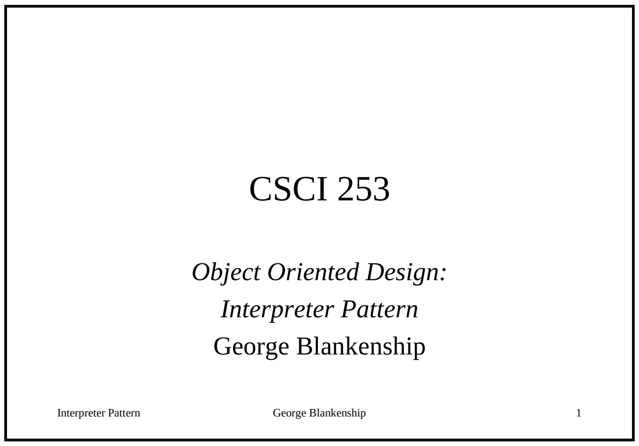
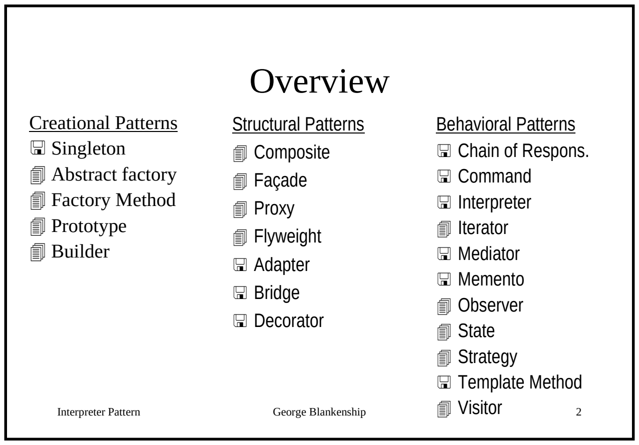


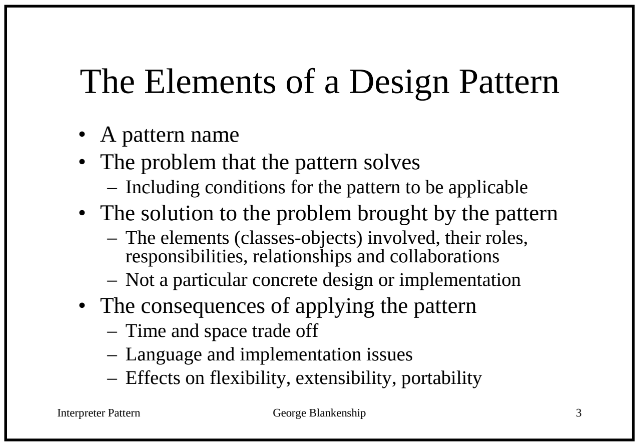
CSCI 253

Object Oriented Design:
Interpreter Pattern
George Blankenship



Overview

Creational Patterns	Structural Patterns	Behavioral Patterns
🖫 Singleton	🗐 Composite	🖫 Chain of Respons.
🗐 Abstract factory	🗐 Façade	🖫 Command
🗐 Factory Method	🗐 Proxy	🖫 Interpreter
🗐 Prototype	🗐 Flyweight	🗐 Iterator
🗐 Builder	🖫 Adapter	🖫 Mediator
	🖫 Bridge	🖫 Memento
	🖫 Decorator	🗐 Observer
		🗐 State
		🗐 Strategy
		🖫 Template Method
		🗐 Visitor

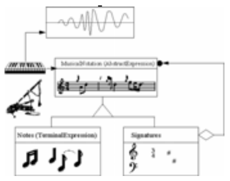


The Elements of a Design Pattern

- A pattern name
- The problem that the pattern solves
  - Including conditions for the pattern to be applicable
- The solution to the problem brought by the pattern
  - The elements (classes-objects) involved, their roles, responsibilities, relationships and collaborations
  - Not a particular concrete design or implementation
- The consequences of applying the pattern
  - Time and space trade off
  - Language and implementation issues
  - Effects on flexibility, extensibility, portability

The Interpreter Pattern: The Problem

he basic idea is to implement a specialized computer language to rapidly solve a defined class of problems. Specialized languages often let a problem be solved several to several hundred times more quickly than a general purpose language would permit.


MusicalNotation (AbstractExpression)
Notes (TerminalExpression)
Signatures

Interpreter Pattern George Blankenship 4

Calculator Grammar

goal -> expr&END
expr -> factor&expr-tail
expr-tail ->^
  expr-tail -> '+' & expr
  expr-tail -> '-' & expr

factor -> term & factor-tail

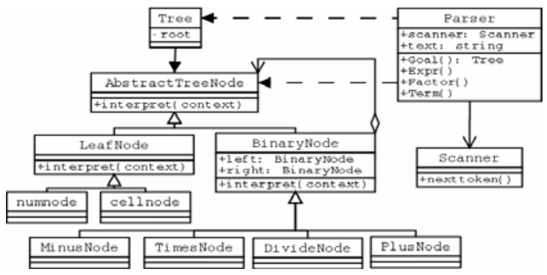
factor-tail ->^
  factor-tail ->'*'&factor
  factor-tail ->'/'&factor

term -> number
  term -> ij
  term -> '(' & expr &')'

tokens: (,), num, ij, +, -, *, /, END

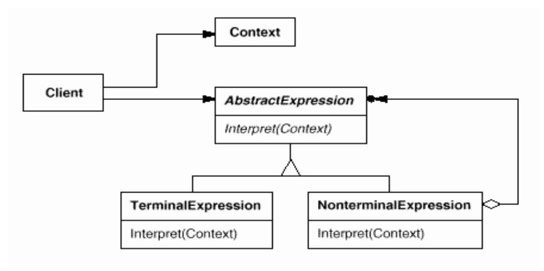
Interpreter Pattern George Blankenship 5

