Advanced Programming Language Features and Software Engineering: Friend or Foe?

Greg Sullivan, AI Lab

gregs@ai.mit.edu

To Learn

- GOF Design Patterns. Overview, some patterns in detail.
- Unusual Language Features, and their use in Design Patterns
  - Multiple dispatch
  - Metaobject protocols,
    - especially instantiation and dispatch
  - Generalized Dynamic Types

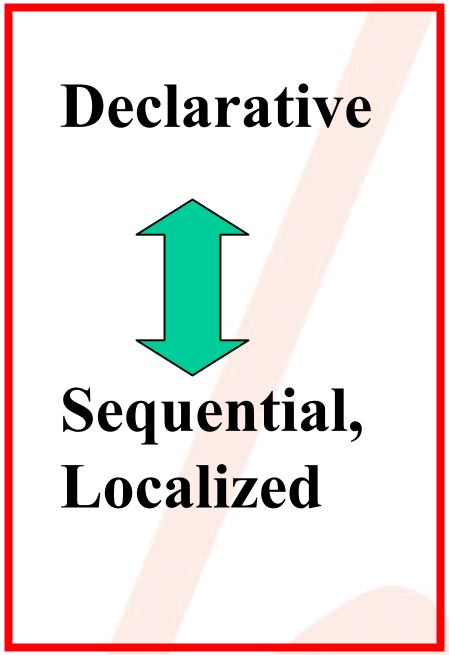
To Ponder, Discuss

- Does programming language design have something to do with software engineering?
- Relation to Programming & Modeling.

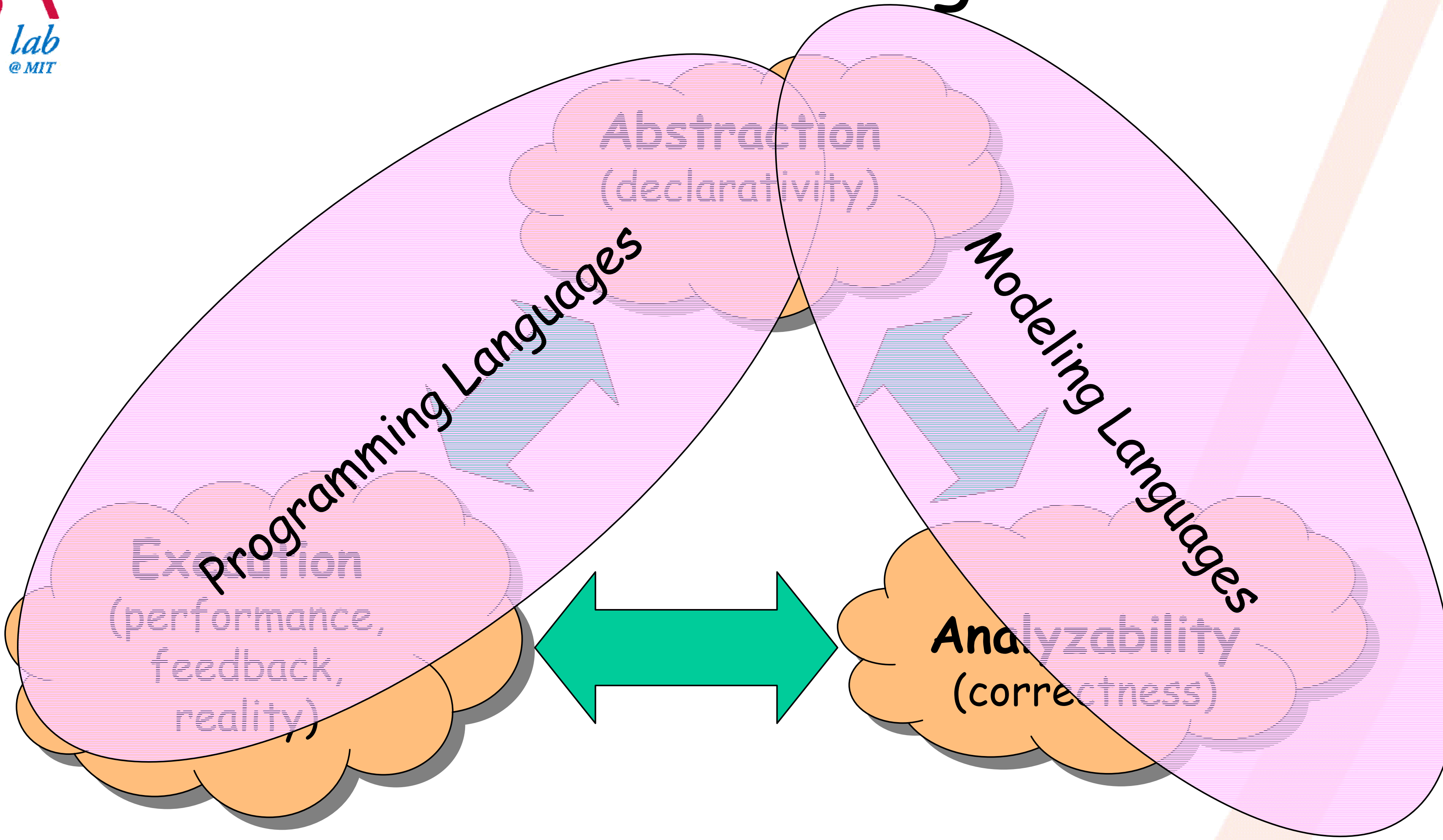
Process of, Languages for

Tinkers and Thinkers

- Tinkers like programming.
- Alloy makes modeling more like programming. More "executable".
- What's important about a **model**?
  - Abstraction
  - Declarative style
- How can programming languages enable more abstraction, declarative style?



