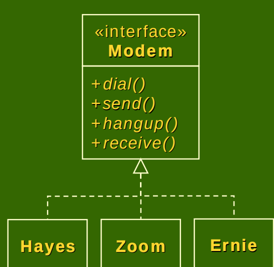
Design Patterns: Set 3

- The VISITOR family
  - VISITOR
  - ACYCLIC VISITOR
  - DECORATOR
  - EXTENSION OBJECT
- STATE

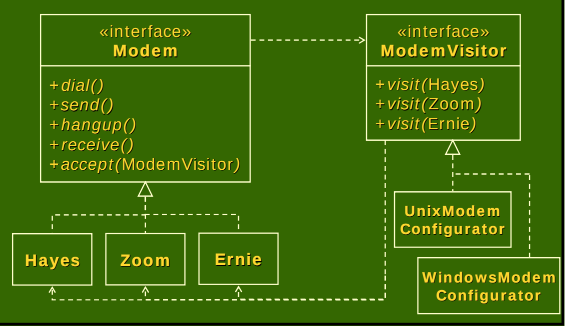
VISITOR

- The VISITOR family allows new methods to be added to existing hierarchies without modifying the hierarchies
- Every derivative of the visited hierarchy has a method in VISITOR
- Dual dispatch: two polymorphic dispatches

Example: Modem Hierarchy



Example: Modem Hierarchy (cont'd)



Example: Modem Hierarchy (cont'd)

```
public interface Modem {
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char receive();
    public void accept(ModemVisitor v);
}

public interface ModemVisitor {
    public void visit(HayesModem modem);
    public void visit(ZoomModem modem);
    public void visit(ErnieModem modem);
}
```

Example: Modem Hierarchy (cont'd)

```
public class HayesModem implements Modem {
    public void accept(ModemVisitor v) {
        v.visit(this);
    }
    /* rest of the implementation omitted */
}

public class UnixModemConfigurator implements ModemVisitor {
    public void visit(HayesModem m) {
        m.setConfigurationString("&s1=4&D=3");
    }
    public void visit(ZoomModem m) {
        m.setConfigurationValue(42);
    }
    public void visit(ErnieModem m) {
        m.setInternalPattern("C is too slow");
    }
}
```

Example: Modem Hierarchy (cont'd)

- To configure a modem for Unix, create an instance of the visitor and pass it to accept
- The appropriate derivative calls `visit(this)`
- New OS configuration can be added by adding a new derivative of the visitor

VISITOR as a Matrix

	Unix	Windows
Hayes	Initialization of Hayes in Unix	Initialization of Hayes in Windows
Zoom	Initialization of Zoom in Unix	Initialization of Zoom in Windows
Ernie	Initialization of Ernie in Unix	Initialization of Ernie in Windows

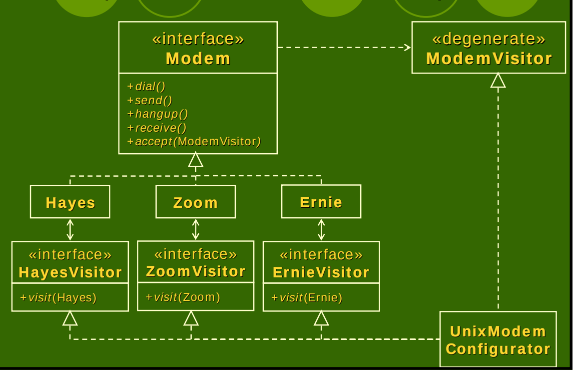
Observations

- In VISITOR
  - the visited hierarchy depends on the base class of the visitor hierarchy
  - the base class of the visitor hierarchy has a function for each derivative of the visited hierarchy
- A cycle of dependencies ties all the visited derivatives together
  - difficult to compile incrementally
  - difficult to add new derivatives of visited hierarchy
- Visitor work well if the hierarchy is not modified often

ACYCLIC VISITOR

- For a volatile hierarchy
  - new derivatives are created
  - quick compilation time is needed
- ACYCLIC VISITOR breaks the dependency cycle by making the visitor base class degenerate (i.e. it has no methods)

Example: Modem Hierarchy



Example: Modem Hierarcy (cont'd)

```
public interface Modem {
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char receive();
    public void accept(ModemVisitor v);
}

public interface ModemVisitor {
}
```

Example: Modem Hierarchy (cont'd)

```
public interface ErnieModemVisitor {
    public void visit(ErnieModem m);
}

public class ErnieModem implements Modem {
    public void accept(ModemVisitor v) {
        try {
            ErnieModemVisitor ev =
                                  (ErnieModemVisitor)v;
            ev.visit(this);
        } catch (ClassCastException e) { }
    }
    /* rest of the implementation omitted */
}
```

