Inheritance and Dynamic Binding: idioms and common patterns

Steven Zeil

September 19, 2013

Contents

Inheritance, subtyping, and dynamic binding are powerful tools. And, like any new tool, it's not always obvious how and when to use them. The notion that inheritance is the programming language realization of a generalization/specialization relationship helps. It also may help to think of it as the "is a" relationship. When we see these kinds of relationships existing among our classes (in analysis), we should at least consider the possibility that we should be using inheritance in our programs.

But not *every* situation where we see a generalization or as-a relationship in analysis deserves to be programmed as inheritance. There's an old saying, "When the only tool you own is a hammer, every problem begins to resemble a nail." We need to avoid trying to distort our designs into an inheritance mold if it does not actually fit.

Fortunately, you are not the first programmer to be faced with the problem of when to use inheritance. There are selected idioms that have evolved in practice that are worth knowing, both as examples of good ways to use inheritance and as counter-examples of ways to abuse inheritance.

1 Inheritance Captures Variant Behavior

Inheritance Captures Variant Behavior

Why bother with inheritance & dynamic binding at all? Because it offers a convenient mechanism for capturing *variant behaviors* among different groups of objects. For example, consider the idea of values stored in a spreadsheet. A Spreadsheet can contain many different kinds of values. Some contain numbers, other strings, others might contain dates, monetary amounts, or other types of data. Typically there is a special kind of "error value" that is stored in a cell when we have formulas that do something "illegal", such as dividing a number by zero or adding a number to a string.

```
string render(int width)
Value* clone()
```

There are some things we expect to be able to do to any value. For values to be useful in a spreadsheet, we must be able to *render* them in a cell whose current column is of some finite width. The render function, in general, could be quite elaborate. For example, given a lot of room, we might be able to render a number as 12509998.0. If we sharing the column containing that cell, though, we might need to render it as 12509998. Shrink it some more and we might need to render it as 1.25E06. Shrink a little more, and we might need to say 1.3E06, then 1E06. Shrink the available space small enough, and a spreadsheet will typically render a number as "***".

There may be other operations required of all values as well, such as `clone()`.

Now, the information we have provided so far is general to all values. But to actually store a numeric value, we need a data member to hold a number (and a function via which we can retrieve it, though we'll ignore that for the moment). Similarly, we can expect that, to store a string value, we would need a data member to store the string.

We will assume, therefore, that

- Numeric values have an attribute d of type `double`
- String values have an attribute s of type `string`

....

A pre-OO Approach to Variant behavior

```
class Value {
public:
  enum ValueKind {Numeric, String, Error};
  Value (double dv);
  Value (string sv);
  Value ();

  ValueKind kind;
  double d;
  string s;

  string render(int width) const;
}
```

Now, if we had never heard of inheritance (of if we were programming C, FORTRAN, Pascal, or any of the other pre-OO languages, we might have come up with an ADT something like this.\footnote{Actually, there are ways to make this more memory-efficient in those languages using constructs called "unions" or "variant records", but they would not change the point of this example. }

- Any given value will presumably have something useful stored in `d` *or* in `s`, but not in both.
  - In the case of an error value, it may not have anything useful in either one.

....

A pre-OO Approach to Variant behavior

```
class Value {
public:
  enum ValueKind {Numeric, String, Error};
  Value (double dv);
  Value (string sv);
  Value ();

  ValueKind kind;
  double d;
  string s;

  string render(int width) const;
}
```

Now, if we had never heard of inheritance (of if we were programming C, FORTRAN, Pascal, or any of the other pre-OO languages, we might have come up with an ADT something like this.\footnote{Actually, there are ways to make this more memory-efficient in those languages using constructs called “unions” or “variant records”, but they would not change the point of this example. }

- `kind` is a "control" data field.
  - It does not actually store useful data of its own.
  - It’s there to tell us which of the variants of value we have stored in any particular value.
  - We’ll use this mainly so that we can branch to code appropriate to that variant.

. . . .

Multi-Way Branching

```
string Value::render(int width) const {
  switch (kind) {
    case Numeric: {
      string buffer;
      ostringstream out (buffer);
      out << dv.d;
      return buffer.substr(0,width);
      }
    case String:
      return sv.s.substr(0.width);
    case Error:
      return string("** error **").substr(0.width);
  }
}
```

For example, to write the body for the `render` function, we would probably do something like this. First, we do a multi-way branch on the `kind` to get to the appropriate code for a numeric, string, or error value. Then in each branch we use a different technique to render the value as a string, then chop the string to the desired width.

- We can expect to see similar multi-way branches used to implement `clone()` or just about every other function we might write for manipulating `Value`s.

. . . .

Variant Behavior under OO

```
class Value {
public:
  virtual string render(int width) const;
};

class NumericValue: public Value {
public:
  NumericValue (double dv);

  double d;
  string render(int width) const;
};

class StringValue: public Value {
public:
  StringValue (string sv);

  string s;

  string render(int width) const;
};

class ErrorValue: public Value {
  ErrorValue ();

  string render(int width) const;
};
```

Now, compare that with the OO approach.

- We represent each variant with a distinct subclass.
  - Only objects of the `NumericValue` class get the `d` data member.
  - Only objects of the `StringValue` class get the `s` data member.
  - None of them get the `kind` data member.
- This saves memory when we have large numbers of values floating about (as in a very large spreadsheet).
- But what's more important is how it affects the code we write for manipulating `Value`s.

. . . .

Variants are Separated

```
string NumericValue::render(int width) const
{
  string buffer;
  ostringstream out (buffer);
  out << d;
  return buffer.substr(0,width);
}


string StringValue::render(int width) const {
  return s.substr(0.width);
}

string ErrorValue::render(int width) const {
  return string("** error **").substr(0.width);
}
```

Here's the OO take on the same `render` function.