The Tool Triangle



Programming Language Features for Abstraction, Declarative style

- Abstraction:
  - Procedural (called abstractions in Λ calculus)
  - Data abstraction (OO, modules/namespaces, ...)
- Declarative:
  - Dynamic dispatch. Adds a level of indirection.
  - Method combination.
  - Exception handling.
  - Aspect-Oriented Programming
    - "crosscutting concerns"
  - Constraint languages
  - Reflection:
    - Traditional: hacking the interpreter. Non-locality. Java reflection – read-only.

GOF Design Patterns

- Most are about adding indirection, abstraction.
  - "Joints"
- An important part of current software engineering dogma.
- Related, as is Alloy, to focus on lightweight methods, XP, etc. Tools for the programmer more than the designer.
- This talk doesn't do the book justice. Much more in the book than code.
- Each of 23 patterns has Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, Related Patterns.

GLOS
Greg's Little Object System

- Not interesting – representative.
- Added to Scheme: multiple dispatch, method combination, multiple inheritance, record subtyping with instantiation protocol.
- First class: functions, methods (functions with argument specializers), generic functions (collections of methods, with combiner function), types.
- Types / Specializers: primitive, and, or, equal (singleton), predicate.

Setup - Mazes

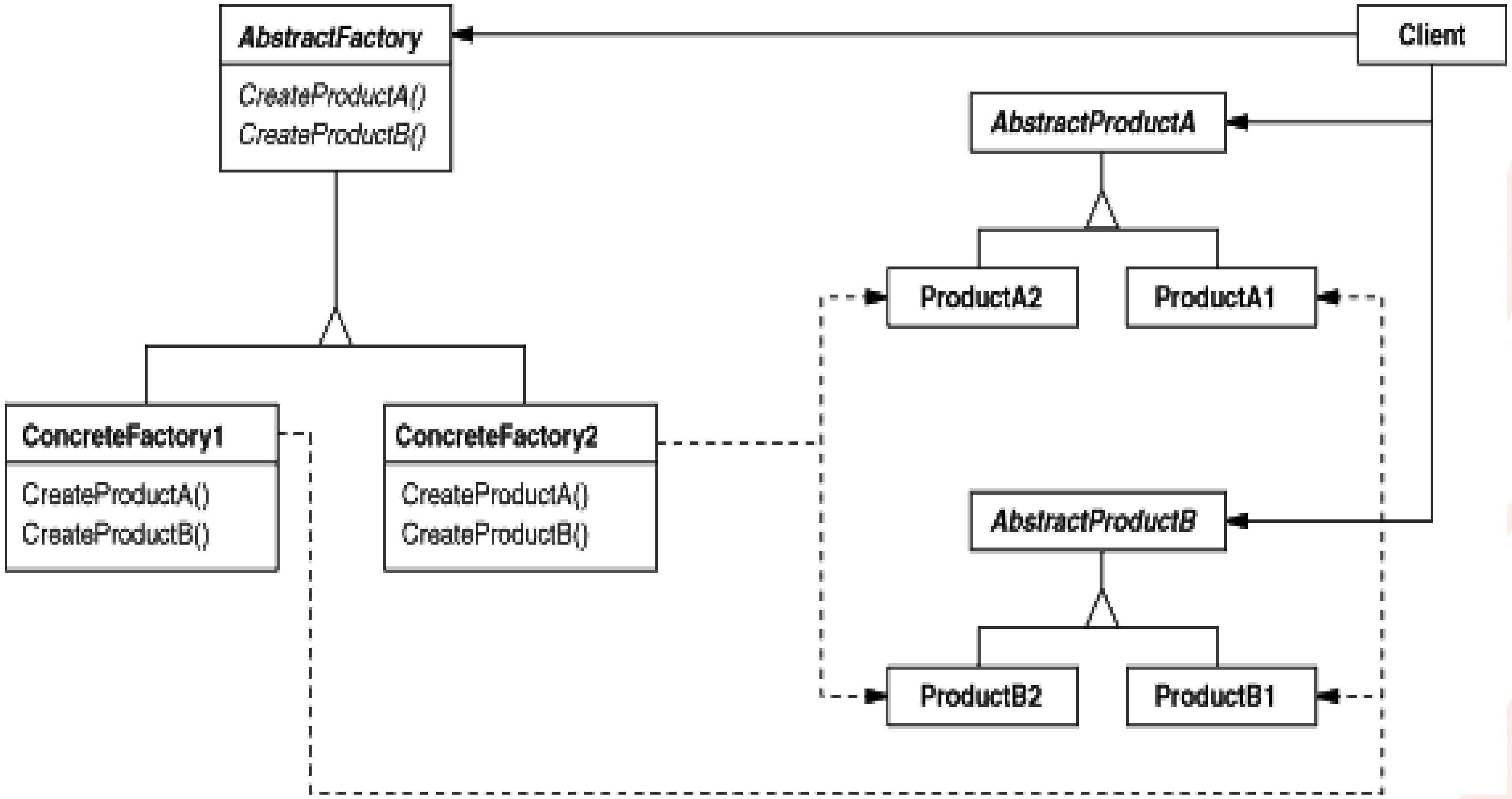
```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = new Maze;
        Room* r1 = new Room(1);
        Room* r2 = new Room(2);
        Door* theDoor = new Door(r1,
r2);

        aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);

        r1->SetSide(North, new Wall);
        r1->SetSide(East, theDoor);
        r1->SetSide(South, new Wall);
        r1->SetSide(West, new Wall);

        r2->SetSide(North, new Wall);
        r2->SetSide(East, new Wall);
        r2->SetSide(South, new Wall);
        r2->SetSide(West, theDoor);

        return aMaze;
    }
```

