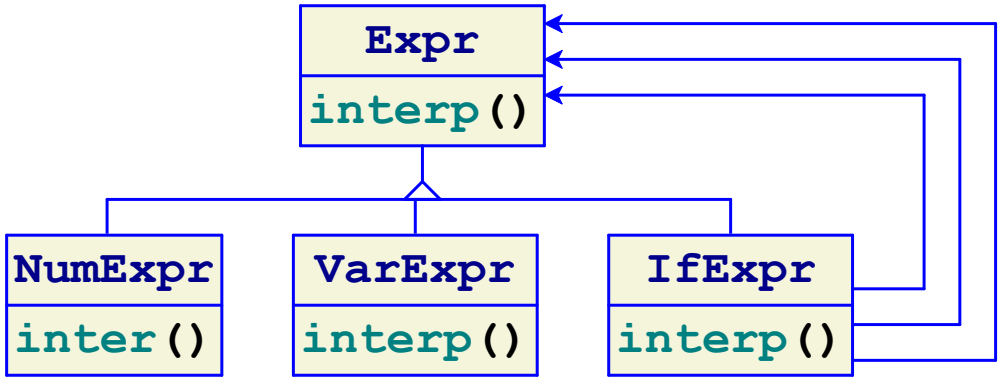
What are Design Patterns?

A ***design pattern*** is a program structure to solve a common kind of problem

Writing code: gives you a template to follow

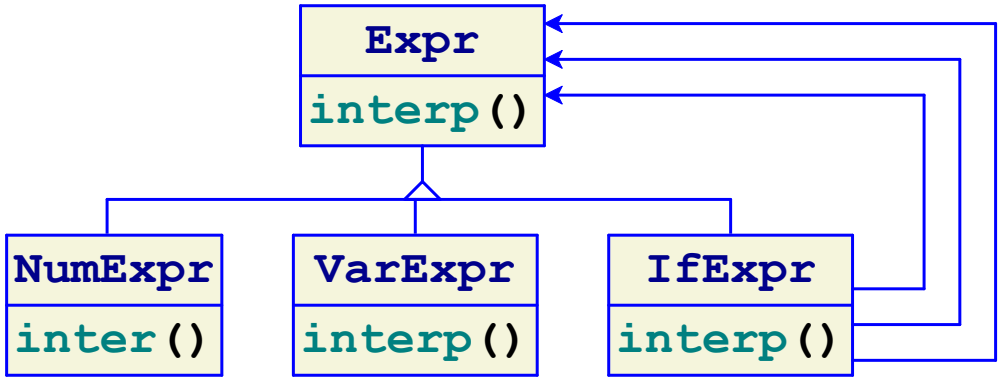
Understanding code: helps you understand how a program, library, or API works

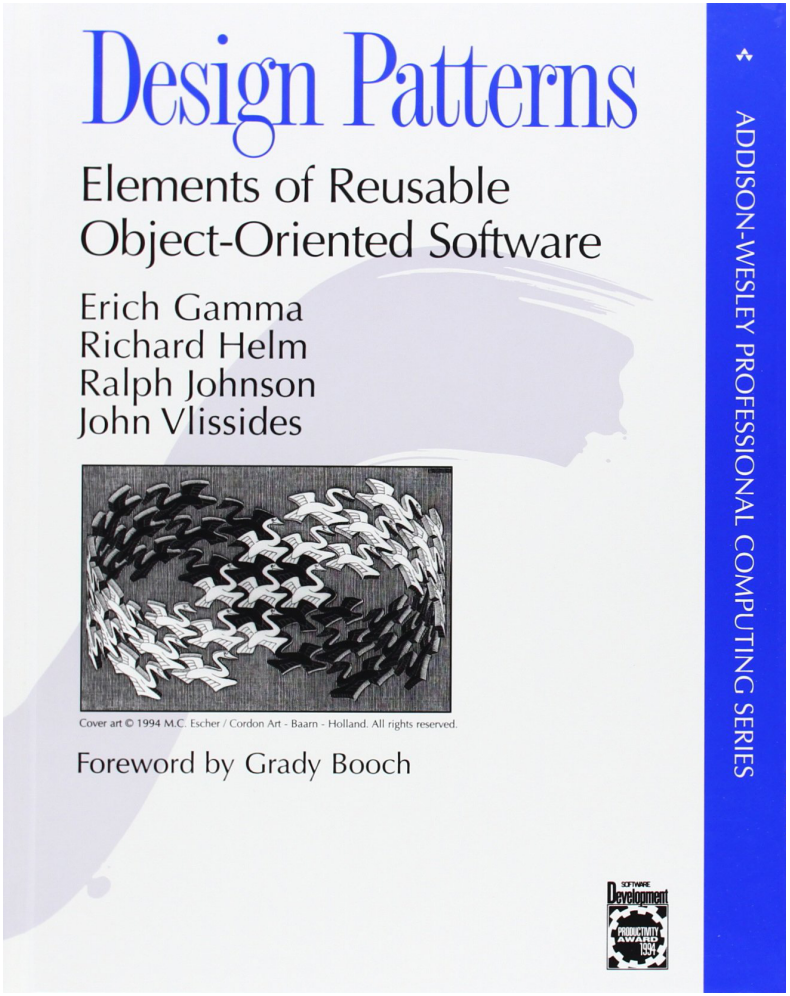
interpreter Pattern



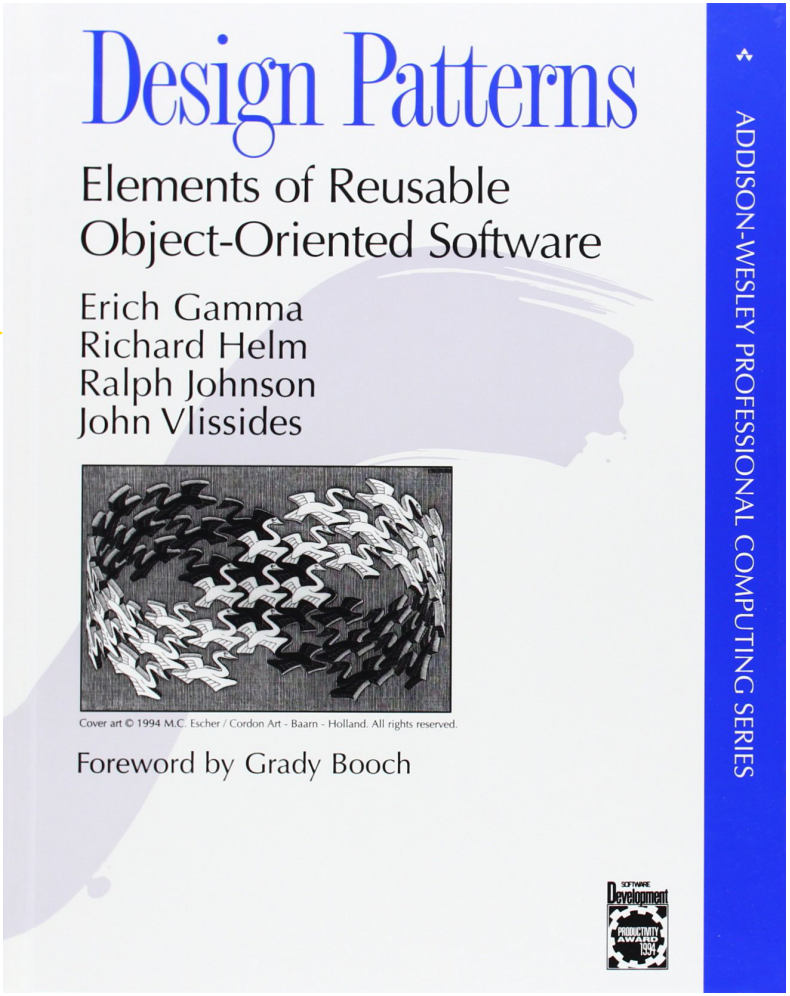
interpreter Pattern

Problem: interpret nested data recursively



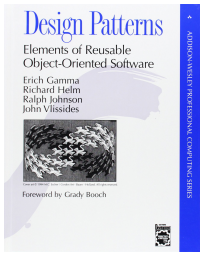

Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

"Gang of four"


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Patterns and Languages

Design patterns are relative to a programming language



is about class-based, object-oriented patterns

One language's **pattern** is another language's built-in **construct**

e.g., iterator

Classifying Patterns

Creational — how objects are created

abstract factory | builder | factory method | prototype | singleton

Structural — how objects are combined

adapter | bridge | composite | decorator | facade | flyweight | proxy

Behavioral — how methods are implemented

chain of responsibility | command | interpreter | mediator | memento

observer | state | strategy | template method | visitor

singleton Pattern

Problem: a class should be instantiated only once

```
class GUIApplication {
  GUIApplication() {
    make_menu_bar();
    ....
  }

  void handle_events();
  void to_foreground();
  ....
};
```

singleton Pattern

```
class GUIApplication {
protected:
  GUIApplication();
  GUIApplication *the_app;

public:
  // returns the_app:
  static GUIApplication *get_instance();

  void handle_events();
  void to_foreground();
  ....
};
```

singleton Pattern

Private constructor

```
class GUIApplication {
protected:
  GUIApplication();
  GUIApplication *the_app;

public:
  // returns the_app:
  static GUIApplication *get_instance();

  void handle_events();
  void to_foreground();
  ....
};
```

singleton Pattern

```
class GUIApplication {
protected:
  GUIApplication();
  GUIApplication *the_app;

public:
  // returns the_app:
  static GUIApplication *get_instance();

  void handle_events();
  void to_foreground();
  ....
};
```

Always returns the same object

singleton Pattern

```
class GUIApplication {
protected:
  GUIApplication();
  GUIApplication *the_app;

public:
  // returns the_app:
  static GUIApplication *get_instance();

  void handle_events();
  void to_foreground();
  ....
};
```

Why not global functions?