- None of the details of how to render specific kinds of value have been changed.
- But we have repackaged that code into subclass-specific bodies.
  - The "variants" are now separate. In a team environment, different people can work on different variants separately.
  - Each subclass operation is simpler.
  - Most important of all, new kinds of values can be added without changing or recompiling the code of the earlier kinds of values.

. . . .

Suppose, for example, that we later decide to add to our spreadsheet the ability to manipulate dates. In the pre-OO style, we would need to look through all our code for any place we had a multi-way branch on the `kind`, then add a new branch with code to handle a date. Heaven help us if we miss one of those branches! If we're *lucky*, our code will crash during testing when we hit that branch with a date value. If we're unlucky, the crash occurs just as we're demo'ing the spreadsheet to upper management.

By contrast, to add dates in the OO style, we declare `DateValue`, a new subclass of `Value`. We write the code to render date values (which we would have to do in any case) and put it in its own `DateValue::render` body. Link it with the existing code, and we're ready to go. None of the existing code had to be touched at all.

Summary

We use inheritance in our programming designs whenever we find ourselves looking at objects that

- come in different varieties,
- each of which has its own slightly different way of doing the "same thing".

Conversely, if we *don't* see that kind of variant behavior, we probably have no need for inheritance.

- Should we have, for example, a subclass for `AlphaNumericStringValues`?
  - Unlikely – I can't think of a way in which the behavior of an alphanumeric string would vary, in the spreadsheet world, from that or an ordinary string.

. . . .

2 Using Inheritance

Using Inheritance

We'll look at 3 idioms describing good ways to use inheritance.

- Specialization
- Specification
- Extension

. . . .

2.1 Specialization

Specialization

When inheritance is used for *specialization*,

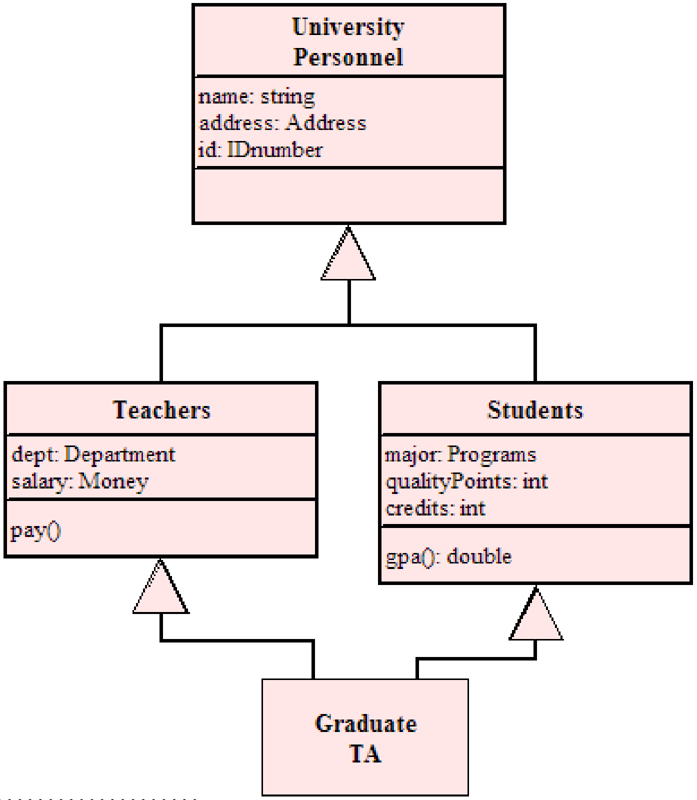
- The new class is a specialized form of the parent class
- but satisfies the specification of the parent in every respect.

  The new class may therefore be substituted for a value of the parent.

This is, in many ways, the “classical” view of inheritance.

. . . .

Recognizing Specialization



- A hallmark of specialization is that the base class
  - has objects of its own
  - may be processed in some applications without any contributions from the subclasses.

. . . .

Consider this hierarchy. It's quite likely that there are University Personnel who are neither teachers nor students (administrators, librarians, staff, etc.). As such, it's entirely likely that some applications (e.g., printing a University telephone directory) will work purely from the attributes and functions common to all University Personnel.

At the same time, there are variant behaviors among the subclasses that would prove useful in some applications. For example, Students have grade point averages (and probably other grade info not shown in this diagram) that would permit us to print grade reports and transcripts.

2.2 Specification

Specification

Inheritance for *specification*[1] takes place when

- a parent class specifies a common interface for all children
- but does not itself implement the behavior
  - Sometimes called the "shared protocol" pattern

. . . .

Defining Protocols

A *protocol* is a set of messages (functions) that can be used together to accomplish some desired task.

- The superclass defines the protocol.
- The subclasses implement the messages of the protocol in their own manner.
- Application code invokes the messages of the protocol without worrying about the individual methods.

. . . .

Recognizing the Specification Idiom of Inheritance

- Base class typically has no instances (objects)
  - Only objects are actually instances of subclasses
- Some operations may not even be implementable in the general base class

. . . .

[1] Yeah, it's a real pain that "specialization" and "specification" both look and sound so much alike. But I didn't make up these terms. And the words really don't mean anything like the same thing in their normal use, so just concentrate on what the words mean instead of what they sound like.