Tedious,

Inflexible

Abstract Factory

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



Abstract Factory in C++

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
       Maze* aMaze = factory.MakeMaze();
       Room* r1 = factory.MakeRoom(1);
       Room* r2 = factory.MakeRoom(2);
       Door* aDoor = factory.MakeDoor(r1, r2);

       aMaze->AddRoom(r1);
       aMaze->AddRoom(r2);

       r1->SetSide(North, factory.MakeWall());
       r1->SetSide(East, aDoor);
       r1->SetSide(South, factory.MakeWall());
       r1->SetSide(West, factory.MakeWall());

       r2->SetSide(North, factory.MakeWall());
       r2->SetSide(East, factory.MakeWall());
       r2->SetSide(South, factory.MakeWall());
       r2->SetSide(West, aDoor);

       return aMaze;
   }
```

```
class MazeFactory {
    public:
        MazeFactory();

        virtual Maze* MakeMaze() const
            { return new Maze; }
        virtual Wall* MakeWall() const
            { return new Wall; }
        virtual Room* MakeRoom(int n) const
            { return new Room(n); }
        virtual Door* MakeDoor(Room* r1, Room* r2)
                const
            { return new Door(r1, r2); }
    };
```

```
class EnchantedMazeFactory : public MazeFactory {
   public:
       EnchantedMazeFactory();

       virtual Room* MakeRoom(int n)  const
           { return new EnchantedRoom(n, CastSpell()); }

       virtual Door* MakeDoor(Room* r1, Room* r2)  const
           { return new DoorNeedingSpell(r1, r2); }

   protected:
       Spell* CastSpell() const;
   };
```

Abstract Factory in Smalltalk

```
createMaze: aFactory
      | room1 room2 aDoor |
      room1 := (aFactory make: #room) number: 1.
      room2 := (aFactory make: #room) number: 2.
      aDoor := (aFactory make: #door) from: room1 to: room2.
      room1 atSide: #north put: (aFactory make: #wall).
      room1 atSide: #east put: aDoor.
      room1 atSide: #south put: (aFactory make: #wall).
      room1 atSide: #west put: (aFactory make: #wall).
      room2 atSide: #north put: (aFactory make: #wall).
      room2 atSide: #east put: (aFactory make: #wall).
      room2 atSide: #south put: (aFactory make: #wall).
      room2 atSide: #west put: aDoor.
      ^ Maze new addRoom: room1; addRoom: room2; yourself
```

Uses 1st class classes

```
make: partName
      ^ (partCatalog at: partName) new
```

```
createMazeFactory
      ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: Room named: #room;
        addPart: Door named: #door;
        yourself)
```

```
createMazeFactory
      ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: EnchantedRoom named: #room;
        addPart: DoorNeedingSpell named: #door;
        yourself)
```

Abstract Factory in GLOS

```
(defgeneric make-maze-element
  (method ((f <maze-factory>) (eltType (== <wall>)))   => <wall>
     (new <wall>))
  (method ((f <maze-factory>) (eltType (== <room>)) :rest args) => <room>
     (apply new <room> args))
  (method ((f <maze-factory>) (eltType (== <door>)) :rest args) => <door>
     (apply new <door> args)))
...
```

Feature: ***multiple dispatch***

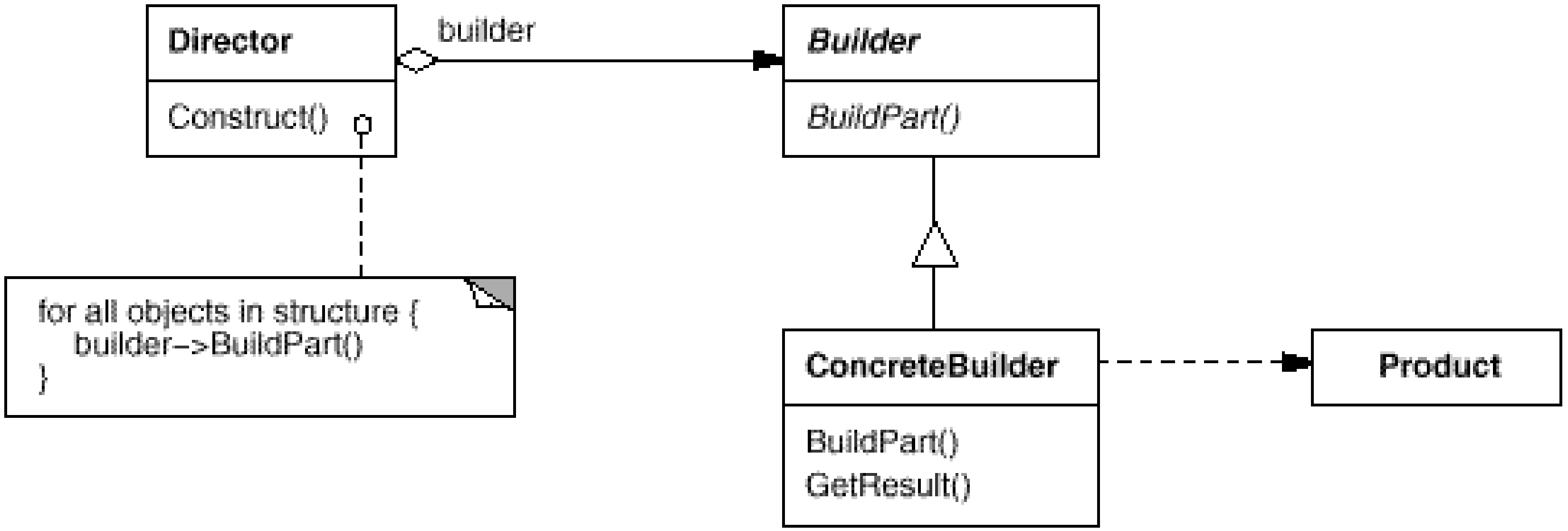
```
(defrectype <enchanted-maze-factory> (<maze-factory>) ())
(defrectype <enchanted-room> (<room>) ())
(gfmethod (make-maze-element (f <enchanted-maze-factory>)
                             (elt-type (== <room>)) :rest args)
   (apply new <enchanted-room> args))
```

Feature: ***singleton types***

