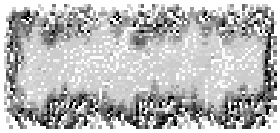
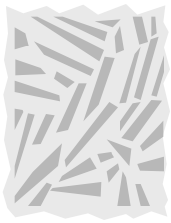
Reuse tutorial, P. Devanbu (Design Patterns Section)

1 1

Outline

1. Patterns---motivation
2. Several Popular patterns
3. Wrap up on Patterns





1 1

2 2

Why Design Patterns

It's the best way to study object-oriented programming!

Object-oriented programming languages are difficult to use properly (specially C++)

- Many features to master.
- Many different ways of design alternatives.
- Mistakes in OO design persist for the system's lifetime.

Analogy:

What's the best way to master a natural language (e.g, English, Tamil, Italian..)

e.g., "jeeze, you stink!".

Vs.:

"Bathe thyself, thou infections, flea-bitten, wart-hog!"

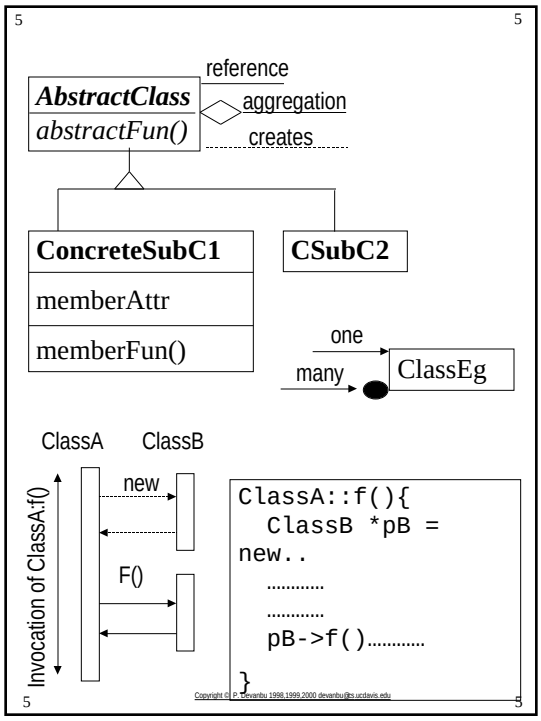
2 2

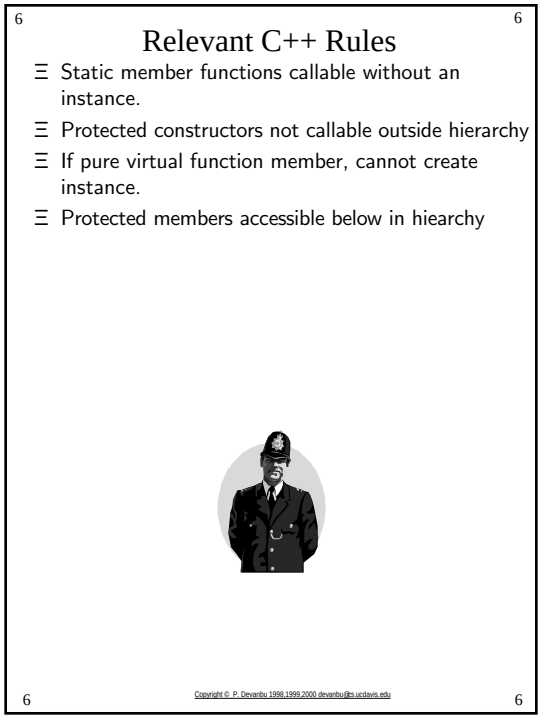
Why Design Patterns?

- **Object-Oriented Design is Hard** Creating a flexible reusable, easy to understand, robust design is nearly impossible the first time.
- **So don't do it for the first time!** Experienced designers have a bag of tricks that they trot out, when they recognize a problem that fits.
- **Analogy: Novels/Plays.** Most novels and plays don't start with a completely blank slate. Typical templates: *tragically flawed heroine*, *heroine overcomes powerful villain with plain old chutzpah*, etc.
- **Other Analogies** Kitchen design, Haiku, etc. etc.