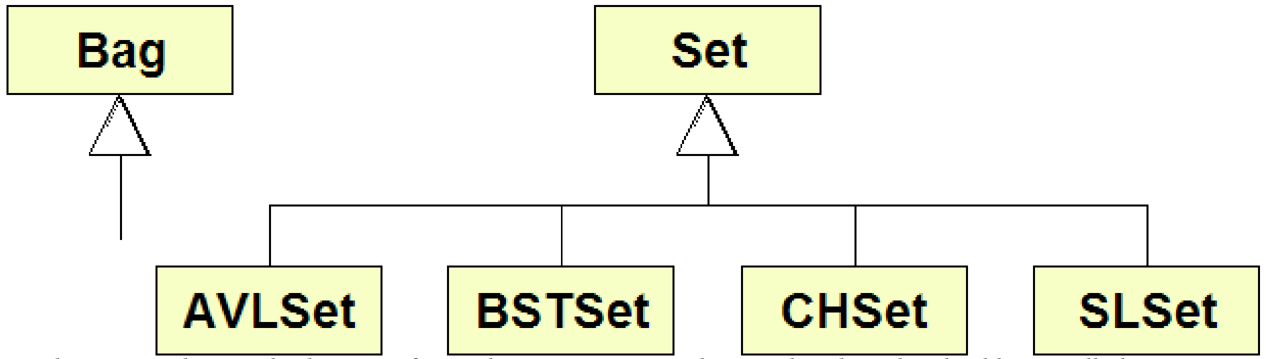
Example: Varying Data Structures

A common requirement in many libraries is to provide different data structures for the same abstraction.

- Allows application code writers to balance speed and storage requirements against expected size and usage patterns.
- Allows code in which the only mention of which data structure is being used comes when the actual objects are constructed

. . . .

libg++



For example, prior to the standardization of C++, the GNU g++ compiler was distributed with a library called `libg++`.

This library as extraordinary for the variety of implementations it provided for each major ADT. For example, the library declared a `Set` class that declared operations like adding an element to a set or checking the set to see if some value were a member of that set. However, the `Set` class itself contained no data structure that could actually hold any data elements.

Instead, a variety of subclasses of `Set` were provided. Each subclass implemented the required operations for being a Set, but provided its own data structure for storing and searching for the elements.

- `AVLSet` stored the data in AVL trees (a balanced binary tree)
- `BSTSet` stored the data in ordinary binary search trees

- CHSet stored the data in a conventional hash table
- SLSet stored the data in a singly-linked list

. . . .

In addition to these, there were subclasses for storing the elements in expandable arrays (similar to the std::vector), in dynamically expandable hash tables, in skip lists, etc.

The idea was that each of these data structures offered a slightly different trade off of speed versus storage, sometimes depending upon the size of the sets being manipulated.

Working with a Specialized Protocol

```
void generateSet (int numElements, Set& s, int *expected)
{
  int elements[MaxSetElement];


  for (int i = 0; i < MaxSetElement; i++)
    {
      elements[i] = i;
      expected[i] = 0;
    }


  // Now scramble the ordering of the elements array
  for (i = 0; i < MaxSetElement; i++)
    {
      int j = rand(MaxSetElement);
      int t = elements[i];
      elements[i] = elements[j];
      elements[j] = t;
    }
  // Insert the first numElements values into s
  s.clear();
  for (i = 0; i < numElements; i++)
    {
      s.add(elements[i]);
      expected[elements[i]] = 1;
    }
}
```

A programmer could write code, like the code shown here, that could work an *any* set.

- It is only in the code that *declared* a new set variable that an actual choice would have to be made.

. . . .

This meant that it was quite easy to write an application using one kind of set, measure the speed and storage performance, and then to change to a different variant of `Set` that would be a better match for the desired performance of the application.

`libg++` provided similar options, not only for sets, but also for bags (sets that permit duplicates, a.k.a. “multi-sets”), maps, and all manner of other container ADTs.

2.2.1 Abstract Base Classes

Adding to a Set Subclass

If we are working with `libg++` This is OK:

```
void foo (Set& s, int x)
{
   s.add(x);
   cout << x << " has been added." << endl;
}

int main ()
{
   BSTSet s;
   foo (s, 23);
    ⋮
```

. . . .

Adding to a General Set

But what should happen here?

```
void foo (Set& s, int x)
{
    s.add(x);
    cout << x << " has been added."  << endl;
}

int main ()
{
    Set s;
    foo (s, 23);
      ⋮
```

- `add()` makes no sense if `Set` doesn't have a data structure that can actually store elements.
    - In fact, `Set s;` makes no sense.

. . . .

How Do We Prevent This?

```
void foo (Set& s, int x)
{
    s.add(x);
    cout << x << " has been added."  << endl;
}
```

```
int main ()
{
    Set s;
    foo (s, 23);
      ⋮
```

- We could add a method so `Set()` that prints an error message when `add()` is called.
- Better is to force a proper choice of data structure at compile time.

. . . .

Abstract Member Functions

```
class Set {
      ⋮
    virtual Set& add (int) = 0;
      ⋮
};
```

- The `= 0` indicates that no method exists in this class for implementing this message.
- `add` is called an *abstract member function*.
  - Subclasses must provide the actual methods (bodies) for these functions.

. . . .

Abstract Classes

An *abstract class* in C++ is any class that

- contains an `= 0` annotation on a member function, or

- inherits such a function and does not provide a method for it.

"Abstract classes" are also known as *pure virtual classes*.

. . . .

Set as an Abstract Class

Set in libg++ is a good example of a class that should be abstract.

- We can't possibly implement `Set::Add`
  - we need to do that in its subclasses.

. . . .

Limitations of Abstract Classes

Abstract classes carry some limitations, designed to make sure we use them in a safe manner.

```
class Set {
   ⋮
  virtual Set& add (int) = 0;
   ⋮
};
  ⋮
void foo (Set& s, int x) // OK
  ⋮

int main () {
   Set s;  // error!
   foo (s, 23);
     ⋮
```

- You cannot construct an object whose type is an abstract class.
- You cannot declare function parameters of an abstract class type when passing parameters "by copy".