- May allow subclassing
- Might switch to multiple

builder Pattern

```
Sandwich s1 = new Sandwich(Bread::wheat,
                           Protein::turkey,
                           /* lettuce */ true, false,
                           /* tomato */ true, false,
                           false, /* mayo */ true, false);

Sandwich s2 = new Sandwich(Bread::sourdough,
                           Protein::none,
                           false, /* sprouts */ true,
                           false, false,
                           false, false, /* ranch */ true);
```

builder Pattern

```
Sandwich s1 = new Sandwich(Bread::wheat,
                           Protein::turkey,
                           /* lettuce */ true, false,
                           /* tomato */ true, false,
                           false, /* mayo */ true, false);

Sandwich s2 = new Sandwich(Bread::sourdough,
                           Protein::none,
                           false, /* sprouts */ true,
                           false, false,
                           false, false, /* ranch */ true);
```

Problem: lots of constructor arguments

builder Pattern

```
SandwichBuilder build(Bread::wheat); // must pick a bread

build.with_protein(Protein::turkey); // add turkey

// chain additional toppings
build.with_lettuce().with_tomato().with_mayo();
if (is_tuesday)
  build.with_bacon();

Sandwich s = build.make_sandwich();
... // toppings in s cannot change
```

builder Pattern

Builder object collects constructor options

```
SandwichBuilder build(Bread::wheat); // must pick a bread

build.with_protein(Protein::turkey); // add turkey

// chain additional toppings
build.with_lettuce().with_tomato().with_mayo();
if (is_tuesday)
  build.with_bacon();

Sandwich s = build.make_sandwich();
... // toppings in s cannot change
```

builder Pattern

```
SandwichBuilder build(Bread::wheat); // must pick a bread

build.with_protein(Protein::turkey); // add turkey

// chain additional toppings
build.with_lettuce().with_tomato().with_mayo();
if (is_tuesday)
  build.with_bacon();

Sandwich s = build.make_sandwich();
... // toppings in s cannot change
```

Builder method instead of `new Sandwich`

factory method Pattern

Problem: class to instantiate can change

```
Room *make_hallway() {
  Room *hall = new Room();

  hall->set_south(new Door());
  hall->set_east(new Wall());
  hall->set_west(new Wall());
  hall->set_north(new Door());

  return hall;
}
```

factory method Pattern

Problem: class to instantiate can change

```
Room *make_hallway() {
  Room *hall = new Room();

  hall->set_south(new Door());
  hall->set_east(new Wall());
  hall->set_west(new Wall());
  hall->set_north(new Door());

  return hall;
}
```

Use with **Room** and **Door** subclasses?

factory method Pattern

```
class MazeGame {
  ....
  Room *make_room() { return new Room(); }
  Wall *make_wall() { return new Wall(); }
  Door *make_door() { return new Door(); }
  ....
};
```

```
Room *make_hallway(MazeGame *game) {
  Room *hall = game->make_room();

  hall->set_south(game->make_door());
  ....

}
```

factory method Pattern

```
class MazeGame {
  ....
  Room *make_room() { return new Room(); }
  Wall *make_wall() { return new Wall(); }
  Door *make_door() { return new Door(); }
  ....
};
```

```
Room *make_hallway(MazeGame *game) {
  Room *hall = game->make_room();

  hall->set_south(game->make_door());
  ....

}
```

Instead of `new Room()`

factory method Pattern

```
class MazeGame {
  ....
  Room *make_room() { return new Room(); }
  Wall *make_wall() { return new Wall(); }
  Door *make_door() { return new Door(); }
  ....
};
```

```
class BombMazeGame : public MazeGame {
  Room *make_room() { return new BombRoom(); }
  ....
};
```

factory method Pattern

Turns a compile-time decision into a run-time decision

```
class MazeGame {
  ....
  Room *make_room() { return new Room(); }
  Wall *make_wall() { return new Wall(); }
  Door *make_door() { return new Door(); }
  ....
};
```

```
class BombMazeGame : public MazeGame {
  Room *make_room() { return new BombRoom(); }
  ....
};
```

dependency injection Pattern (not in )

```
check_homework(string filename) {
  Compiler *c = homework7_compiler;
  Runner *r = new homework7_runner;
  r->run(c->compile(filename));
}
```

Problem: object to use can change

dependency injection Pattern (not in )

```
check_homework(string filename) {
  Compiler *c = homework7_compiler;
  Runner *r = new homework7_runner;
  r->run(c->compile(filename));
}
```

⇒

```
check_homework(string filename,
               Compiler *c,
               Runner *r) {
  r->run(c->compile(filename));
}
```

dependency injection Pattern (not in )

```
check_homework(string filename) {
  Compiler *c = homework7_compiler;
  Runner *r = new homework7_runner;
  r->run(c->compile(filename));
}
```

⇒

```
check_homework(string filename,
               Dictionary *tools) {
  Compiler *c = tools->get("compiler");
  Runner *r = tools->get("runner");
  r->run(c->compile(filename));
}
```

dependency injection Pattern (not in )

```
check_homework(string filename) {
  Compiler *c = homework7_compiler;
  Runner *r = new homework7_runner;
  r->run(c->compile(filename));
}
```

⇒

```
check_homework(string filename,
               ToolMaker *tools) {
  Compiler *c = tools->make_compiler();
  Runner *r = tools->make_runner();
  r->run(c->compile(filename));
}
```

dependency injection Pattern (not in )

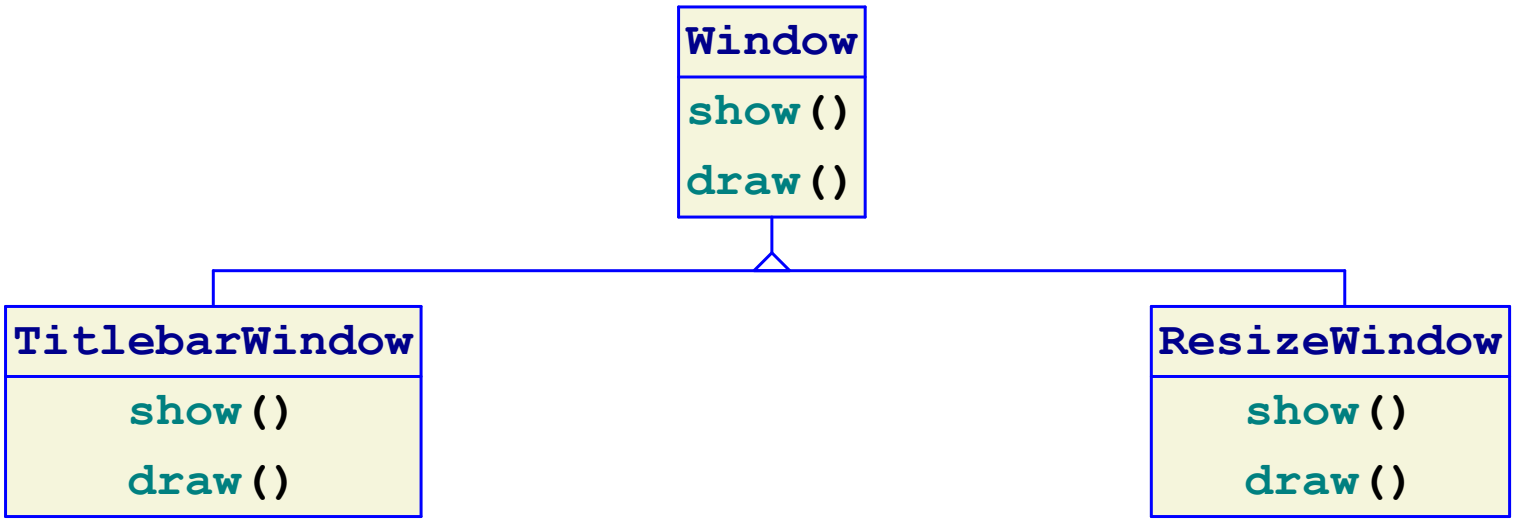
```
check_homework(string filename) {
  Compiler *c = homework7_compiler;
  Runner *r = new homework7_runner;
  r->run(c->compile(filename));
}
```

