Design Patterns:
An Essential Component of CS Curricula

Owen Astrachan, Geoffrey Berry
Landon Cox, Garrett Mitchener
Department of Computer Science
Duke University

ola@cs.duke.edu
http://www.cs.duke.edu/~ola
http://www.cs.duke.edu/csed

NSF DUE-9554910, CCR-9702550

NSF support for this project

- DUE-9554910, Course, Curriculum, and Development
  - Applied Apprenticeship Approach to CS2, inter-institutional collaboration: Duke, Appalachian State University, North Carolina Central University
  - Develop instructional frameworks for assignments, labs, courses
    - apprentice learning: read, modify, write
    - assignment re-use, develop frameworks that allow re-use, but permit different assignments each semester
- CCR-9702550, CAREER
  - Using and Developing Design Patterns in Undergraudate Computer Science Courses
  - instructional material and curricular suggestions for studying and using patterns (primarily first three courses)

Explaining, Using, Developing Patterns

- What is a pattern, why patterns are important
- Why software infrastructure is essential in explaining, understanding, and using patterns
- Programming Patterns, Design Patterns, other kinds of patterns
- Incorporating patterns into courses, rethinking courses

Overview of talk/discussion/interaction

- Patterns, genesis, use in developing software
- programming patterns, object-oriented design patterns
- before and after, class libraries, software cadavers, pattern infrastructure
- using patterns in courses, developing pattern-based courses
- discussion, future work

What is a pattern?

- **"… a three part rule, which expresses a relation between a certain context, a problem, and a solution. The pattern is, in short, at the same time a thing, … , and the rule which tells us how to create that thing, and when we must create it."**

  Christopher Alexander

➤ **name**	*factory, aka virtual constructor*
➤ **problem**	*delegate creation responsibility: expression tree nodes*
➤ **solution**	*createFoo() method returns aFoo, bFoo,...*
➤ **consequences**	*potentially lots of subclassing, ...*

- **more a recipe than a plan, micro-architecture, frameworks, language idioms made abstract, less than a principle but more than a heuristic**
- **patterns capture important practice in a form that makes the practice accessible**

Patterns are discovered, not invented

- You encounter the same “pattern” in developing solutions to programming or design problems
  - develop the pattern into an appropriate form that makes it accessible to others
  - fit the pattern into a language of other, related patterns

- Patterns transcend programming languages, but not (always) programming paradigms
  - OO folk started the patterns movement
  - language idioms, programming templates, programming patterns, case studies

Pattern/Programming Interlude

- Microsoft interview question (1998)

3	3	5	5	7	8	8	8
3	5	7	8				

- Dutch National Flag problem (1976)

🟥	⬜	🟦	🟦	🟥	⬜	🟦	⬜	🟥
🟥	🟥	🟥	⬜	⬜	⬜	🟦	🟦	🟦

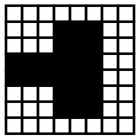
- Remove Zeros (AP 1987)

2	1	0	5	0	0	8	4
2	1	5	8	4			

- Quicksort partition (1961, 1986)

4	3	8	9	1	6	0	5
3	1	0	4	8	9	6	5

- Run-length encoding (SIGCSE 1998)



```
11 3 5 3 2 6 2 6 5 3 5 3 5 3 10
```

Solving (related) problems

- Sometimes it is not clear that problems are related, or how problems are related
- Educators sometimes do not know what they know, so cannot convey knowledge of how to solve problems to students
  - often students don't see programming problems as related, or see them related by language features rather than by higher-level features/dependencies
  - it's often difficult for students to appreciate why one method of solving a problem is better without a context to see the solution in force more than once
- Using patterns can help make knowledge gleaned over many years accessible to those new to the field
  - patterns may be useful in connecting problems and providing a framework for categorizing solutions

One loop for linear structures

- Algorithmically, a problem may seem to call for multiple loops to match intuition on how control structures are used to program a solution to the problem, but data is stored sequentially, e.g., in an array or file. Programming based on control leads to more problems than programming based on structure.

  Therefore, use the structure of the data to guide the programmed solution: one loop for sequential data with appropriately guarded conditionals to implement the control

  Consequences: one loop really means loop according to structure, do not add loops for control: what does the code look like for run-length encoding example?