  - but you can pass pointers/references to the abstract class type.

....

Abstract Classes & Specification

- Base classes in inheritance-for-specification are often abstract
  - Base class exists to define a protocol
  - Not to provide actual objects
- Our `value.h` spreadsheet *Value* class is abstract.

  We cannot expect, for example, to write a `render` function that would work for all values. We have to rely on the subclasses to provide the code for rendering themselves.
- So is our spreadsheet `expression.h` *Expression* class.

  Look at the declaration and see which functions are marked as abstract. Ask yourself if you could implement any of those for all possible expressions. For example, can you write a rule to `evaluate` any expression? No. Can you write a rule to evaluate a `PlusNode`? Or perhaps a `NumericConstant`? Those seem much more plausible.

....

2.3 Extension

Extension

In this style of inheritance, a limited number of "new" abilities is grafted onto an otherwise unchanged superclass.

```
class FlashingString: public StringValue {
  bool _flash;
public:
  FlashingString (std::string);
  void flash();
  void stopFlashing();
};
```

. . . .

Are Extensions OK?

- Extensions are often criticized as "hacks" reflecting afterthoughts & poor hierarchy designs.
- There is often a cleaner way to achieve the same design
  - A "socially acceptable" form of extension is the *mixin*

. . . .

Mixins

A mixin is a class that makes little sense by itself, but provides a specialized capability when used as a base class.

- Mixins in C++ often wind up involving multiple inheritance.

. . . .

Mixin Example: Noisy

Noisy is a mixin I use when debugging.

```
#ifndef NOISY_H
#define NOISY_H

class Noisy
{
  static int lastID;
  int id;
public:
  Noisy();
  Noisy(const Noisy&);
  virtual ~Noisy();

  void operator= (const Noisy&);
};

#endif
```

```
#include "noisy.h"
#include <iostream>

using namespace std;

int Noisy::lastID = 0;

Noisy::Noisy()
{
  id = lastID++;
  cerr << "Created object " << id << endl;
}

Noisy::Noisy(const Noisy& x)
{
  id = lastID++;
  cerr << "Copied object " << id << " from object " << x.id << endl;
}

Noisy::~Noisy()
{
  cerr << "Destroyed object " << id << endl;
}


void Noisy::operator= (const Noisy&) {}
```

- Keep track of when and where constructors and destructors are being invoked
- Also helps to catch excessive copying.

. . . .

Using Noisy

```
class TreeSet
  : public Set, public Noisy
{
      ⋮
```

- This particular mixin also works if used as a data member.
  - Should we worry about multiple inheritance conflicts?
    - Probably not, because `Noisy` declares so few public members it is unlikely to conflict with the "real" member names of my classes.

. . . .

Mixin Example: checking for memory leaks

```
#ifndef COUNTED_H
#define COUNTED_H
```

```
class Counted
{
  static int numCreated;
  static int numDestroyed;
public:
  Counted();
  Counted(const Counted&);
  virtual ~Counted();

  static void report();

};

#endif
```

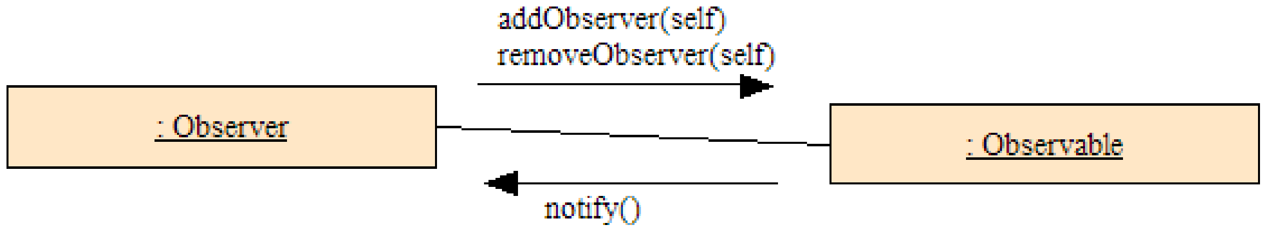
```
#include "counted.h"
#include <iostream>

using namespace std;

int Counted::numCreated = 0;
int Counted::numDestroyed = 0;

Counted::Counted()
{
  ++numCreated;
}

Counted::Counted(const Counted& x)
{
  ++numCreated;
}

Counted::~Counted()
{
  ++numDestroyed;
}



void Counted::report()
{
  cerr << "Created " << numCreated << " objects." << endl;
  cerr << "Destroyed " << numDestroyed << " objects." << endl;
}
```

- A similar example is offered by the Counted class.
- Lets me see if I have created more objects than I have destroyed.

. . . .

3 The Observer Pattern

The Observer Pattern



- A "design pattern" or idiom of OO programming
- Based on 2 mixins
  - Observer
    - can ask an Observable object for notification of any changes
  - Observable
    - will keep track of a list of Observers and notify them when its state changes

```
#ifndef OBSERVER_H
#define OBSERVER_H

//
// An Observer can register itself with any Observable object
// cell by calling the obervable's addObserver() function. Subsequently,
// the oberver wil be notified whenever the oberver calls its
// notifyObservers() function (usually whenever the obervable object's
// value has changed.
//
// Notification occurs by calling the notify() function declared here.
```

```
class Observable;

class Observer
{
public:
  virtual void notify (Observable* changedObject) = 0;
};
#endif
```

Here is an `Observer` mixin. Note that there's not a whole lot to it. `Observable` need to be able to `notify` observers.

