Loop Invariants

- Want to reason about the correctness of a proposed iterative solution
- Loop invariants provide a means to effectively about the correctness of code

```c
while !done do
  // what is true at every step
  // Update/iterate
  // maintain invariant
od
```

Bean Can game

- Can contains N black beans and M white beans initially
- Emptied according the following repeated process
  - Select two beans from the can
  - If the beans are:
    - *same color*: put a black bean back in the can
    - *different colors*: put a white bean back in the can
  - Player who chooses the color of the remaining bean wins the game
- Analyze the link between the initial state and the final state
- Identify a property that is preserved as beans are removed from the can
  - *Invariant* that characterizes the removal process

Bean Can Algorithm

```c
while (num-beans-in-can > 1) do
  pick 2 beans randomly
  if bean1-color == bean2-color then
    put-back black bean
  else
    put-back white bean
od
```

Bean Can Analysis

- What happens each turn?
  - Number of beans in can is decreased by one
  - Number of white beans is either reduced by 2 or 0
  - Number of black beans is either reduced by 1 or 0
- Examine the final states for 2 bean and 3 bean initial states
- Any guesses for the correct strategy?

- What is the process invariant?
The Game of Nim

- Two Piles of counters with N and M counters in each pile
- 2 players take turns:
  - Remove some number of counters (≥ 1) from one pile
  - Player who removes last counter wins
- Properties
  - Complete information: could exhaustively search for winning solution
  - Impartial: same moves are available for each player

Nim Analysis

- Denote state by \( (x, y) \): number of counters in each pile
- What about simple case of \((1, 1)\)?
- For whom is \((1, 1)\) a “safe” state?
- How about \((1, 2)\) or \((1, 3)\)?
- How about \((2, 2)\)?
- What is the invariant to be preserved by the winning player?

Nim Algorithm

```plaintext
// reach a state \((x, y)\) where \(x = y\) on opponent’s turn and then follow below algorithm

while !empty(pile1) && !empty(pile2) do
  let opponent remove \(q\) counters from a pile
  remove \(q\) counters from other pile
od
```

Numbers from Ends

- Game begins with some even number of numbers on a line
  - 10 5 7 9 6 12
- Players take turns removing numbers from the ends while keeping running sum of numbers collected so far
- Player with largest sum wins
- Complete information but how to win without search?
Patterns

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

- Alexander et. al., 1977
- A text on architecture!

- What is a programming or design pattern?
- Why are patterns important?

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

Programming Problems

- Microsoft interview question (1998)
- Dutch National Flag problem (1976)
- Remove Zeros (AP 1987)
- Quicksort partition (1961, 1986)
- Run-length encoding (SIGCSE 1998)
Removing Duplicates

```java
void crunch(ArrayList<String> list)
{
    int lastUniqueIndex = 0;
    String lastUnique = list.get(0);
    for(int k=1; k < list.size(); k++)
    {
        String current = list.get(k);
        if (current != lastUnique)
        {
            list.set(++lastUniqueIndex, current);
            lastUnique = current;
        }
    }
    for (int k=list.size()-1; k > lastUniqueIndex; k--)
    list.remove(k);
}
```

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?

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

Using linear data structures

- We’ve studied vectors, stacks, queues, which to use?
  - It depends on the application
  - Vector is multipurpose, why not always use it?
    - Make it clear to programmer what’s being done
    - Other reasons?
- Other linear ADTs exist
  - List: add-to-front, add-to-back, insert anywhere, iterate
    - Alternative: create, head, tail (see Clist<> in tapestry)
    - Linked-list nodes are concrete implementation
  - Deque: add-to-front, add-to-back, random access
    - Why is this “better” than a vector?
    - How to implement?