expression = new OrExp(
    new AndExp(new Constant(true), x),
    new AndExp(y, new NotExp(x))
);

context.Assign(x, false);
context.Assign(y, true);

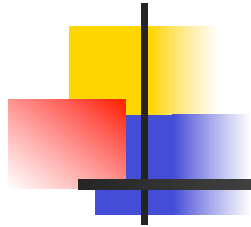
bool result = expression->Evaluate(context);

VariableExp* z = new VariableExp("Z");
NotExp not_z(z);

BooleanExp* replacement = expression->Replace("Y", not_z);

context.Assign(z, true);

result = replacement->Evaluate(context);
```



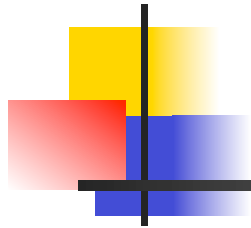
IN Related Patterns

- Related Patterns
 - Composite
 - Iterator
 - Flyweight
 - Visitor



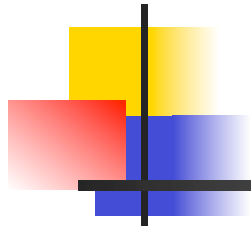
Mediator (ME)

- 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.
- Motivation
 - Object-oriented design encourages the distribution of behavior among objects.
 - distribution may result in many connections between objects (up to every object know about every other)
 - it can be difficult to change the system's behavior in any significant way



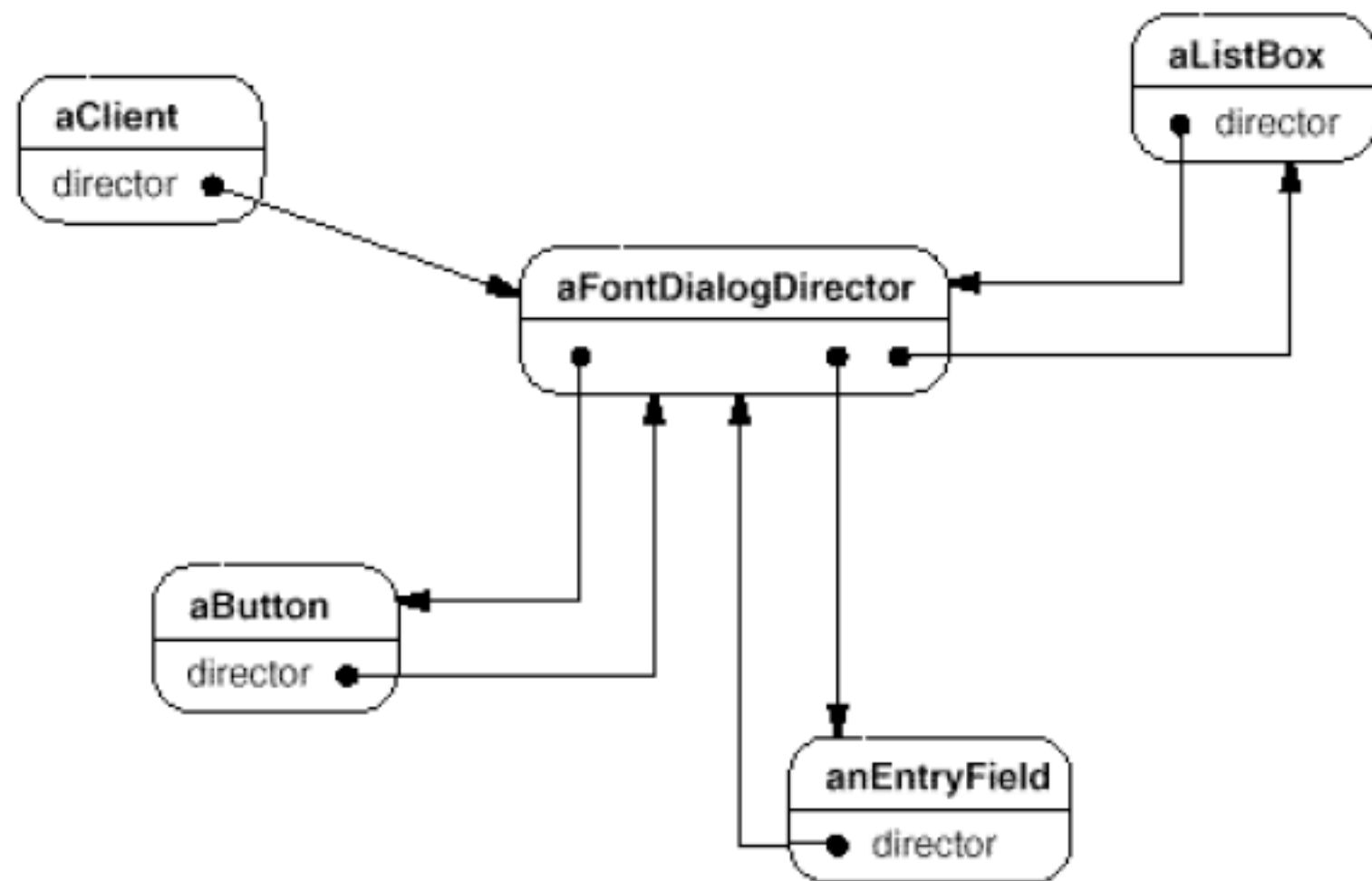
ME Motivation (continued)

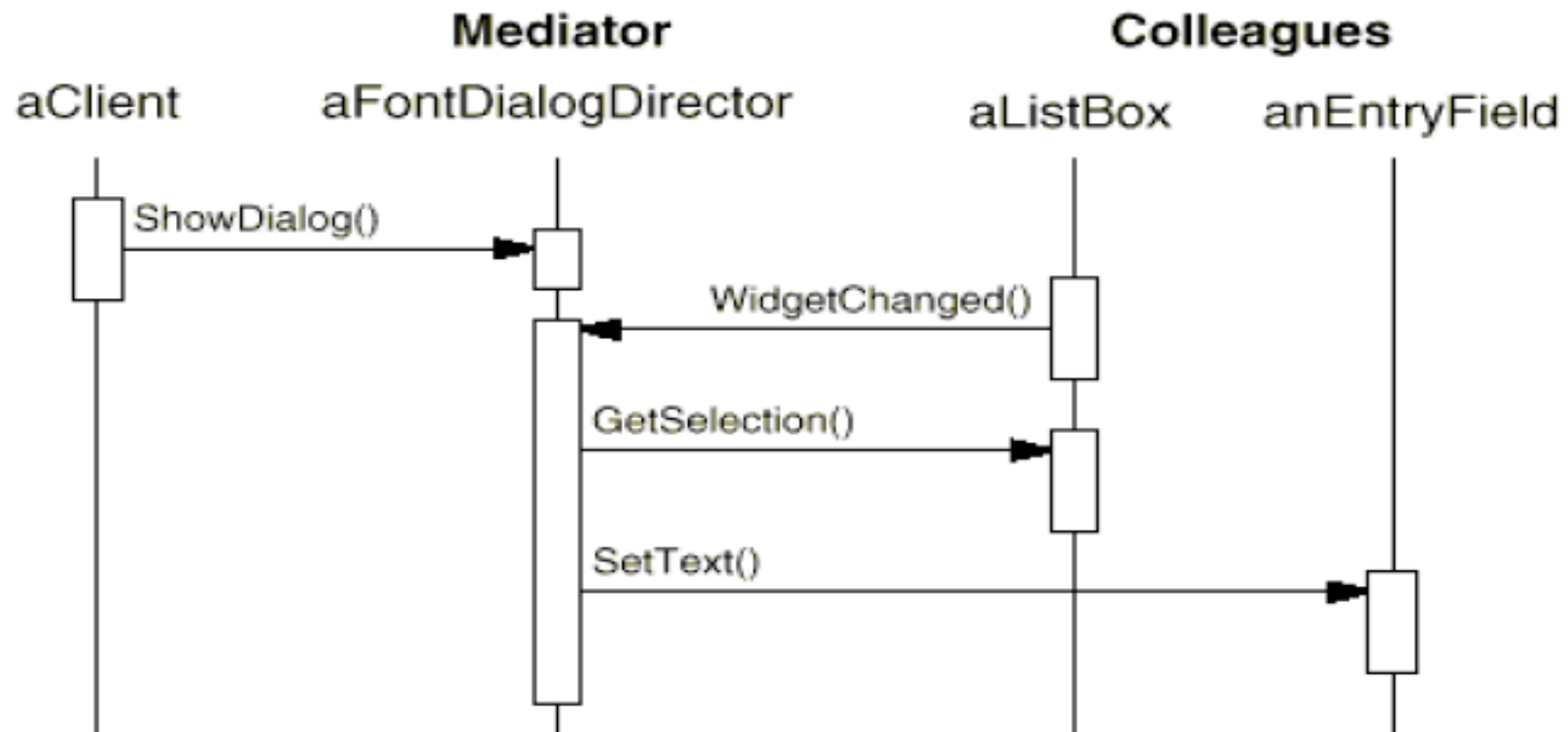
- Consider the implementation of dialog boxes in a graphical user interface.
 - E.g. a dialog box presenting a collection of widgets such as buttons, menus, and entry fields
 - Often there are dependencies between the widgets in the dialog.
 - E.g. Selecting an entry in a list of choices called a **list box** might change the contents of an entry field.
 - Different dialog boxes will have different dependencies between widgets.



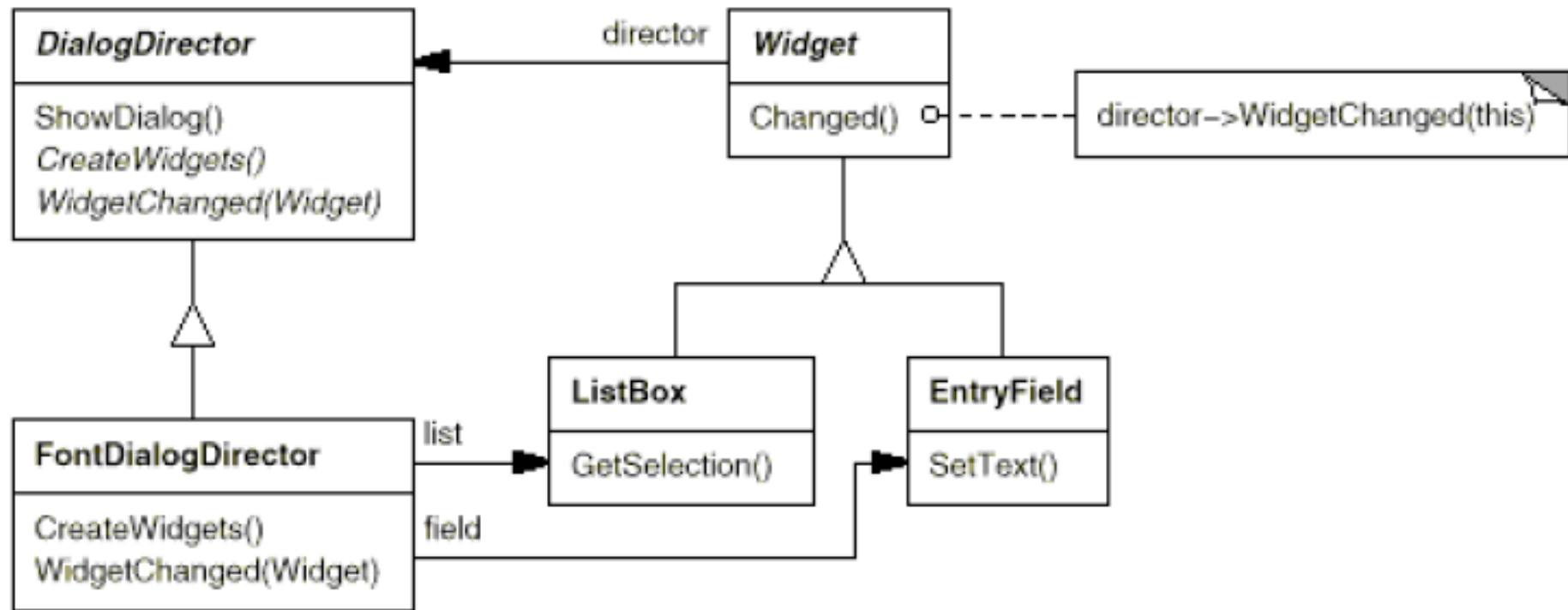
ME Motivation (3)

- Avoid these problems by encapsulating collective behavior in a separate **mediator** object.
 - responsible for controlling and coordinating the interactions of a group of objects.
 - an intermediary that keeps objects in the group from referring to each other explicitly.
 - The objects only know the mediator, thereby reducing the number of interconnections.

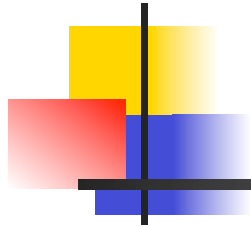




1. The list box tells its director that it's changed.
2. The director gets the selection from the list box.
3. The director passes the selection to the entry field.
4. Now that the entry field contains some text, the director enables button(s) for initiating an action (e.g., "demibold," "oblique").



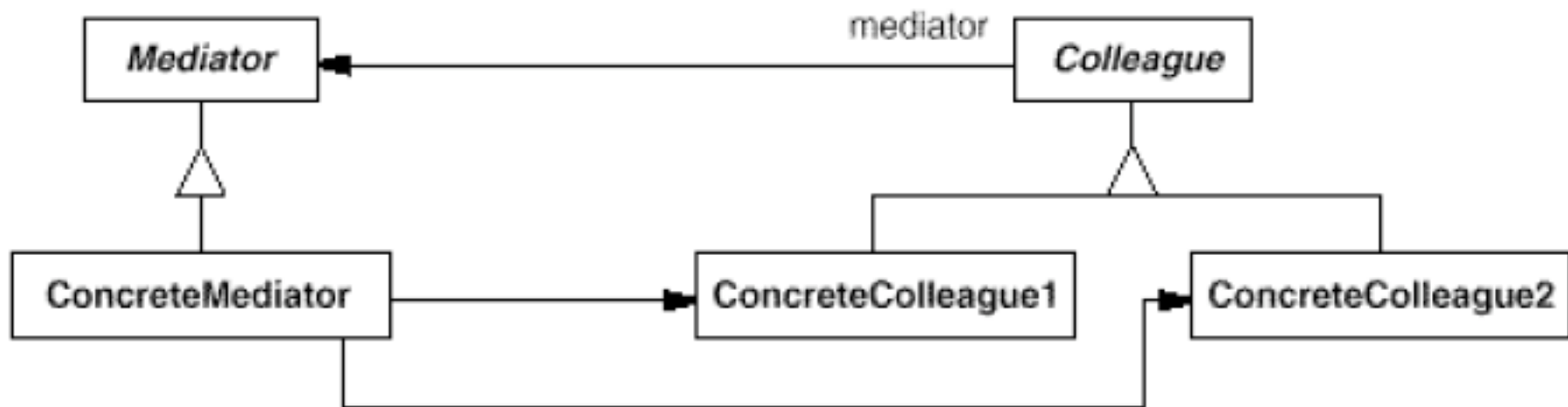
- DialogDirector defines the overall behavior of a dialog.
- Clients call the ShowDialog operation to display the dialog.
- CreateWidgets is an abstract operation for creating the widgets
- WidgetChanged is an abstract operation for informing their director that they have changed.
- CreateWidgets is overridden to create the proper widgets, and they override WidgetChanged to handle the changes.



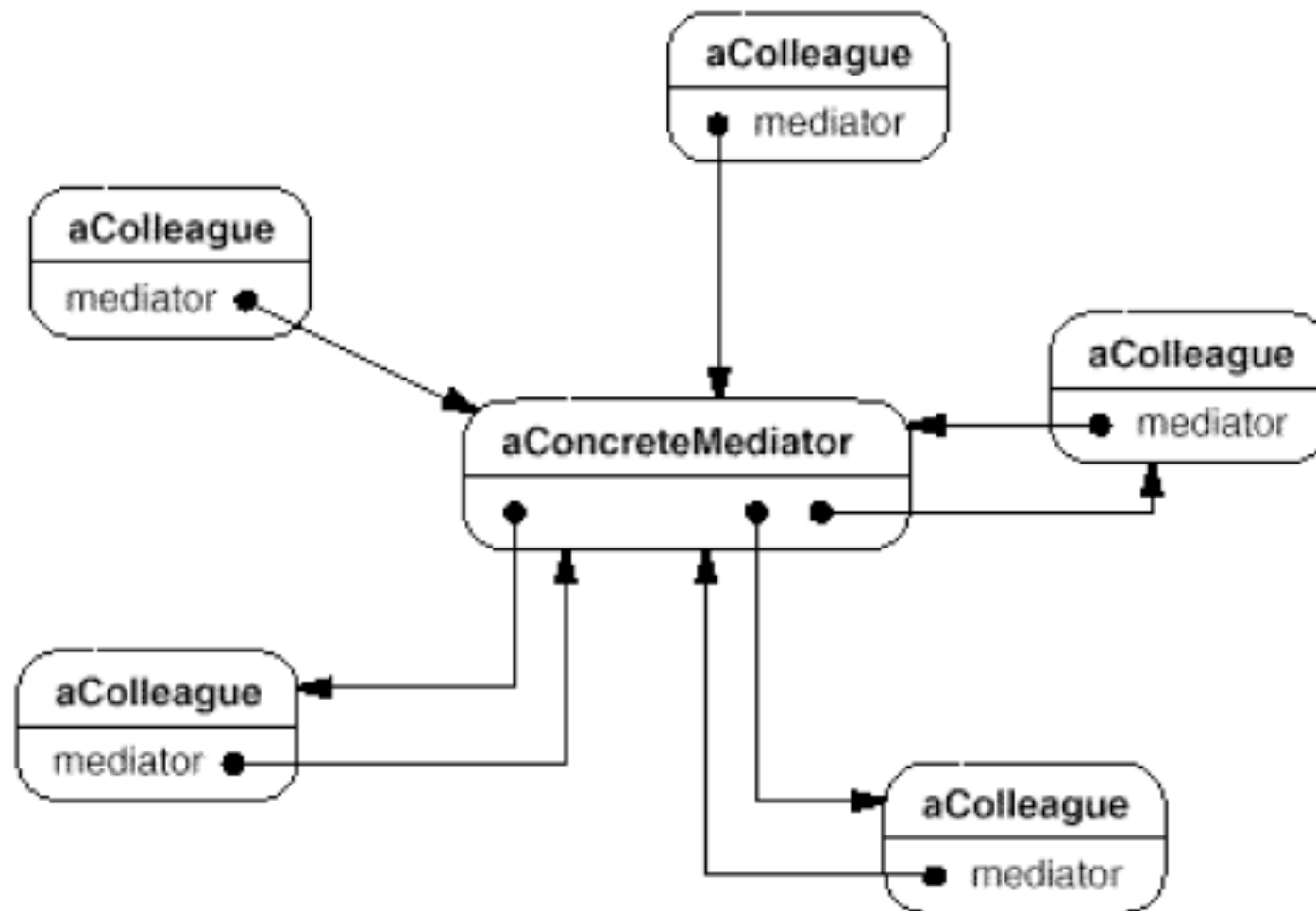
ME Applicability

- Use the ME 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 communicates with many other objects.
 - a behavior that's distributed between several classes should be customizable without a lot of subclassing.

ME Structure



ME Structure





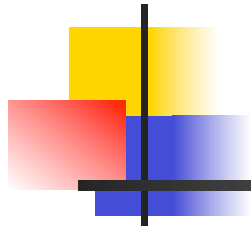
ME Participants and Collaborations

- Participants
 - Mediator (DialogDirector)
 - defines an interface for communicating with Colleague
 - ConcreteMediator (FontDialogDirector)
 - implements cooperative behavior by coordinating Colleague objects
 - knows and maintains its colleagues.
 - Colleague classes (ListBox, EntryField)
 - each Colleague class knows its Mediator object
 - each colleague communicates through its mediator
- Collaborations
 - Clients access a Singleton instance solely through Singleton's Instance operation.



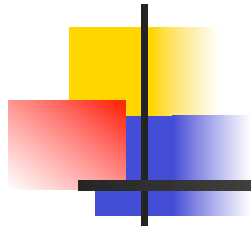
ME Consequences (1)

- It limits subclassing.
 - behavior is localized
 - Changing it requires subclassing Mediator only
 - Colleague classes can be reused as is.
- It decouples colleagues.
 - promotes loose coupling between colleagues.
 - You can vary and reuse Colleague and Mediator classes independently.



ME Consequences (2)

- It simplifies object protocols.
 - replaces many-to-many interactions with one-to-many interactions (ME and colleagues)
 - One-to-many is easier to understand, maintain, and extend.
- It abstracts how objects cooperate.
 - lets you focus on how objects interact apart from their individual behavior.
 - can help clarify how objects interact in a system.
- It centralizes control.
 - ME trades complexity of interaction for complexity in the mediator.
 - the mediator may become a monolith that's hard to maintain.



ME Implementation

- Omitting the abstract Mediator class.
 - There's no need to define an abstract Mediator class when colleagues work with only one mediator.
- Colleague-Mediator communication.
 - Colleagues have to communicate with their mediator when an event of interest occurs.
 - Implement the Mediator as an Observer
 - defines a specialized notification interface in Mediator that lets colleagues be more direct in their communication.



ME Sample Code