⇒

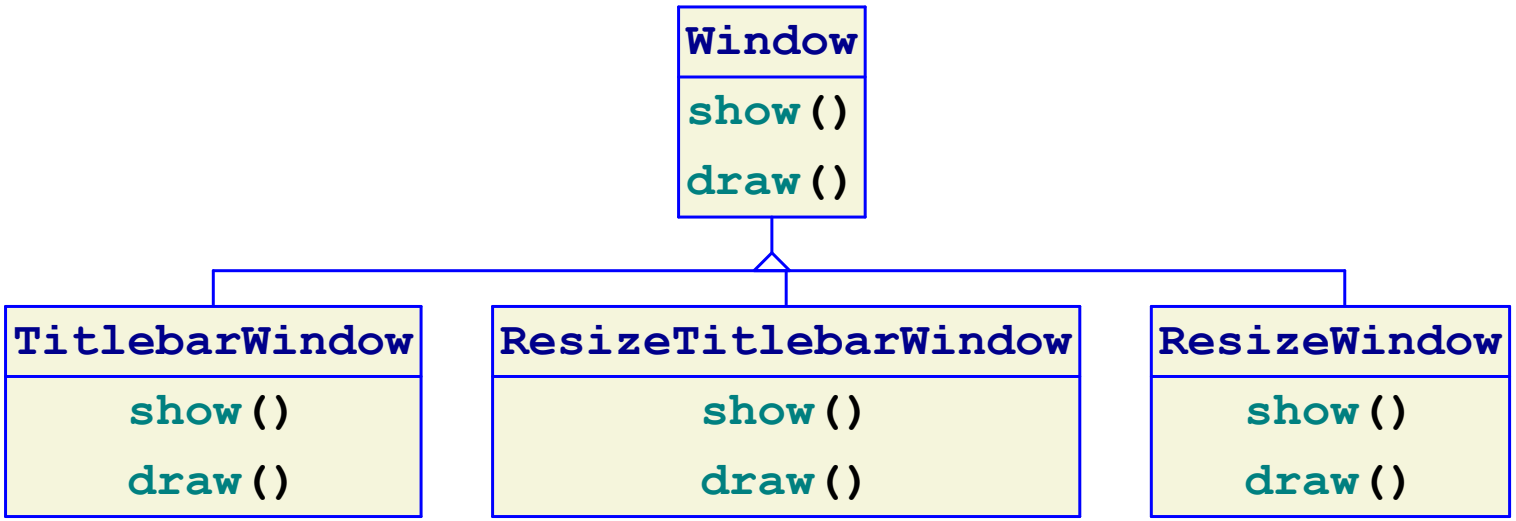
```
check_homework(string filename,
               ToolMaker *tools) {
  Compiler *c = tools->make_compiler();
  Runner *r = tools->make_runner();
  r->run(c->compile(filename));
}
```