Here is the Java equivalent. The Observer/Observable pattern is so common that it is part of the standard Java API, and has been for quite some time. It has some minor differences. The function is called "update" instead of "notify" and can take a second parameter, but the basic idea is the same.

```
#ifndef OBSERVABLE_H
#define OBSERVABLE_H

#include "observerptrseq.h"

// An Observable object allows any number of Observers to register
// with it. When a significant change has occured to the Observable object,
// it calls notifyObservers() and each registered observer will be notified.
// (See also oberver.h)

class Observer;

class Observable
{
public:

  // Add and remove observers
  void addObserver (Observer* observer);
  void removeObserver (Observer* observer);

  //   For each registered Observer, call notify(this)
  void notifyObservers();

private:
  ObserverPtrSequence observers;
};


#endif
```

`Observable` is not much more complicated. It has functions allowing an observer to register itself for future notifications, and a utility function that the observable object calls to talk its current list of observers and notify each of them.

```
#include "observable.h"
#include "observer.h"


// An Observable object allows any number of Observers to register
// with it. When a significant change has occured to the Observable object,
// it calls notifyObservers() and each registered observer will be notified.
//

// Add and remove observers
void Observable::addObserver (Observer* observer)
{
  observers.addToFront (observer);
}


void Observable::removeObserver (Observer* observer)
```

```
{
  ObserverPtrSequence::Position p = observers.find(observer);
  if (p != 0)
    observers.remove(p);
}

//   For each registered Observer, call hasChanged(this)
void Observable::notifyObservers()
{
  for (ObserverPtrSequence::Position p = observers.front();
       p != 0; p = observers.getNext(p))
    {
      observers.at(p)->notify(this);
    }
}
```

The implementation details are above.

The Java version of Observable is similar.

. . . .

3.1 Applications of Observer

3.1.1 Example: Propagating Changes in a Spreadsheet

Example: Propagating Changes in a Spreadsheet

Anyone who has used a spreadsheet has observed the way that, when one cell changes value, all the cells that mention that first cell in their formulas change, then all the cells the mention those cells change, and so on, in a characteristic "ripple" effect until all the effects of the original change have played out.

There are several ways to program that effect. One of the more elegant is to use the Observer/Observable pattern.

. . . .

Cells Observe Each Other

```
class Cell: public Observable, Observer
{
public:
```

The idea is that cells will observe one another.

- Suppose cell B2 contains the formula “2*A1 + B1”.
  - Then B2 will observe A1 and B1.

    It will use the `Observable::addObserver` function to add itself as an observer of both those cells.
  - If something were to happen to A1 that changes its value (e.g., we click on A1 and then enter a new value), then
    - A1 will call `notifyObservers()`, which goes through its list of observers, including, eventually including B2, notifying each one.
    - When B2 is notified, it can re-evaluate its formula, change its value, and notify **its** observers.

. . . .

Changing a Cell Formula

```
void Cell::putFormula(Expression* e)
{
  if (theFormula != 0)
    {  ❶
      CellNameSequence oldReferences = theFormula->collectReferences();
      for (CellNameSequence::Position p = oldReferences.front();
           p != 0; p = oldReferences.getNext(p))
        {
          Cell* c = theSheet.getCell(oldReferences.at(p));
          if (c != 0)
```

```
            c->removeObserver (this);
        }
      delete theFormula;
    }
  theFormula = e; ❷
  if (e != 0)
    {     ❸
      CellNameSequence newReferences = e->collectReferences();
      for (CellNameSequence::Position p = newReferences.front();
           p != 0; p = newReferences.getNext(p))
        {
          Cell* c = theSheet.getCell(newReferences.at(p));
          if (c != 0)
            c->addObserver (this);
        }
    }
  theSheet.cellHasNewFormula (this); ❹
}
```

Here's the code that's actually invoked to change the expression stored in a cell.

❶ The first if statement, and it's loop, looks at the expression already stored in the cell. It loops through all cells already named in the expression and tells them that this cell is no longer observing them (`removeObserver`).

❷ After that, we store the new expression (`e`) into the cell (`theFormula` is a data member of `Cell`).

❸ The next if statement and the loop inside look almost like the first one. But now we are looking at the new formula, and calling `addObserver` instead. So now any cells mentioned in the new expression will notify this one when their values change.

❹ At this point, we have set up the network of cells observing other cells. This particular cell has been given a new expression, so there's a good change its value will change once we evaluate that expression. For technical reasons, we don't do so immediately. Instead, we tell the spreadsheet that this cell has a new formula. The spreadsheet keeps a queue of cells that need to be re-evaluated, and processes them one at a time.

. . . .

Evaluating a Cell's Formula

```
const Value* Cell::evaluateFormula()
{
  Value* newValue = (theFormula == 0)
    ? new StringValue()
    : theFormula->evaluate(theSheet);   ❶

  if (theValue != 0 && *newValue == *theValue) ❷
    delete newValue;    ❸
  else
    {    ❹
      delete theValue;
      theValue = newValue;
      notifyObservers();   ❺
    }
  return theValue;
}
```

Eventually the spreadsheets calls this function on our recently changed cell.

❶ We start by checking to see if the formula is not null. If it is not, we evaluate it to get the value of the new expression, `newValue`.

❷ We make sure the cell's old value (stored in the cell's data member `theValue`) is not null, then check to see if it is equal to the new value.

❸ If they are equal, we don't need the new value and can throw it away.

❹ If they are not equal, we throw out the old value and save the new one. Now our cell's value has definitely changed.

❺ So what do we do? We notify our observers.

. . . .

Notifying a Cell's Observers

```
void Cell::notify (Observable* changedCell)
{
  theSheet.cellRequiresEvaluation (this);
}
```

What does an observing cell do when it is notified? It tells the spreadsheet that it needs to be re-evaluated.