Example: Modem Hierarchy (cont'd)

```
public class UnixModemConfigurator implements
             ModemVisitor, HayesVisitor, ZoomVisitor,
                                         ErnieVisitor {
    public void visit(HayesModem m) {
        m.setConfigurationString("&s1=4&D=3");
    }
    public void visit(ZoomModem m) {
        m.setConfigurationValue(42);
    }
    public void visit(ErnieModem m) {
        m.setInternalPattern("C is too slow");
    }
}
```

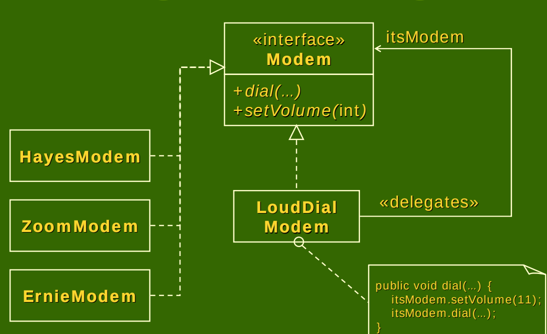
Observations

- Breaking the dependency cycle ⇒
  - easier to add visited derivatives
  - solution is much more complex
  - timing of the type casting is hard to characterize
- ACYCLIC VISITOR is like a sparse matrix
  - visitor classes do no have to implement visit functions for all visited derivatives

DECORATOR

- Allows attaching additional responsibilities to an object dynamically (i.e. at runtime)
- Provides a flexible alternative to subclassing for extending functionality
- Allows adding responsibilities to an object without adding methods to its interface

Example: Loud Dial Modem



Example: Loud Dial Modem (cont'd)

```
public interface Modem {
    public void dial String(pno);
    public void setSpeakerVolume(int volume);
}

public class HayesModem implements Modem {
    private String itsPhoneNumber;
    private int itsSpeakerVolume;

    public void dial(String pno) {
        itsPhoneNumber = pno;
    }

    public void setSpeakerVolume(int volume) {
        itsSpeakerVolume = volume;
}   }
```

Example: Loud Dial Modem (cont'd)

```
public class LoudDialModem implements Modem {
    private Modem itsModem;

    public LoudDialModem(Modem m) {
        itsModem = m;
    }

    public void dial(String pno) {
        itsModem.setSpeakerVolume(11);
        itsModem.dial(pno);
    }

    public void setSpeakerVolume(int volume) {
        itsModem.setSpeakerVolume(volume);
}   }
```

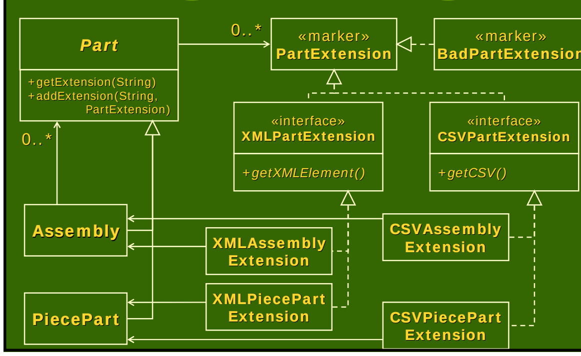
Observations

- Multiple decorators: base class decorator
  - supplies the delegation code
  - actual decorators derive from the base class and override only those methods they need
- Cf.
  - Java I/O streams:
    ```
    BufferedReader keyboard =
        new BufferedReader(
        new InputStreamReader(System.in));
    ```
  - `javax.swing.JScrollPane`

EXTENSION OBJECT

- More complex than VISITOR but more powerful
- Each object in the hierarchy
  - maintains a list of special extension objects
  - provides a method that allows the extension object to be looked up by name
- Extension object provides methods that manipulate the original hierarchy object

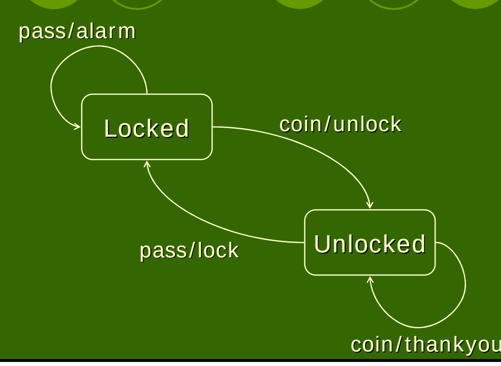
Example: Bill-of-Materials



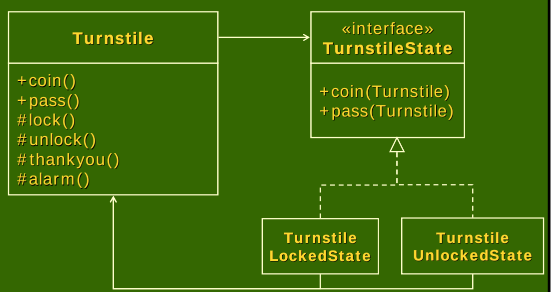
STATE

- Allows an object to alter its behaviour when its internal state changes
  - the object will appear to change its class
- Typically used to change the behaviour according to a state transition diagram
- Other implementations for an FSM
  - nested switch/case statement
  - transition table