dependency injection
via
factory methods

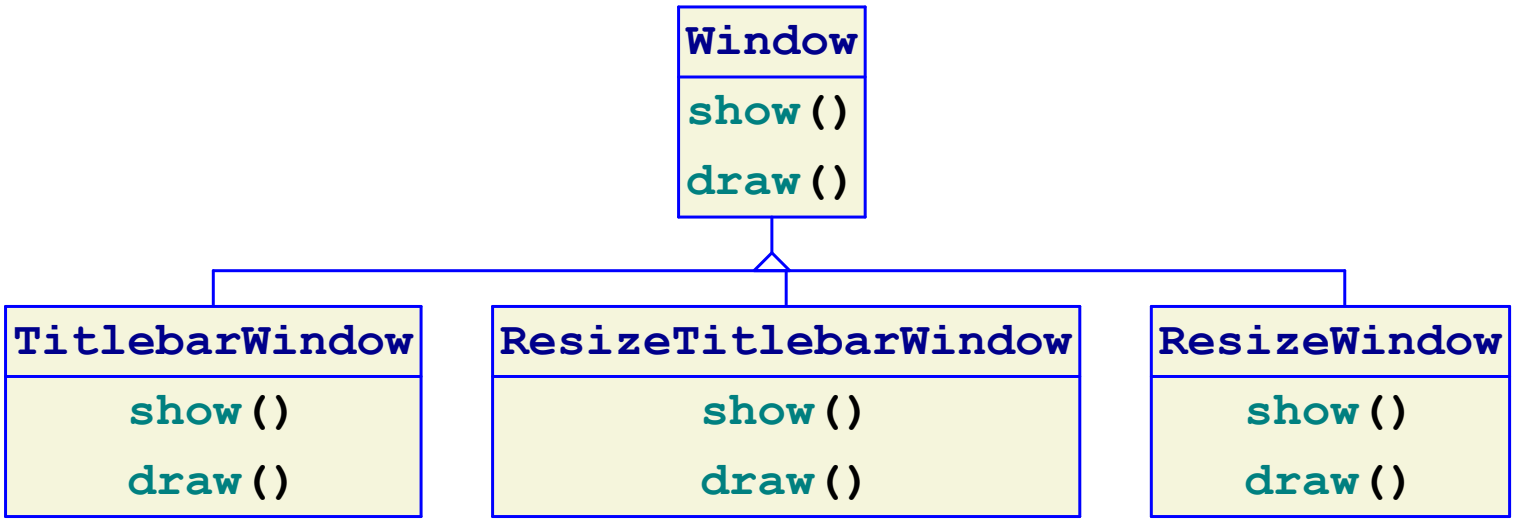
decorator Pattern



decorator Pattern

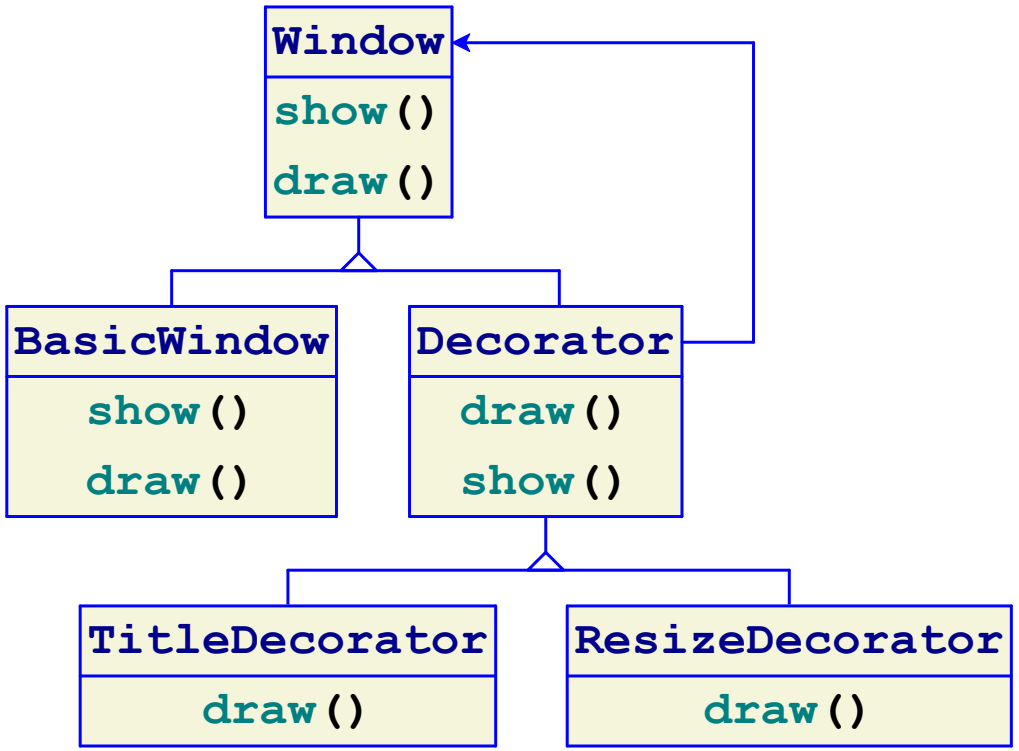


decorator Pattern

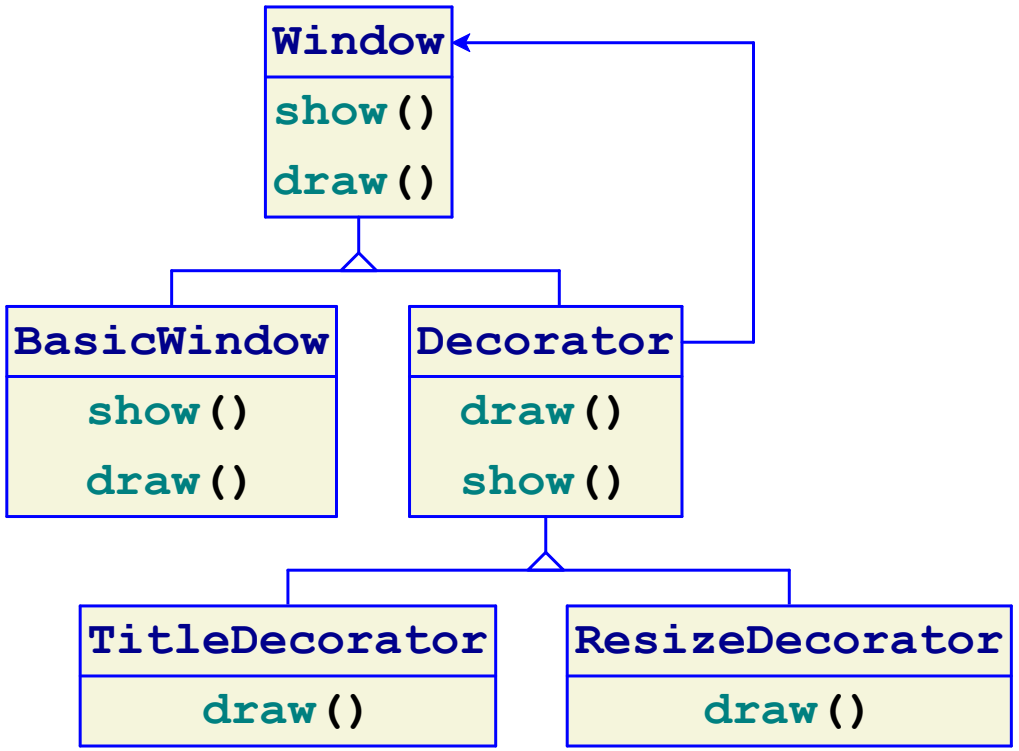


Problem: combinations create too many subclasses

decorator Pattern



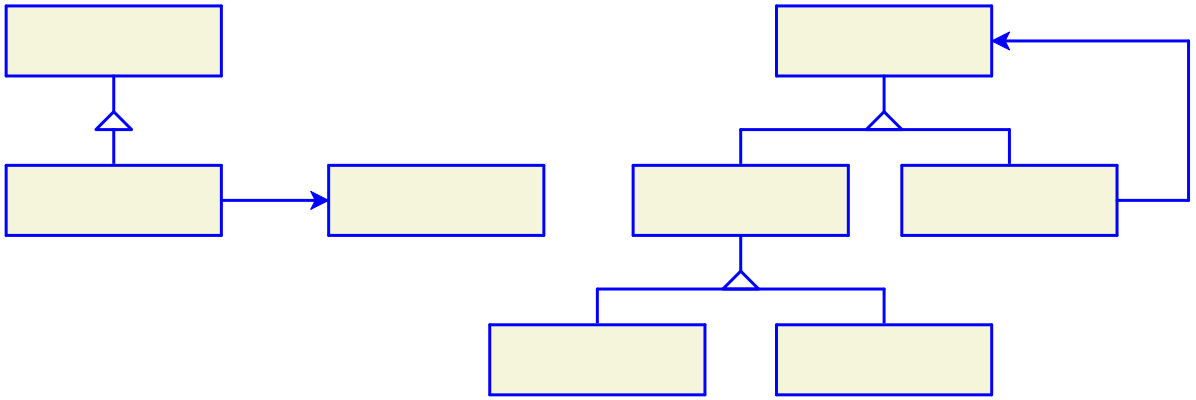
decorator Pattern



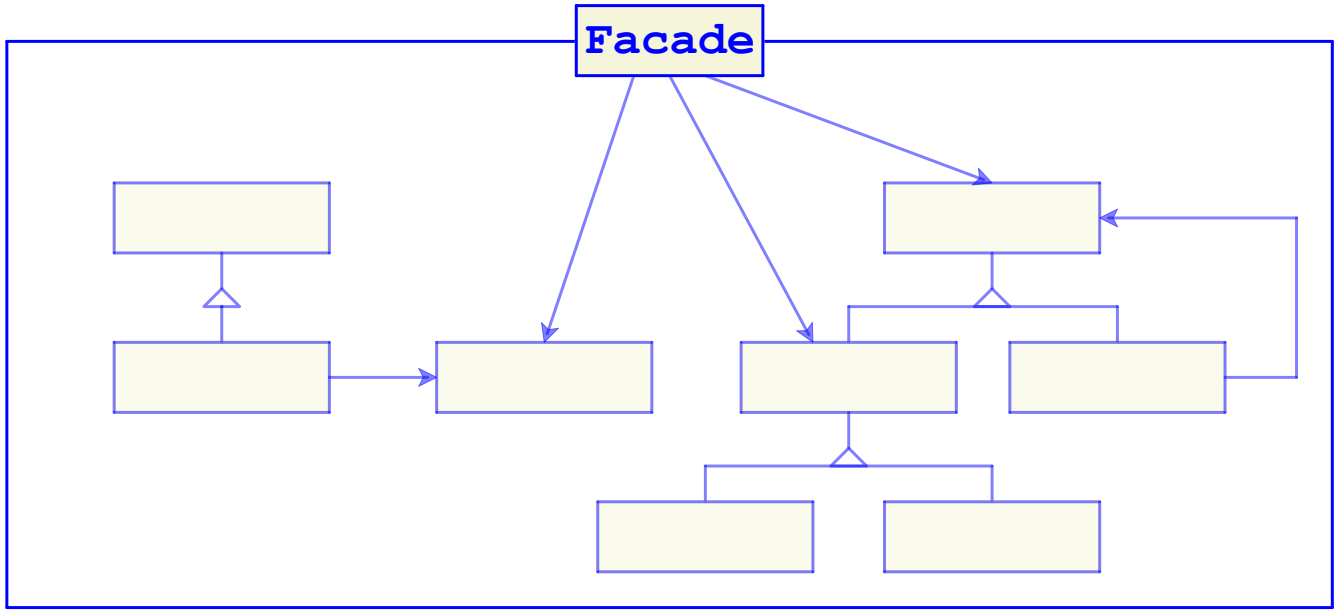

```
new BasicWindow()
new TitleDecorator(new BasicWindow())
new ResizeDecorator(new TitleDecorator(new BasicWindow()))
```

facade Pattern

Problem: complex and changing class structure exposed to clients

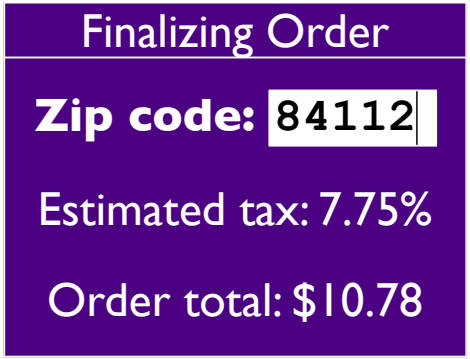


facade Pattern



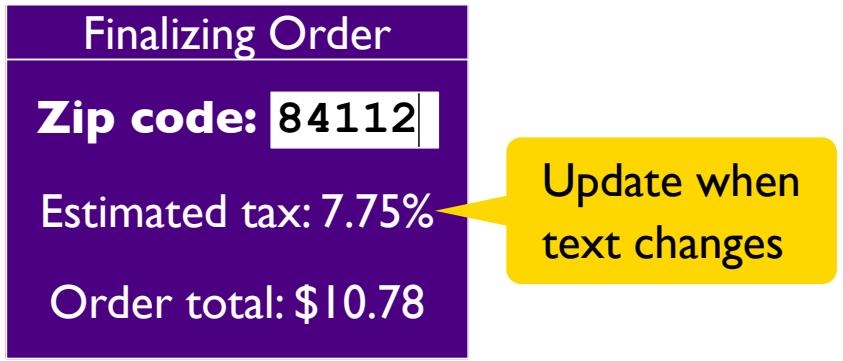
observer Pattern

```
class TextField {




  handle_key(Event key_evt) {
     ....

  }
  ....




};
```



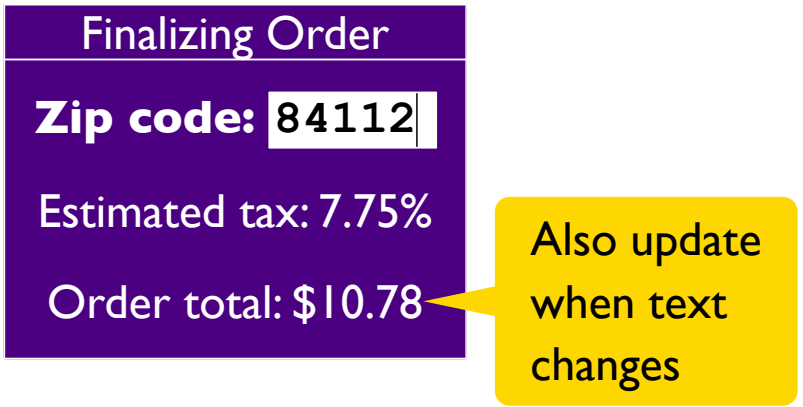
observer Pattern

```
class TextField {




  handle_key(Event key_evt) {
     ....

  }
  ....




};
```



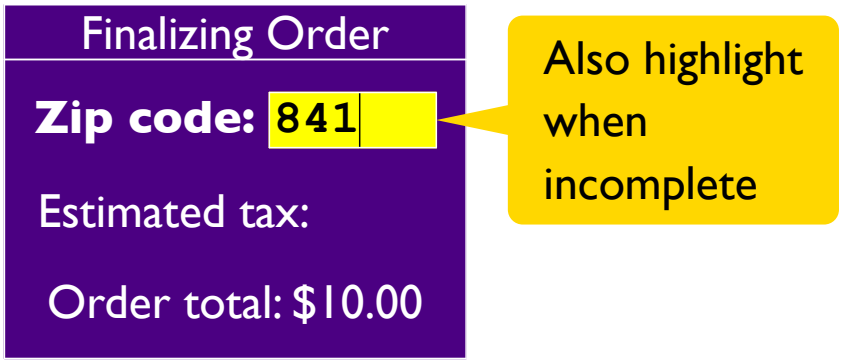
observer Pattern

```
class TextField {




  handle_key(Event key_evt) {
     ....

  }
  ....




};
```



observer Pattern

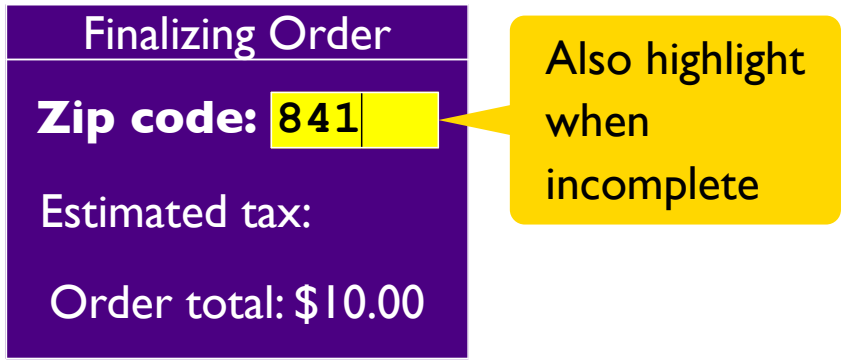
```
class TextField {




  handle_key(Event key_evt) {
     ....

  }
  ....




};
```



observer Pattern

```
class TextField {




  handle_key(Event key_evt) {
     ....

  }
  ....




};
```