- Again, the spreadsheet puts the cell into a queue, but eventually calls `evaluateFormula` on that cell,
  - which may change value and notify *its* observers,
  - which will tell the spreadsheet that they need to be re-evaluated,
  - which will eventually call `evaluateFormula` on them,

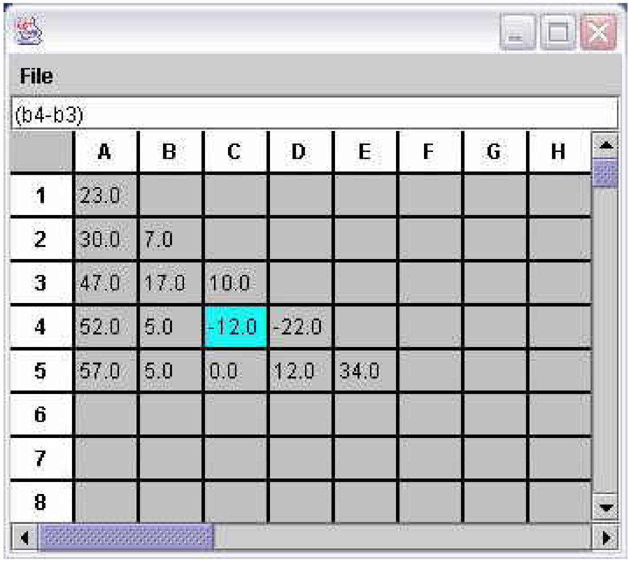
and so on.

Eventually the propagation trickles to an end, as we eventually re-evaluate cells that either do not change value or that are not themselves mentioned in the formulae of any other cells.

. . . .

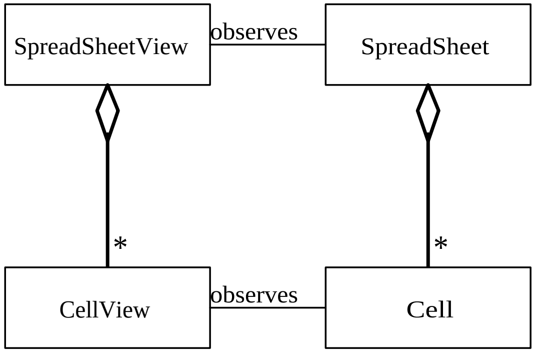
3.1.2 Example: Observer and GUIs

Example: Observer and GUIs



(b4-b3)

	A	B	C	D	E	F	G	H
1	23.0							
2	30.0	7.0						
3	47.0	17.0	10.0					
4	52.0	5.0	-12.0	-22.0				
5	57.0	5.0	0.0	12.0	34.0			
6								
7								
8								



cellView [0,0] observes ssheet.getCell(" a1")

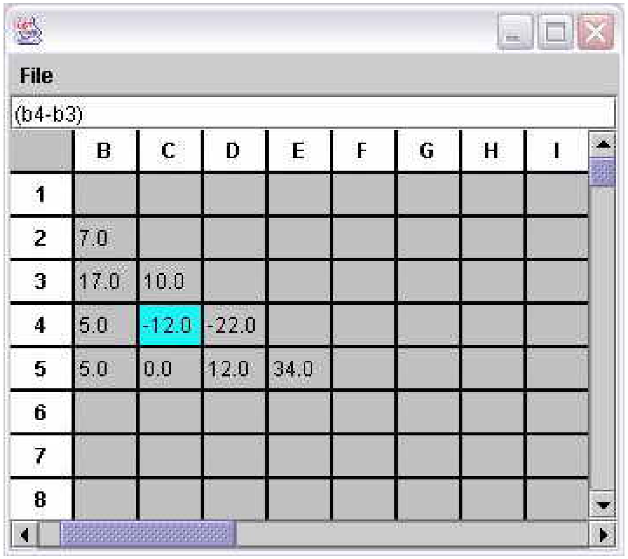
cellView [0,1] observes ssheet.getCell(" a2")

A spreadsheet GUI contains a rectangular array of `CellViews`. Each `CellView` observes one `Cell`

- When value in a cell changes, it notifies its observers, as we have already seen. Some of those observers are other cells, as described earlier. But one of those observers might be a `CellView`.
- The observing `CellView` then redraws itself with the new value

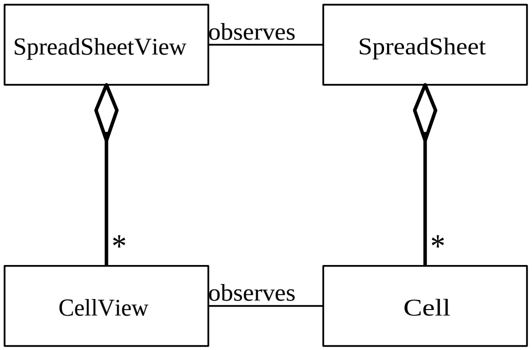
. . . .

Scrolling the Spreadsheet



(b4-b3)

	B	C	D	E	F	G	H	I
1								
2	7.0							
3	17.0	10.0						
4	5.0	-12.0	-22.0					
5	5.0	0.0	12.0	34.0				
6								
7								
8								



cellView [0,0] observes ssheet.getCell(" b1")

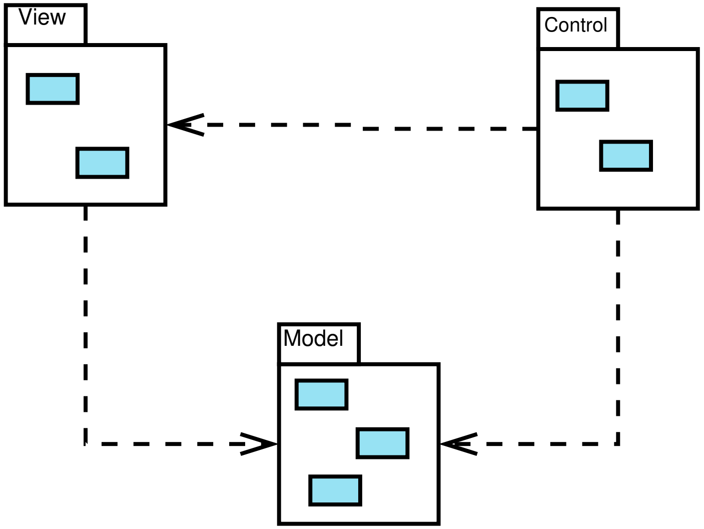
cellView [0,1] observes ssheet.getCell(" b2")

Not every cell will have a `CellView` observer.