```
class DialogDirector {
public:
    virtual ~DialogDirector();

    virtual void ShowDialog();
    virtual void WidgetChanged(Widget*) = 0;

protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};
```



```
class Widget {
public:
    Widget(DialogDirector*);
    virtual void Changed();

    virtual void HandleMouse(MouseEvent& event);
    // ...
private:
    DialogDirector* _director;
};

void Widget::Changed () {
    _director->WidgetChanged(this);
}
```

```
class ListBox : public Widget {
public:
    ListBox(DialogDirector*);

    virtual const char* GetSelection();
    virtual void SetList(List<char*>* listItems);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

class EntryField : public Widget {
public:
    EntryField(DialogDirector*);

    virtual void SetText(const char* text);
    virtual const char* GetText();
    virtual void HandleMouse(MouseEvent& event);
    // ...
};
```

```
class Button : public Widget {
public:
    Button(DialogDirector*);

    virtual void SetText(const char* text);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();
}
```

```
class FontDialogDirector : public DialogDirector {
public:
    FontDialogDirector();
    virtual ~FontDialogDirector();
    virtual void WidgetChanged(Widget*);

protected:
    virtual void CreateWidgets();

private:
    Button* _ok;
    Button* _cancel;
    ListBox* _fontList;
    EntryField* _fontName;
};
```

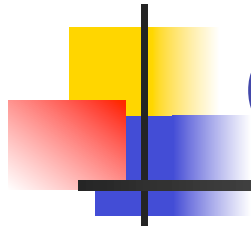
```
void FontDialogDirector::CreateWidgets () {  
    _ok = new Button(this);  
    _cancel = new Button(this);  
    _fontList = new ListBox(this);  
    _fontName = new EntryField(this);  
  
    // fill the listBox with the available font names  
  
    // assemble the widgets in the dialog  
}
```

```
void FontDialogDirector::WidgetChanged (
    Widget* theChangedWidget
) {
    if (theChangedWidget == _fontList) {
        _fontName->SetText(_fontList->GetSelection());
    } else if (theChangedWidget == _ok) {
        // apply font change and dismiss dialog
        // ...
    } else if (theChangedWidget == _cancel) {
        // dismiss dialog
    }
}
```



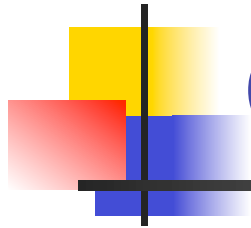
ME Known Uses and Related Patterns

- Known Uses
 - ...many applications
- Related Patterns
 - Façade
 - Observer



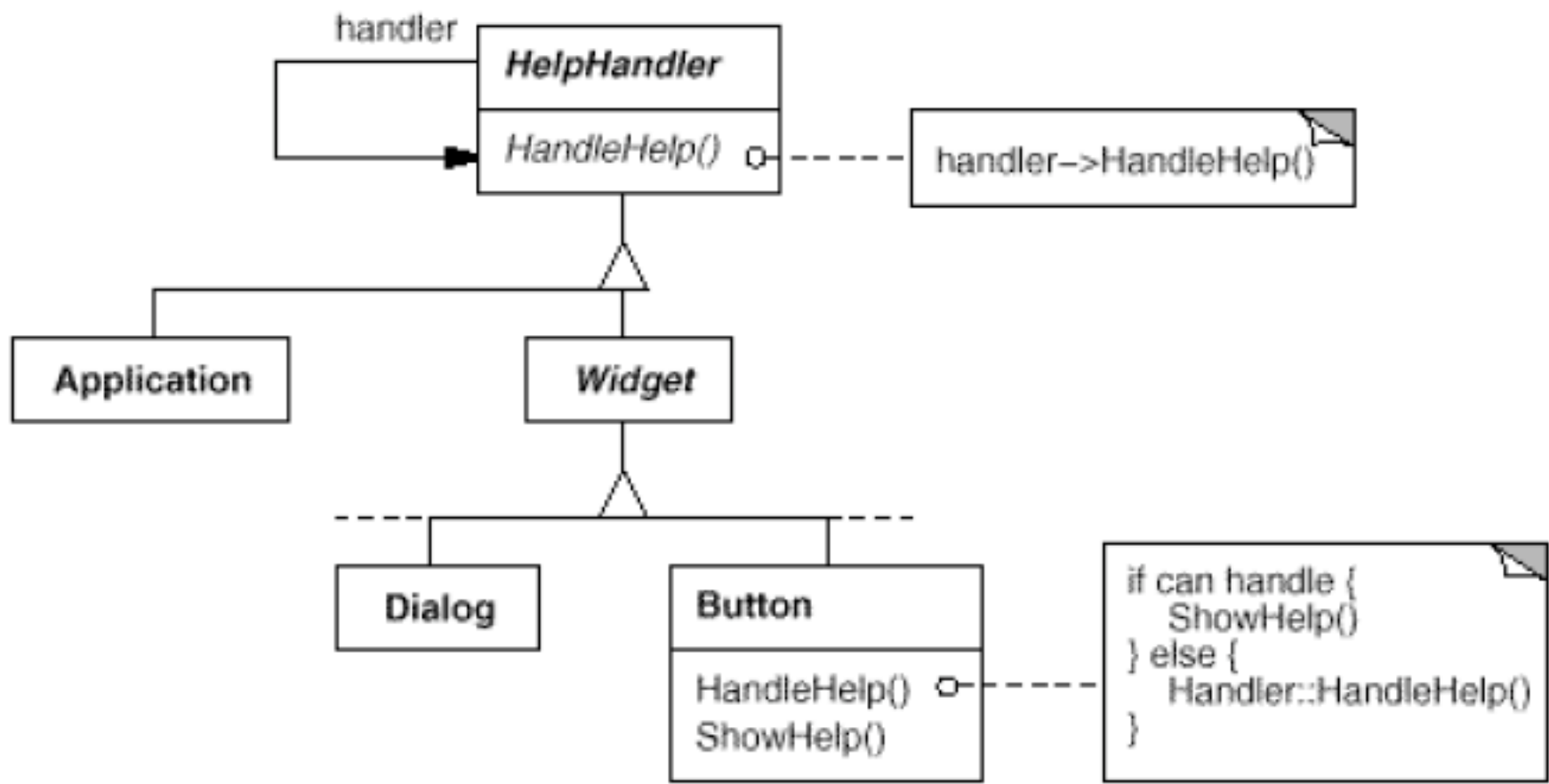
Chain of Responsibility (CoR)

- 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.
- Motivation
 - Consider a context-sensitive help facility for a GUI.
 - Obtain help information on any part of the interface just by clicking on it.
 - The help depends on the selected part of the interface
 - If no specific help information exists a more general help message is displayed

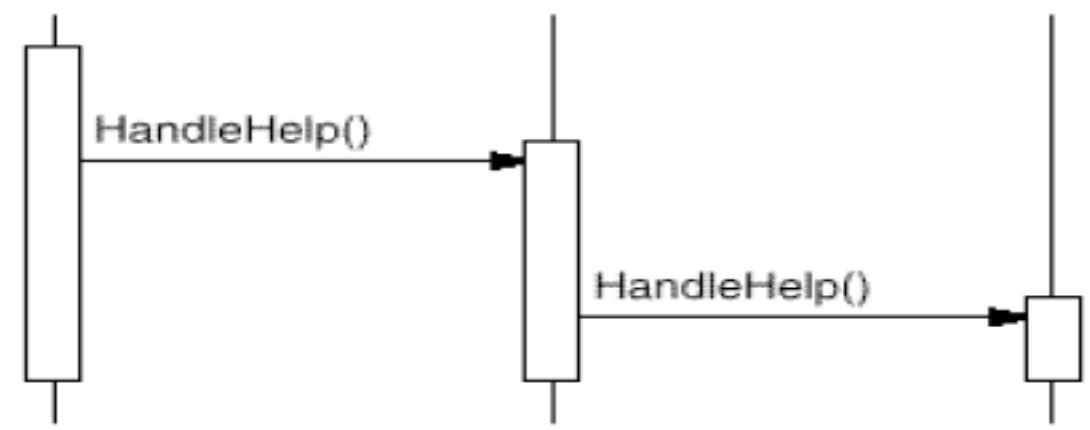


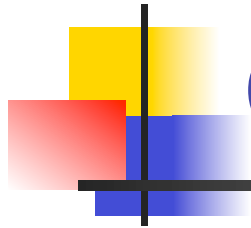
CoR Motivation (continued)

- The problem here is that the object that ultimately *provides* the help isn't known explicitly to the object (e.g., the button) that *initiates* the help request.
- The idea of CoR is to decouple senders and receivers by giving multiple objects a chance to handle a request.
 - The request gets passed along a chain of objects until one of them handles it.



aPrintButton aPrintDialog anApplication

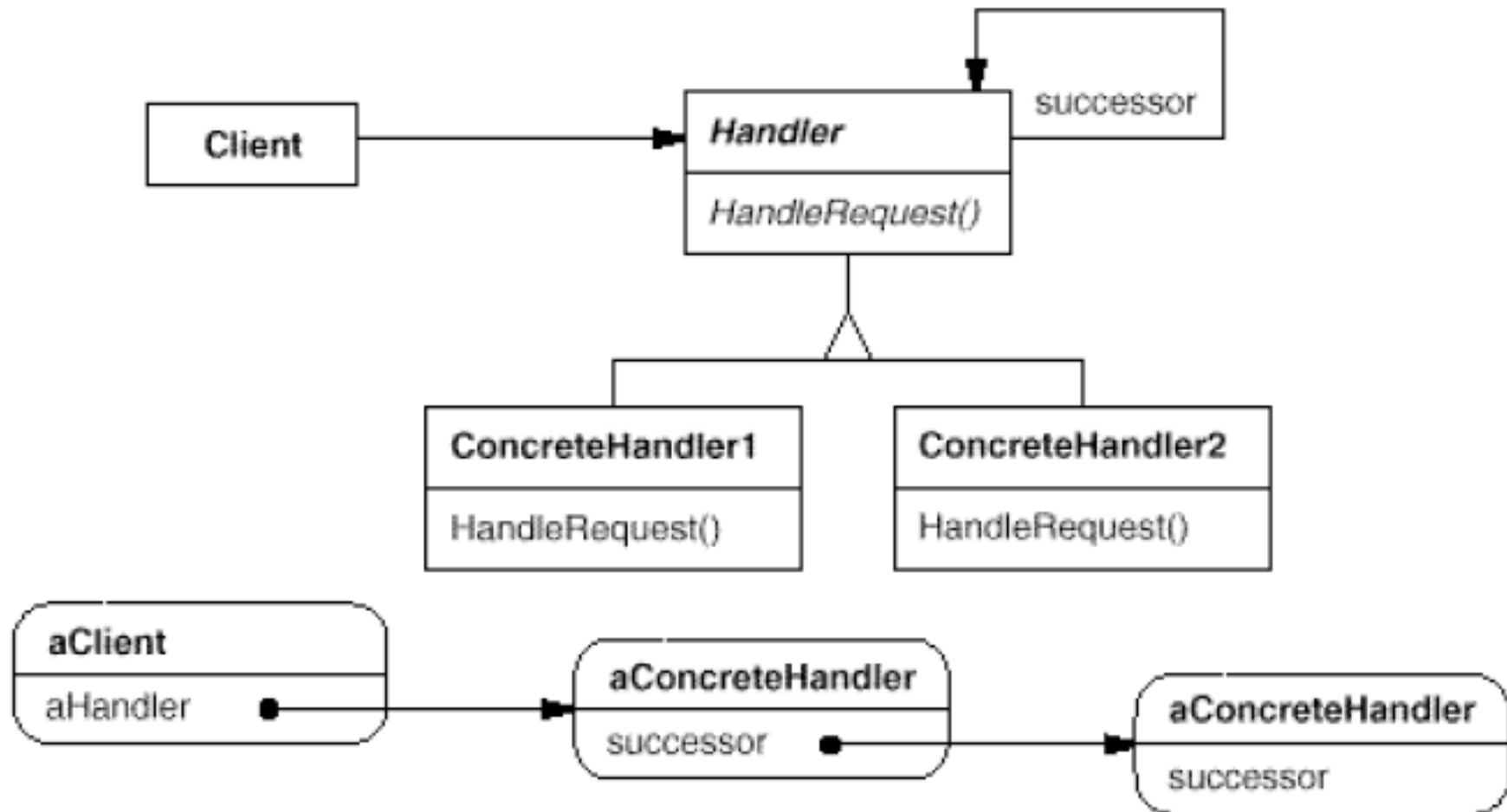


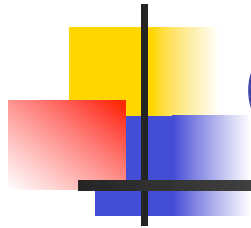


CoR Applicability

- Use the CoR when
 - more than one object may handle a request, and the handler isn't known *a priori*.
 - The handler should be 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.

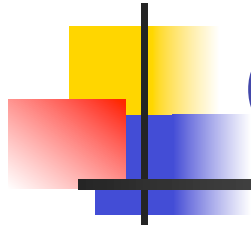
CoR Structure





CoR Participants

- Handler (HelpHandler)
 - defines an interface for handling requests.
 - (optional) implements the successor
- ConcreteHandler (PrintButton, PrintDialog)
 - handles requests it is responsible for.
 - can access its successor.
 - If the ConcreteHandler can handle the to its successor.
- Client
 - initiates the request to a ConcreteHandler



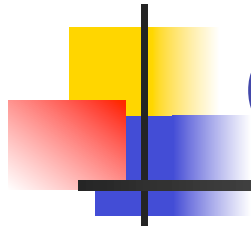
CoR Collaborations

- When a client issues a request, the request propagates along the chain until a ConcreteHandler takes responsibility for handling it.



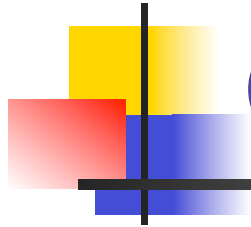
CoR Consequences (1)

- Reduced coupling.
 - CoR frees an object from knowing which other object handles a request.
 - An object only has to know that a request will be handled "appropriately."
 - Both the receiver and the sender have no explicit knowledge of each other
 - an object in the chain doesn't have to know about the chain's structure.
 - CoR can simplify object interconnections.
 - objects keep a single reference to their successor.



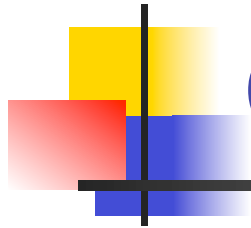
CoR Consequences (2)

- Added flexibility in assigning responsibilities to objects.
 - You can add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time.
 - You can combine this with subclassing to specialize handlers statically.
- Receipt isn't guaranteed.
 - Since a request has no explicit receiver, there's no guarantee it'll be handled
 - A request can also go unhandled when the chain is not configured properly.



CoR Implementation (1)

- Implementing the successor chain.
 - Define new links (usually in the Handler, but ConcreteHandlers could define them instead).
 - Use existing links.
 - use existing object references to form the successor chain.
 - E.g. parent references in a part-whole hierarchy can define a part's successor (see Composite)
 - It works well when the links support the chain you need.
 - If the structure doesn't reflect the required chain of responsibility then you'll have to define redundant links.



CoR Implementation (2)

- Connecting successors.
 - If there are no preexisting references for defining a chain, then you'll have to introduce them yourself.
 - the Handler usually maintains the successor as well.
 - That lets the handler provide a default implementation of `HandleRequest` that forwards the request to the successor (if any).
 - `ConcreteHandler` doesn't have to override the forwarding operation always



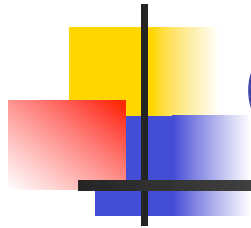
CoR Implementation (3)

- Representing requests.
 - the request is a hard-coded operation invocation
 - convenient and safe, but only a fixed set of requests
 - Use a single handler function that takes a request code as parameter.
 - the sender and receiver agree on the encoding request
 - more flexible, but:
 - requires conditional statements based on the “request code”
 - no type-safe way to pass parameters
 - Obviously this is less safe than invoking an operation directly.
 - we can use Request class representing requests explicitly,
 - new kinds of requests can be defined by subclassing.
 - Handlers must know the kind of request



CoR Implementation (4)

```
void Handler::HandleRequest (Request* theRequest) {  
    switch (theRequest->GetKind()) {  
        case Help:  
            // cast argument to appropriate type  
            HandleHelp((HelpRequest*) theRequest);  
            break;  
  
        case Print:  
            HandlePrint((PrintRequest*) theRequest);  
            // ...  
            break;  
  
        default:  
            // ...  
            break;  
    }  
}
```



CoR Sample Code

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
    HelpHandler(HelpHandler* = 0, Topic = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
    Topic _topic;
};
```

```

HelpHandler::HelpHandler (
    HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}

void HelpHandler::HandleHelp () {
    if (_successor != 0) {
        _successor->HandleHelp();
    }
}

class Widget : public HelpHandler {
protected:
    Widget(Widget* parent, Topic t = NO_HELP_TOPIC);
private:
    Widget* _parent;
};

Widget::Widget (Widget* w, Topic t) : HelpHandler(w, t) {
    _parent = w;
}

```

```

class Button : public Widget {
public:
    Button(Widget* d, Topic t = NO_HELP_TOPIC);

    virtual void HandleHelp();
    // Widget operations that Button overrides...
};

Button::Button (Widget* h, Topic t) : Widget(h, t) { }

void Button::HandleHelp () {
    if (HasHelp()) {
        // offer help on the button
    } else {
        HelpHandler::HandleHelp();
    }
}

```

```

class Dialog : public Widget {
public:
    Dialog(HelperHandler* h, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();

    // Widget operations that Dialog overrides...
    // ...
};

Dialog::Dialog (HelperHandler* h,  Topic t) : Widget(0) {
    SetHandler(h, t);
}

void Dialog::HandleHelp () {
    if (HasHelp()) {
        // offer help on the dialog
    } else {
        HelperHandler::HandleHelp();
    }
}

```



```
class Application : public HelpHandler {
public:
    Application(Topic t) : HelpHandler(0, t) { }

    virtual void HandleHelp();
    // application-specific operations...
};

void Application::HandleHelp () {
    // show a list of help topics
}
```

```
const Topic PRINT_TOPIC = 1;
const Topic PAPER_ORIENTATION_TOPIC = 2;
const Topic APPLICATION_TOPIC = 3;

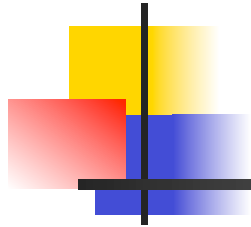
Application* application = new Application(APPLICATION_TOPIC);
Dialog* dialog = new Dialog(application, PRINT_TOPIC);
Button* button = new Button(dialog, PAPER_ORIENTATION_TOPIC);

button->HandleHelp();
```



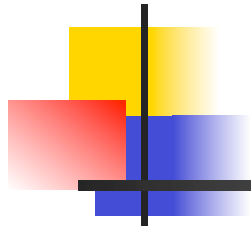
CoR Known Uses and Related Patterns

- Known Uses
 - ...many applications
- Related Patterns
 - Composite



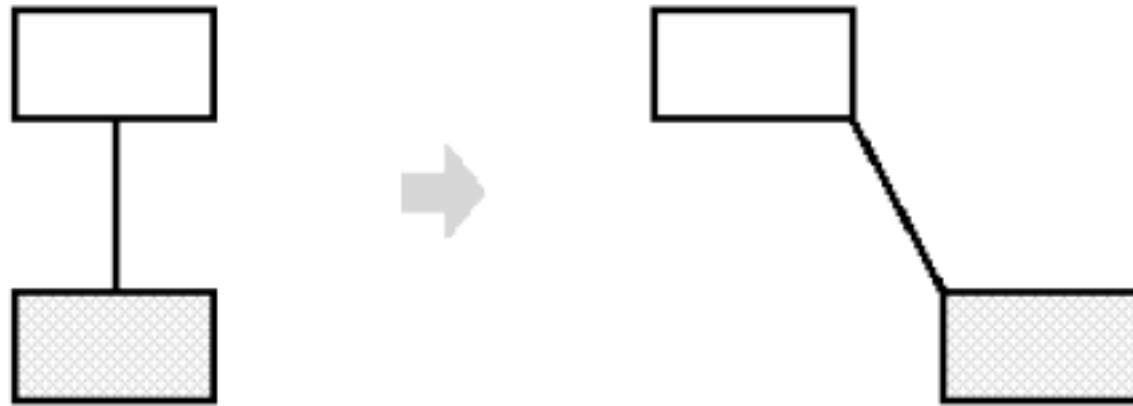
Memento (MMT)

- Intent
 - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- Also Known As
 - Token



MMT Motivation (1)

- Sometimes it's necessary to record the internal state of an object.
 - To implement checkpoints or undo mechanisms
 - You must save state information so that you can restore the objects state
 - The state is usually inaccessible to other objects
 - Exposing this state would violate encapsulation
- Consider for example a graphical editor that supports connectivity between objects.
 - A user can connect two rectangles with a line, and the rectangles stay connected when the user moves either of them.