```
(define room1 (make-maze-element the-factory <room> 'number 1))
(define room2 (make-maze-element the-factory <room> 'number 2))
(define door1 (make-maze-element the-factory <door> 'from room1 'to room2))
```

Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.



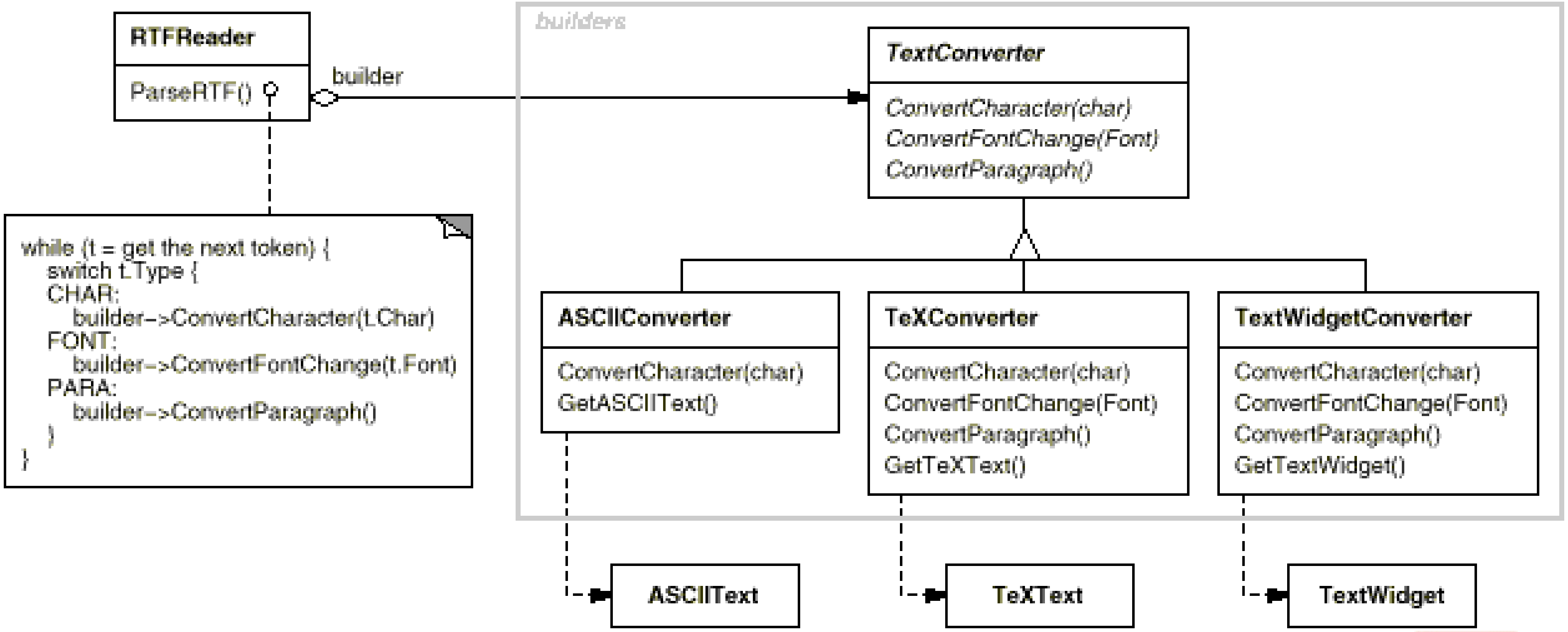
Builder in C++

```
class MazeBuilder {
   public:
       virtual void BuildMaze() { }
       virtual void BuildRoom(int room) { }
       virtual void BuildDoor(int roomFrom, int roomTo)
           { }
       virtual Maze* GetMaze() { return 0; }
   protected:
       MazeBuilder();
   };



Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
       builder.BuildMaze();
       builder.BuildRoom(1);
       builder.BuildRoom(2);
       builder.BuildDoor(1, 2);
       return builder.GetMaze();
   }
```

and GLOS

```
(defrectype <maze-builder> () ())
(defgeneric create-maze
  (method ((game <maze-game>)
           (builder <maze-builder>))
    (build-maze builder game)
    (build-room builder game 1)
    (build-room builder game 2)
    (build-door builder game 1 2)
    (get-maze builder)))
```

Builder, continued



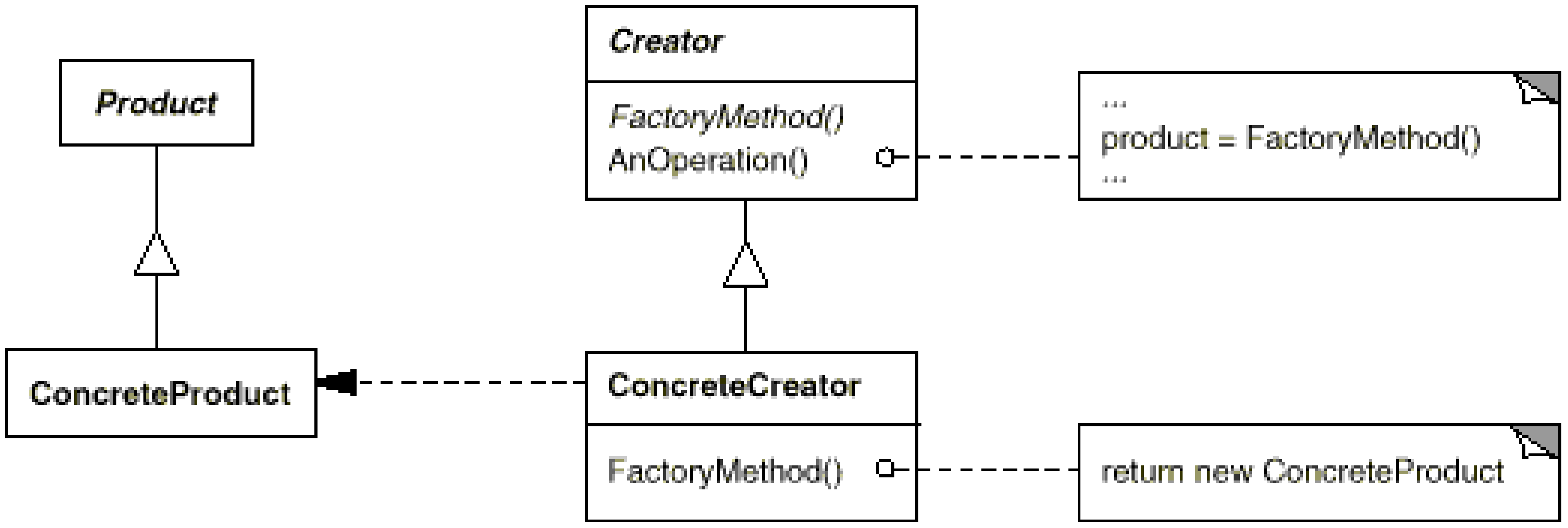
```
while (t = get the next token) {
    switch t.Type {
    CHAR:
        builder->ConvertCharacter(t.Char)
    FONT:
        builder->ConvertFontChange(t.Font)
    PARA:
        builder->ConvertParagraph()
    }
}
```

Builder with Multiple Dispatch