C++ BACKGROUND

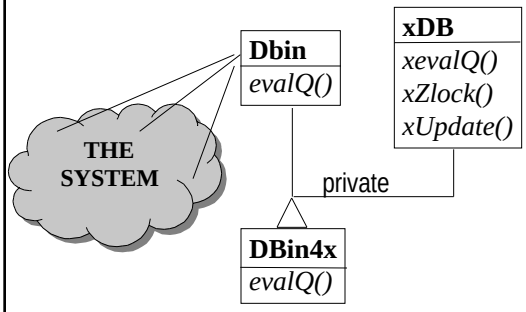
```
class Polygon {
public:
   virtual Polygon();
   virtual ~Polygon();
   virtual int getArea() =0;
   int getSides() return sides;
protected:
   void setSides(int n) ¶sides =n; ◇;
private:
   int sides
}
class Triangle : public Polygon ¶
public:
   Triangle←) ¶ ↙sides=3, ▷℘,
   int getArea () ¶ ◇;
private:
   Vector sideVector[3];
◇
class Pentagon : public Polygon ¶
public:
   Pentagon← ←¶ ↙sides=5, ℘,
   int getArea () ¶↙◇ ;
private:
```





Typical Design Problem

- **Problem:** Your system makes extensive use of a database subsystem. However, a) we want to be able to use different databases depending on whichever one is cheaper (abstraction) and b) we want programmers to only use a "least common denominator feature set" from the database market.



Adapter Discussion

Applicability Use the adapter pattern when:

1. An available implementation does not match the required interface
2. You want to create a reusable "wall" between a known subsystem and an unknown or unforeseeable set of implementations.

Participants These are the elements of an adapter pattern instance:

1. *Adaptee*: the "foreign" class being adapted.
2. *Target*: the interface the foreign class hides" behind.
3. *Adapter*: the implementation class that implements the *target* interface (public inheritance) using the methods of the *adaptee* class (private inheritance).
4. *Client(s)*: classes using the services available via *target*

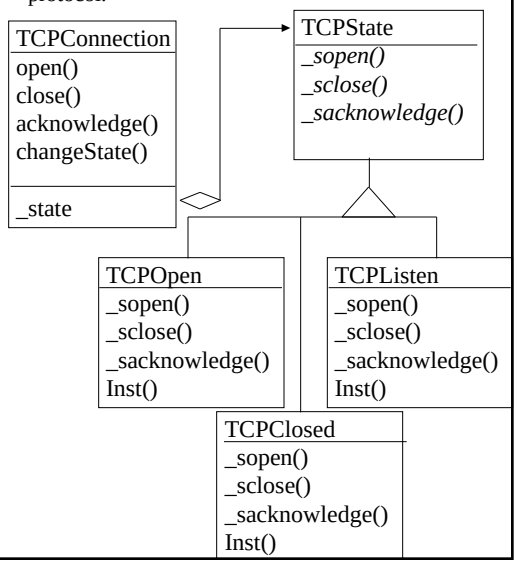
Collaborations This is how it works:
Clients call the *target* interface, whence (via the virtual function) calls are dispatched to the *adapter* class, which in turn calls the member functions of the *adaptee*.

Consequences There are some trade-offs.
1. Runtime Overhead: virtual function despatch + function call $\leftarrow client \rightarrow adapter \rightarrow adaptee$)

2. Won't track *adaptee*'s evolution, i.e, over-ridden and newly introduced methods in *adaptee* not available.

State Pattern

Problem: An object should change behaviour when it's internal state changes; the object should appear to be instance of different classes. (*e.g.*) states in a protocol.



10 10

Implementations

```
TCPConnection::open()
   {
   _state-
   >_sopen(this);
}

TCPConnection::close()
   {
 _state-
   >_sclose(this);
}

TCPConnection::
   changestate(
   TCPConnection
   *newS) {
   _state=newS;
}
```