What about efficiency?

One loop/linear: efficiency issues

- **Consider Quicksort partition example**
  - ➤ **see code samples from different text books**
  - ➤ **consider student developed Quicksort**
    - **write from scratch? Get code from a book?**
  - ➤ **Quicksort in practice: *Engineering a Sort Function,* Bentley and McIlroy, Software Practice and Experience, November 1993**
    - **what about equal keys?**
- **Consequence of pattern: possible that some efficiency may be lost**
  - ➤ **when is efficiency important? Lessons for students?**
  - ➤ **Make it run, make it right, make it fast, make it small**

Coding Pattern

- **Name:**
  - ➤ one loop for linear structures
- **Problem:**
  - ➤ Sequential data, e.g., in an array or a file, must be processed to perform some algorithmic task. At first it may seem that multiple (nested) loops are needed, but developing such loops correctly is often hard in practice.
- **Solution:**
  - ➤ Let the structure of the data guide the coding solution. Use one loop with guarded/if statements when processing one-dimensional, linear/sequential data
- **Consequences:**
  - ➤ Code is simpler to reason about, facilitates develop of loop invariants, possibly leads to (slightly?) less efficient code

Simply Understood Code

- People need to stare at code in order to understand it well enough to feel secure making changes to it. Spending time switching from window to window or scrolling up and down to see all the relevant portions of a code fragment takes attention away from understanding the code and gaining confidence to modify it.
- People can more readily understand things that they can read in their natural text reading order; for Western culture this is generally left to right, top to bottom.
- If code cannot be confidently understood, it will be accidentally broken.
- *Therefore,* Arrange the important parts of the code so it fits on one page. Make that code understandable to a person reading it from top to bottom. Do not require the code to be repeatedly scanned in order to understand how data is used and how control moves about.
- `http://c2.com/cgi/wiki?SimplyUnderstoodCode`

OO Design Patterns

- **Assumption: the object-oriented paradigm is simply better than the structured programming paradigm. OO is here to stay, and will be used with increasing frequency in CS courses, including (especially?) CS1 and CS2**
  - **Consequence: it is hard, especially at first, to write and develop code in an object-oriented way, especially for those whose synapses have been seemingly hard-wired from years of Fortran, Pascal, C, …**
  - **Problem: most (all?) CS1 and CS2 texts do not show good examples of OO programming and design**
  - **Solution: design patterns can help with both problems, but the classic software pattern literature (e.g., GOF) is not easily digestible**
- **Limit the number of patterns studied, use before and after**

Before and After

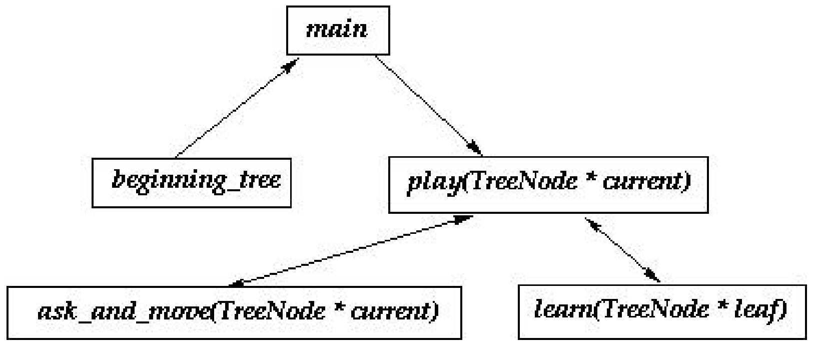
- Problem: the “animal guessing game”

  is it a mammal? *Yes*
  does it have stripes? *No*
  does it live in Africa? *No*
  is it a moose? *No*
  I give up, what animal were you thinking of? *Platypus*
  do you want to play again? ...

- Classic data structures/CS2 assignment
  - ➤ uses binary trees, engaging (potentially) to students
  - ➤ read/store from/to a file
  - ➤ what lessons are learned from this assignment? What are the goals in assigning it?