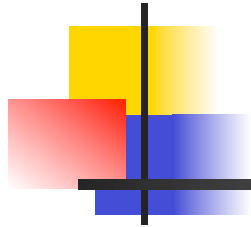
A **ConstraintSolver** class records connections as they are made and generates mathematical equations that describe them. The undo mechanism must work with ConstraintSolver.

To solve the problem we use a memento object.

A **memento** is an object that stores a snapshot of the internal state of another object—the memento's **originator**.

The undo mechanism will request a memento from the originator

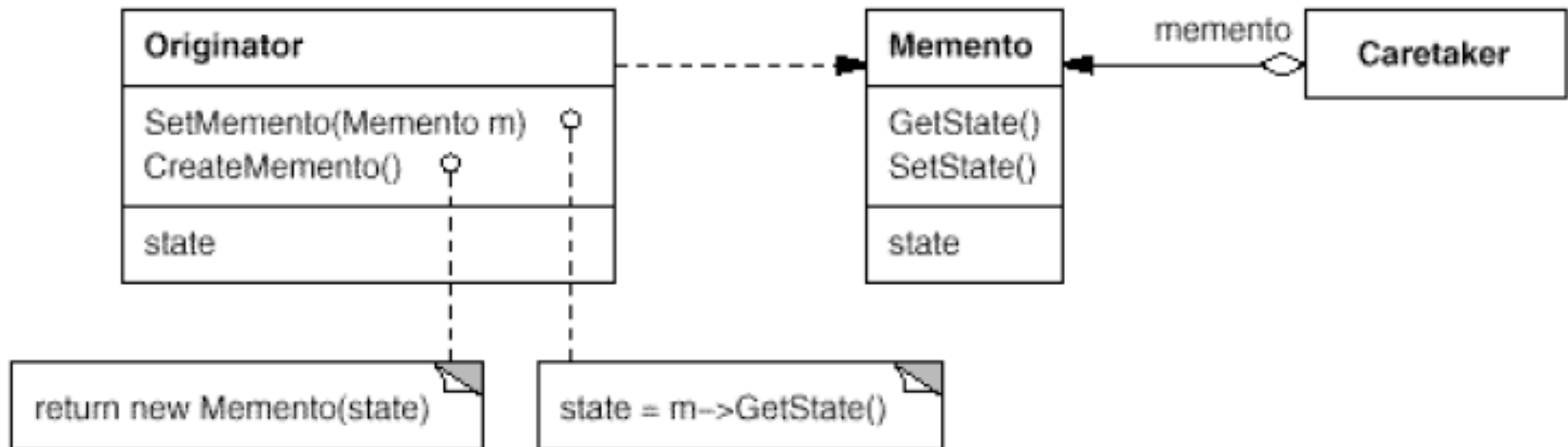
The memento is "opaque" to other objects.



MMT Applicability

- Use the Memento pattern 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 implementation details and break the object's encapsulation.

MMT Structure





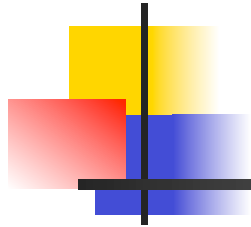
MMT Participants (1)

- Memento (SolverState)
 - stores internal state of the Originator object.
 - may store as much or as little of the originator's internal state as necessary at its originator's discretion.
 - protects against access by objects other than the originator.
 - MMTs have effectively two interfaces.
 - Caretaker sees a narrow interface to the Memento
 - it can only pass the memento to other objects.
 - Originator sees a wide interface
 - it access all the data necessary to restore itself to its previous state.



MMT Participants (2)

- Originator (ConstraintSolver)
 - creates a memento containing a snapshot of its current internal state.
 - uses the memento to restore its internal state.
- Caretaker (undo mechanism)
 - is responsible for the memento's safekeeping.
 - never operates on or examines the contents of a memento.



MMT Collaborations

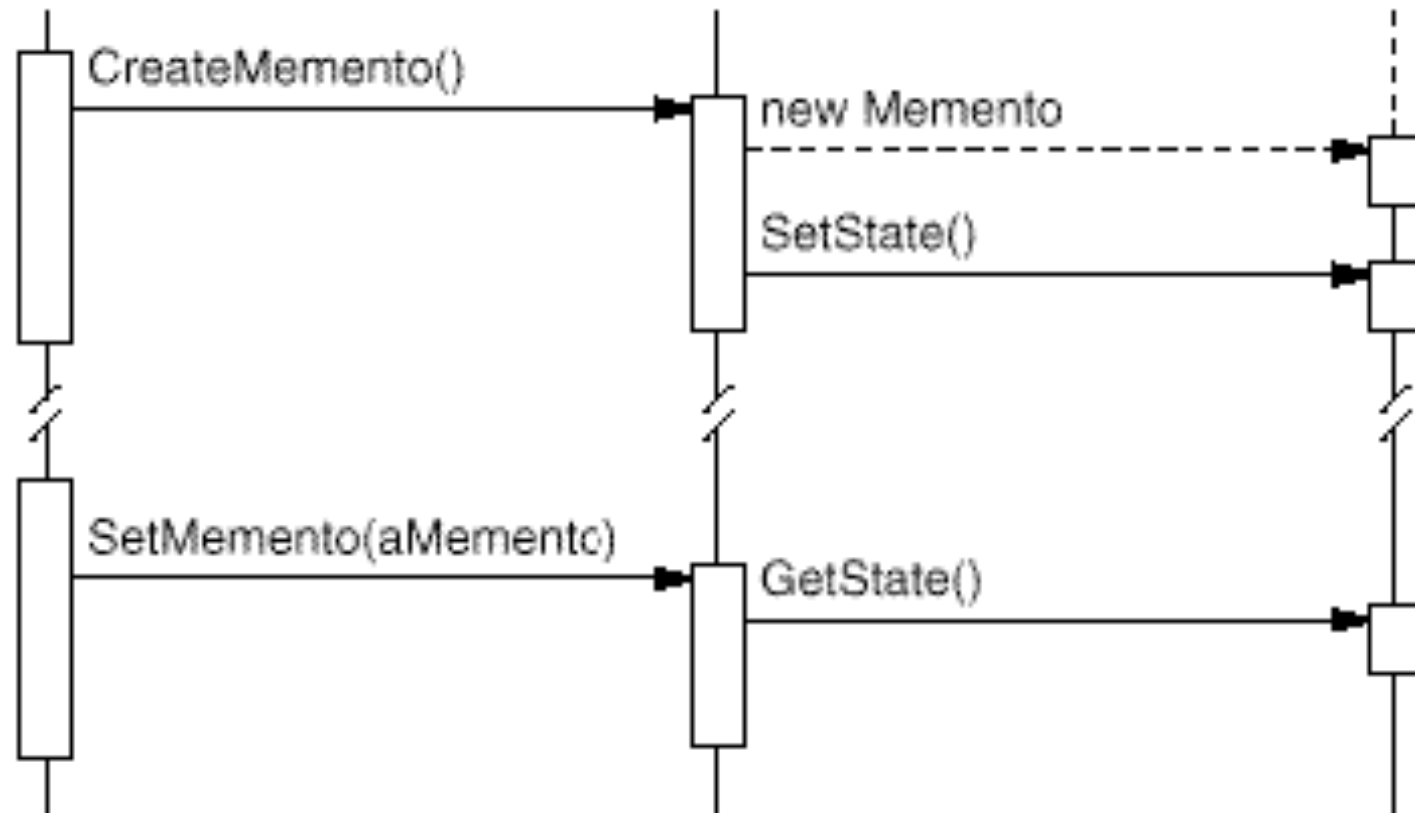
- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator
- Sometimes the caretaker won't pass the memento back to the originator, because the originator might never need to revert to an earlier state.
- Mementos are passive. Only the originator that created a memento will assign or retrieve its state.

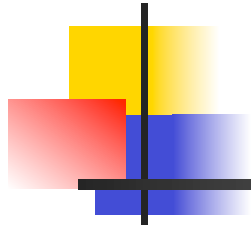
MMT Collaborations

aCaretaker

anOriginator

aMemento





MMT Consequences (1)

- Preserving encapsulation boundaries.
 - MMT shields other objects from potentially complex Originator internals, thereby preserving encapsulation boundaries.
- MMT simplifies Originator.
 - Having clients manage the state they ask for simplifies Originator and keeps clients from having to notify originators when they're done.



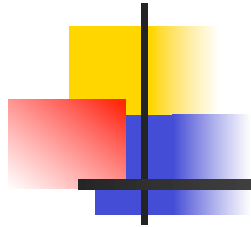
MMT Consequences (2)

- Using mementos might be expensive.
 - Mementos might incur considerable overhead if
 - Originator must copy large amounts of information
 - clients create and return mementos to the originator often enough.
 - Unless encapsulating and restoring Originator state is cheap, the pattern might not be appropriate.
- Defining narrow and wide interfaces.
 - It may be difficult in some languages to ensure that only the originator can access the memento's state.
- Hidden costs in caring for mementos.
 - A caretaker is responsible for deleting the MMTs it cares for.
 - Hence an otherwise lightweight caretaker might incur large storage costs when it stores mementos.



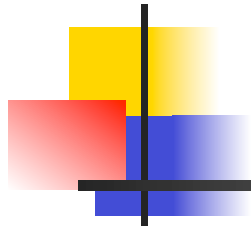
MMT Implementation (1)

- Language support.
 - Mementos have two interfaces:
 - a wide one for originators and
 - a narrow one for other objects.
 - C++ lets you do this by making the Originator a friend of Memento and making Memento's wide interface private.
 - Only the narrow interface should be declared public



MMT Implementation (2)

- Storing incremental changes.
 - If MMT are created in a predictable sequence, then Memento can save just the incremental changes.
 - E.g. undoable commands in a history list
 - A specific order in which commands can be undone and redone => mementos can store just the incremental changes



MMT Sample Code

```
class Graphic;
    // base class for graphical objects in the graphical editor

class MoveCommand {
public:
    MoveCommand(Graphic* target, const Point& delta);
    void Execute();
    void Unexecute();
private:
    ConstraintSolverMemento* _state;
    Point _delta;
    Graphic* _target;
};
```

```

class ConstraintSolver {
public:
    static ConstraintSolver* Instance();

    void Solve();
    void AddConstraint(
        Graphic* startConnection, Graphic* endConnection
    );
    void RemoveConstraint(
        Graphic* startConnection, Graphic* endConnection
    );

    ConstraintSolverMemento* CreateMemento();
    void SetMemento(ConstraintSolverMemento*);
private:
    // nontrivial state and operations for enforcing
    // connectivity semantics
};

class ConstraintSolverMemento {
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();

    // private constraint solver state
};

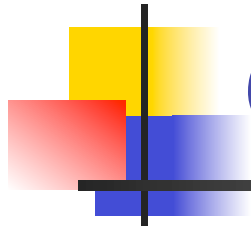
```

```
void MoveCommand::Execute () {  
    ConstraintSolver* solver = ConstraintSolver::Instance();  
    _state = solver->CreateMemento(); // create a memento  
    _target->Move(_delta);  
    solver->Solve();  
}  
  
void MoveCommand::Unexecute () {  
    ConstraintSolver* solver = ConstraintSolver::Instance();  
    _target->Move(-_delta);  
    solver->SetMemento(_state); // restore solver state  
    solver->Solve();  
}
```



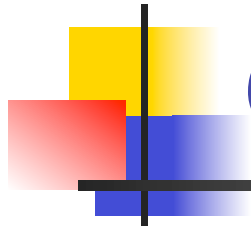
MMT Related Patterns

- Command
- Iterator



Observer (OB)

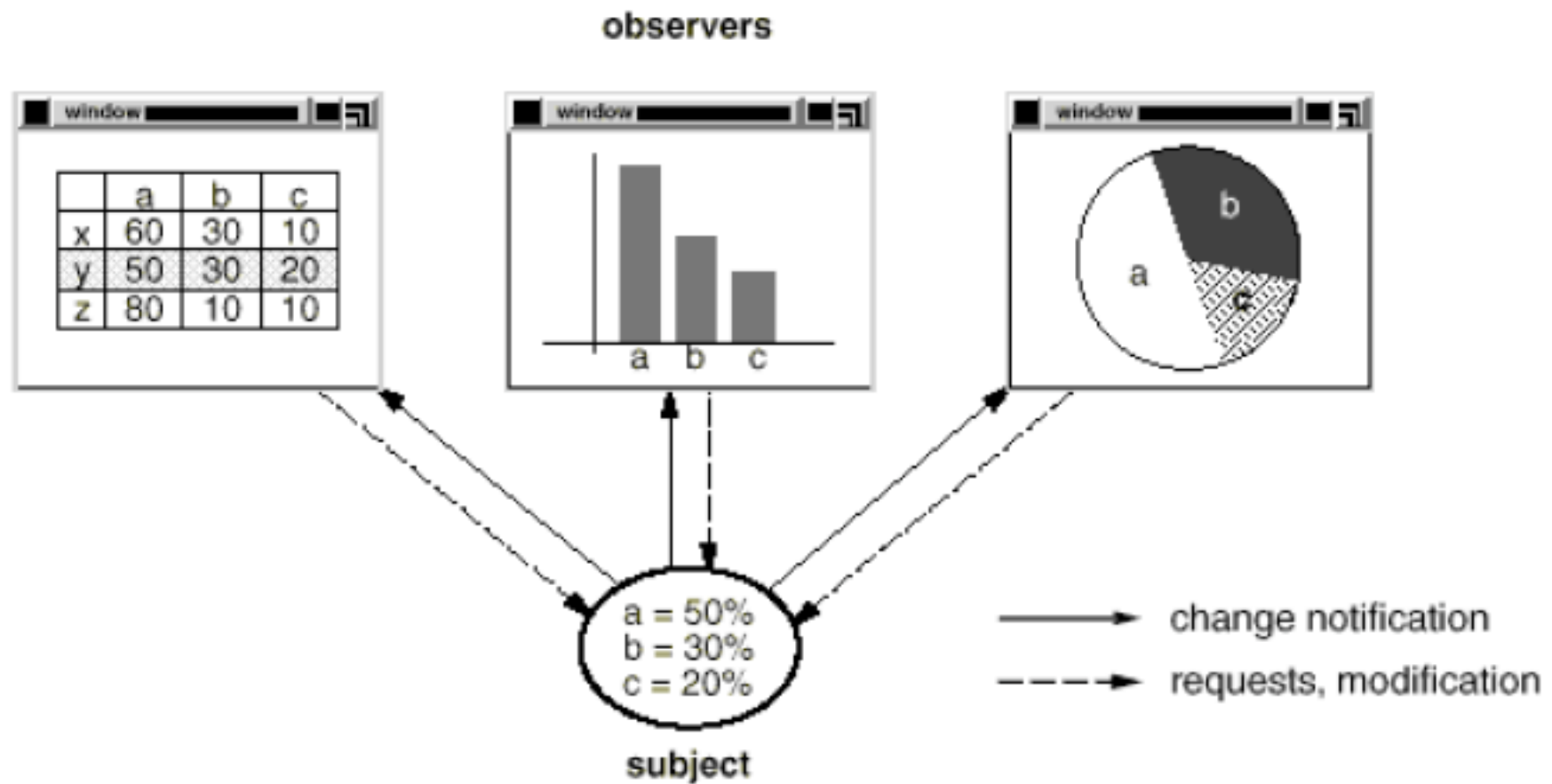
- Intent
 - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Also Known As
 - Dependents, Publish-Subscribe

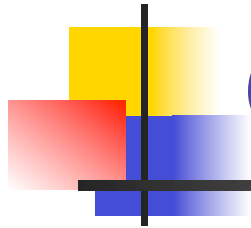


OB Motivation (1)

- Given a collection of cooperating classes there is the need to maintain consistency
- You don't want to make the classes tightly coupled => that reduces their reusability.
- E.g. graphical user interface toolkits where:
 - separate the presentational aspects of the user interface from the underlying application data
 - Classes defining application data and presentations can be reused independently.
 - They can work together, too

OB Motivation (1)





OB Motivation (2)

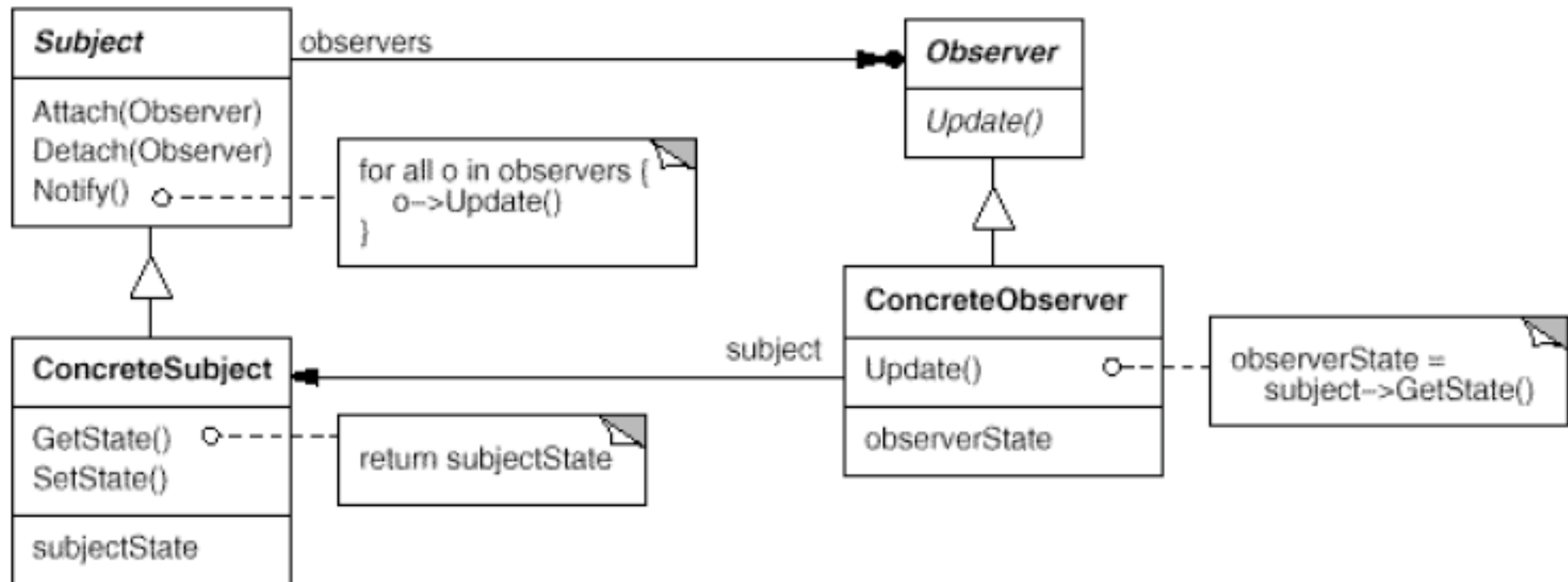
- The data object is represented by
 - a spreadsheet and/or a bar chart
 - no limit on the number of dependent objects to two
 - there may be any number of different user interfaces to the same data.
- The key objects: **subject** and **observer**.
 - A subject may have any number of dependent observers.
 - All observers are notified whenever the subject undergoes a change in state.
 - AKA **publish-subscribe**
 - Subject=publisher; Observer=Subscriber



OB Applicability

- Use OB when:
 - an abstraction has two aspects, one dependent on the other.
 - Encapsulating these aspects in separate objects lets you vary and reuse them independently.
 - a change to one object requires changing others, and you don't know how many objects there are.
 - an object notifies other objects without making assumptions about who these objects are.
 - In other words, you don't want these objects tightly coupled.