```
(add-method* convert
   (method ((builder <tex-converter>) (token <char>))
      ... convert TeX character ...)
   (method ((builder <tex-converter>) (token <font>))
      ... convert TeX font change ...)
   (method ((builder <text-widget-converter>) (token <char>))
      ... convert text widget character ...)
   (method ((builder <text-widget-converter>) (token <font>))
      ... convert text widget font change ...))
```

Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



```
...
product = FactoryMethod()
...
```

```
return new ConcreteProduct
```

Factory Method - C++

```
class MazeGame {
   public:
       Maze* CreateMaze();

   // factory methods:

       virtual Maze* MakeMaze() const
           { return new Maze; }
       virtual Room* MakeRoom(int n) const
           { return new Room(n); }
       virtual Wall* MakeWall() const
           { return new Wall; }
       virtual Door* MakeDoor(Room* r1, Room* r2) const
           { return new Door(r1, r2); }
   };
```

```
Maze* MazeGame::CreateMaze () {
     Maze* aMaze = MakeMaze();
     Room* r1 = MakeRoom(1);
     Room* r2 = MakeRoom(2);
     Door* theDoor = MakeDoor(r1, r2);

     aMaze->AddRoom(r1);
     aMaze->AddRoom(r2);

     r1->SetSide(North, MakeWall());
     r1->SetSide(East, theDoor);
  ...
       return aMaze;
   }
```

```
class EnchantedMazeGame : public MazeGame {
   public:
       EnchantedMazeGame();

       virtual Room* MakeRoom(int n) const
           { return new EnchantedRoom(n, CastSpell()); }

       virtual Door* MakeDoor(Room* r1, Room* r2) const
           { return new DoorNeedingSpell(r1, r2); }
  ...
```

Factory Method - GLOS

```
(gfmethod (make (c (== <wall>)))
   (make <bombed-wall>))
(gfmethod (make (c (== <room>)) (n <int>))
   (make <bombed-room>))
```

Feature:

Instantiation Protocol

- in CLOS, Dylan. Any others?
- in GLOS, **`new`** first calls **`make`**, then **`initialize`**

Parameterized Factory Method

in C++

```
Product* MyCreator::Create (ProductId id) {
       if (id == YOURS)   return new MyProduct;
       if (id == MINE)    return new YourProduct;
           // N.B.: switched YOURS and MINE

       if (id == THEIRS) return new TheirProduct;

       return Creator::Create(id); // called if all others fail
   }
```

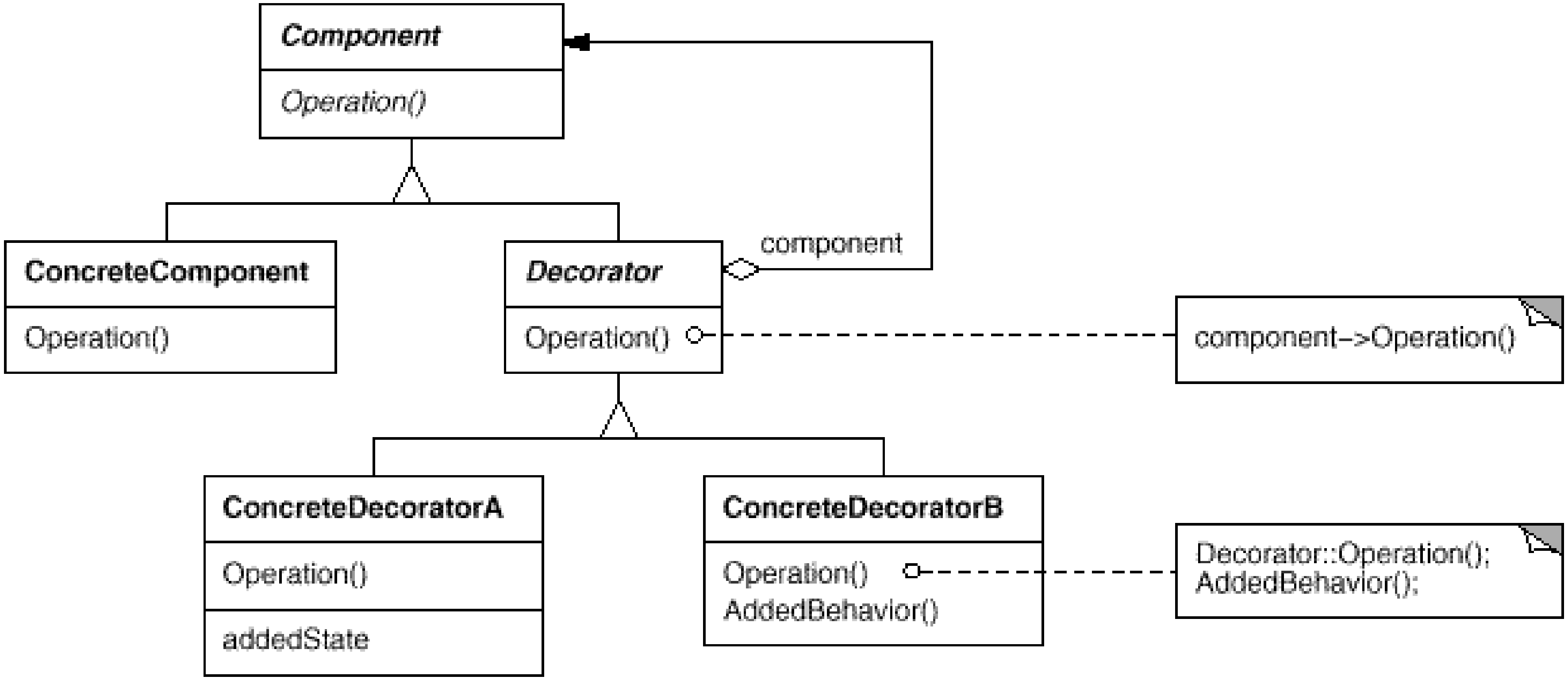
in GLOS

multiple dispatch, singleton types, again