Tree (Calculator)



Interpreter Pattern George Blankenship 6

The Interpreter Pattern: Structure

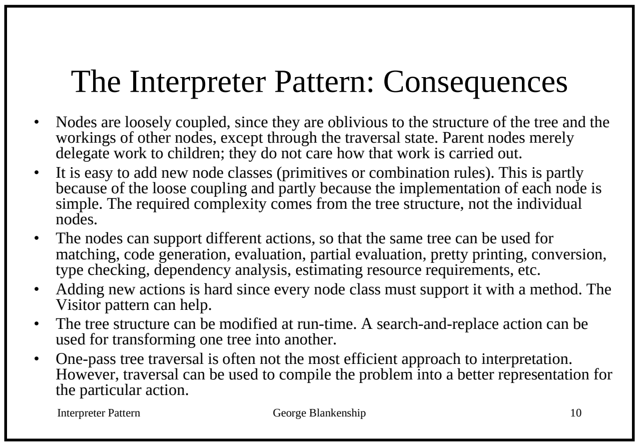


The Interpreter Pattern: Participants

- **AbstractExpression** (RegularExpression)
  - declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.
- **TerminalExpression** (LiteralExpression)
  - implements an Interpret operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in a sentence.
- **NonterminalExpression** (AlternationExpression, RepetitionExpression, SequenceExpressions)
  - one such class is required for every rule $R ::= R_1\ R_2 \ldots R_n$ in the grammar.
  - maintains instance variables of type AbstractExpression for each of the symbols $R_1 \ldots R_n$.
  - implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing $R_1 \ldots R_n$.
- **Context**
  - contains information that's global to the interpreter.
- **Client**
  - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes.
  - invokes the Interpret operation.

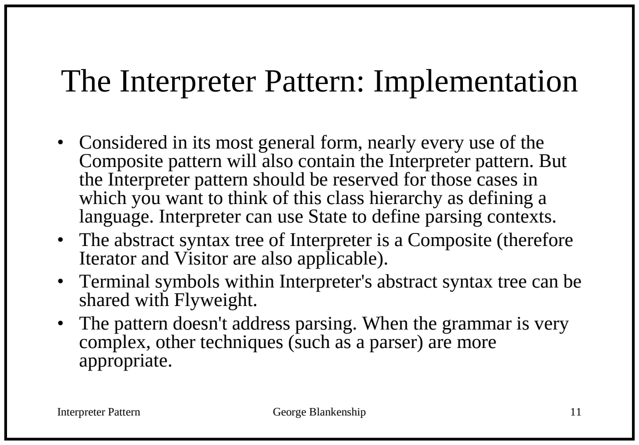
The Interpreter Pattern: Collaboration

- Decide if a "little language" offers a justifiable return on investment.
- Define a grammar for the language.
- Map each production in the grammar to a class.
- Organize the suite of classes into the structure of the Composite pattern.
- Define an interpret (Context) method in the Composite hierarchy.
- The Context object encapsulates the current state of the input and output as the former is parsed and the latter is accumulated. It is manipulated by each grammar class as the "interpreting" process transforms the input into the output.



The Interpreter Pattern: Consequences

- Nodes are loosely coupled, since they are oblivious to the structure of the tree and the workings of other nodes, except through the traversal state. Parent nodes merely delegate work to children; they do not care how that work is carried out.
- It is easy to add new node classes (primitives or combination rules). This is partly because of the loose coupling and partly because the implementation of each node is simple. The required complexity comes from the tree structure, not the individual nodes.
- The nodes can support different actions, so that the same tree can be used for matching, code generation, evaluation, partial evaluation, pretty printing, conversion, type checking, dependency analysis, estimating resource requirements, etc.
- Adding new actions is hard since every node class must support it with a method. The Visitor pattern can help.
- The tree structure can be modified at run-time. A search-and-replace action can be used for transforming one tree into another.
- One-pass tree traversal is often not the most efficient approach to interpretation. However, traversal can be used to compile the problem into a better representation for the particular action.



The Interpreter Pattern: Implementation

- Considered in its most general form, nearly every use of the Composite pattern will also contain the Interpreter pattern. But the Interpreter pattern should be reserved for those cases in which you want to think of this class hierarchy as defining a language. Interpreter can use State to define parsing contexts.
- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.
- The pattern doesn't address parsing. When the grammar is very complex, other techniques (such as a parser) are more appropriate.