Structured/Class-based approach

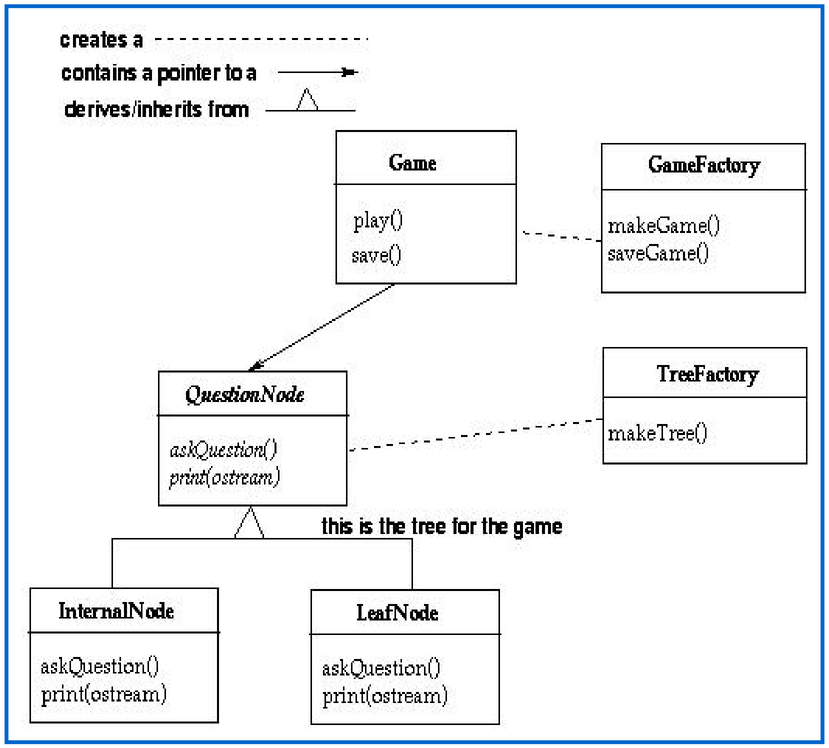
- This solution (see code handout) is from Main and Savitch, *Data Structures and Other Objects using C++*, A-W
  - ➤ How are tree nodes created? What's a more object-oriented way of doing this?
  - ➤ Where is the control for playing a game?
  - ➤ Where would we add code for reading/saving from/to a file?
- What programming lessons can we and our students extrapolate from this exercise?
- Each program we show to students, or ask students to write, is a precious resource



Using OO techniques/patterns

- (my) Fundamental law of OO programming and design

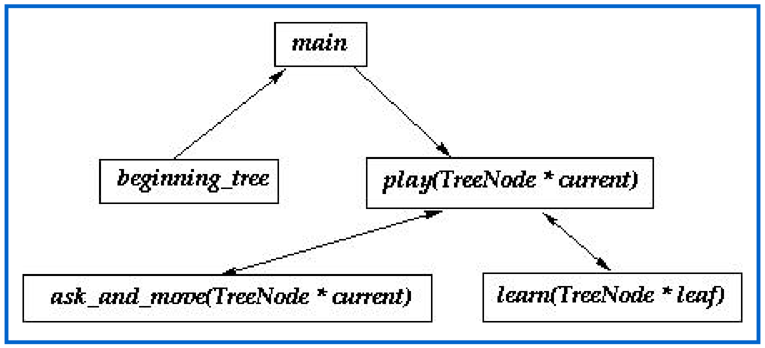
  Ask not what you can do to an object, ask what an object can do to itself
- Nodes ask questions, nodes print themselves (to a file)
- How to differentiate between internal and external nodes?
  - Inheritance hierarchy, use abstract base class
- Who creates nodes, the game?
  - A node factory
  - A game factory