```
(add-method* make
  (method ((c (== <product>)) (id (== 'mine)))
         (make <my-product>))
  (method ((c (== <product>)) (id (== 'yours)))
         (make <your-product>)))
```

Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Decorator - C++

```
class BorderDecorator : public Decorator {
   public:
       BorderDecorator(VisualComponent*, int borderWidth);

       virtual void Draw();
   private:
       void DrawBorder(int);
   private:
       int _width;
   };

   void BorderDecorator::Draw () {
       Decorator::Draw();
       DrawBorder(_width);
   }
```

```
Window* window = new Window;
TextView* textView = new TextView;

window->SetContents(
       new BorderDecorator(
           new ScrollDecorator(textView), 1
       )
   );
```

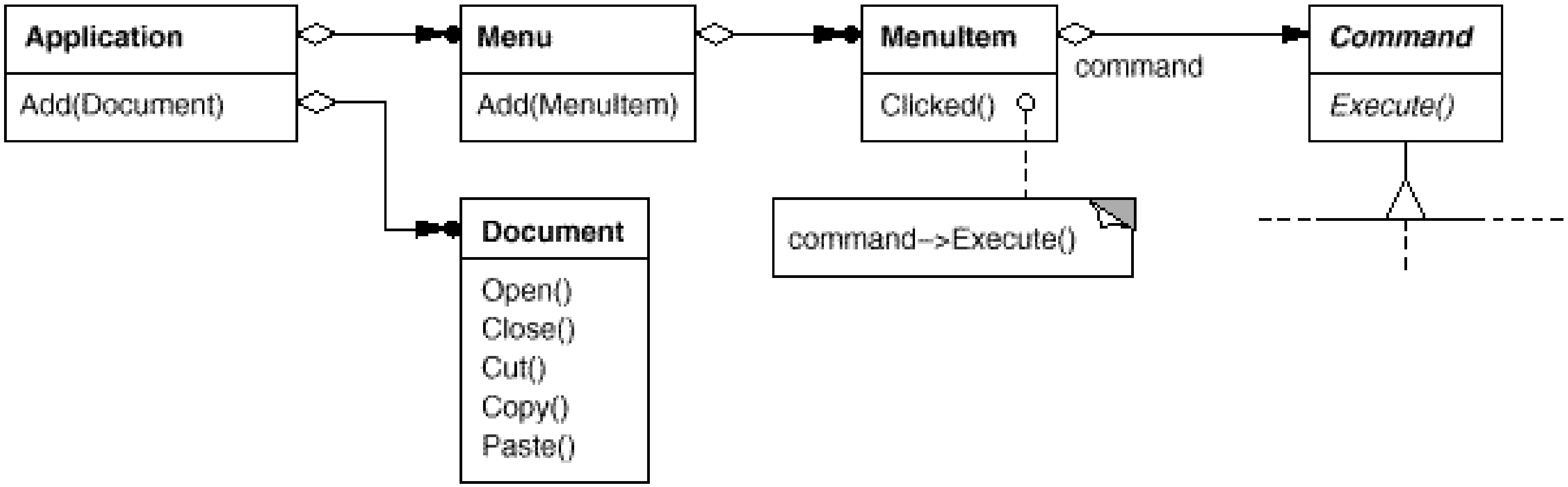
Decorator - GLOS

```
(defgeneric draw
  (method ((comp <visual-component>))
        (format true "drawing visual-component~%"))
  (method ((w <window>))
        (draw (window-contents w)))

(defmethod (decorate (component <visual-component>)
                     (decoration <visual-component>))
  (add-after-method draw
        (method ((c (== component)))
             (draw decoration))))

(defrectype <border-decorator> (<visual-component>)
  ((width <int>))
  (width border-width set-border-width!))