OB Structure





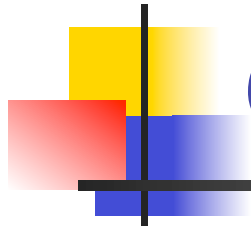
OB Participants (1)

- Subject

- knows its observers. Any number of Observer objects may observe a subject.
- provides an interface for attaching and detaching Observer objects.

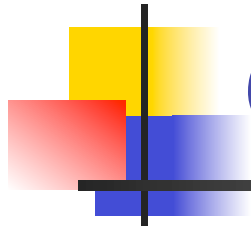
- Observer

- defines an updating interface for objects that should be notified of changes in a subject.



OB Participants (2)

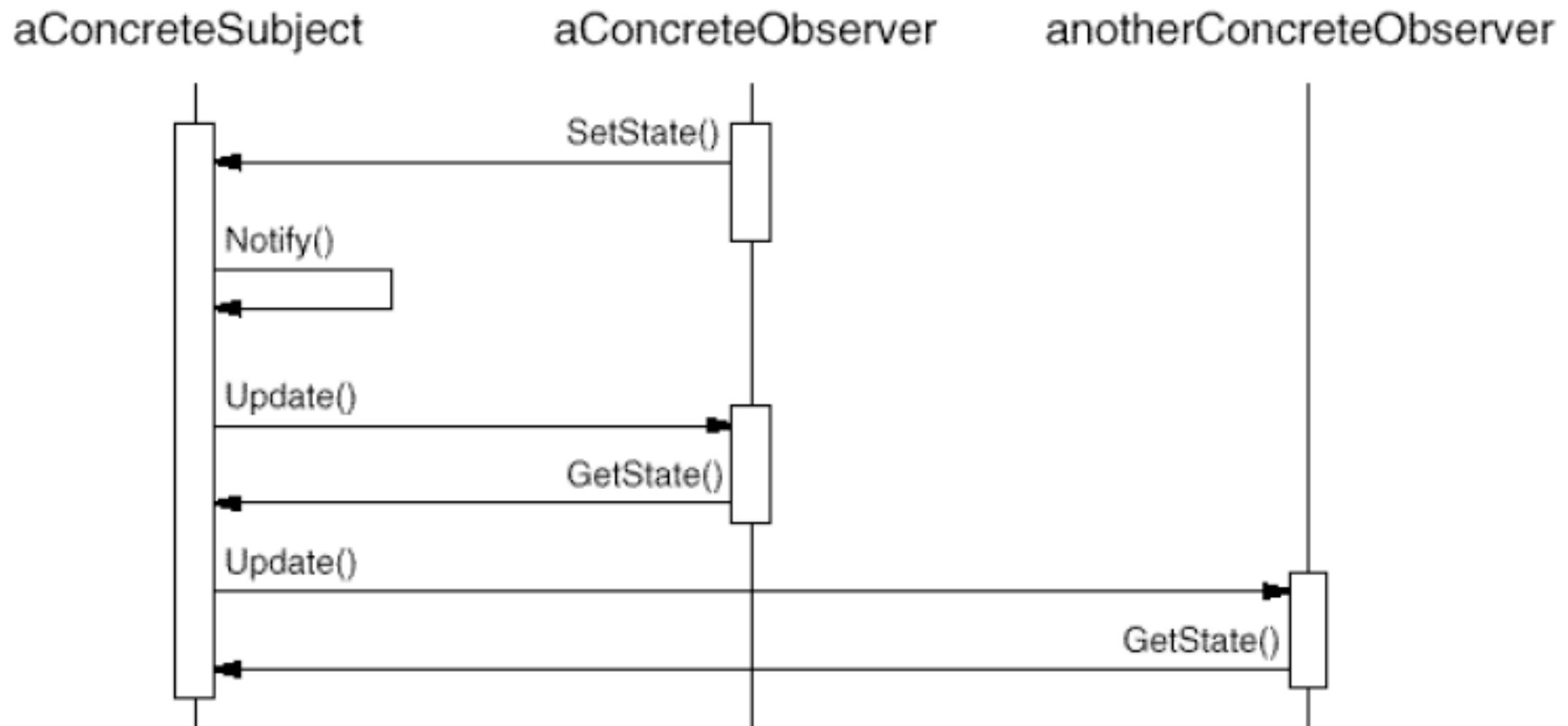
- ConcreteSubject
 - stores state of interest to ConcreteObserver objects.
 - sends a notification to its observers when its state changes.
- ConcreteObserver
 - maintains a reference to a ConcreteSubject object.
 - stores state that should stay consistent with the subject's.
 - implements the Observer updating interface to keep its state consistent with the subject's.



OB Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

OB Collaborations





OB Consequences (1)

- OB lets you vary subjects and observers independently.
- You can reuse subjects without reusing their observers, and vice versa.
- OB lets you add observers without modifying the subject or other observers.



OB Consequences (2)

- Abstract coupling between Subject and Observer.
 - A subject knows its list of observers each conforming to the abstract Observer
 - A subject doesn't know the concrete class of any observer.
 - the coupling is abstract and minimal.
- Loose coupling => Observer and Subject can belong to different layers of abstraction in a system.
 - A lower-level subject can communicate and inform a higher-level observer



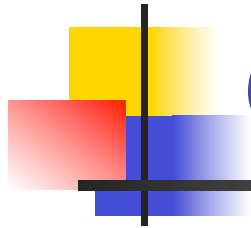
OB Consequences (2)

- Support for broadcast communication.
 - the notification that a subject sends needn't specify its receiver.
 - The notification is broadcast automatically to all interested objects that subscribed to it.
 - This gives you the freedom to add and remove observers at any time.
- Unexpected updates.
 - Observers do not know each other's presence
 - An innocuous operation on the subject may cause a cascade of updates
 - Dependency criteria may lead to spurious updates, which can be hard to track down.
 - aggravated by the fact that the simple update protocol provides no details on what changed in the subject.



OB Implementation (1)

- Mapping subjects to their observers.
 - store references to the observers
 - may be too expensive when there are many subjects and few observers.
 - Use an associative look-up (e.g., a hash table) to maintain the subject-to-observer mapping.
 - No overhead for subject with no observers
 - that increases the cost of accessing the observers.
- Observing more than one subject.
 - If an observer depends on several subjects
 - extend the Update interface in such cases to let the observer know which subject is sending the notification.
 - The subject can simply pass itself as a parameter



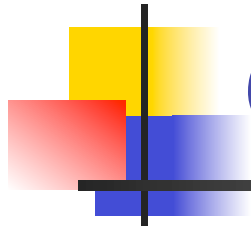
OB Implementation (2)

- Who triggers the update (calls Notify)?
 - state-setting operations on Subject call Notify
 - clients don't have to remember to call Notify on the subject.
 - several consecutive operations will cause several consecutive updates, which may be inefficient.
 - Make clients responsible for calling Notify
 - the update can be triggered after a series of state changes has been made
 - avoiding needless intermediate updates.
 - clients have an added responsibility to trigger the update.
 - That makes errors more likely, since clients might forget to call Notify.



OB Implementation (3)

- Dangling references to deleted subjects.
 - Deleting a subject should not produce dangling references in its observers.
 - the subject notify its observers as it is deleted
- Making sure Subject state is self-consistent before notification.
 - Subject state must be self-consistent before calling Notify
 - observers query the subject for its current state in the course of updating their own state.
 - Use a template method before calling notify.



OB Implementation (4)

- Avoiding observer-specific update protocols
 - **push model:**
 - the subject sends observers detailed information about the change, whether they want it or not.
 - assumes subjects know something about their observers'
 - might make observers less reusable,
 - **pull model:**
 - the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.
 - emphasizes the subject's ignorance of its observers, whereas the push model
 - may be inefficient, because Observer classes must ascertain what changed without help from the Subject.

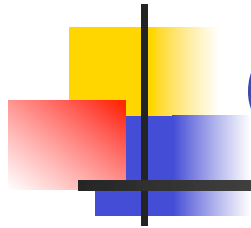


OB Implementation (5)

- Specifying modifications of interest explicitly.
 - improve update efficiency registering observers only for specific events of interest.
 - the subject informs only those observers that have registered interest in that event.

```
void Subject::Attach(Observer*, Aspect& interest);
```

```
void Observer::Update(Subject*, Aspect& interest);
```

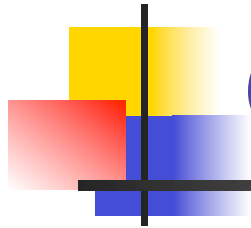


OB Implementation (6)

- Specifying modifications of interest explicitly.
 - improve update efficiency registering observers only for specific events of interest.
 - the subject informs only those observers that have registered interest in that event.

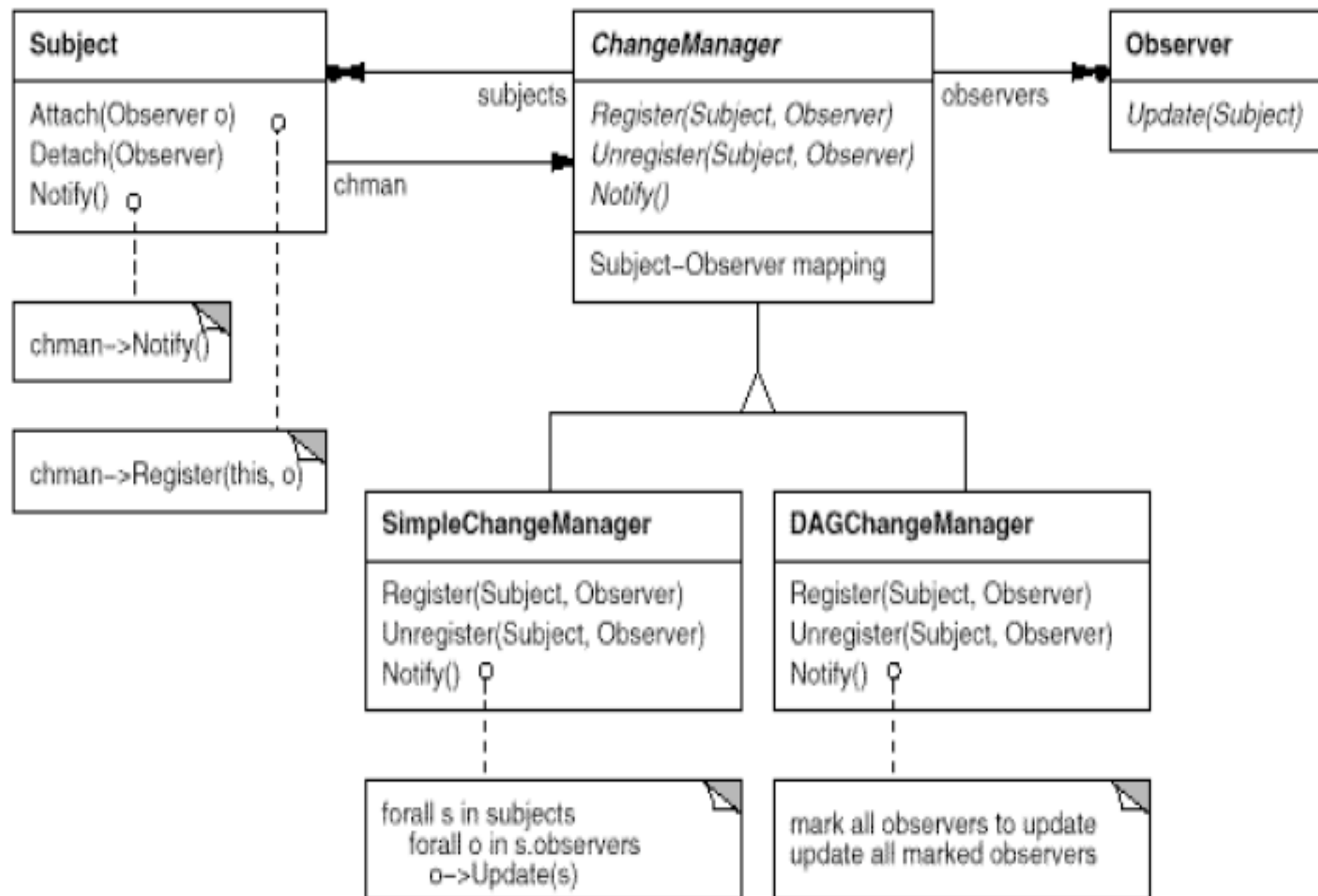
```
void Subject::Attach(Observer*, Aspect& interest);
```

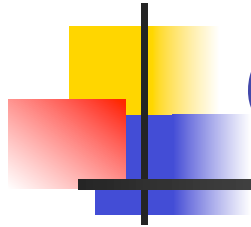
```
void Observer::Update(Subject*, Aspect& interest);
```



OB Implementation (7)

- Encapsulating complex update semantics.
 - If When the dependency between subjects and observers is complex, a ChangeManager may be used.
 - The Change Manager
 - maps a subject to its observers
 - defines a particular update strategy.
 - updates all dependent observers at the request of a subject.
 - It is a Mediator





OB Implementation (8)

- Combining the Subject and Observer classes.
 - If language lack multiple inheritance (like Smalltalk) don't define separate Subject and Observer classes but combine their interfaces in one class.
 - Objects act as both a subject and an observer without multiple inheritance.



OB Sample Code

```
class Observer {  
public:  
    virtual ~ Observer();  
    virtual void Update(Subject* theChangedSubject) = 0;  
protected:  
    Observer();  
};
```

```

class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}

```

```
class ClockTimer : public Subject {
public:
    ClockTimer();

    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();

    void Tick();
};

void ClockTimer::Tick () {
    // update internal time-keeping state
    // ...
    Notify();
}
```

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();

    virtual void Update(Subject*);
        // overrides Observer operation

    virtual void Draw();
        // overrides Widget operation;
        // defines how to draw the digital clock
private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock:: DigitalClock () {
    _subject->Detach(this);
}
```

```
void DigitalClock::Update (Subject* theChangedSubject) {  
    if (theChangedSubject == _subject) {  
        Draw();  
    }  
}  
  
void DigitalClock::Draw () {  
    // get the new values from the subject  
  
    int hour = _subject->GetHour();  
    int minute = _subject->GetMinute();  
    // etc.  
  
    // draw the digital clock  
}
```

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};
```

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```



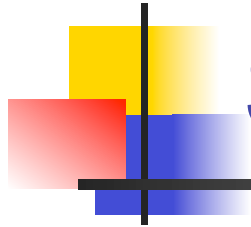

OB Known Uses and Related Patterns

- Known Uses
 - ... MVC
- Related Patterns
 - Mediator
 - Singleton



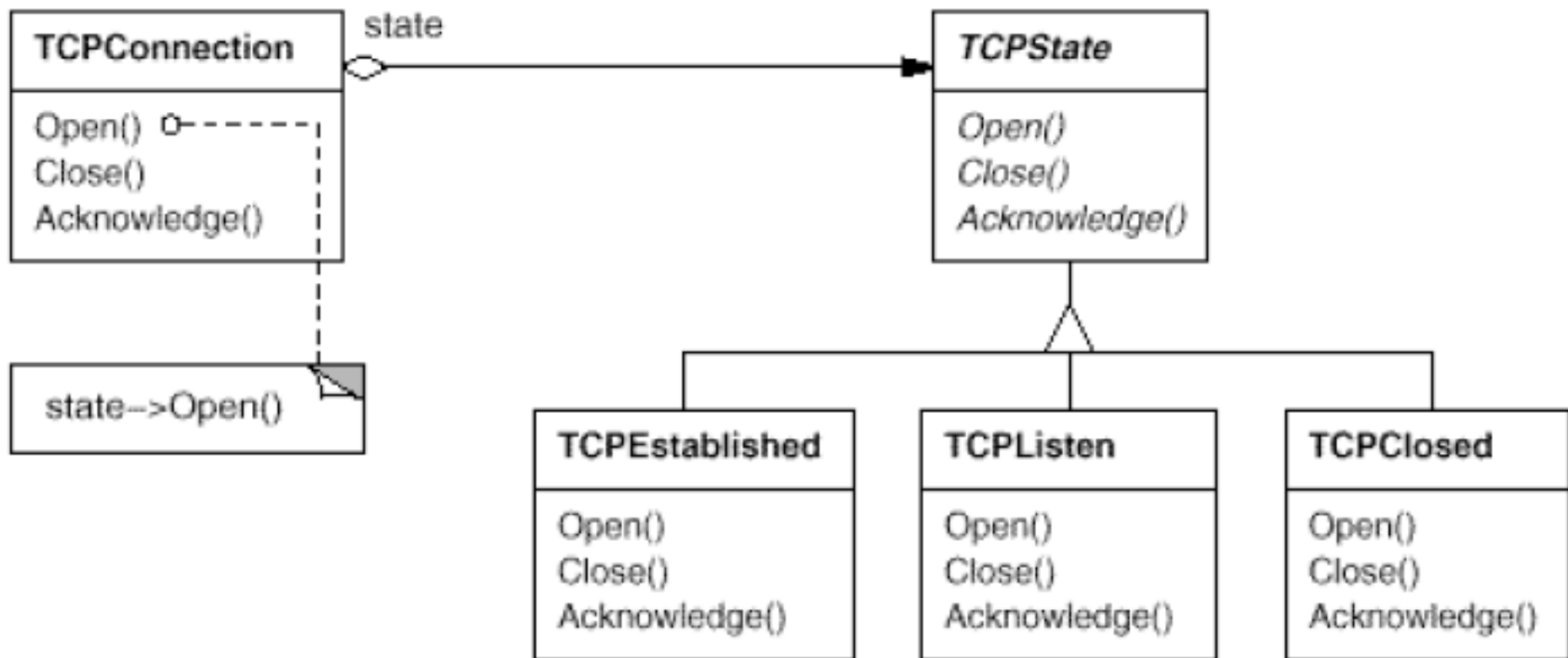
State (ST)

- Intent
 - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Also Known As
 - Objects for States



ST Motivation

- Consider a class TCPConnection that represents a network connection.
- A TCPConnection object can be in one of several different states:
 - Established, Listening, Closed.
- When a TCPConnection object receives requests from other objects, it responds differently depending on its current state.