Problem: many independent reactions to a change



observer Pattern

```
class TextField {
  vector<TextObserver> observers;
  AddObserver(TextObserver o) {
    observers.push(o);
  }

  handle_key(Event key_evt) {
    ....
    notify();
  }
  ....

  void notify() {
    for (TextObserver o : observers)
      o->changed();
  }
};
```

```
class TextObserver {
  virtual void changed() = 0;
};
```

Finalizing Order

Zip code: 84112

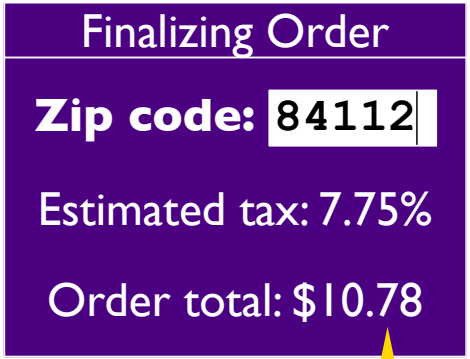
Estimated tax: 7.75%

Order total: $10.78

observer Pattern

```
class TextField {
  vector<TextObserver> observers;
  AddObserver(TextObserver o) {
    observers.push(o);
  }

  handle_key(Event key_evt) {
    ....
    notify();
  }
  ....

  void notify() {
    for (TextObserver o : observers)
      o->changed();
  }
};
```

```
class TextObserver {
  virtual void changed() = 0;
};
```



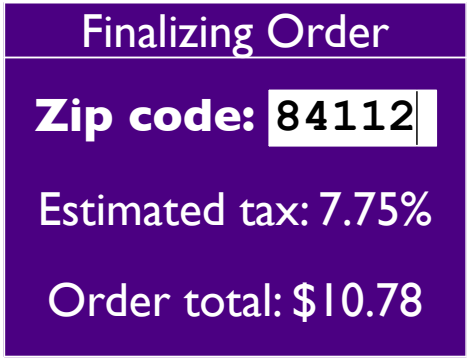
Create and register a **`TextObserver`** object for each reaction

observer Pattern

```
class TextField {
  vector<TextObserver> observers;
  AddObserver(TextObserver o) {
    observers.push(o);
  }

  handle_key(Event key_evt) {
    ....
    notify();
  }
  ....

  void notify() {
    for (TextObserver o : observers)
      o->changed();
  }
};
```

Called after any change

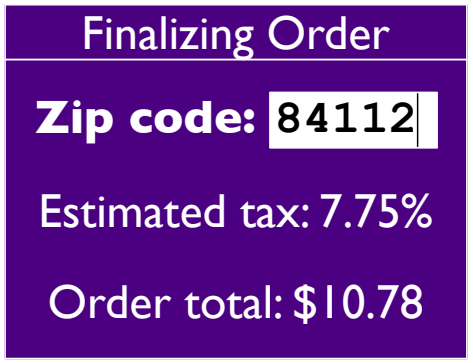
```
class TextObserver {
  virtual void changed() = 0;
};
```



state Pattern

```
class TextField {
  enum { ready, incomplete, disabled } state;

  void draw() {
    if (state == ready)
       ....
    else if (state == incomplete)
       ....
    else if (state == disabled)
       ....
  }

  handle_key(Event key_evt) {
     .... // state cases again
  }

  ....
};
```

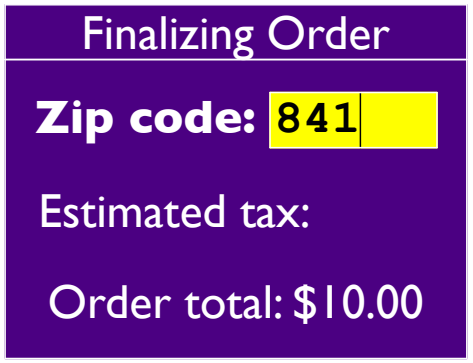
ready



state Pattern

```
class TextField {
  enum { ready, incomplete, disabled } state;

  void draw() {
    if (state == ready)
       ....
    else if (state == incomplete)
       ....
    else if (state == disabled)
       ....
  }

  handle_key(Event key_evt) {
     .... // state cases again
  }

  ....
};
```

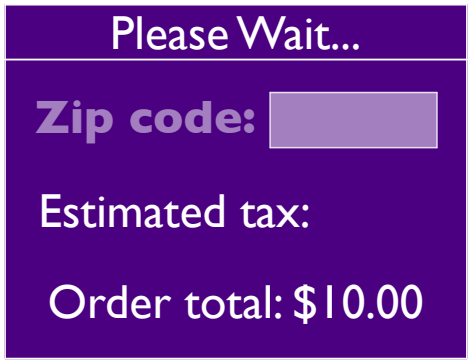
`incomplete`



state Pattern

```
class TextField {
  enum { ready, incomplete, disabled } state;

  void draw() {
    if (state == ready)
       ....
    else if (state == incomplete)
       ....
    else if (state == disabled)
       ....
  }

  handle_key(Event key_evt) {
     .... // state cases again
  }

  ....
};
```

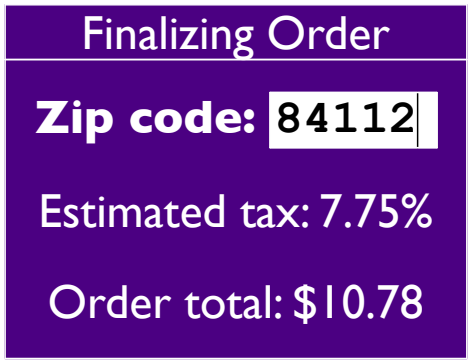
disabled



state Pattern

```
class TextField {
  enum { ready, incomplete, disabled } state;

  void draw() {
    if (state == ready)
       ....
    else if (state == incomplete)
       ....
    else if (state == disabled)
       ....
  }

  handle_key(Event key_evt) {
     .... // state cases again
  }

  ....
};
```

ready



Every method has cases for `state`

state Pattern

```
class TextField {
  enum { ready, incomplete, disabled } state;

  void draw() {
    if (state == ready)
       ....
    else if (state == incomplete)
       ....
    else if (state == disabled)
       ....
  }

  handle_key(Event key_evt) {
     .... // state cases again
  }

  ....
};
```