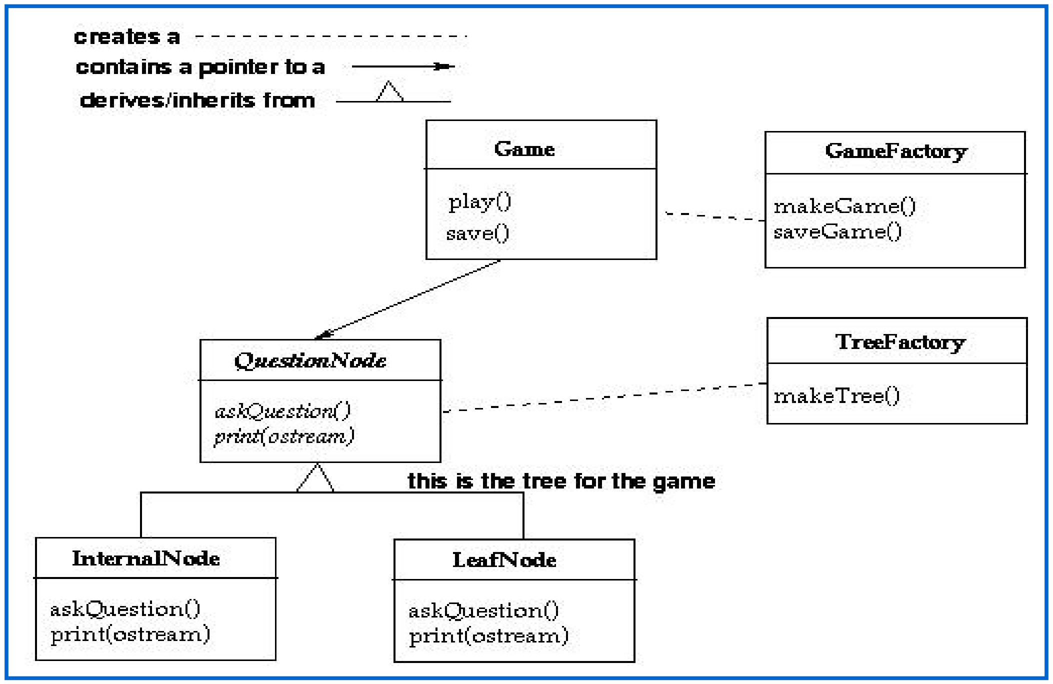
This diagram (and code) from Duke's CS2 course, materials developed for AAA project, see web for details

Before and After

- (roughly) 300 lines of code for both, OO version has 100 lines of .h/.cpp for load and save



- where is code added to read a directory of data files and give user a choice of game?



- Upside of OO
  - transferable design and practice,...
- Downside of OO
  - arguably more complex

Software frameworks/cadavers

- **Experience with OO programming and design shows that design patterns are useful**
  - ➤ **where do we get the experience?**
  - ➤ **how do we give experience to our students?**
  - ➤ **what can we use to illustrate patterns in practice?**
  - ➤ **what patterns should we emphasize, how, when?**

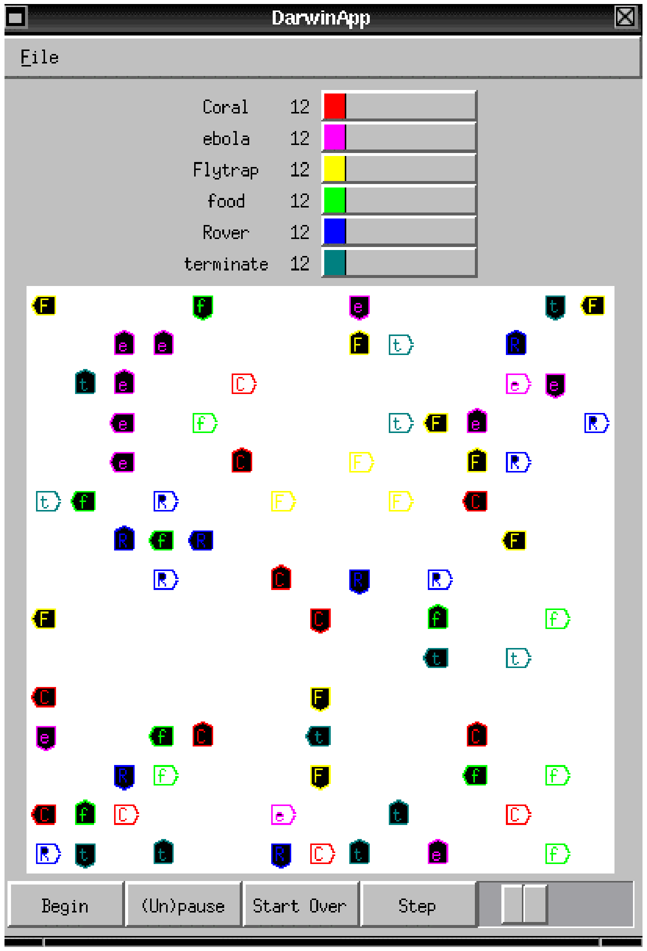
Good design comes from experience, experience comes from bad design