(gfmethod (draw (comp <border-decorator>))
        (draw-border (border-width comp)))
(define tv1 (new <text-view>))
(define w1 (new <window> 'contents tv1))
(decorate tv1 (new <scroll-decorator>))
(decorate tv1 (new <border-decorator> 'width 4))
```

Feature:
method combination

Command

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



Command in C++

```
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();
       }
   }
```

Command in GLOS

```
(define (make-open-command app)
  (lambda ()
    (let ((name (ask-user)))
      (if name
          (let ((doc (new <document> name)))
            (add app doc)
            (open doc)))))))
...
(add-menuitem some-menu
              (make-open-command the-app))
...
((menuitem-command some-menuitem))
```

Feature:
first class functions

- In all functional languages, including Smalltalk.
- Doesn't account for undo feature of pattern.

Iterator (Internal) - C++

```
template <class Item>
class FilteringListTraverser {
public:
    FilteringListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
    virtual bool TestItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
template <class Item>
void FilteringListTraverser<Item>::Traverse () {
    bool result = false;
    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        if (TestItem(_iterator.CurrentItem())) {
            result =
ProcessItem(_iterator.CurrentItem());
            if (result == false) {
                break;
            }
        }
    }
    return result;
}
```

```
class HighlyPaidEmployees :
    public
FilteringListTraverser<Employee*> {
public:
    HighlyPaidEmployees(List<Employee*>*
aList, int n) :
           FilteringListTraverser<Employee*
>(aList),
           _min(n) { }
protected:
    bool ProcessItem(Employee* const&);
    bool TestItem(const Employee&);
private:
    int _total;
    int _count;
};
bool HighlyPaidEmployees::ProcessItem
          (Employee* const& e) {
    _count++;
    e->Print();
}
bool HighlyPaidEmployees::TestItem
          (Employee* const& e) {
    return e->Salary() > _min
}

List<Employee*>* employees;
// ...
HighlyPaidEmployees pa(employees, 100000);
pa.Traverse();
```

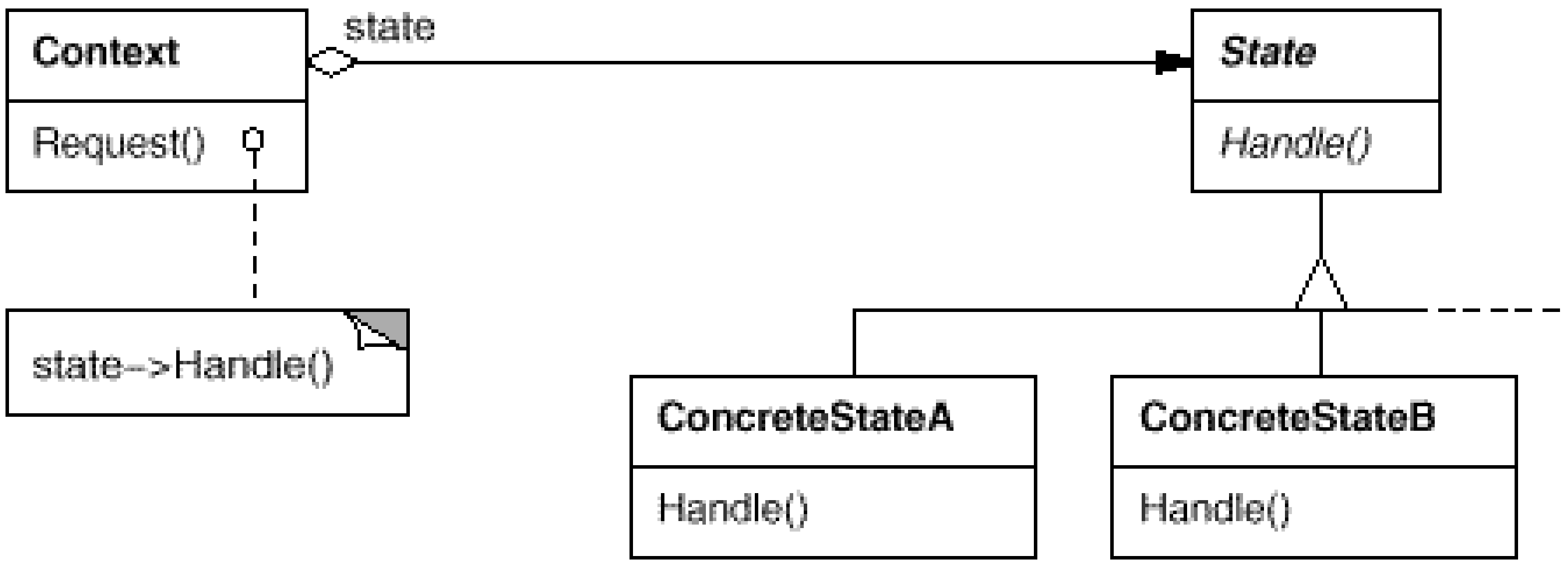
Internal Iterator in GLOS

```
(define employees (list ...))
(filter (lambda (e)
           (> (employee-salary e) 100000))
        employees)
```

- See also iteration protocol of Dylan

State

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



State in C++

```
class TCPState;
class TCPConnection {
public:
    void Open();
    void Close();
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
class TCPState {
public:
    virtual void Open(TCPConnection*);
    virtual void Close(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
TCPConnection::TCPConnection () {
    _state = TCPClosed::Instance();
}
void TCPConnection::ChangeState (TCPState* s) {
    _state = s;
}
void TCPConnection::Open () {
    _state->Open(this);
}
void TCPConnection::Close () {
    _state->Close(this);
}
```