Example: Turnstile FSM



Example: Turnstile



Example: Turnstile (cont'd)

```
public interface TurnstileState {
    public void coin(Turnstile t);
    public void pass(Turnstile t);
}

public class LockedTurnstileState implements TurnstileState {
    public void coin(Turnstile t) {
        t.setUnlocked();
        t.unlock();
    }

    public void pass(Turnstile t) { t.alarm(); }
}

public class UnlockedTurnstileState implements TurnstileState {
    public void coin(Turnstile t) { t.thankyou(); }

    public void pass(Turnstile t) {
        t.setLocked();
        t.lock();
}   }
```

Example: Turnstile (cont'd)

```
public class Turnstile {
    private static TurnstileState lockedState = new LockedTurnstileState();
    private static TurnstileState unlockedState = new UnlockedTurnstileState();

    private TurnstileController turnstileController;
    private TurnstileState state = lockedState;

    public Turnstile(TurnstileController action) {
        turnstileController = action;
    }

    public void coin()           { state.coin(this); }
    public void pass()           { state.pass(this); }
    public void setLocked()      { state = lockedState; }
    public void setUnLocked()    { state = unlockedState; }
    public boolean isLocked()    { return state == lockedState; }
    public boolean isUnlocked()  { return state == unlockedState; }
    protected void thankyou()    { turnstileController.thankyou(); }
    protected void alarm()       { turnstileController.alarm(); }
    protected void lock()        { turnstileController.lock(); }
    protected void unlock()      { turnstileController.unlock(); }
}
```

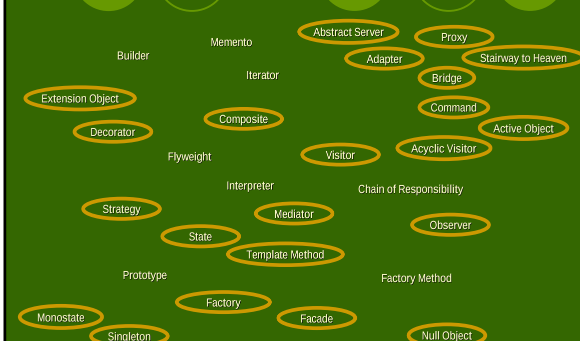
STATE vs. STRATEGY

- Common
  - context class
  - delegation to a polymorphic base class that has several derivatives
- Difference
  - STATE: derivatives hold a reference back to the context class
  - STRATEGY: no such constraint or intent
- All instances of STATE are also instances of STRATEGY

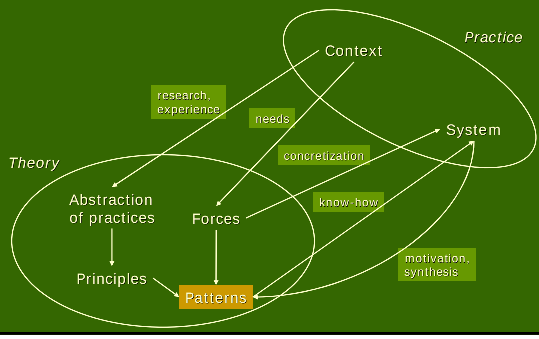
Observations

- Very strong separation between actions and the logic of state machine
  - action in the context class
  - logic distributed through the derivatives of the state class
- Simple to change one without affecting the other
  - reuse the context class with different state logic
  - create subclasses of context class that modify the action without affecting the logic
- Costs
  - writing state derivatives is tedious
  - the logic is distributed, no single place to see it all

Design Patterns (revisited)



Principles, Patterns, and Practices



Examinations

- Examination dates
  - May 15, 2006
  - June 20, 2006
  - September, 2006 (exact date to be announced)
- Confirm the times and places at http://www.it.utu.fi/opetus/tentit/
- If you are not a student of University of Turku, you must register to receive the credits
- Remember to enrol in time!

Examinations (cont'd)

- Questions:
  - based on both lectures and course textbook
  - three questions, à 10 points
  - to pass the examination, at least 15 points (50%) are required
  - questions are in English, but you can answer in English or in Finnish
- Note: You can use the textbook in the examination

Fin.