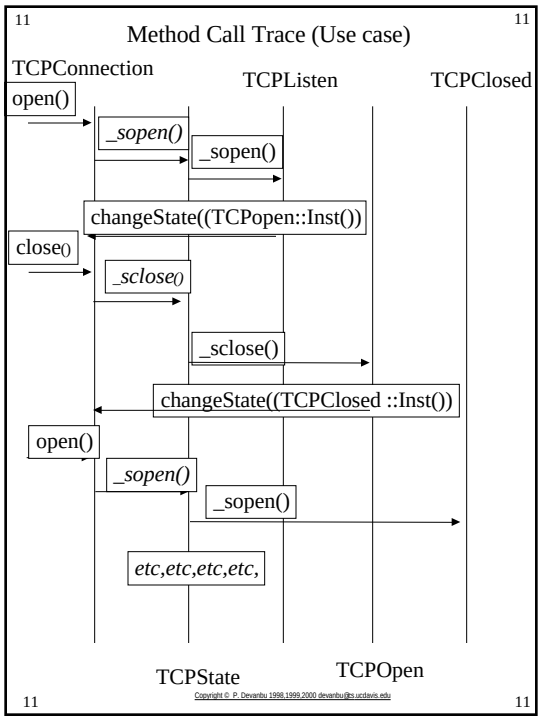
```
TCPListen::_sopen(
   TCPConnection *tC)
   {
   ...
   ...
   tC->changestate(
        TCPOpen::Inst()
               );
   ..

}

TCPOpen::_sopen(
   TCPConnection *tC)
   {
   ...
   ...
   error(….)
   ..

}
```

10 10



12 12

The ***State*** Pattern

Applicability Use the ***State*** pattern when:

- an objects behaviour depends on state, and it must change it's behaviour at run time, based on state.
- Large switch statements (on input actions) for each state. This "objectifies" the behaviour for each state.

Participants:

- *Context (*`TCPConnection`*)* used by (protocol) clients
- *State (*`TCPState`*)* enscapsulates state behaviour.
- *ConcreteState (*`TCPOpen`*)* implements a specific state's behaviour.

Collaboration: *Context* delegates stimuli via *state* to specific *ConcreteState* implementation. *Context* passes itself as argument for further applicable *State* changes. ***State*** is a ***Singleton*** pattern, usually!

Consequences:

Helps catch certain types of errors! (what??) but not others.

State objects are shared; can't have instance values that are client-dependent.

Another example: drawing tools!

12 12

Pattern Categories

Creational:

For solving problems related to creating object instances (e.g., Singleton, Builder, Factory…)

Structural:

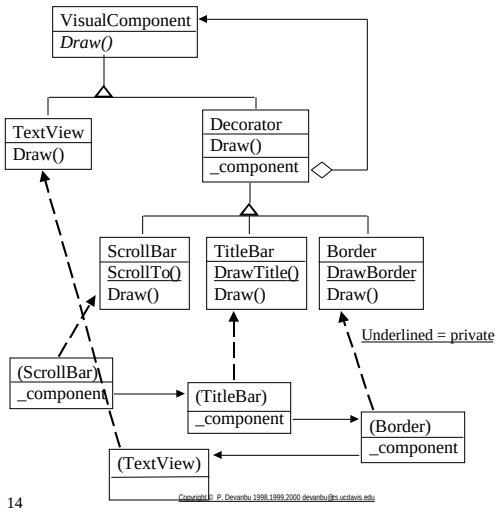
For solving problems relating structural relationships between parts of an OO system (e.g., Adapter, Façade, Bridge, Composite, etc…)

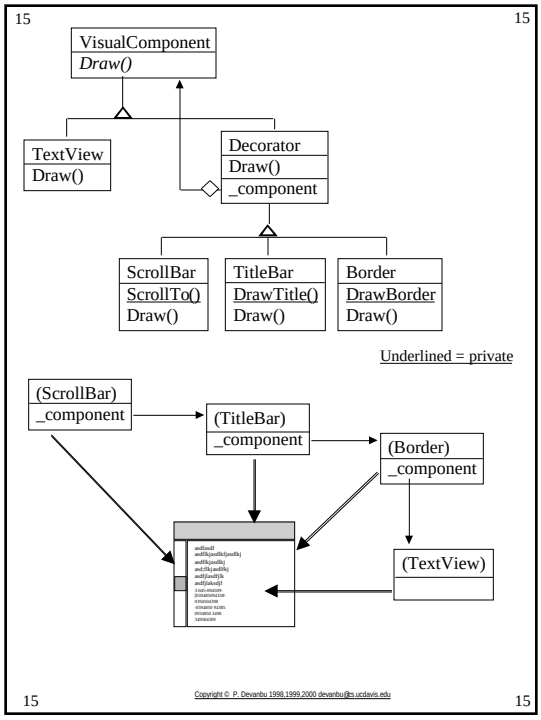
Behavioral:

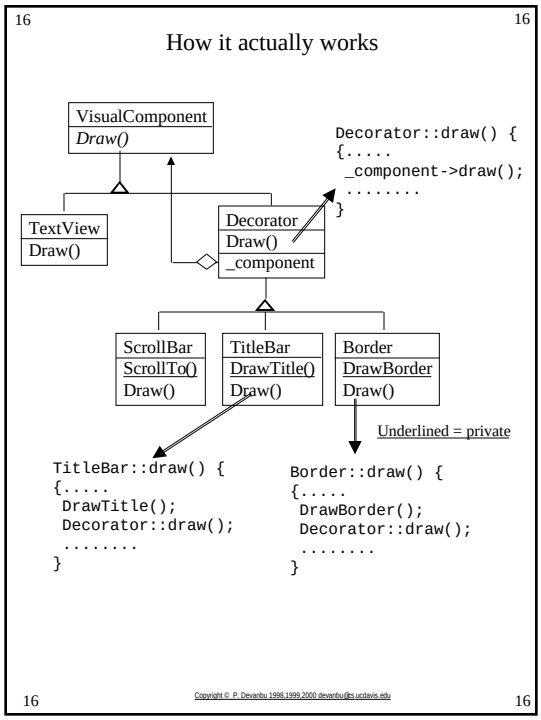
For design problems that require a particular type of behavioral or operational abstractions. (e.g., State, Decorator, Observer, Visitor etc…)

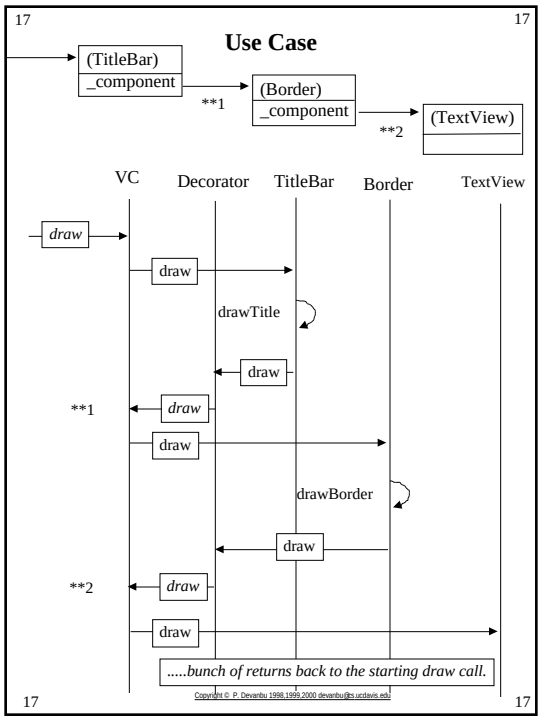
The ***Decorator*** Pattern

Add additional functionality to objects dynamically. E.g., take a window and dynamically add borders, scrolling, titlebar etc to it. (without multiple inheritance etc).









18 18

Decorator Pattern

Applicability Use the ***Decorator*** pattern when:

- Add/remove functionality to individual instances, without affecting other instances.
- Subtyping is impractical: too many combinations.

Participants:

- *Component* (`VisualComponent`) defines the interface to which functionality is added.
- *ConcreteComponent* (`TextView`) a class to which additional functionality can be added
- *Decorator* (`Decorator`) maintain a reference to a "decorated" component object and defines an interface identical to *Component.*
- *ConcreteDecorator* (`Border`) adds function to *Component.*

Collaboration: *Decorator* forwards requests to it's *Component.* May also do some function before/after forwarding request.

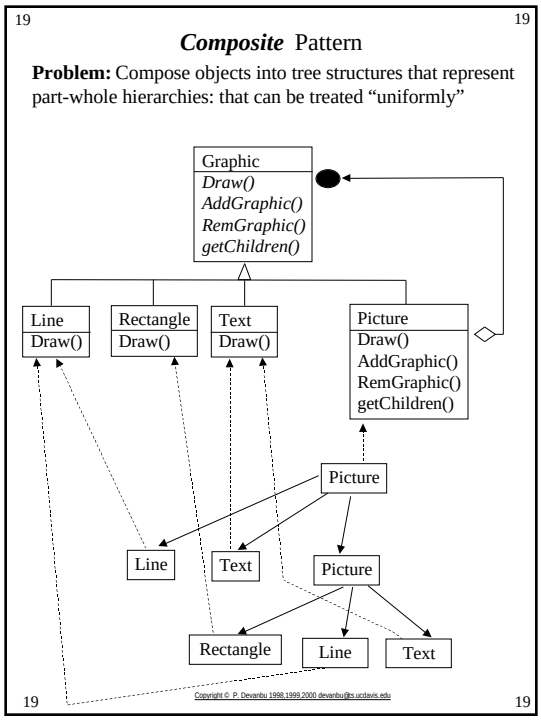
Consequences:

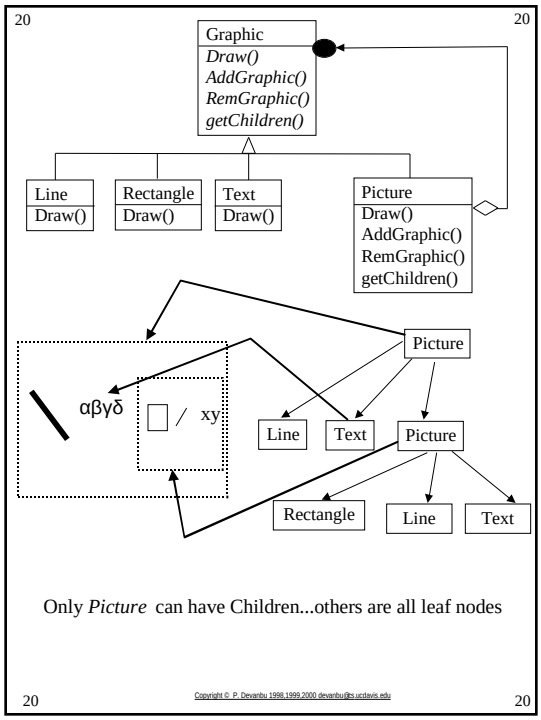
Avoids multiple inheritances, class proliferation one for each combination..

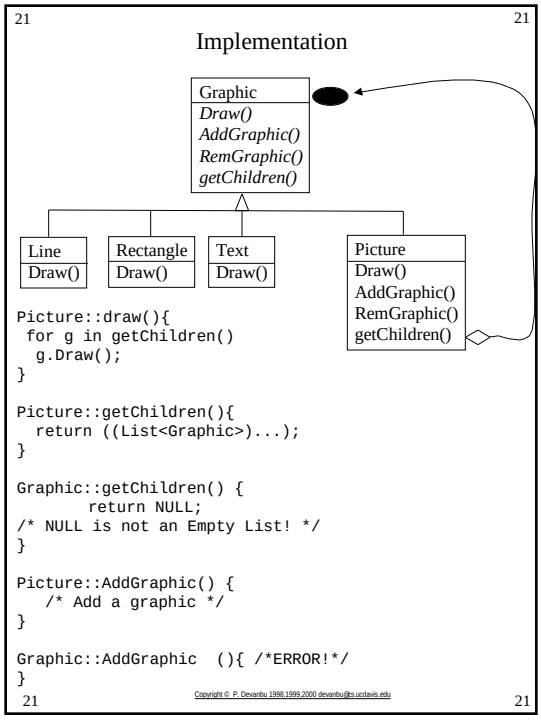
However, instance proliferation !!!

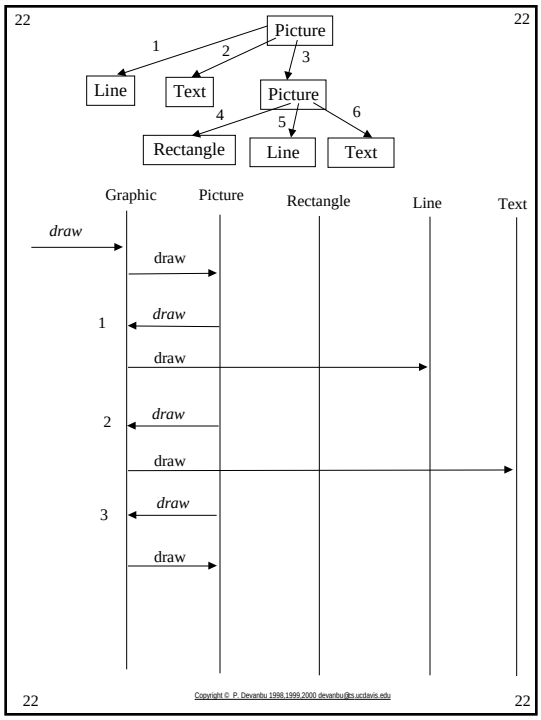
Example: Travel Prefences: seat, diet, smoking, status....