- That's because most of the cells in a spreadsheet are not actually visible at a given time.
- Whenever we scroll the spreadsheet GUI, the `CellViews` each register themselves with a different cell, depending on how far we have scrolled.

. . . .

Model-View-Controller (MVC) Pattern



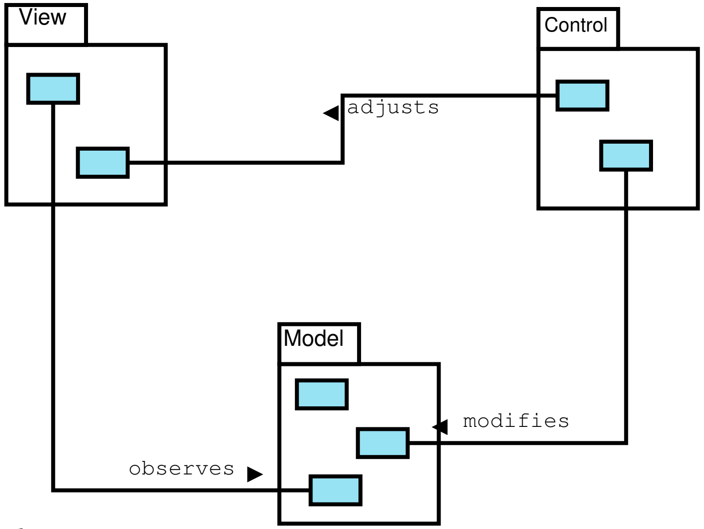
- A powerful approach to GUI construction
- Separate the system in 3 subsystems
  - Model: the "real" data managed by the program
  - View: portrayal of the model
    - updated as the data changes
  - Controller: input & interaction handlers
- View observes the Model

. . . .

This pattern has many advantages. GUI code is hard to test. Keeping the core data independent of the GUI means we can test it using unit or similar "easy" techniques. We can also change the GUI entirely without altering the core classes. For example, my implementation of the spreadsheet has both a graphic form (as shown in the prior section) and a text-only interface that can be used over a simple telnet connection.

Separating the control code means that we can test the view by driving it from a custom Control that simply issues a pre-scripted set of calls on the view and model classes

MVC Interactions



How do we actually accomplish this?

- The Oberver/Observable pattern is one important component of the MVC.
  - It allows the Model classes to notify appropriate parts of the GUI without knowing anything detailed about the GUI class interfaces.

....

4 Abusing Inheritance

Just as there are idioms for good uses of inheritance, there are some that programmers have tried, and that new programmers often “rediscover”, that are really not advisable.

4.1 Construction

Occurs when a subclass is created that uses the superclass as the implementing data structure

- May violate the is-a relationship normally indicated by inheritance.

```
struct Cell {
   int     data;
   Cell* next;

   Cell (int i) {data = i; next = this;}
   Cell (int i, Cell* n)
             {data = i; next = n};
};
```

Here is an example, drawn from a textbook that shall remain anonymous. This cell (no relation to the notion of a cell in a spreadsheet) class defines a "utility" linked list node.

```
class CircularList {
    Cell* rear;
public:
    CircularList();
    bool empty();
    void addFront (int);
    int removeFront ();
    int addRear (int);
};
```

We can use this to create a circular list (a linked list in which the last node in the list points back to the first node). Circular lists are useful in that they support efficient adding at either end and removing from one end.

```
class Queue: public CircularList {
public:
    Queue();
    void enterq (int x) { addRear(x);}
    int leaveq()         {removeFront();}
    // inherit bool empty();
};


class Stack: public CircularList {
public:
    Stack();
    void push (int x) {addFront(x);}
    void pop() {removeFront();}
    // inherit bool empty()
};
```

This list could be used to implement stacks and queues.
So we can write applications like the following:

```
    Queue q;
    q.enterq (23);
    q.enterq (42);
    q.leaveq ();
    bool b = q.empty();
    q.addFront (21);  // oops!
```

The last call is a problem. We should not be able to add to the front of a queue.

- Public inheritance makes too much additional interface visible

– need to hide the underlying operations.

4.1.1 Private Inheritance

```
class Queue: private CircularList {
public:
    Queue();
    void enterq (int x) { addRear(x);}
    int leaveq()         {removeFront();}
    CircularList::empty;
};


class Stack: private CircularList {
public:
    Stack();
    void push (int x) {addFront(x);}
    void pop() {removeFront();}
    CircularList::empty;
};
```

It is possible to solve this problem by private inheritance. We inherit, but all inherited members are private in the subclass.

4.1.2 Private Inheritance

Private inheritance hides inherited members from application code.

- It can be used to share data structure and methods without affecting the interface
- Contrast this with public inheritance, where the major motivation is to share protocol/interface.

4.1.3 What's Wrong with This?

- With private inheritance (or judicious use of protected members), inheritance for the purpose of code sharing is possible,

- But
  - the resulting inheritance hierarchy is often counter-intuitive and may clash with other inheritance motivations.
  - it is usually easier to do the same thing using aggregation/composition rather than inheritance:

```
class Queue: {
   CircularList c;
public:
   Queue();
   void enterq (int x) { c.addRear(x);}
   int leaveq()        {c.removeFront();}
   bool empty()        {return c.empty();}
};
```

Doesn't this just seem simpler?