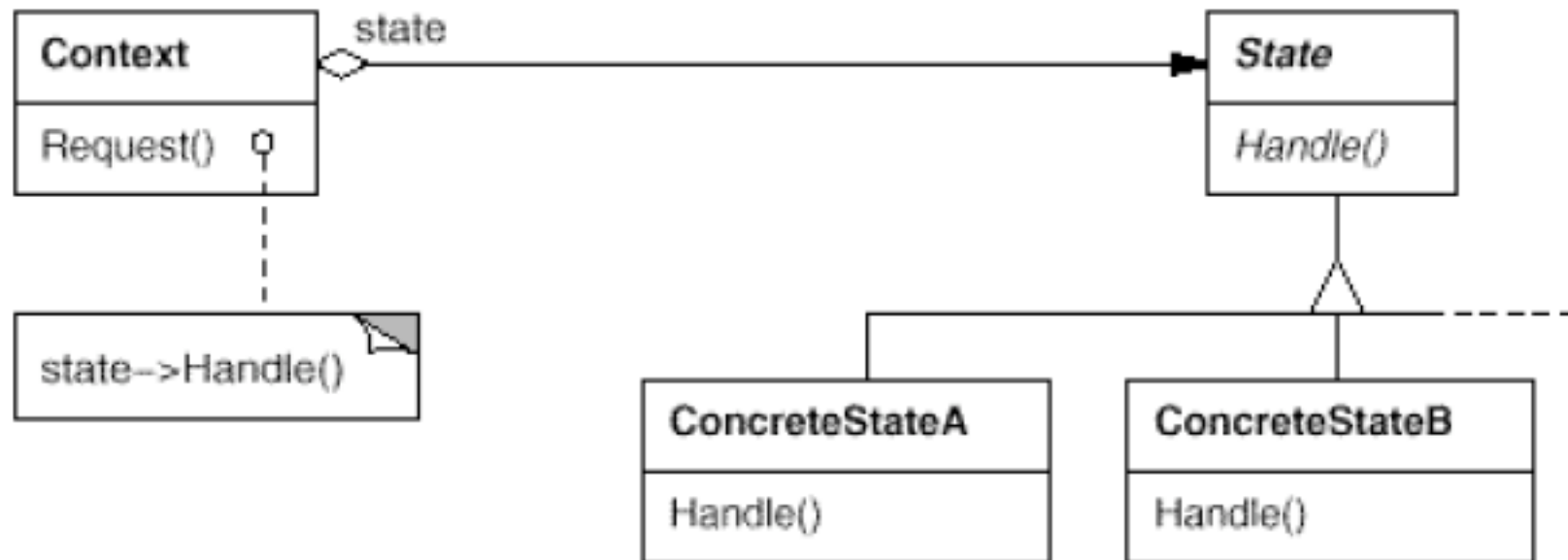
The TCPState class declares an interface common to all classes that represent different operational states.
Subclasses of TCPState implement state-specific behavior.

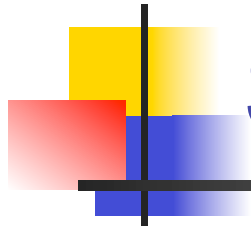


ST Applicability

- Use ST 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 that depend on the object's state.
 - enumerated constants + conditional structure
VS
a separate class for each branch of the conditional
 - treat the object's state as an object in its own right that can vary independently from other objects.

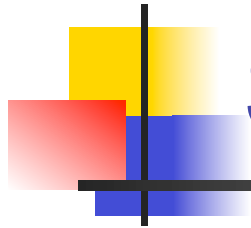
ST Structure





ST Participants

- Context (TCPConnection)
 - defines the interface of interest to clients.
 - maintains an instance of a ConcreteState subclass that defines the current state.
- State (TCPState)
 - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed)
 - each subclass implements a behavior associated with a state of the Context.



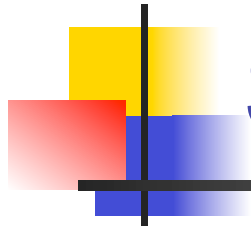
ST Collaborations

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request.
 - This lets the State object access the context if necessary.
- Context is the primary interface for clients.
 - Clients can configure a context with State objects.
 - Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.



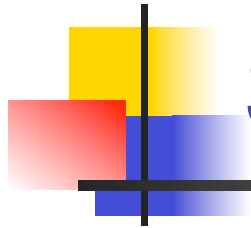
ST Consequences (1)

- It localizes state-specific behavior and partitions behavior for different states.
- ST puts all behavior associated with a particular state into one object.
 - new states and transitions can be added easily by defining new subclasses.
- large conditional statements are undesirable.
 - make the code less explicit
 - are difficult to modify and extend.
- ST imposes structure on the code and makes its intent clearer.



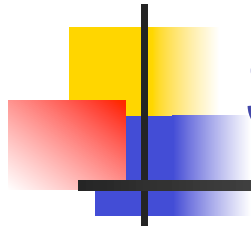
ST Consequences (2)

- It makes state transitions explicit.
 - If state is defined in terms of internal data values => state transitions have no explicit representation
 - with separate objects for different states makes the transitions more explicit.
 - State objects can protect the Context from inconsistent internal states
 - Transitions rebind one variable not several
- State objects can be shared.
 - If State objects have no instance variables contexts can share a State object (Flyweight)
 - no intrinsic state only behavior.



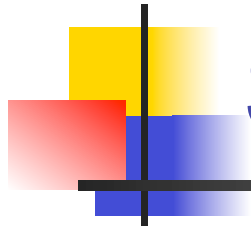
ST Implementation (1)

- Who defines the state transitions?
 - ST does not specify which participant defines the criteria for state transitions.
 - They can be implemented entirely in the Context.
 - the State subclasses themselves specify their successor state and when to make the transition.
 - This requires adding an interface to the Context that lets State objects set the Context's current state explicitly.
 - A disadvantage is that one State subclass will have knowledge of at least one other, which introduces implementation dependencies between subclasses.



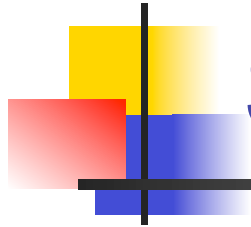
ST Implementation (2)

- A table-based alternative.
 - In C++ you can impose structure on state-driven code
 - use tables to map inputs to state transitions.
 - For each state the table maps every possible input to a succeeding state.
 - It converts conditional code (and virtual functions, in the case of the State pattern) into a table look-up.
 - The main advantage is regularity
 - modify data instead of changing program code.
 - There are some disadvantages:
 - A table look-up is often less efficient than a (virtual) function call.
 - transition criteria less explicit and harder to understand.
 - Is difficult add actions to accompany the state transitions.
 - ST pattern models state-specific behavior, whereas table-driven approach defines state transitions.



ST Implementation (3)

- Creating and destroying State objects.
 - to create State objects only when they are needed
 - preferable when the states that will be entered aren't known at run-time, and contexts change state infrequently
 - important if the State objects store a lot of information.
 - to create them ahead of time and never destroying them.
 - better when state changes occur rapidly
 - Instantiation costs are paid once up-front,
 - no destruction costs at all.
 - Context must keep references to all states that might be entered (it may be inconvenient).
- Using dynamic inheritance.
 - Changing the behavior could be accomplished by changing the object's class at run-time
 - It is not possible in most object-oriented languages



ST Sample Code

- We now give the C++ code for the TCP connection example described in the Motivation

```
class TCPOctetStream;
class TCPState;

class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen();
    void PassiveOpen();
    void Close();

    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
```

```
class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
```



```
void TCPConnection::PassiveOpen () {  
    _state->PassiveOpen(this);  
}  
  
void TCPConnection::Close () {  
    _state->Close(this);  
}  
  
void TCPConnection::Acknowledge () {  
    _state->Acknowledge(this);  
}  
  
void TCPConnection::Synchronize () {  
    _state->Synchronize(this);  
}  
  
TCPConnection::TCPConnection () {  
    _state = TCPClosed::Instance();  
}  
  
void TCPConnection::ChangeState (TCPState* s) {  
    _state = s;  
}
```

```
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();

    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};

class TCPListen : public TCPState {
public:
    static TCPState* Instance();

    virtual void Send(TCPConnection*);
    // ...
};

class TCPClosed : public TCPState {
public:
    static TCPState* Instance();

    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    // ...
};
```

```

void TCPClosed::ActiveOpen (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.

    ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::PassiveOpen (TCPConnection* t) {
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Close (TCPConnection* t) {
    // send FIN, receive ACK of FIN

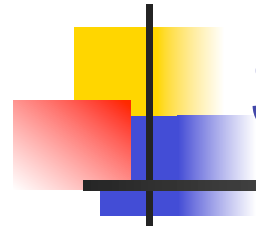
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Transmit (
    TCPConnection* t, TCPOctetStream* o
) {
    t->ProcessOctet(o);
}

void TCPListen::Send (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.

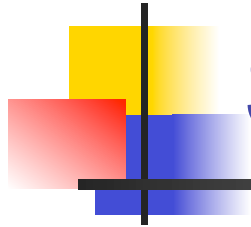
    ChangeState(t, TCPEstablished::Instance());
}

```



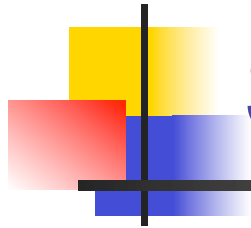
ST Related Patterns

- Flyweighth
- Singleton



Strategy (STG)

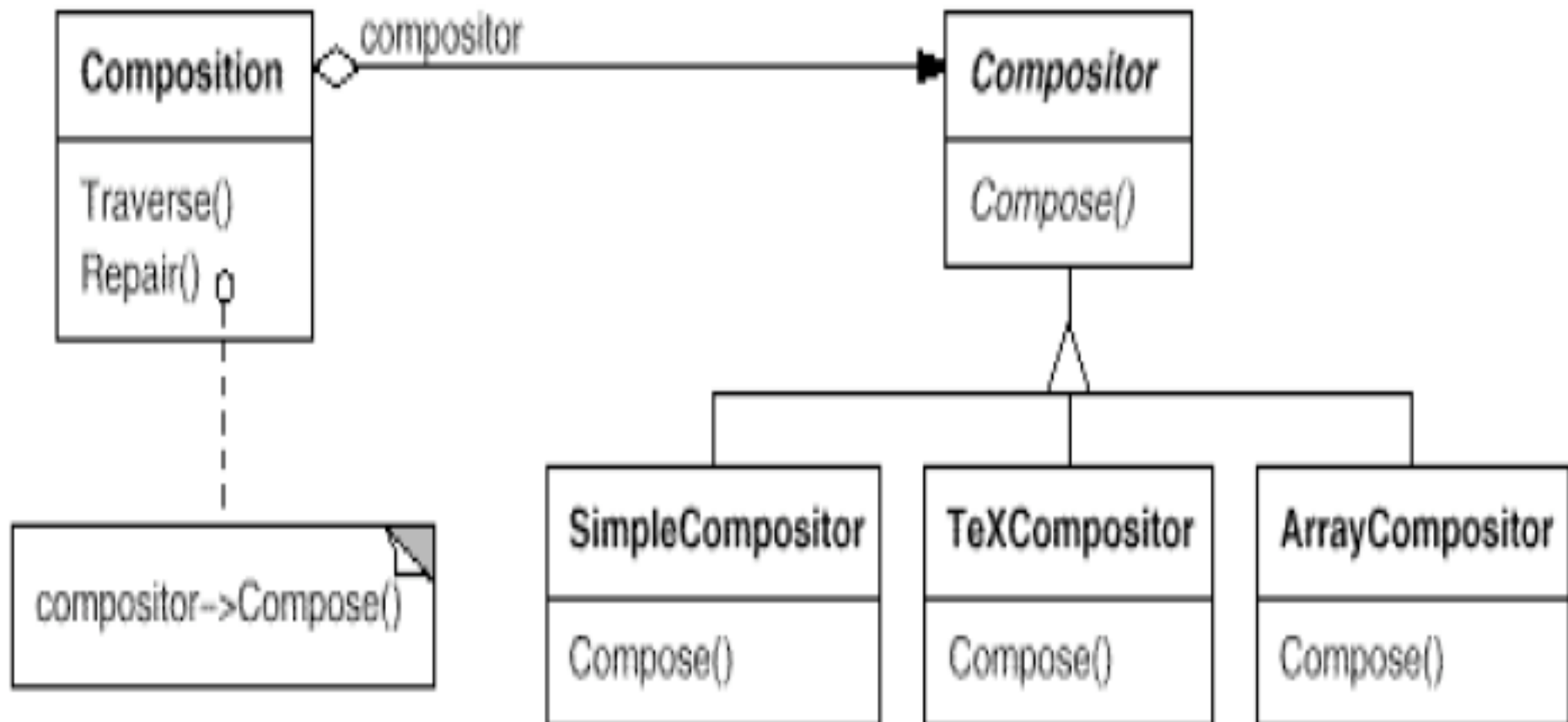
- Intent
 - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Also Known As
 - Policy

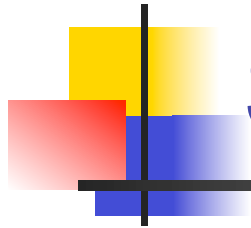


STG Motivation

- Many algorithms exist for breaking a stream of text into lines.
 - Hard-wiring all of them isn't desirable:
 - That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.
 - Different algorithms will be appropriate at different times.
 - Support only the one which is needed
 - It's difficult to add new algorithms and vary existing ones when linebreaking is an integral part of a client.
 - Avoid these problems by defining classes that encapsulate different linebreaking algorithms.
 - Such an algorithm is called a strategy.

STG Motivation

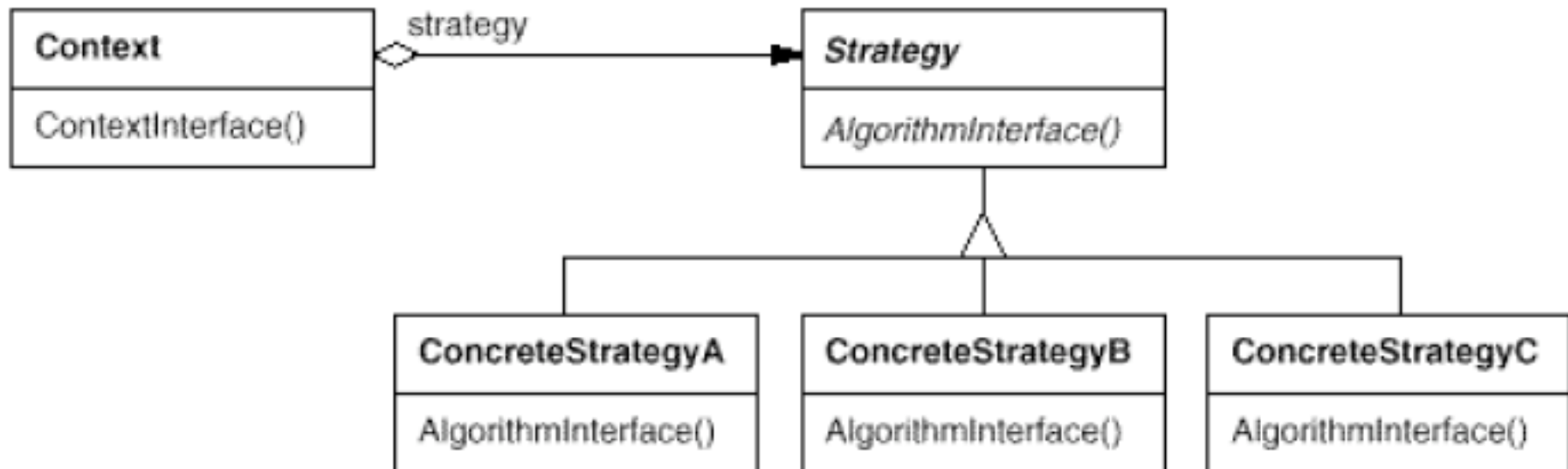


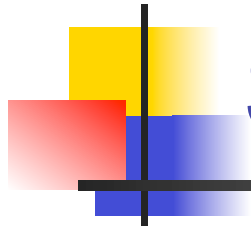


STG Applicability

- Use STG 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.
 - E.g. you might reflect different space/time trade-offs.
 - If an algorithm uses data that clients shouldn't know about.
 - Avoid exposing complex, algorithm-specific data structures.
 - a class defines many behaviors as multiple conditional statements in its operations.
 - move conditional branches into their own Strategy class.

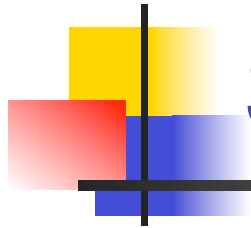
STG Structure





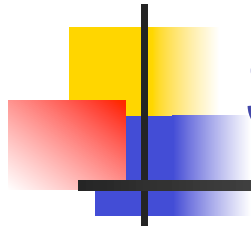
STG Participants

- Strategy (Compositor)
 - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy (SimpleCompositor ...)
 - implements the algorithm using the STG interface.
- Context (Composition)
 - is configured with a ConcreteStrategy object.
 - maintains a reference to a Strategy object.
 - may define an interface that lets Strategy access its data.



STG Collaborations

- Strategy and Context interact to implement the chosen algorithm.
 - A context may pass all data required by the algorithm to the strategy when the algorithm is called.
 - Or, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context
 - clients interact with the context exclusively.
- There is often a family of ConcreteStrategy classes for a client to choose from.



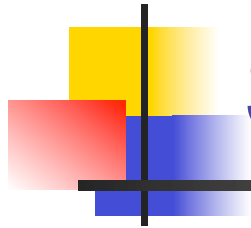
STG Consequences (1)

- Families of related algorithms.
 - STG define a family of algorithms for contexts to reuse; inheritance can help factor out common functionality of the algorithms.
- An alternative to subclassing.
 - By subclassing a Context class hardwires the behavior into Context.
 - It mixes the algorithm implementation with Context's,
 - you can't vary the algorithm dynamically.
 - STG lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.



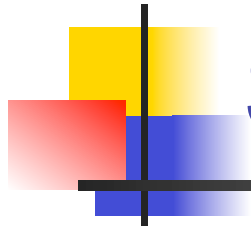
STG Consequences (2)

- Strategies eliminate conditional statements.
 - STG eliminates conditional statements
 - many conditional statements => apply STG.
- A choice of implementations.
 - Chose different implementations of the same behavior (having different time\space trade-offs)
- Clients must be aware of different Strategies
 - (potential drawback) a client must understand how Strategies differ => Clients might be exposed to implementation issues.
 - use the Strategy pattern only when the variation in behavior is relevant to clients.



STG Consequences (3)

- Communication overhead between Strategy and Context.
 - it's likely that some ConcreteStrategies won't use all the information passed to them
 - simple ConcreteStrategies may use none of it!
 - Time may be wasted creating and initializing parameters that never get used.
 - In that case you need tighter coupling between Strategy and Context.
- Increased number of objects.
 - Strategies increase the number of objects
 - => implement strategies as stateless objects that contexts can share (see Flyweight)



STG Implementation (1)

- Defining the Strategy and Context interfaces.
 - A ConcreteStrategy needs efficient access to any data it needs from a context, and vice versa.
 - Two approaches
 - Context pass data in parameters to Strategy operations
 - This keeps Strategy and Context decoupled.
 - Context might pass data the Strategy doesn't need.
 - Context pass itself as an argument
 - eliminating the need to pass anything at all.
 - the strategy can request exactly what it needs.
 - Context must define a more elaborate interface to its data
 - Strategy and Context are coupled more closely.