ready

Finalizing Order

Zip code: 84112

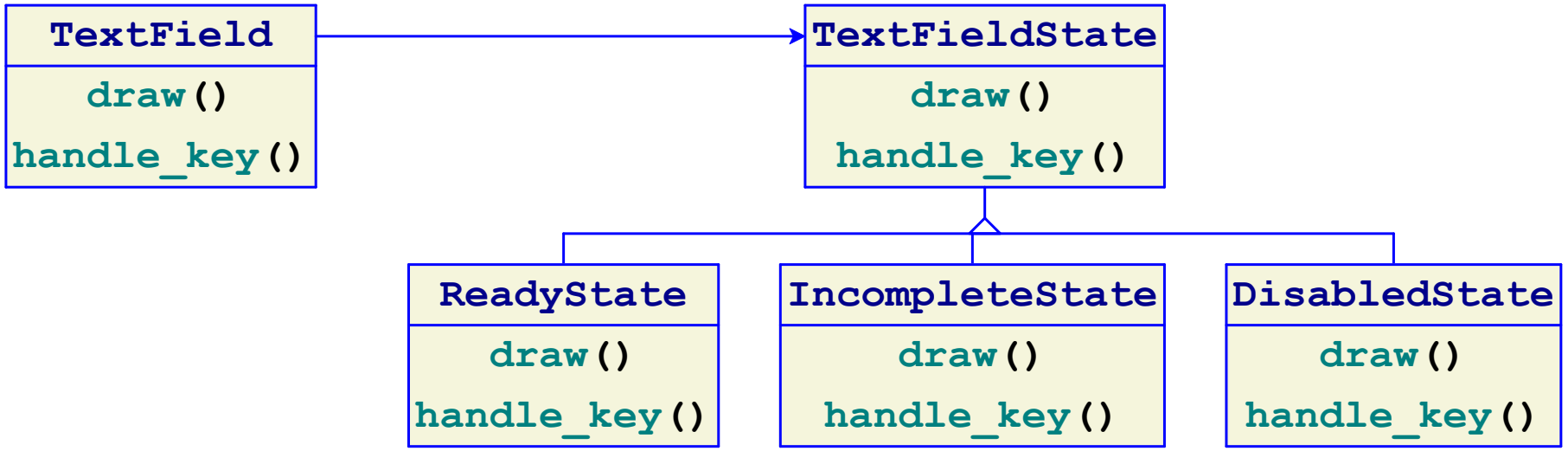
Estimated tax: 7.75%

Order total: $10.78

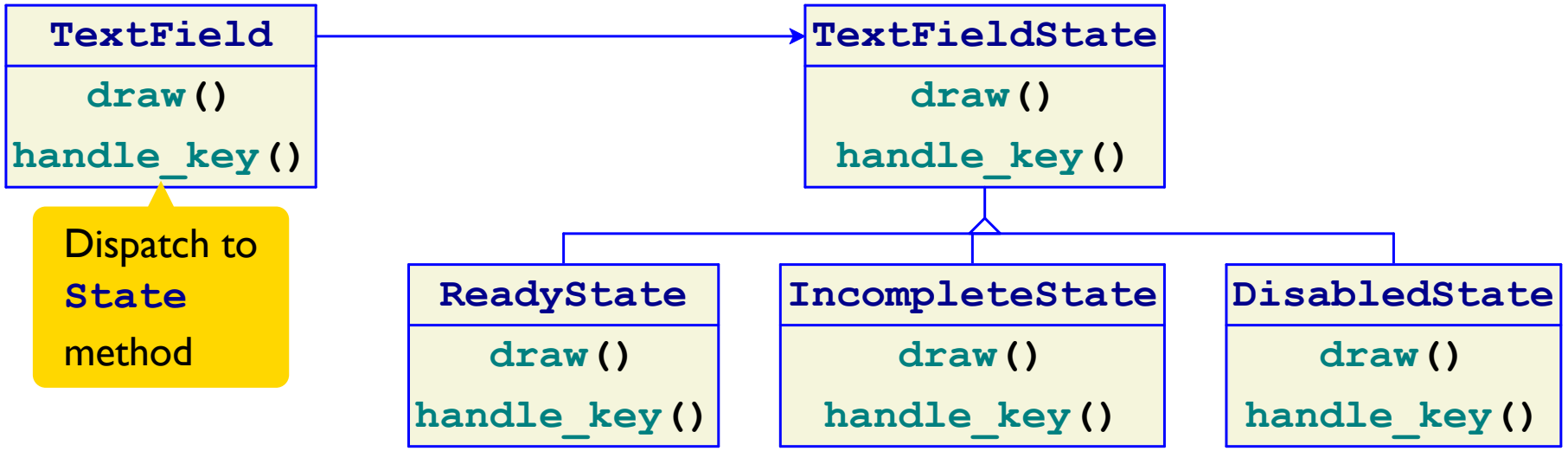
Every method has cases for `state`

Problem: state conditionals proliferate

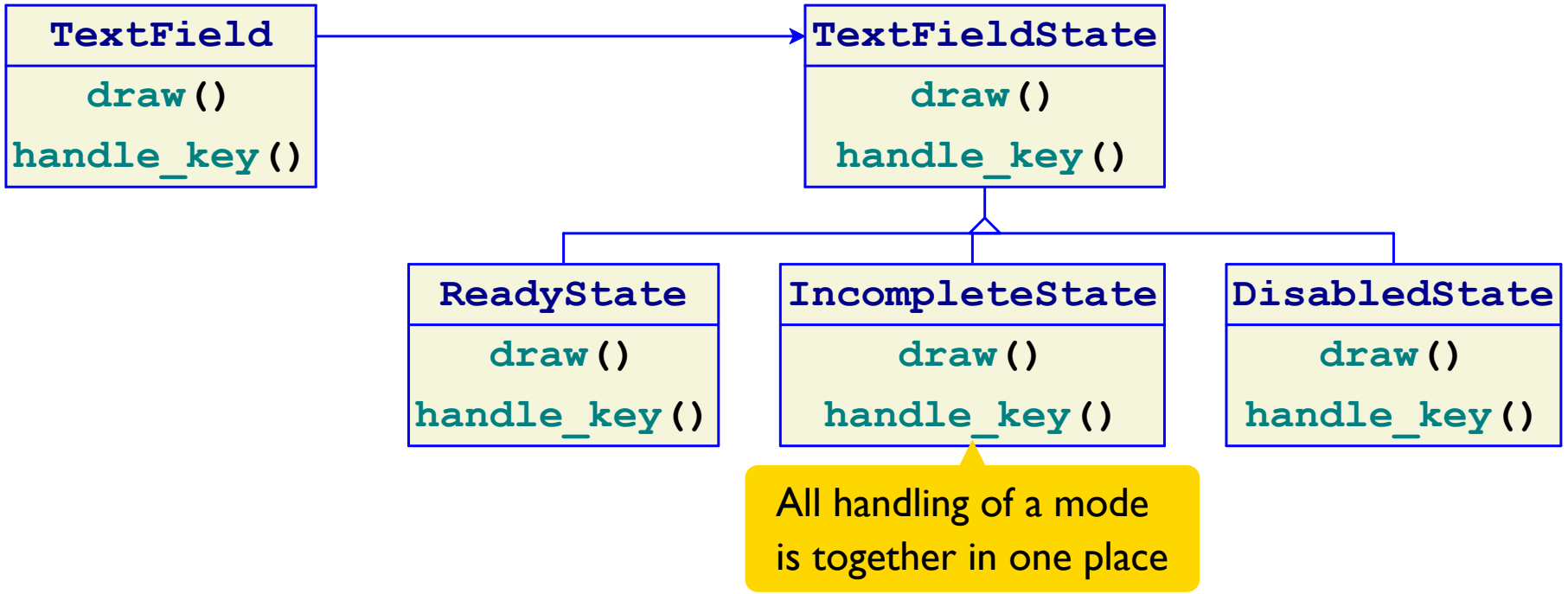
state Pattern



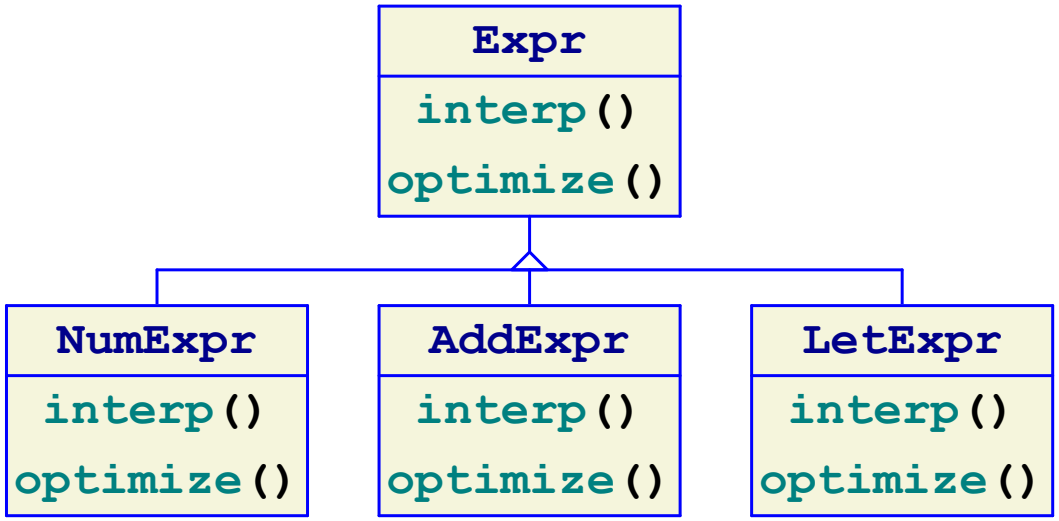
state Pattern



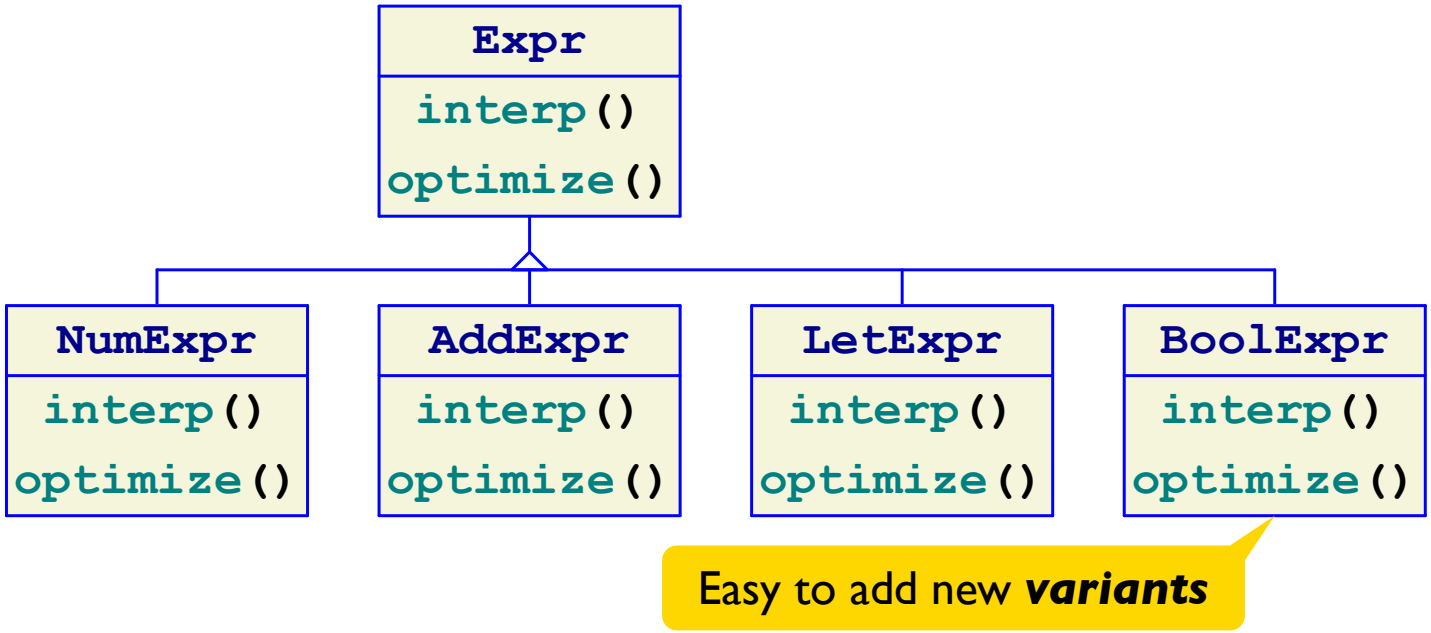
state Pattern



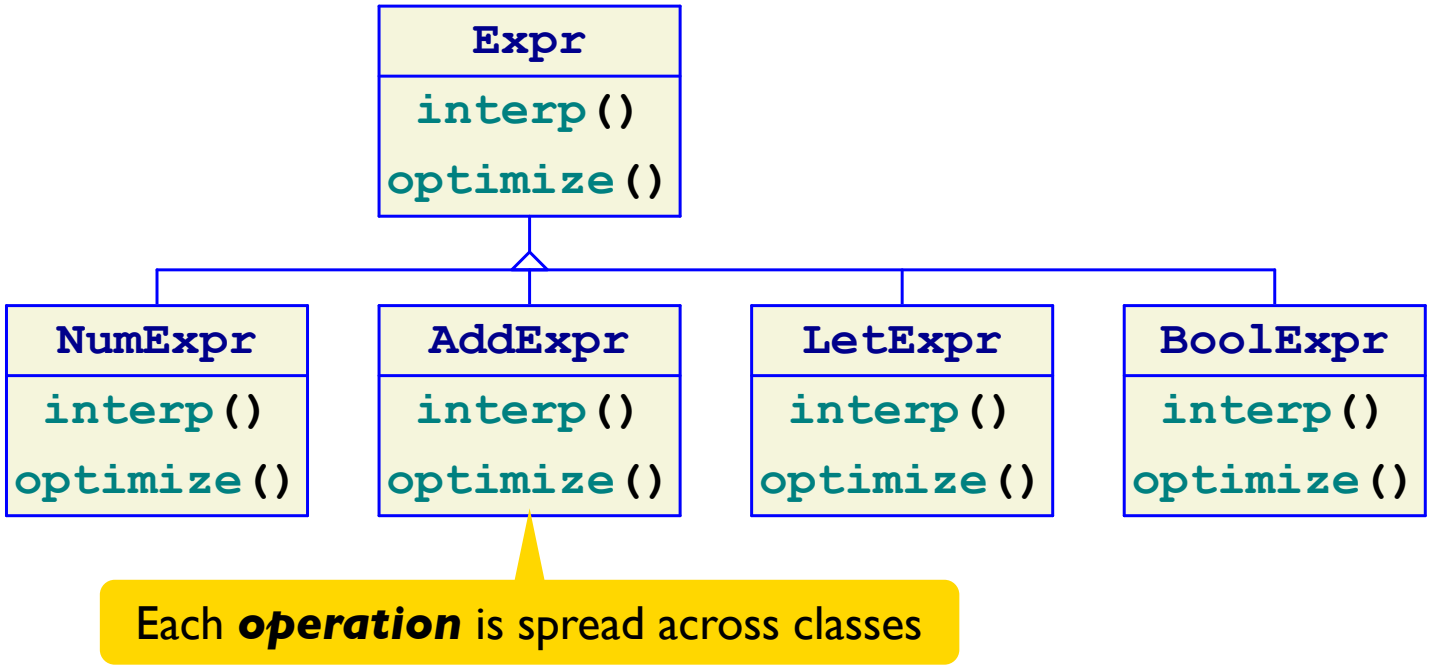
visitor Pattern



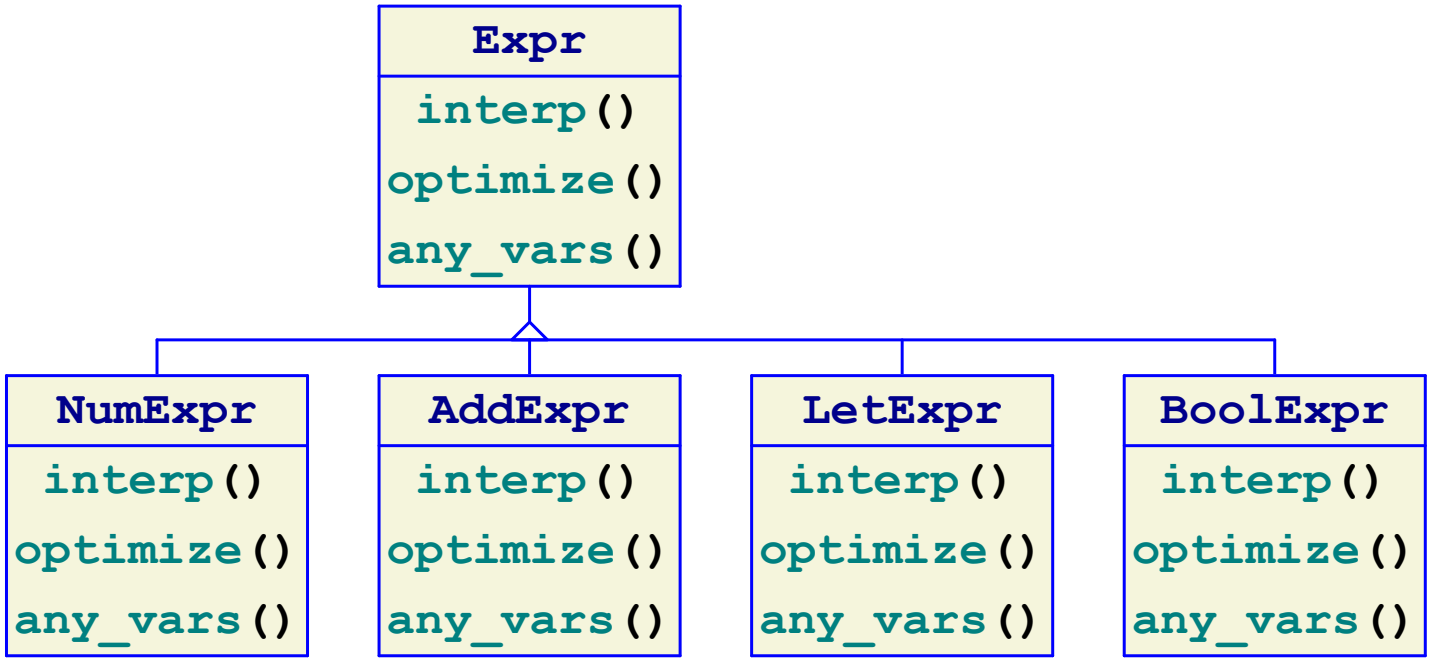
visitor Pattern



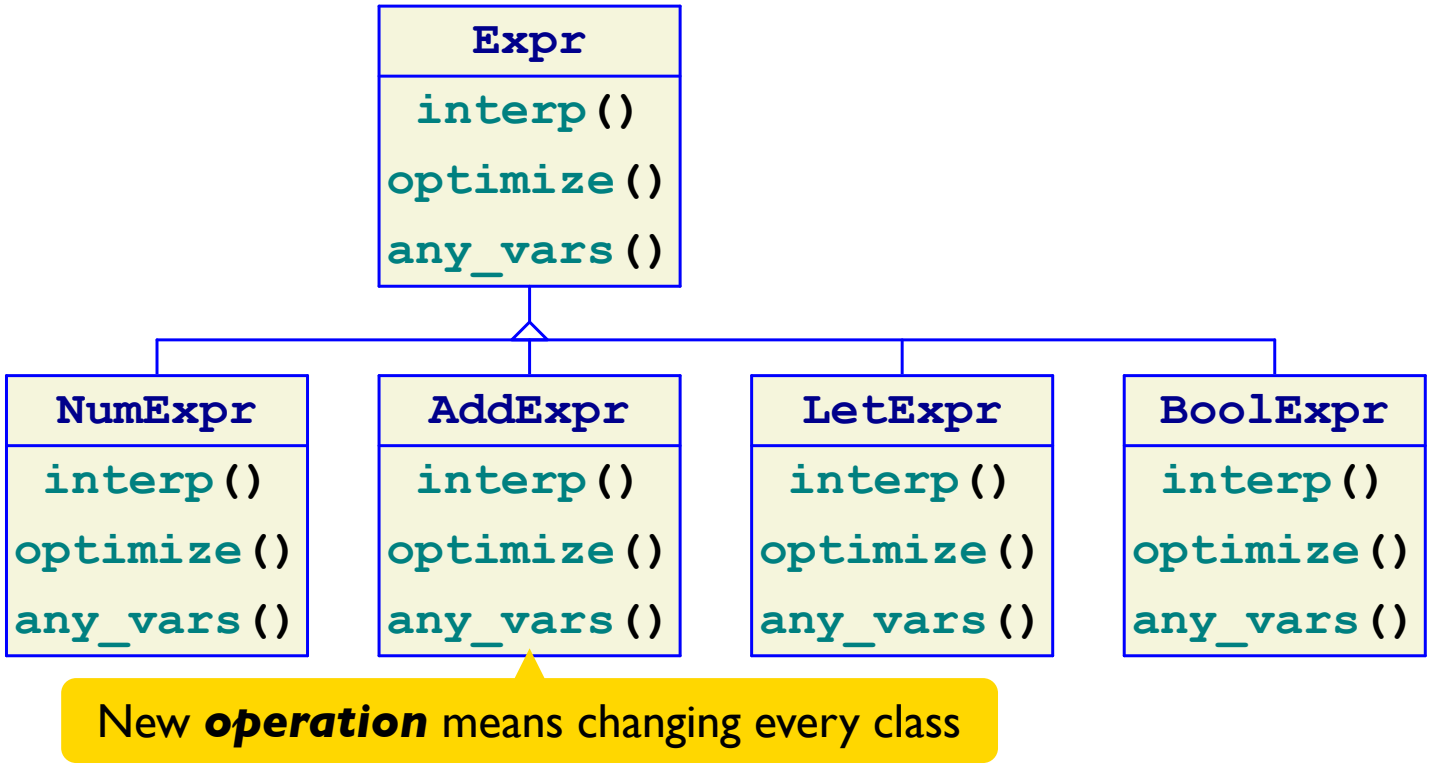
visitor Pattern



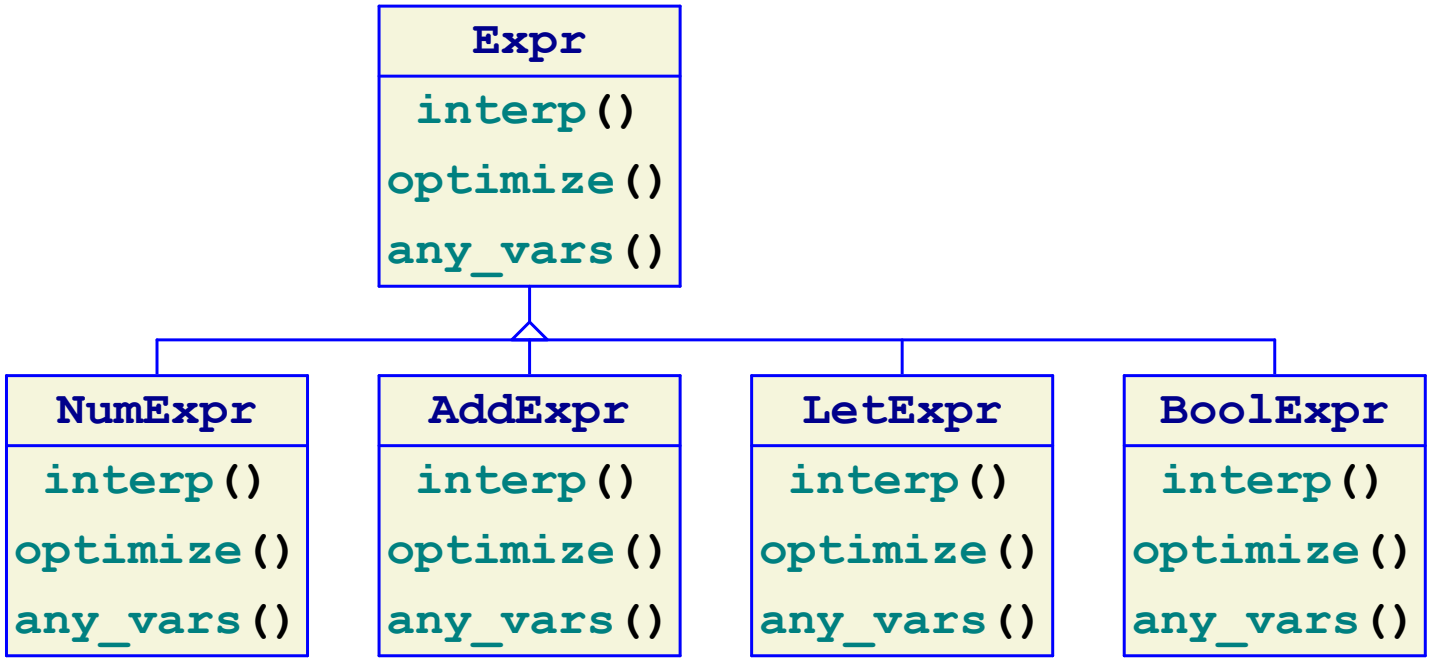
visitor Pattern



visitor Pattern



visitor Pattern



Problem: extensible set of independent operations

visitor Pattern

```
class Expr {
  virtual void* visit(ExprVisitor *visitor) = 0;
};
```

```
class ExprVisitor {
  virtual void* visit_num(int rep) = 0;
  virtual void* visit_add(Expr *lhs, Expr *rhs) = 0;
  virtual void* visit_let(string name, Expr *rhs, Expr* body) = 0;
  ....
};
```

visitor Pattern

```
class Expr {
  virtual void* visit(ExprVisitor *visitor) = 0;
};
```

Generic return type to accomodate any operation

```