18 18









Composite Pattern

Applicability Use the ***Composite*** pattern:

- to represent part/whole hiearchies.
- When both parts and wholes need to get some uniform treatment.

Participants:

- *Component* (`Graphic`) defines the interface for all objects in a part/whole hierarchy
- *Composite* (`Picture`) a class that has children in the hierarchy; defines the behaviour, and aggregates components.
- *Leaf* (`Text,Line..`)
- *Client* (`Not Shown`) Uses *Component's* interface.

Collaboration: *Client* uses *Component's interface* to invoke methods on the entire hierarchy. If *Component* is a *Leaf*, executed directly. Otherwise the method is passed off to children.

Consequences:

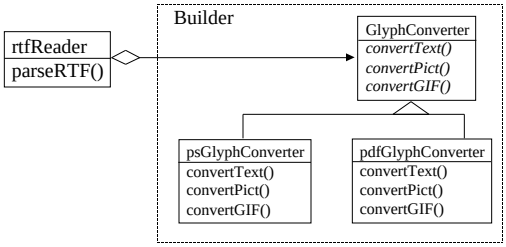
Client doesn't have to worry about iteration!

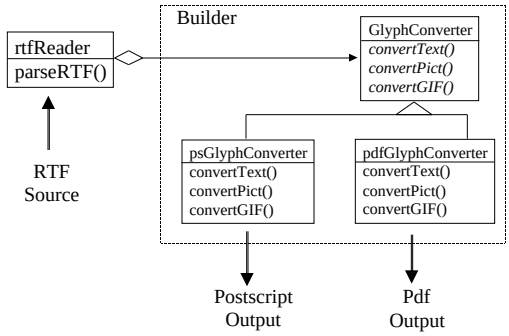
Can evolve by adding new types of leaves.

Another example of a hierachy: (Hint: "physical world?")

Builder Pattern

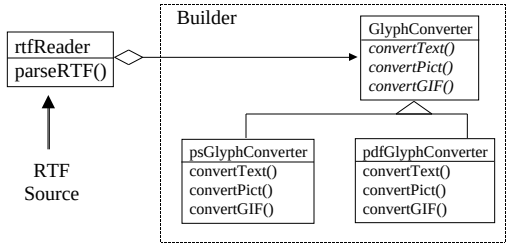
Problem: Separate the algorithm for constructing a complex object from the actual representation of the object. (Example: PostScript or PDF from RTF).





Rather than output, could also create a different internal (datastructure) representation.

Implementation




```
rtfReader::parseRTF(builder){
 while(a = getNextRTFitem)
   if (isText(a))
      builder->convertText(a);
   else if(isPict(a))
      builder->convertPict(a);
   else if(isGIF(a))
      builder->convertGIF(a);
   else /* error */
}
```

```
rtfReader myRtf;
/* For Postscript: */
myRtf.parseRTF(new psGlyphConverter)
/* For PDF: */
myRtf.parseRTF(new pdfGlyphConverter)
```

Builder pattern

Applicability Use the ***Builder*** pattern:

- when the algorithm for construction is separable from the representation.
- Different representations could be desirable.

Participants:

- *Director* (`rtfReader`) Has the algorithm for building the representation
- *Builder* (`glyphConverter`) an interface for creating parts of a representation
- *ConcreteBuilder* (`psConverter,pdfConverter`) implements *Builder* ; creates specific rep. of particular parts.
- *Product*(`pdf, ps etc`)The different representations.

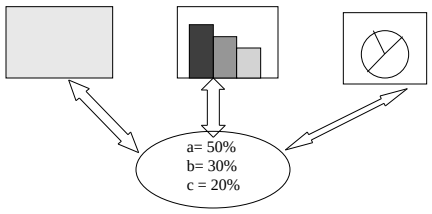
Collaboration: A Director, given a builder for a specific rep., calls the builder to construct different pieces of a complex representation..