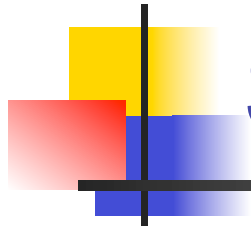


STG Implementation (2)

- Strategies as template parameters.
 - the Strategy is selected at compile-time,
 - it does not have to be changed at run-time

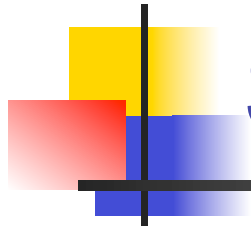
```
template <class AStrategy>                class MyStrategy {
class Context {                            public:
    void Operation()                      void DoAlgorithm();
    { theStrategy.DoAlgorithm(); }      };
    // ...
private:
    AStrategy theStrategy;
};
Context<MyStrategy> aContext;
```

- no need to define an abstract Strategy
- Static binding increase efficiency.



STG Implementation (3)

- Making Strategy objects optional.
 - Context checks if it has a Strategy object before accessing it.
 - If there is one, it uses it normally.
 - If there isn't one, Context carries out default behavior.
 - Clients don't have to deal with Strategy objects at all unless they don't like the default behavior.



STG Sample Code

- Let's see the high-level code for the Motivation example
- The Composition class maintains a collection of Component instances, which represent text and graphical elements in a document.
- A composition arranges component objects into lines using an instance of a Compositor subclass, which encapsulates a linebreaking strategy.

```

class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components;           // the list of components
    int _componentCount;              // the number of components
    int _lineWidth;                   // the Composition's line width
    int* _lineBreaks;                 // the position of linebreaks
                                     // in components
    int _lineCount;                   // the number of lines
};

class Compositor {
public:
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    ) = 0;
protected:
    Compositor();
};

```

```
class SimpleCompositor : public Compositor {
public:
    SimpleCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

```

void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
    int* breaks;

    // prepare the arrays with the desired component sizes
    // ...

    // determine where the breaks are:
class TeXCompositor : public Compositor {
public:
    TeXCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};

```

```

class ArrayCompositor : public Compositor {
public:
    ArrayCompositor(int interval);

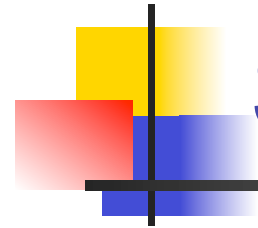
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};

```

```

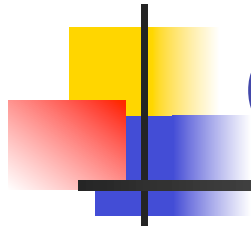
Composition* quick = new Composition(new SimpleCompositor);
Composition* slick = new Composition(new TeXCompositor);
Composition* iconic = new Composition(new ArrayCompositor(100))

```



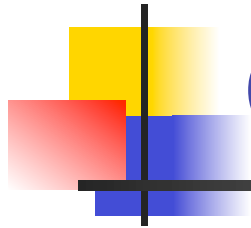
STG Related Patterns

- Flyweight



Command (CMD)

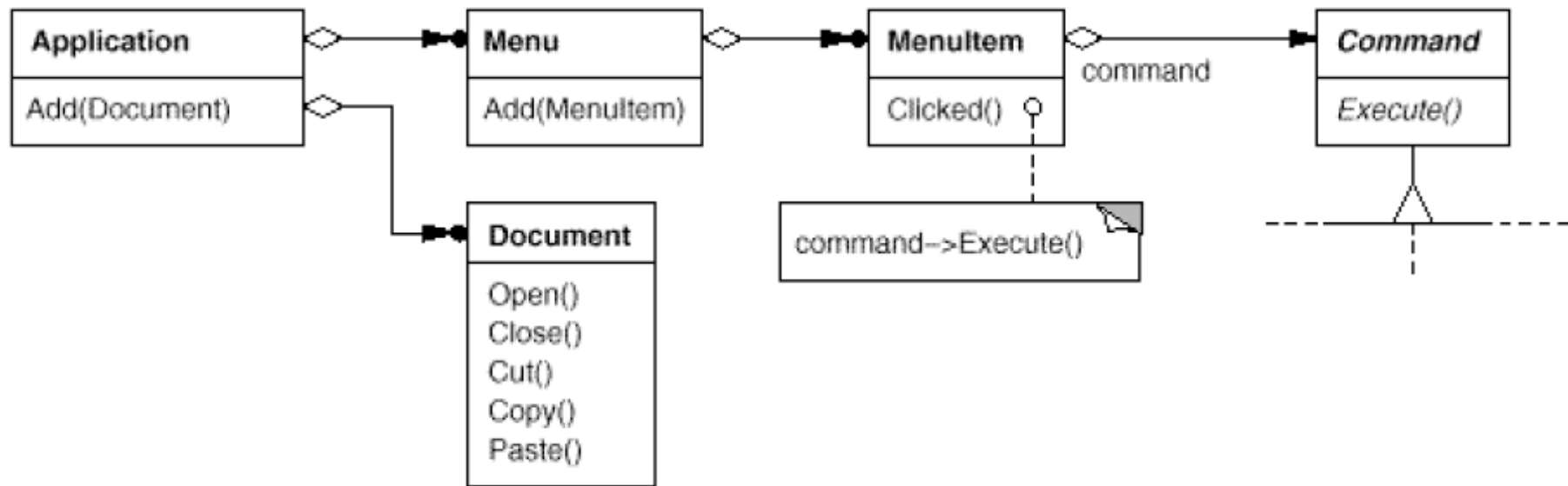
- Intent
 - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Also Known As
 - Action, Transaction



CMD Motivation (1)

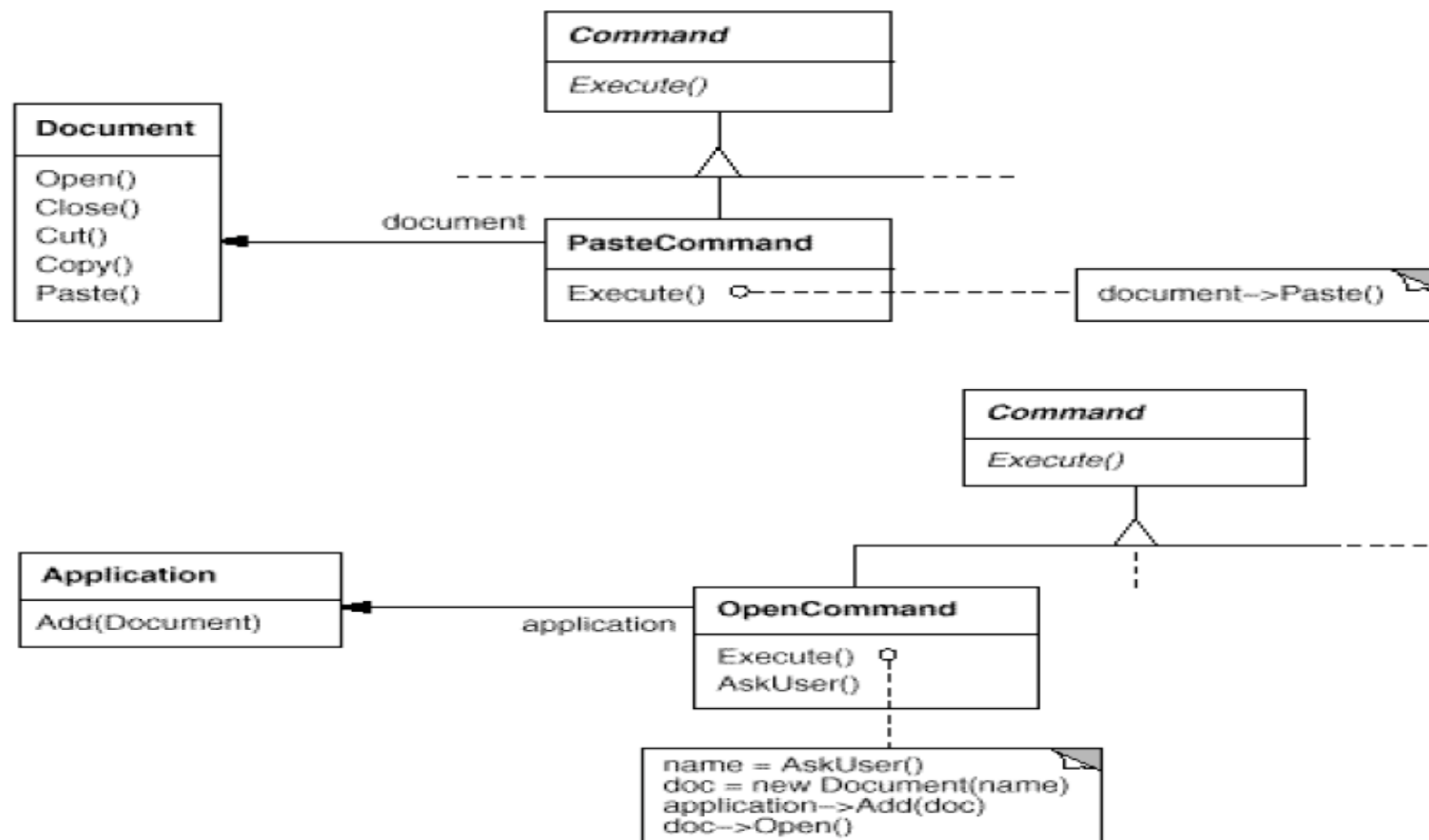
- Consider an user interface toolkits include objects like buttons and menus that carry out a request in response to user input.
 - only applications know what should be done on which object => request can't be implemented in the button or menu
- The Command pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object. This object can be stored and passed around like other objects.

CMD Motivation (2)



- The application configures each MenuItem with an instance of a concrete Command subclass.
- The user selects a MenuItem that calls Execute on its command, and Execute carries out the operation.

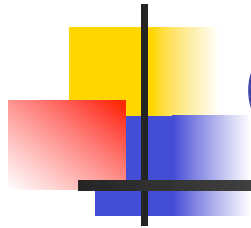
CMD Motivation (3)





CMD Applicability ⁽¹⁾

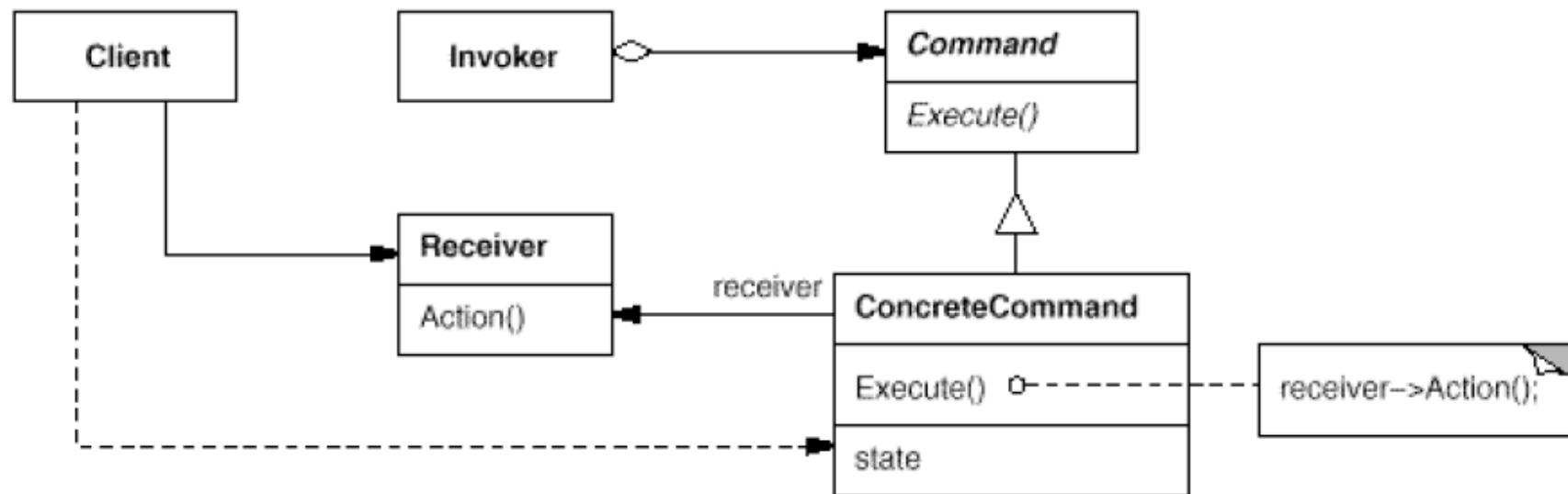
- Use the Command pattern when you want to:
 - Parameterize objects by an action to perform
 - Commands are an object-oriented replacement for callbacks
 - Specify, queue, and execute requests at different times
 - CMD can have a lifetime independent of the original request (can be address space independent)
 - support undo
 - storing state for reversing its effects in the command itself.
 - CMD must have an added Unexecute operation
 - Executed commands are stored in a history list (by MMT?)



CMD Applicability (2)

- support logging changes may be reapplied in case of a system crash.
 - By augmenting CMD with load and store operations
- structure a system around high-level operations built on primitives operations.
 - CMD offers a way to model transactions
 - easy to extend the system with new transactions

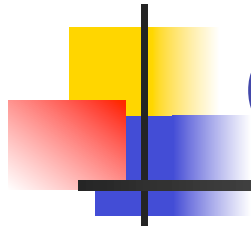
CMD Structure





CMD Participants

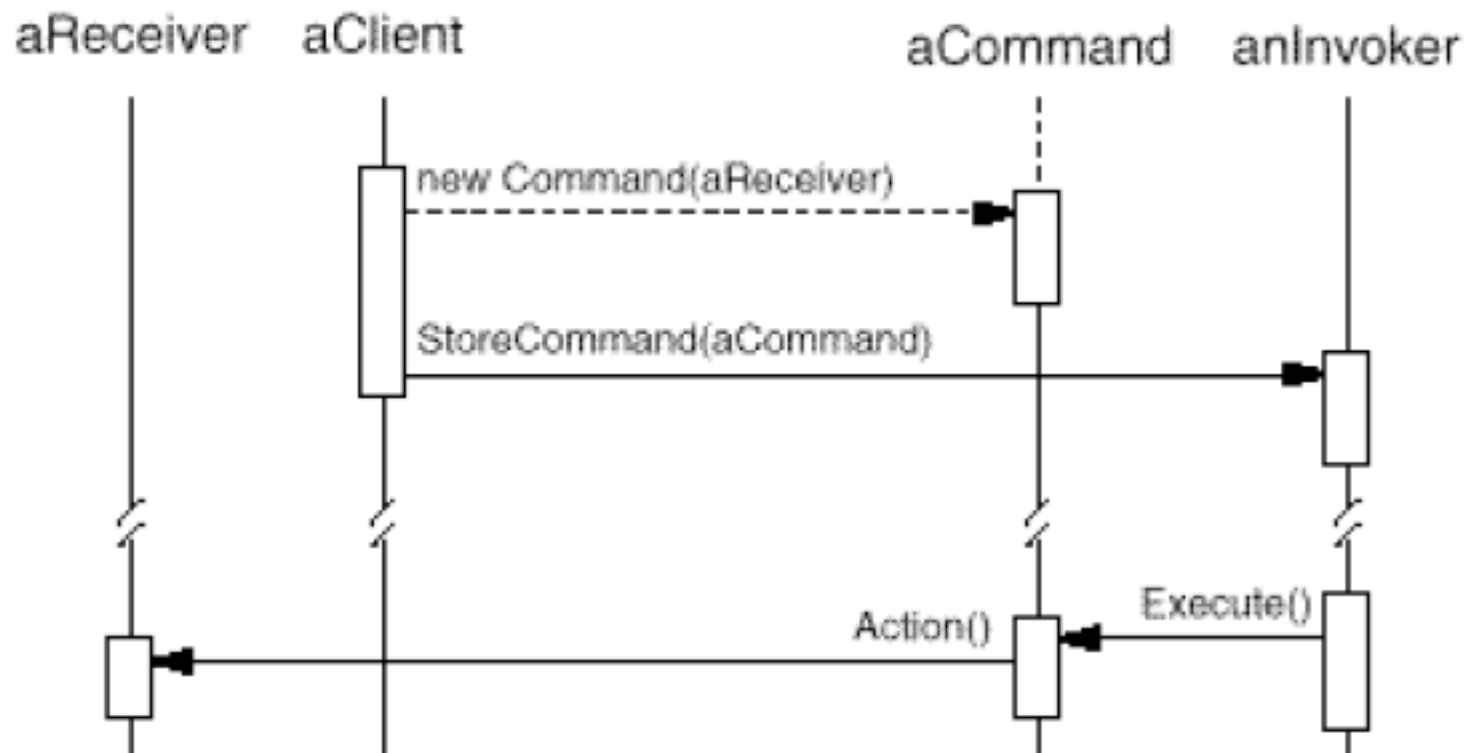
- Command
 - declares an interface for executing an operation.
- ConcreteCommand (PasteCommand, OpenCom...)
 - defines a binding between a Receiver object and an action.
 - implements Execute by invoking the corresponding operation(s) on Receiver.
- Client (Application)
 - creates a ConcreteCommand object and sets its receiver.
- Invoker (MenuItem)
 - asks the command to carry out the request.
- Receiver (Document, Application)
 - knows how to perform the operations associated requests

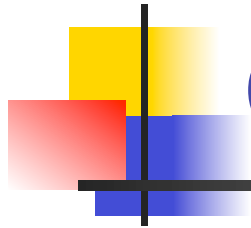


CMD Collaborations (1)

- The client creates a ConcreteCommand object and specifies its receiver
- An Invoker object stores the ConcreteCommand object
- The invoker issues a request by calling Execute on the command
 - When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carry out the request

CMD Collaborations (2)





CMD Consequences

- Decouples the object that invokes the operation from the one that knows how to perform it
- Commands are first-class objects
 - They can be manipulated and extended like any other object
- You can assemble commands into a composite command
 - Composite commands are an instance of Composite
- It's easy to add new Commands
 - you don't have to change existing classes



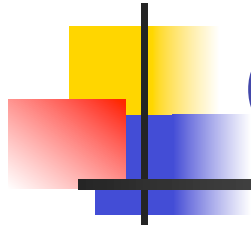
CMD Implementation (1)

- How intelligent should a command be?
 - from merely defining a binding between a receiver and the request's actions,
 - To implementing everything itself
 - commands that are independent of existing classes, suitable receivers might not exist
 - In between CMDs that have enough knowledge to find their receiver dynamically
- Supporting undo and redo
 - By providing a way to reverse execution (e.g., an Unexecute or Undo operation)
 - To support one level of undo, an application needs to store only the command that was executed last.
 - For multiple-level undo and redo, the application needs a history list of commands (CMD may become a Prototype)



CMD Implementation (2)

- How intelligent should a command be?
 - from merely defining a binding between a receiver and the request's actions,
 - To implementing everything itself
 - commands that are independent of existing classes, suitable receivers might not exist
 - In between CMDs that have enough knowledge to find their receiver dynamically
- Supporting undo and redo
 - By providing a way to reverse execution (e.g., an Unexecute or Undo operation)
 - To support one level of undo, an application needs to store only the command that was executed last.
 - For multiple-level undo and redo, the application needs a history list of commands (CMD may become a Prototype)



CMD Implementation (3)

- Avoiding error accumulation in the undo process.
 - Hysteresis can be a problem in ensuring a reliable, semantics-preserving undo/redo mechanism.
 - To avoid error accumulations CMD may be a MMT
 - CMD stores more information to ensure that objects are restored to their original state.
- Using C++ templates to avoid subclassing:
 - If CMDs aren't undoable and don't require arguments



CMD Sample Code

```
class Command {  
public:  
    virtual ~Command();  
  
    virtual void Execute() = 0;  
protected:  
    Command();  
};
```

```

class OpenCommand : public Command {
public:
    OpenCommand(Application*);

    virtual void Execute();
protected:
    virtual const char* AskUser();
private:
    Application* _application;
    char* _response;
};