- With inline function expansion, this need not be any slower than the inherited versions.

4.2 Subset != Subclass

- We often say that inheritance captures an "is a" relationship.
  - But "is a" is vague and subject to many interpretations.
  - A common interpretation is "is a specialized form of".

4.2.1 A Bird in the Hand

```
class Animal;
class Bird: public Animal { ...
class BlueJay: public Bird { ...
```

- Seems logical, right?

4.2.2 is Worth 2 in the Sky?

Let's postulate some members for Birds:

```
class Bird: public Animal {
    Bird();
    double altitude() const;

    void fly();
        // post-condition: altitude() > 0.
      ⋮
};

class BlueJay: public Bird
{
     ⋮
```

4.2.3 Is an Ostrich a-kind-of Bird?

```
class Ostrich: public Bird
{
   Ostrich();
   // Inherits
   // double altitude() const;

   // void fly();
   //post-condition: altitude() > 0.
```

- Now, Ostrich can provide its own method for fly()

4.2.4

```
void Ostrich::fly()
//post-condition: altitude() > 0.
{
   plummet();
}
```

- but it can't satisfy that post-condition.
    - An unpleasant surprise for someone traversing a list of Bird's.

4.2.5 Substitutability

The ostrich/bird hierarchy violates the substitutability principle:

- Class A should be a public subclass of B only if a value of class A can logically be substituted wherever a value of type B is expected.
    - Instead of is-a, perhaps we should say that public inheritance captures an "observes the protocol of" relation.
- You can view this dilemma as a failure of analysis:
    - Who said all birds could fly?
    - A FlyingBirds subclass of Bird would clarify the situation.
- Anyone want to tackle a platypus?

4.2.6 Fixing a Broken Hierarchy

When two classes share code and data but do not share an is-a (protocol) relationship.

- Mouse & Tablet
- Stack& Queue

- Better to factor shared code and data into a common parent class
  - e.g., Pointing Device
- or factor shared code and data into a common implementation structure, accessed via composition
  - e.g., implement Stack & Queue via a LinkedList data member

Appendices

A value.h

```
#ifndef VALUE_H
#define VALUE_H

#include <string>
#include <typeinfo>

//
// Represents a value that might be obtained for some spreadsheet cell
// when its formula was evaluated.
//
// Values may come in many forms. At the very least, we can expect that
// our spreadsheet will support numeric and string values, and will
// probably need an "error" or "invalid" value type as well. Later we may
// want to add addiitonal value kinds, such as currency or dates.
//
class Value
{
public:
  virtual ~Value() {}


  virtual std::string render (unsigned maxWidth) const = 0;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.


  virtual Value* clone() const = 0;
  // make a copy of this value

protected:
  virtual bool isEqual (const Value& v) const = 0;
  //pre: typeid(*this) == typeid(v)
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

  friend bool operator== (const Value&, const Value&);
};

inline
bool operator== (const Value& left, const Value& right)
{
  return (typeid(left) == typeid(right))
    && left.isEqual(right);
}

#endif
```

B expression.h

```
#ifndef EXPRESSION_H
#define EXPRESSION_H

#include <string>
#include <iostream>

#include "cellnameseq.h"

class SpreadSheet;
class Value;

// Expressions can be thought of as trees.  Each non-leaf node of the tree
// contains an operator, and the children of that node are the subexpressions
// (operands) that the operator operates upon.  Constants, cell references,
// and the like form the leaves of the tree.
//
// For example, the expression (a2 + 2) * c26 is equivalent to the tree:
//
//                  *
//                 /
```

```
//                +   c26
//               /
```

```
//            a2   2

class Expression
{
public:
```

```
virtual ~Expression() {}


// How many operands does this expression node have?
virtual unsigned arity() const = 0;

// Get the k_th operand
virtual const Expression* operand(unsigned k) const = 0;
//pre: k < arity()




// Evaluate this expression
virtual Value* evaluate(const SpreadSheet&) const = 0;



// Copy this expression (deep copy), altering any cell references
// by the indicated offsets except where the row or column is "fixed"
// by a preceding $. E.g., if e is  2*D4+C$2/$A$1, then
// e.copy(1,2) is 2*E6+D$2/$A$1, e.copy(-1,4) is 2*C8+B$2/$A$1
virtual Expression* clone (int colOffset, int rowOffset) const = 0;



virtual CellNameSequence collectReferences() const;


static Expression* get (std::istream& in, char terminator);
```

```
  static Expression* get (const std::string& in);
  virtual void put (std::ostream& out) const;




  // The following control how the expression gets printed by
  // the default implementation of put(ostream&)

  virtual bool isInline() const = 0;
  // if false, print as functionName(comma-separated-list)
  // if true, print in inline form

  virtual int precedence() const = 0;
  // Parentheses are placed around an expression whenever its precedence
  // is lower than the precedence of an operator (expression) applied to it.
  // E.g., * has higher precedence than +, so we print 3*(a1+1) but not
  // (3*a1)+1

  virtual string getOperator() const = 0;
  // Returns the name of the operator for printing purposes.
  // For constants, this is the string version of the constant value.




};




inline std::istream& operator>> (std::istream& in, Expression*& e)
{
  string line;
  getline(in, line);
  e = Expression::get (line);
  return in;
}


inline std::ostream& operator<< (std::ostream& out, const Expression& e)
{
  e.put (out);
  return out;
}

inline std::ostream& operator<< (std::ostream& out, const Expression* e)
{
  e->put (out);
  return out;
}

#endif
```