Consequences:

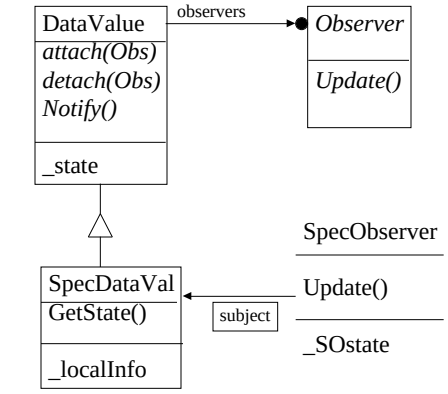
Can vary the representation without varying the construction process!

Example: producing different things from a parse tree: control flow graph, machine code, etc.

Observer Pattern.



Observer Pattern




```
forall o in observers  {
        o->Update()
}
```

```
_Sostate =
  subject->GetState()
```


Applicability Use the observer pattern when:
1. When a monitored object changes state, a bunch of interested parties" needs to get notified.
2. The monitored object can keep track of the interested parties.

Participants: These are the elements of an observer pattern instance:
1. *Subject:* The object being monitored (the Room). Provides interfaces for attaching & detaching observers.
2. *Observers:* The monitoring entities. Provides the update interface.
3. *ConcreteSubject:* An implementation of a subject. Stores info of interest to observer.
4. *ConcreteObserver(s):* Implements the update interface, and has state info that is consistent with subjects (via the update).

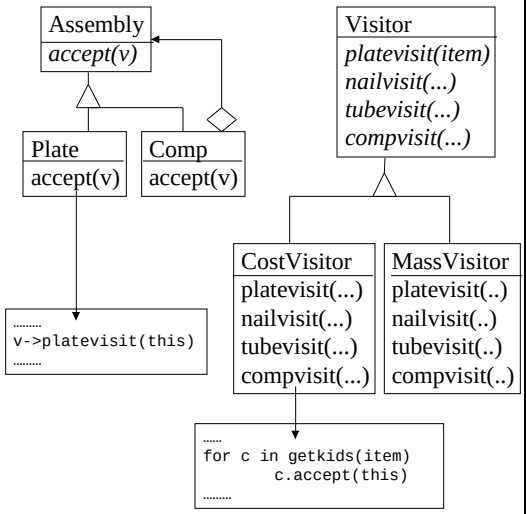
Collaborations This is how it works: *ConcreteSubject* notifies the *observers* by calling the `notify` member function of Subject. Then *ConcreteObserver* updates its state by calling member `getInfo` of *ConcreteSubject.*

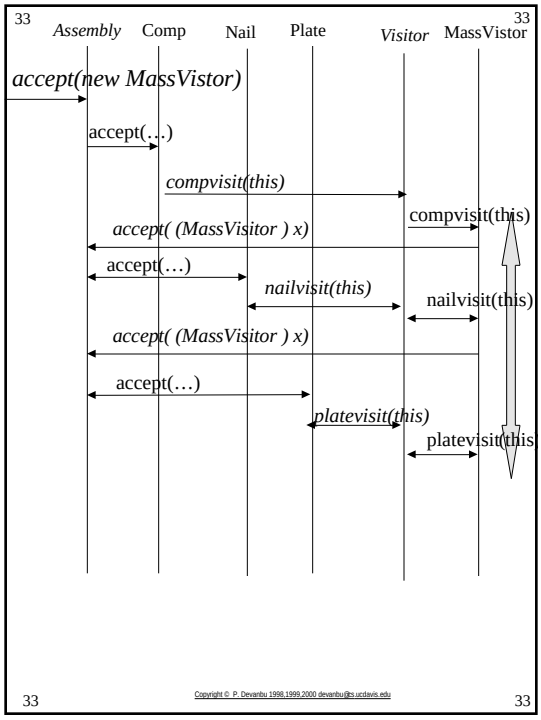
Consequences Some considerations:
• No way to pre-judge the impact of a `notify` call. *ConcreteSubject* may get held up!
• *ConcreteSubject* should be in a consistent, stable state, before calling `notify`; otherwise *ConcreteObservers* may get confused.