OpenCommand::OpenCommand (Application* a) {
    _application = a;
}

void OpenCommand::Execute () {
    const char* name = AskUser();

    if (name != 0) {
        Document* document = new Document(name);
        _application->Add(document);
        document->Open();
    }
}

```

```

template <class Receiver>
class SimpleCommand : public Command {
public:
    typedef void (Receiver::* Action)();

    SimpleCommand(Receiver* r, Action a) :
        _receiver(r), _action(a) { }

    virtual void Execute();
private:
    Action _action;
    Receiver* _receiver;
};

```

```

template <class Receiver>
void SimpleCommand<Receiver>::Execute () {
    (_receiver->* _action)();
}

```



```

class MacroCommand : public Command {
public:
    MacroCommand();
    virtual ~MacroCommand();
    virtual void Add(Command*);
    virtual void Remove(Command*);

    virtual void Execute();
private:
    List<Command*>* _cmds;
};

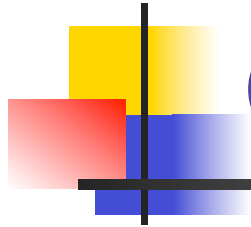
void MacroCommand::Execute () {
    ListIterator<Command*> i(_cmds);

    for (i.First(); !i.IsDone(); i.Next()) {
        Command* c = i.CurrentItem();
        c->Execute();
    }
}

void MacroCommand::Add (Command* c) {
    _cmds->Append(c);
}

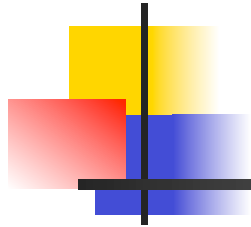
void MacroCommand::Remove (Command* c) {
    _cmds->Remove(c);
}

```



CMD Related Patterns

- A Composite can be used to implement MacroCommands
- A Memento can keep state the command requires to undo its effect.
- A command that must be copied before being placed on the history list acts as a Prototype.

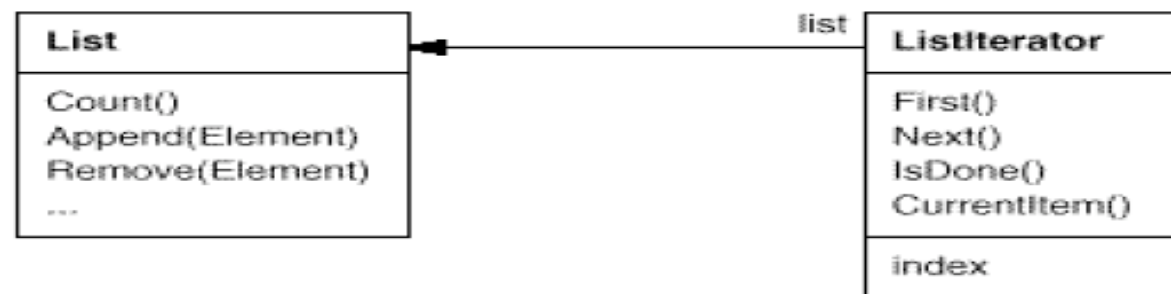


Iterator (ITR)

- Intent
 - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Also Known As
 - Cursor

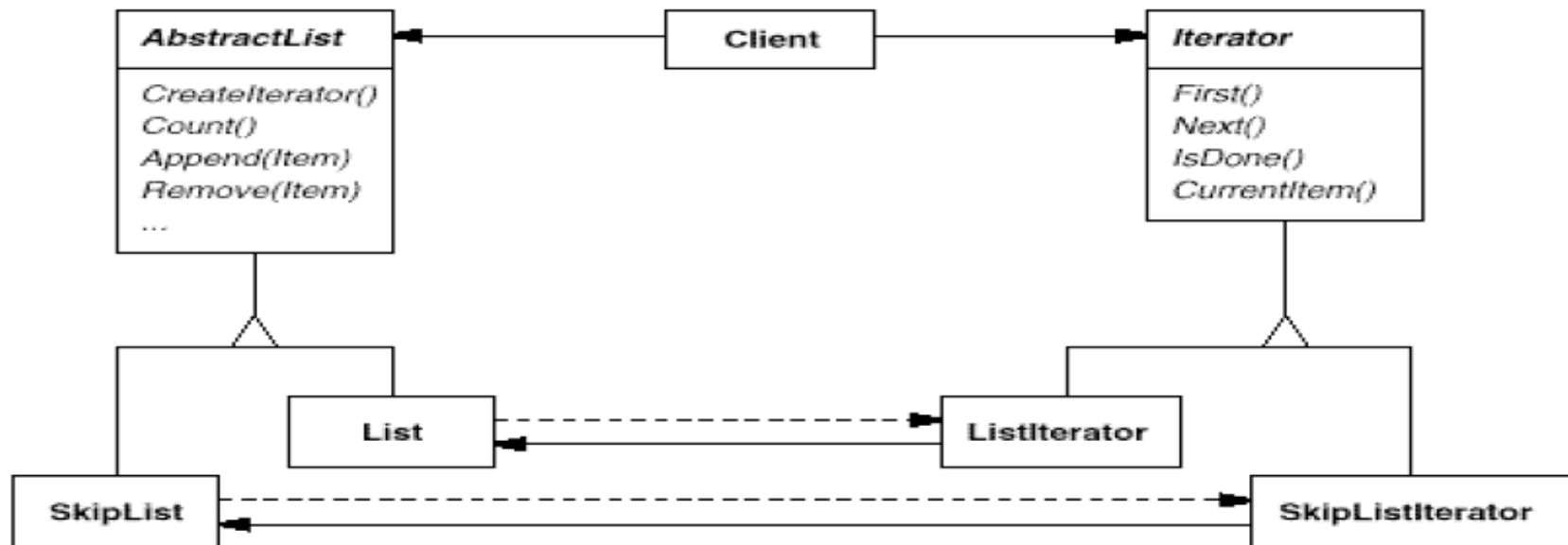
ITR Motivation (1)

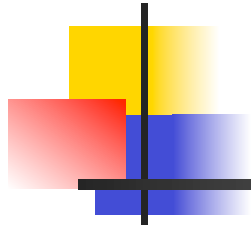
- An aggregate object such as a list
 - should give a way to access elements without exposing its internals
 - might be traversed in different ways, depending on some task
 - might also need to have more than one traversal pending on it.
 - ...but you don't want to bloat the List interface with operations
- Key idea:
 - take the responsibility for access and traversal out of the list object and put it into an iterator object



ITR Motivation (2)

- Separating the traversal from the List:
 - define iterators for different traversal policies without touching the List
- The Iterator and the list are coupled:
 - the client commits to a particular aggregate structure → change the aggregate class without changing client code via **polymorphic iteration**

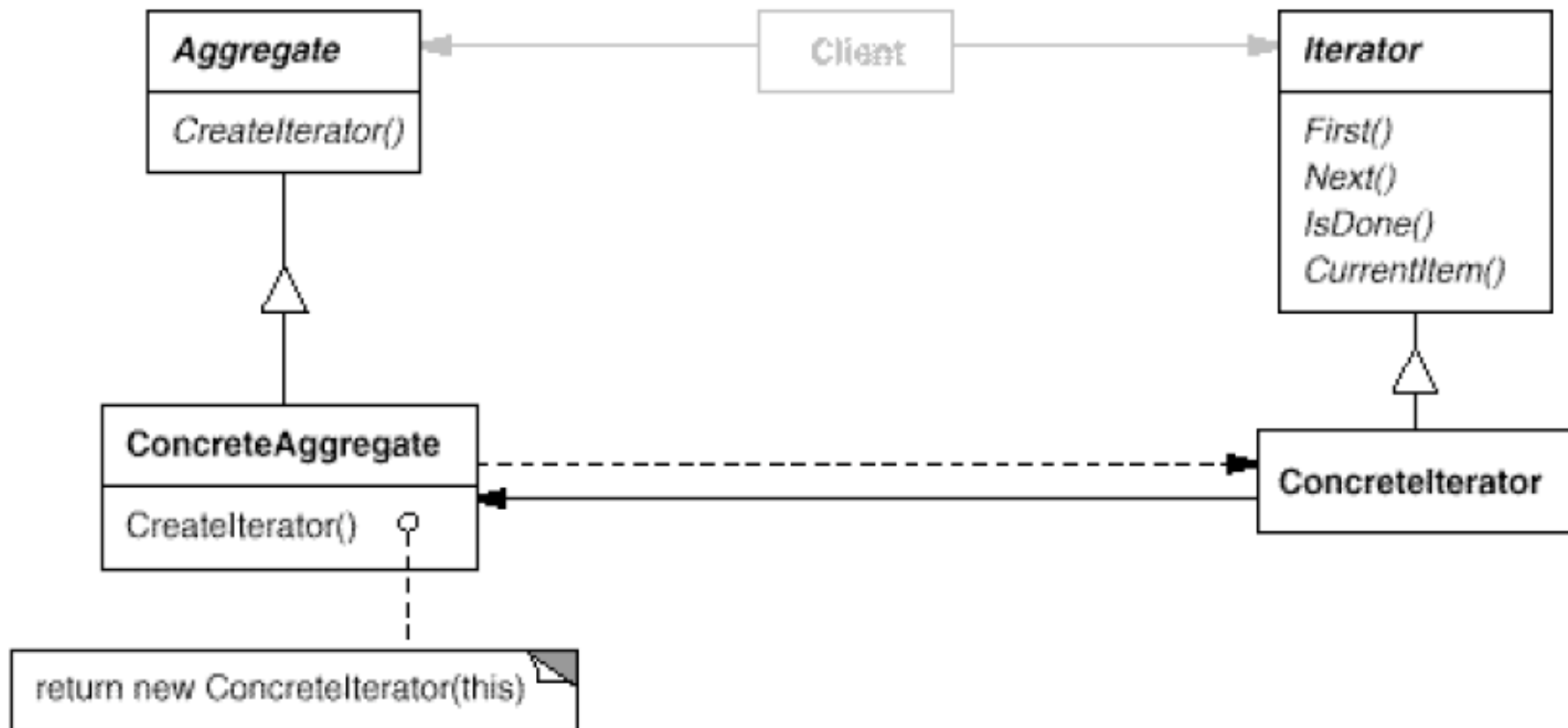


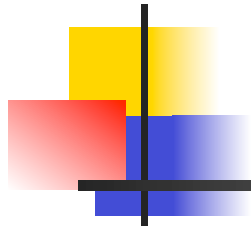


ITR Applicability

- Use the Iterator pattern
 - to access an aggregate object's contents without exposing its internal representation.
 - to support multiple traversals of aggregate objects.
 - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

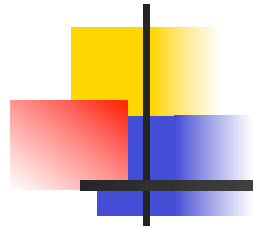
ITR Structure





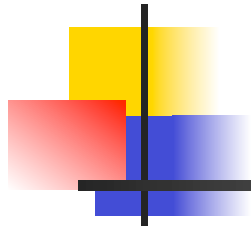
ITR Participants

- Iterator
 - defines an interface for accessing and traversing elements
- ConcreteIterator
 - implements the Iterator interface keeps track of the current position in the traversal of the aggregate
- Aggregate
 - defines an interface for creating an Iterator object
- ConcreteAggregate
 - implements the Iterator creation interface to return an instance of the proper ConcreteIterator



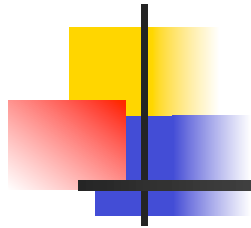
ITR Collaborations

- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal



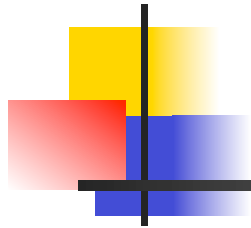
ITR Consequences

- ITR supports variations in the traversal of an aggregate.
 - Complex aggregates may be traversed in many ways.
- ITR make it easy to change the traversal algorithm:
 - Just replace the iterator instance with a different one.
- Iterators simplify the Aggregate interface
 - Iterator's traversal interface obviates the need for a similar interface in Aggregate
- More than one traversal can be pending on an aggregate
 - ITR keeps track of its own traversal state.



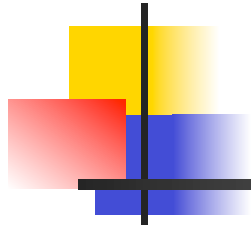
ITR Implementation (1)

- Who controls the iteration? ... IT or the client?
 - external vs internal iterators
 - Internal iterator: has an operation to perform, and the iterator applies that operation to every element in the aggregate
 - External iterator (weak in a language like C++) are more flexible
- Who defines the traversal algorithm? ... IT or the aggregate?
 - The aggregate:
 - use the iterator as a Cursor to store just the state of the iteration
 - A client will invoke the Next operation on the aggregate with the cursor as an argument, and the Next operation will change the state of the cursor
 - The iterator:
 - the traversal algorithm might need to access the private variables of the aggregate: violates the encapsulation of the aggregate
 - it's easy to use different iteration algorithms on the same aggregate



ITR Implementation (2)

- How robust is the iterator?
 - Robust iterator: ensures that insertions and removals won't interfere with traversal
 - Copy the aggregate and traverse the copy, but that's too expensive
 - Rely on registering the iterator with the aggregate: the aggregate fixes registered iterators
- Additional Iterator operations.
 - The minimal interface: First, Next, IsDone, and CurrentItem.
 - Some additional operations might prove useful (e.g., Previous or SkipTo)
- Using polymorphic iterators in C++.
 - Have their cost and should be used only when there's a need
 - Another drawback: the client is responsible for deleting them
 - The Proxy pattern provides a remedy



ITR Implementation (3)

- Iterators may have privileged access.
 - The iterator and the aggregate are tightly coupled:
 - use friend in C++ (add a friend for each iterator)
 - include protected operations for accessing members of the aggregate
- Iterators for composites
 - External iterators difficult to implement over Composite-like
 - Use an internal iterator:
 - It can record the current position simply by calling itself recursively, thereby storing the path in the call stack
- Null iterators
 - A degenerate iterator that's helpful for handling boundary conditions
 - A NullIterator is always done with traversal: IsDone returns true
 - Can make traversing tree-structured aggregates easier and uniform



ITR Sample Code

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};

template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

```

template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;

private:
    const List<Item>* _list;
    long _current;
};

template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {
}

template <class Item>
void ListIterator<Item>::First () {
    _current = 0;
}

template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBounds;
    }
    return _list->Get(_current);
}

```

```
void PrintEmployees (Iterator<Employee*>& i) {  
    for (i.First(); !i.IsDone(); i.Next()) {  
        i.CurrentItem()->Print();  
    }  
}
```

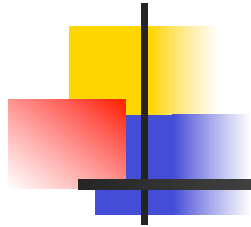
```
List<Employee*>* employees;  
// ...  
ListIterator<Employee*> forward(employees);  
ReverseListIterator<Employee*> backward(employees);  
  
PrintEmployees(forward);  
PrintEmployees(backward);
```



```
// Implementing polymorphic Iterators
```

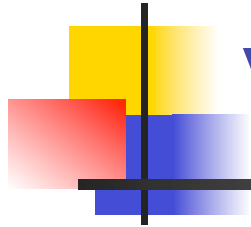
```
template <class Item>
class AbstractList {
public:
    virtual Iterator<Item>* CreateIterator() const = 0;
    // ...
};