class ExprVisitor {
  virtual void* visit_num(int rep) = 0;
  virtual void* visit_add(Expr *lhs, Expr *rhs) = 0;
  virtual void* visit_let(string name, Expr *rhs, Expr* body) = 0;
  ....
};
```

visitor Pattern

```
class Expr {
  virtual void* visit(ExprVisitor *visitor) = 0;
};
```

Subclass to implement an operation

```
class ExprVisitor {
  virtual void* visit_num(int rep) = 0;
  virtual void* visit_add(Expr *lhs, Expr *rhs) = 0;
  virtual void* visit_let(string name, Expr *rhs, Expr* body) = 0;
  ....
};
```

visitor Pattern

```
class Expr {
  virtual void* visit(ExprVisitor *visitor) = 0;
};
```

One method per variant

```
class ExprVisitor {
  virtual void* visit_num(int rep) = 0;
  virtual void* visit_add(Expr *lhs, Expr *rhs) = 0;
  virtual void* visit_let(string name, Expr *rhs, Expr* body) = 0;
  ....
};
```

visitor Pattern

```
class Expr {
  virtual void* visit(ExprVisitor *visitor) = 0;
};
```

Method receives fields of variant

```
class ExprVisitor {
  virtual void* visit_num(int rep) = 0;
  virtual void* visit_add(Expr *lhs, Expr *rhs) = 0;
  virtual void* visit_let(string name, Expr *rhs, Expr* body) = 0;
  ....
};
```

visitor Pattern

```