A CAD/CAM Application

- Consider a composite pattern instantiated for a CAD/CAM application and consider how we *singly* dispatch a draw or volume operation: with:
  - Base class: Assembly,
  - Composite: CompoundObject (Comp), and
  - Single Objects: plate, screw, tube, bolt etc.
- Two main subsystems in the CAD/CAM application: an *authoring* or *design tool*, and an *analysis/computation/simulation* side.
  - Which subsystems depend on the part hierarchy?
  - Where is the code for a specific analysis function? the required data items?
  - What does the analysis/computation/simulation side do (the general structure of these operations?)
  - What are the dependencies? (cf: re-compilation?)
  - Where should the analysis methods be placed?
- Can we separate out two "independent hierarchies" so that changes to the analysis side don't affect the part hiearchy, and therefore can be evolved independently, and encapsulated separately?
- *Multiple* Dispatch!

Visitor Pattern





34 34

Applicability Use the ***Visitor*** pattern:

- When there is 1) an object structure with many subclasses, and 2) operations with sub-class specific functions.
- object structure changes less often than operations, and operations should be grouped.

Participants:

- *Element* (`Assembly`) Abstract element that accepts operations as *Visitors.*
- *Visitor* (`Visitor`) an interface for operations of visitors (one for each *ConcreteElement*
- *ConcreteVisitor* (`CostVisitor`) implements all operations of *Visitor*.
- *ConcreteElement*(`Nail`)Implements accept by delegating to member virtual function of *Visitor.*

Collaboration: An operation is initiated by invoking accept on the top-level element with a visitor object as argument…from then on visitor object is responsible for computing the operation.

Consequences:

Visitors make adding new operations easy.
Visitor encapsulates related operations and data values.
Adding new ConcreteElement is **hard.**
Note: Need double dispatch via two virtual functions!!

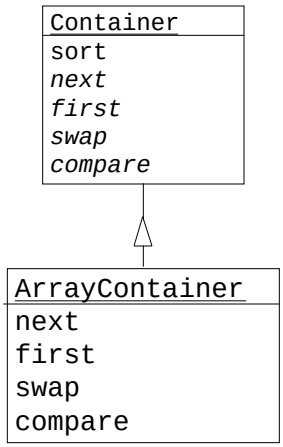
Example: producing different things from a parse tree, such as control flow, type checking, data flow problem solutions, etc.

34 34

35 35

Template Method

Assume that you have an abstract algorithm which depends upon some primitive functions that may vary with the exact application. How can we implement and distribute this algorithm?



35 35

36 36

Applicability Use the ***Template Method*** pattern:

•When there is 1) there is an abstract algorithm that can be reused and 2) when intellectual property may be at stake 3) performance could be compromised.

•Algorithm changes less often than the primitive operations in use.

Participants: Yada, yada.

Collaboration: The template method invokes the primitive operations as virtual functions, which are implemented in the derived class

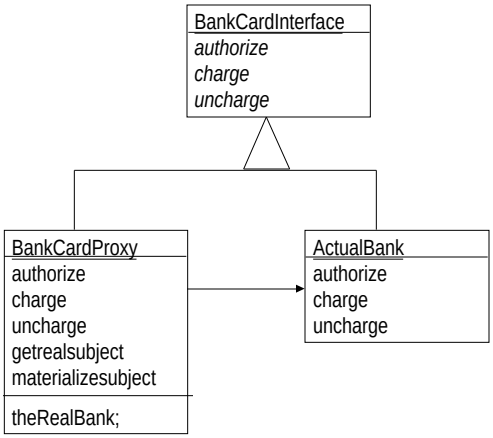
Consequences:

Easy to create several instantiations of an abstract algorithm.
The code for the abstract algorithm is hidden from prying eyes (and tools)
So what's the downside?

36 36

Virtual Proxy

The actual implementation of an object is sitting somewhere, possibly in different places, sometimes cached, etc. How can we allow the client to remain ignorant of how to actually find it, but still use it normally?




```
Int validate(….) {
    return (getrealsubject()->validate());
}

/* All requests look exactly the same as above */
BankCardInterface *getrealsubject (){
  if(theRealBank == 0) {
    theRealBank = materializesubject();
  }
  return (theRealBank);
}
```

Conclusion

- **How do patterns help a designer?**
- **What are the major types of design patterns:**
  - Behavioural Patterns
  - Creational Patterns
  - Structural Patterns
- **How can design patterns help with old code or libraries, frameworks, etc ?**
- **How are patterns described?**
- **What if someone says: "Eureka! I've invented a pattern!"**