template <class Item>
Iterator<Item>* List<Item>::CreateIterator () const {
    return new ListIterator<Item>(this);
}
```



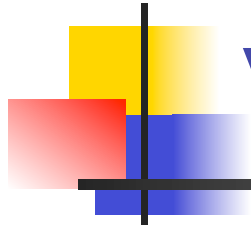
ITR Related Patterns

- Composite
- Factory Method
- Memento
 - ITR can use a memento to capture the state of an iteration



Visitor (VST)

- 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

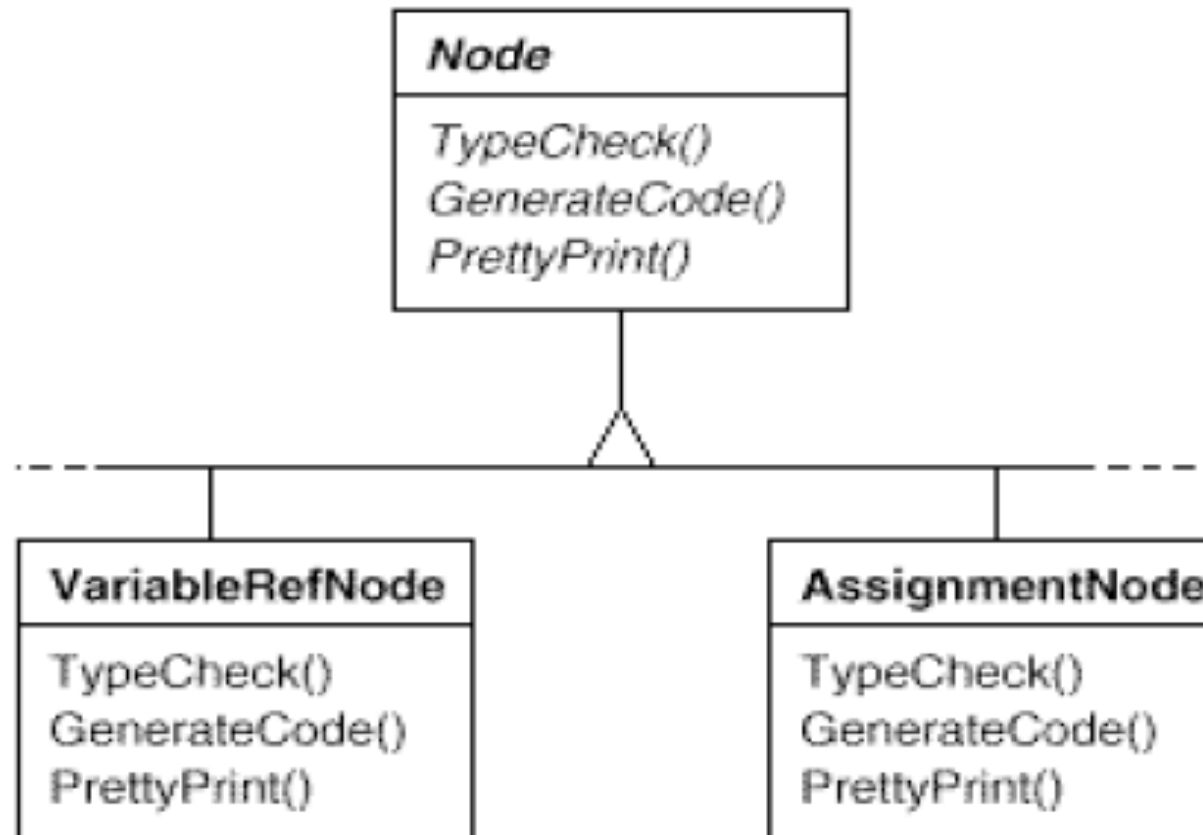


VST Motivation (1)

- Consider a compiler that represents programs as abstract syntax trees
 - It will need to perform operations on abstract syntax trees
 - for "static semantic" analyses like checking that all variables are defined
 - also needs to generate code ... etc.
 - Operations will need to treat nodes that represent assignment statements differently from nodes that represent variables or arithmetic expressions...



VST Motivation (2)

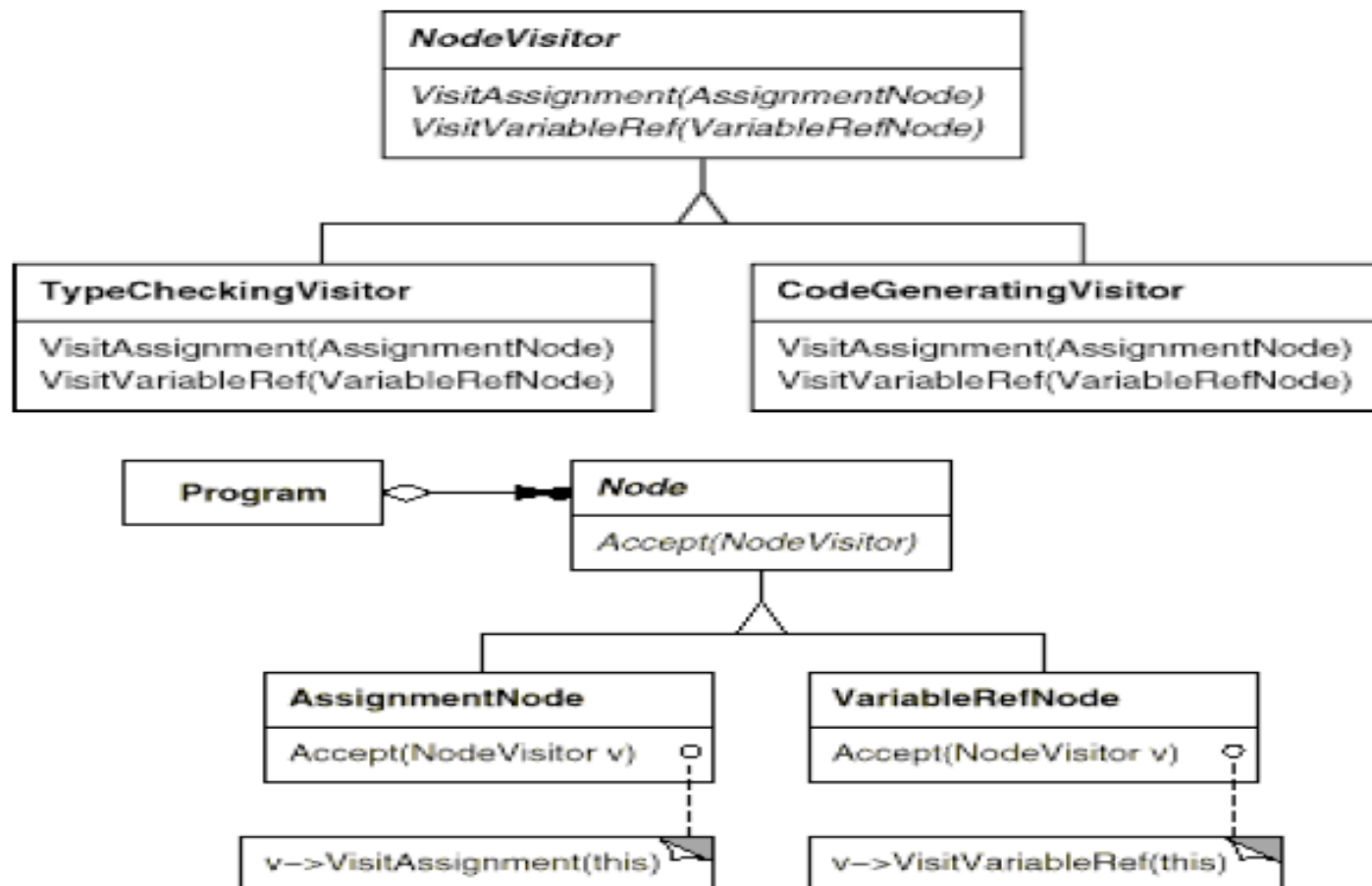




VST Motivation (3)

- The problem:
 - all these operations on the abstract syntax tree are distributed
=> a system that's hard to understand, maintain, and change
 - type-checking code mixed with pretty-printing code or flow analysis code thus adding a new operation requires recompiling
- The Visitor:
 - Packages related operations from each class in a separate object
 - each new operation could be added separately
 - node classes are independent of the operations that apply to them.
 - is passed it to elements of the abstract syntax; an element "accepts" the visitor, the VST will then execute the operation (used to be in the class of the element) for that element

VST Motivation (4)

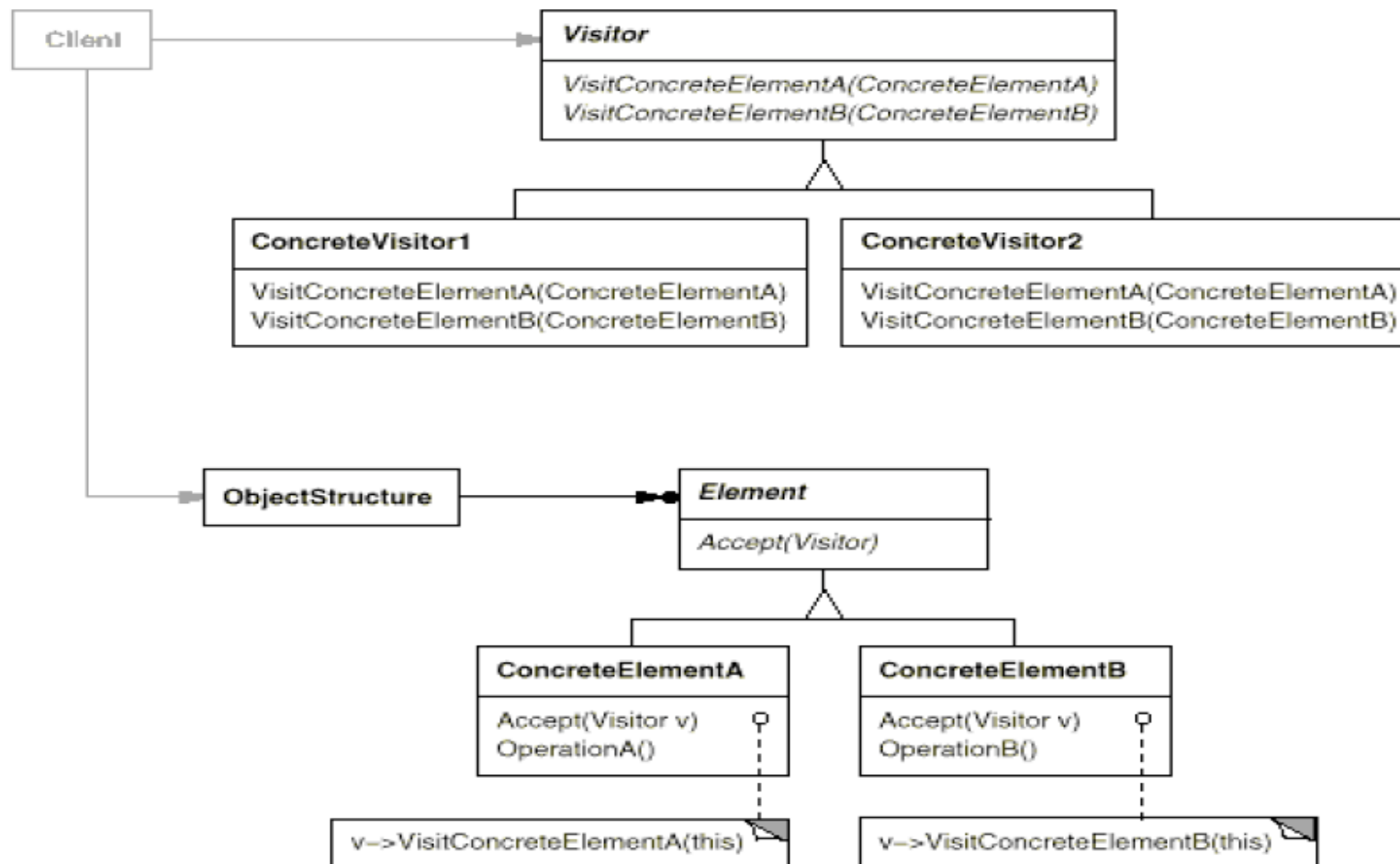


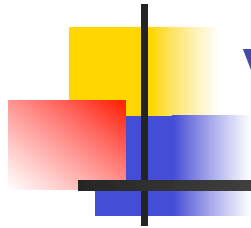


VST Applicability

- Use VST 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
 - avoid "polluting" their classes with these operations
 - keep related operations together by defining them in one class
 - If classes defining the object structure rarely change, and you often want to define new operations over the structure
 - Changing structure classes => redefining all visitors
 - If the structure of classes changes often => define the operations in those classes

VST Structure





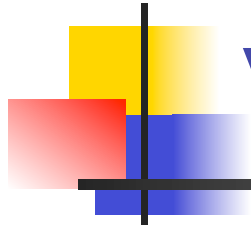
VST Participants (1)

- Visitor (NodeVisitor)
 - declares a Visit operation for each class of ConcreteElement
 - The operation's signature identifies the class that sends the Visit request
 - VST can access the element directly through its particular interface
- ConcreteVisitor (TypeCheckingVisitor)
 - implements each operation declared by Visitor (a fragment of the algorithm)
 - provides the context for the algorithm and stores its local state
- Element (Node)
 - defines an Accept operation taking a visitor



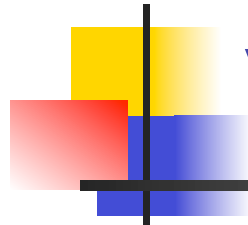
VST Participants (2)

- ConcreteElement (AssignmentNode, VariableRefNode)
 - implements an Accept operation that takes a visitor as an argument
- ObjectStructure (Program)
 - can enumerate its elements
 - may provide a high-level interface to allow the visitor to visit its elements
 - may either be a Composite or a collection such as a list or a set

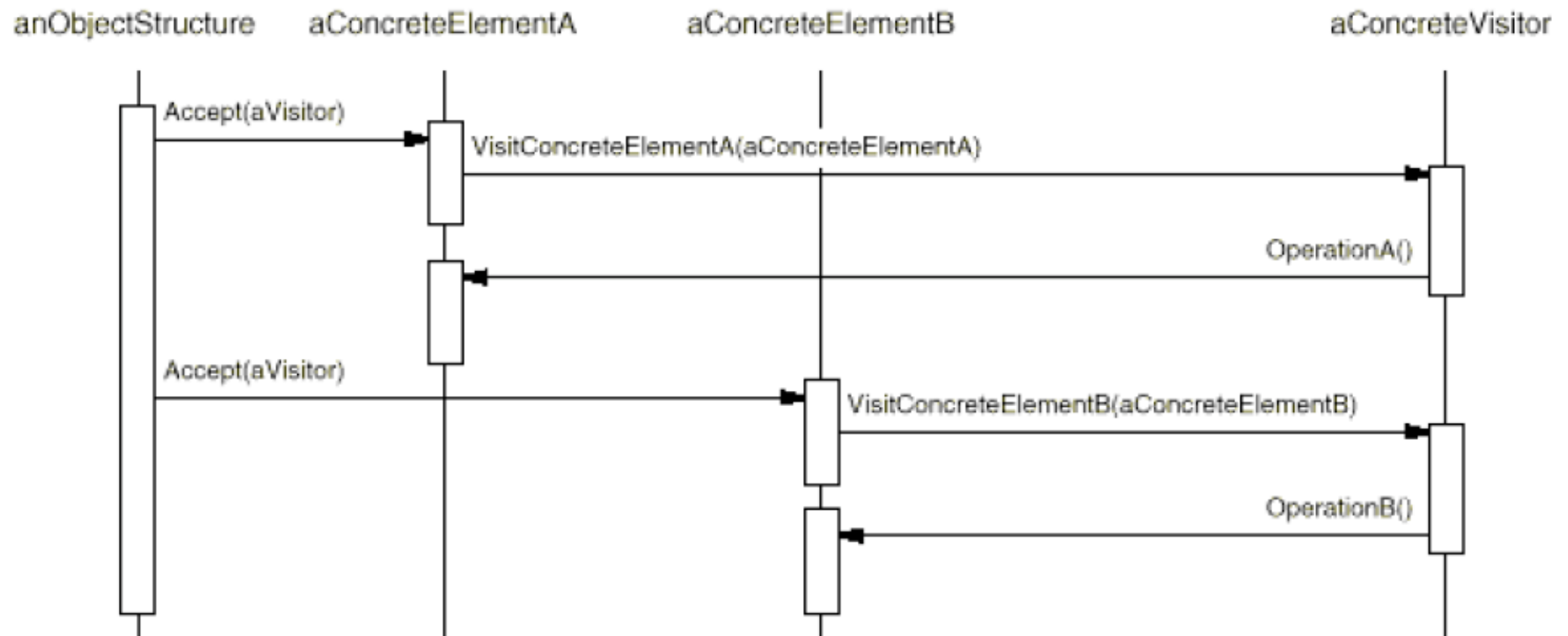


VST Collaborations (1)

- A client
 - creates a ConcreteVisitor object and
 - traverse the object structure, visiting each element
- When an element is visited
 - it calls the Visitor operation that corresponds to its class.
 - The element supplies itself to let the visitor access its state, if necessary.

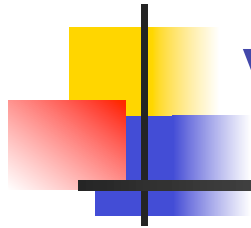


VST Collaborations (2)





VST Consequences



VST Implementation (1)

- Each object structure will have an associated Visitor class
 - This abstract visitor class declares a VisitConcreteElement operation for each class of ConcreteElement defining the object structure.
- Each Visit operation on the Visitor declares its argument to be a particular ConcreteElement
 - allowing the Visitor to access the interface of the ConcreteElement directly
- ConcreteVisitor classes override each Visit operation
 - to implement visitor-specific behavior for the corresponding ConcreteElement class

```
class Visitor {
public:
    virtual void VisitElementA(ElementA*);
    virtual void VisitElementB(ElementB*);

    // and so on for other concrete elements
protected:
    Visitor();
};

class Element {
public:
    virtual ~Element();
    virtual void Accept(Visitor&) = 0;
protected:
    Element();
};

class ElementA : public Element {
public:
    ElementA();
    virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};

class ElementB : public Element {
public:
    ElementB();
    virtual void Accept(Visitor& v) { v.VisitElementB(this); }
};
```



```
class CompositeElement : public Element {
public:
    virtual void Accept(Visitor&);
private:
    List<Element*>* _children;
};

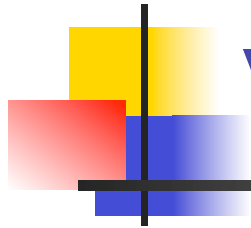
void CompositeElement::Accept (Visitor& v) {
    ListIterator<Element*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Accept(v);
    }
    v.VisitCompositeElement(this);
}
```



VST Implementation (3)

- VST lets you add operations to classes without changing them
- The key to the Visitor pattern:
 - The operation executed depends on the type both Visitor and Element visited
 - Double dispatch
 - Execution depends on the kind of request and the types of two receivers
 - Accept is a double-dispatch operation.
 - Some programming languages support it directly



VST Implementation (2)

- Who is responsible for traversing the object structure?
 - The object structure, the visitor, or a separate iterator
 - Often the object structure is responsible for iteration
 - A collection will simply iterate over its elements, calling the Accept operation on each
 - A composite will commonly traverse itself by calling Accept recursively
 - An iterator to visit the elements
 - In C++, you could use either an internal or external iterator
 - An internal iterator will not cause double-dispatching
 - The traversal algorithm in the visitor
 - Duplicating the traversal code in each ConcreteVisitor for each aggregate ConcreteElement.
 - Good for particularly complex traversals

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Accept(EquipmentVisitor&);
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```



VST Sample Code

```
class EquipmentVisitor {
public:
    virtual ~EquipmentVisitor();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);

    // and so on for other concrete subclasses of Equipment
protected:
    EquipmentVisitor();
};
```

```
void FloppyDisk::Accept (EquipmentVisitor& visitor) {  
    visitor.VisitFloppyDisk(this);  
}  
  
void Chassis::Accept (EquipmentVisitor& visitor) {  
    for (  
        ListIterator i(_parts);  
        !i.IsDone();  
        i.Next()  
    ) {  
        i.CurrentItem()->Accept(visitor);  
    }  
    visitor.VisitChassis(this);  
}
```

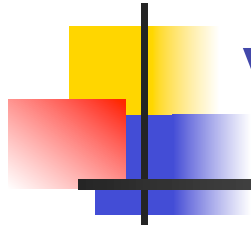
```
class PricingVisitor : public EquipmentVisitor {
public:
    PricingVisitor();

    Currency& GetTotalPrice();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
    // ...
private:
    Currency _total;
};

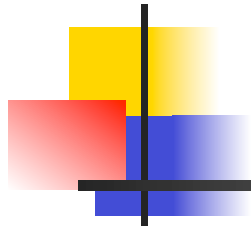
void PricingVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _total += e->NetPrice();

    void PricingVisitor::VisitChassis (Chassis* e) {
        _total += e->DiscountPrice();
    }
}
```



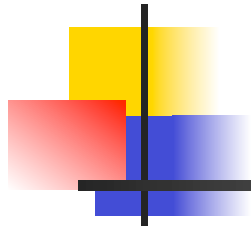
VST Related Patterns

- Composite
 - Visitors can be used to apply an operation over an object structure defined by Composite
- Interpreter
 - Visitor may be applied to do the interpretation



Discussion (1)

- Encapsulating variation
 - an object encapsulates an changing aspect
 - STG encapsulates an algorithm
 - ST encapsulates a state-dependent behavior
 - ME encapsulates the protocol between objects
 - ITR encapsulates the way you access and traverse the components of an aggregate
- Communication between an open-ended number of objects (e.g., CoR)



Discussion (2)

- Objects as Arguments
 - VST is the argument to a polymorphic Accept operation on the objects it visits
 - CMD and MMT act as magic tokens (request, state) to be passed around and invoked later
 - CMD is polymorphic, MMT is “opaque”: narrow interface
- Encapsulated or Distributed Communication?
 - Mediator vs Observer
 - OB distributes, ME centralizes
 - ME easier to follow, OB easy to extend



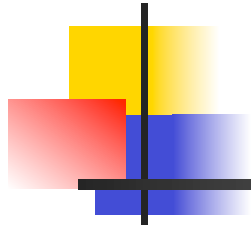
Discussion (3)

- Decoupling Senders and Receivers
 - CMD, OB, ME, CoR
 - CMD: a separate object lets the sender work with different receivers
 - OB:
 - Subjects decoupled from observers by an interface for signaling changes in subjects
 - looser sender-receiver binding than CMD, with multiple receivers varying at running time



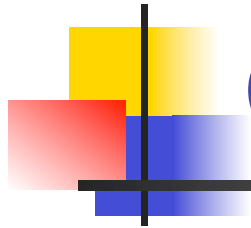
Discussion (3)

- ME:
 - Objects refer to each other indirectly through a ME
 - less subclassing by centralized communication, weaker type safety
- CoR:
 - Decoupling request by passing the it along a chain of potential receivers
 - good if sender and the receiver are already part of the system's structure



Discussion (4)

- Common pattern combinations:
 - CoR uses TM
 - CoR uses CMD to represent requests
 - IN uses ST for parsing contexts
 - VST to implement IN evaluations
 - VST works well with Composite
 - CoR integrates with Composite



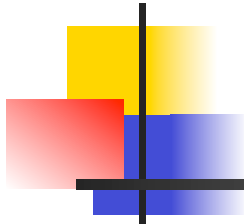
Conclusion (1)

*“Well-designed object-oriented systems are just like this
—they have multiple patterns embedded in them—
but not because their designers necessarily thought in these
terms. Composition at the pattern level rather than
the class or object levels lets us achieve the same synergy
with greater ease.” [GoF]*



Conclusion (2)

- What to Expect from Design Patterns?
 - A Common Design Vocabulary
 - A Documentation and Learning Aid
 - An Adjunct to Existing Methods
 - A Target for Refactoring



Conclusion (3)

“It is possible to make buildings by stringing together patterns, in a rather loose way. A building made like this, is an assembly of patterns. It is not dense. It is not profound.

But it is also possible to put patterns together in such a way that many patterns overlap in the same physical space: the building is very dense; it has many meanings captured in a small space; and through this density, it becomes profound.” [Alexander]