class NumExpr {
  int rep;
  void* visit(ExprVisitor *visitor) {
    return visitor->visit_num(rep);
  }
};
```

```
class AddExpr {
  Expr *lhs;
  Expr *rhs;
  void* visit(ExprVisitor *visitor) {
    return visitor->visit_add(lhs, rhs);
  }
};
```

...

visitor Pattern

```
class NumExpr {
  int rep;
  void* visit(ExprVisitor *visitor) {
    return visitor->visit_num(rep);
  }
};
```

Implement `visit` once and for all

```
class AddExpr {
  Expr *lhs;
  Expr *rhs;
  void* visit(ExprVisitor *visitor) {
    return visitor->visit_add(lhs, rhs);
  }
};
```

...

visitor Pattern

```
class InterpVisitor : public ExprVisitor {

  void* visit_num(int rep) { return new NumVal(rep); }

  void* visit_add(Expr *lhs, Expr *rhs) {
     Val *lhs_val = (Val *)lhs->visit(this);
     Val *rhs_val = (Val *)rhs->visit(this);
     return lhs_val->add_to(rhs_val);
  }

  ....
};
```

```
expr->visit(new InterpVisitor())
```

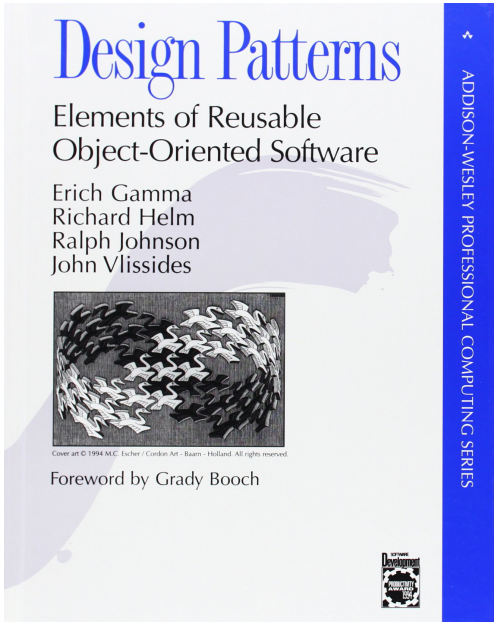
visitor Pattern

```
class OptimizeVisitor : public ExprVisitor {
  ....
};
```

```
class AnyVarsVisitor : public ExprVisitor {
  ....
};
```

...

Additional Resources



Wikipedia page

Pluralsight: “Design Patterns in Java”