```
void TCPState::Open (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::ChangeState
        (TCPConnection* t, TCPState* s) {
    t->ChangeState(s);
}
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();
    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 Open(TCPConnection*);
    // ...
};
void TCPClosed::Open (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.
    ChangeState(t,
TCPEstablished::Instance());
}
void TCPEstablished::Close (TCPConnection*
t) {
    // send FIN, receive ACK of FIN
    ChangeState(t, TCPListen::Instance());
}
```

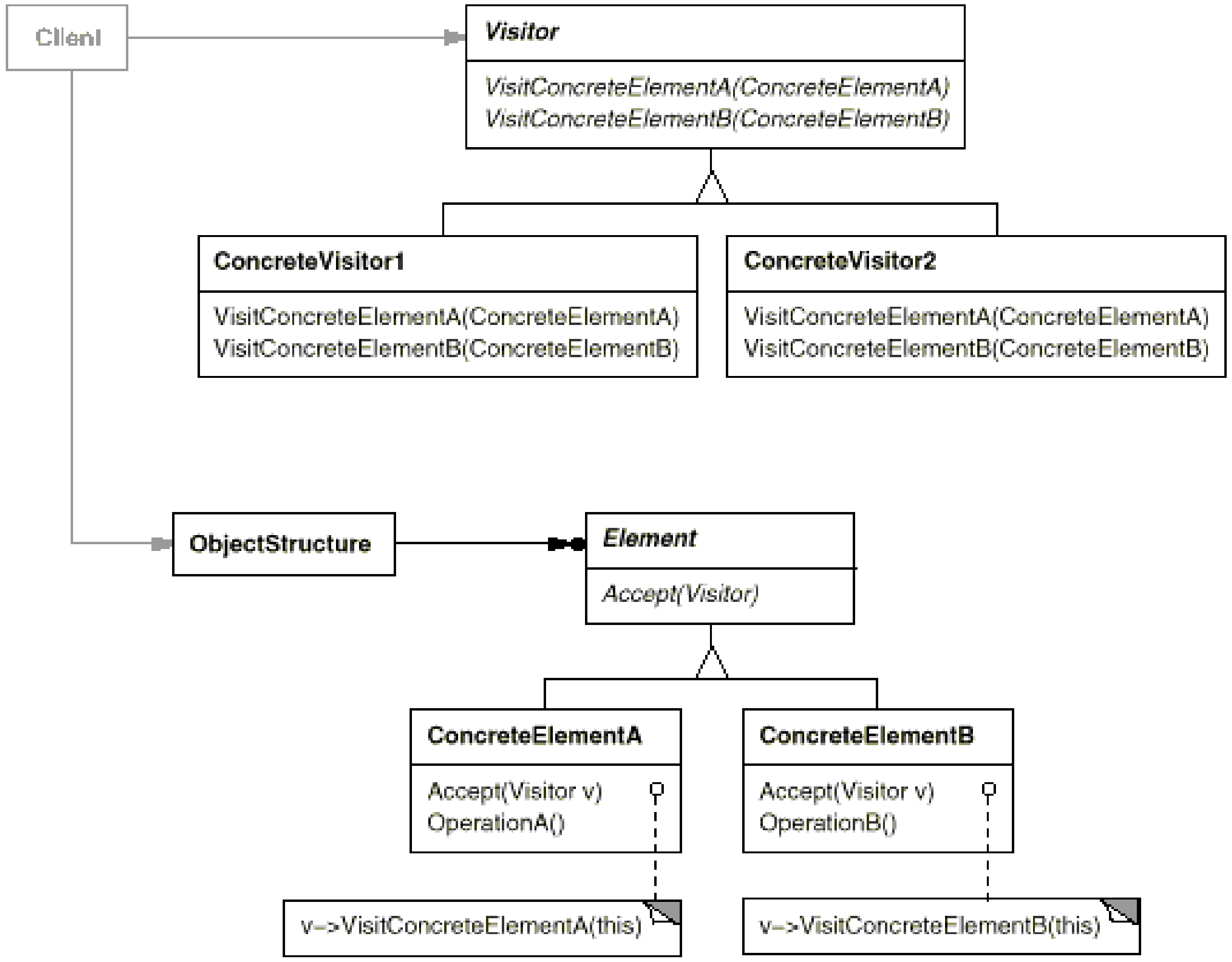
State in GLOS

```
(defrectype <tcp-connection> ()
  ((status <symbol> 'closed))
  (status connection-status
          set-connection-status!))
(define <open>
  (and? <tcp-connection>
        (lambda (c)
          (eq? 'open (connection-status c)))))
(define <closed>
  (and? <tcp-connection>
        (lambda (c)
          (eq? 'closed (connection-status c)))))
(defgeneric open
  (method ((c <tcp-connection>))
          (error "Cannot open connection."))
  (method ((c <closed>))
          (set-connection-status! c 'open)))
(defgeneric transmit
  (method ((c <tcp-connection>) data)
          (error "Cannot transmit on connection."))
  (method ((c <open>) data)
          (format true "Transmitting: ~a~%" data)))
(defgeneric close
  (method ((c <tcp-connection>))
          (error "Cannot close connection."))
  (method ((c <open>))
          (set-connection-status! c 'closed)))
```

Feature:
predicate types

Visitor

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.



Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Visitor, continued

- Mostly solved with multiple dispatch (along with data structure traversal). In single dispatch languages, this is referred to as “double dispatch”.

Summary: Language Features

- Protocols:
  - Instantiation
  - Method call

Relation to Modeling

- Modeling allow for pre-runtime verification
- Many of these language features, adding dynamism, reflection, abstraction, make programs **more** difficult to analyze statically. Oops.
- Alloy focuses most on structural – these language features mostly dynamic.

Discussion

- Are all Design Patterns solvable with advanced language features? (I used to think so) Not so simple:
  - some GOF patterns appear universal (Template Method)
  - any language, no matter what features, will manifest its own design patterns.
  - GOF book discusses design tradeoffs.

Discussion, Continued

- Can Design Patterns be made 1st class?
  - Can we instantiate design patterns?
  - Maybe.

Design Pattern Catalog

Creational Patterns

Abstract Factory (87)
: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder (97)
: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Factory Method (107)
: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Prototype (117)
: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton (127)
: Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

Adapter (139)
: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge (151)
: Decouple an abstraction from its implementation so that the two can vary independently.

Composite (163)
: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator (175)
: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade (185)
: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Flyweight (195)
: Use sharing to support large numbers of fine-grained objects efficiently.

Proxy (207)
: Provide a surrogate or placeholder for another object to control access to it.

Behavioral Patterns

Chain of Responsibility (223)
: 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.

Command (233)
: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Interpreter (243)
: Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator (257)
: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator (273)
: 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.

Memento (283)
: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer (293)
: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

State (305)
: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy (315)
: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template Method (325)
: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor (331)
: 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.

More

- Chain of Responsibility: shows how 1st class generics can be used.