Fred Brooks/Henry Petroski

- **Good design also comes from experiences with good design**
  - ➤ **use several software cadavers to show patterns in practice**

Darwin, example of software cadaver

- Originally developed at Stanford for use in their courses
  - ➤ Ported to Duke in Java and C++ (qt widget toolkit)
  - ➤ Used in COSEN, program for women and minorities to explain computer science
- Simulation/ALife
  - ➤ beings inhabit a world, beings can change species
  - ➤ behavior is programmed using DULL (Darwin Unstructured Lattice Language)
  - ➤ students implement parts of program



Coral	12	🟥
ebola	12	🟪
Flytrap	12	🟨
food	12	🟩
Rover	12	🟦
terminate	12	🟦

Factory pattern

- **Provide interfaces for different parts of the program, expect students to implement the interface**
  - ➤ **Students should not need to implement the entire program at once; vary the assignment, get students to grow the solution**
  - ➤ **Student implements class *Creature*, how is *Creature* used in the rest of the program [that student may not write]?**
    - **Students create a CreatureFactory that returns FooCreature, which implements/subclasses Creature**

```
Factory * fact = new CreatureFactory;
Creature * c = fact->makeCreature();
```

- **A factory is often implemented as a singleton, but avoid too-much-too-soon syndrome**

Observer/Observable

- When the creatures move, the world should show their movement
  - when a program executes, the program view changes
  - each observable (creature) notifies its observers (world listener, program listener) when observable changes
  - separate the model from the view, especially useful when developing GUI programs, allows multiple views of the same model
- Use pattern in one course, show again in a subsequent assignment/course, eventually students adopt/understand
  - spiral approach to using patterns is essential for success

Iterator

- Collections must support access to their elements, use a separate class for access, e.g., supporting forward, backward, sequential, random access
  - iterators are essential in STL, Enumeration used in Java 1.1, Iterator used in 1.2

- Use the Iterator pattern early in CS1 to hide platform-specific code (e.g., in C++), to hide complicated code (e.g., I/O in Java), and to introduce an important pattern
  - WordStreamIterator in C++ and in Java
  - Directory reading/processing classes in both languages

- Internal iterators useful also, even in non OO languages

Design patterns you shouldn't miss

- Iterator
  - useful in many contexts, see previous examples, integral to both C++ and Java
- Factory
  - essential for developing OO programs/classes, e.g., create iterator from a Java 1.2 List? `list.iterator()`
- Composite
  - essential in GUI/Widget programming, widgets contain collections of other widgets
- Adapter/Façade
  - replug-and-play, hide details
- Observer/Observable, Publish/Subscribe, MVC
  - separate the model from the view, smart updates

Patterns discussed in other venues

- Template pattern, function objects
  - see Gelfand, Goodrich, Tamassia (SIGCSE 98)
- Singleton, null-object, state pattern
  - see Nguyen (SIGCSE 98)
- Scaling
  - see Fell, Proulx, Rasala (SIGCSE 98)
- Elementary programming patterns
  - see Wallingford (SIGCSE 96)
  - ChiliPloP (98)

- Do we need to worry about what a pattern is?
  - Growing literature, must contributors conform?

Course and Curricular changes

- Courses must become problem oriented rather than language oriented
    - ➤ texts should explain language and programming details in a context, one that is rich in computer science, software design (or engineering), and intrigue
- Patterns may be part of a new way to organize courses and curricula
    - ➤ one course is not enough, two courses are not enough, think long-term, develop material over a three-course sequence
- Provide pattern-languages/catalogues, facilitate use and application of patterns
    - ➤ cannot throw GOF (or equivalent) at beginners