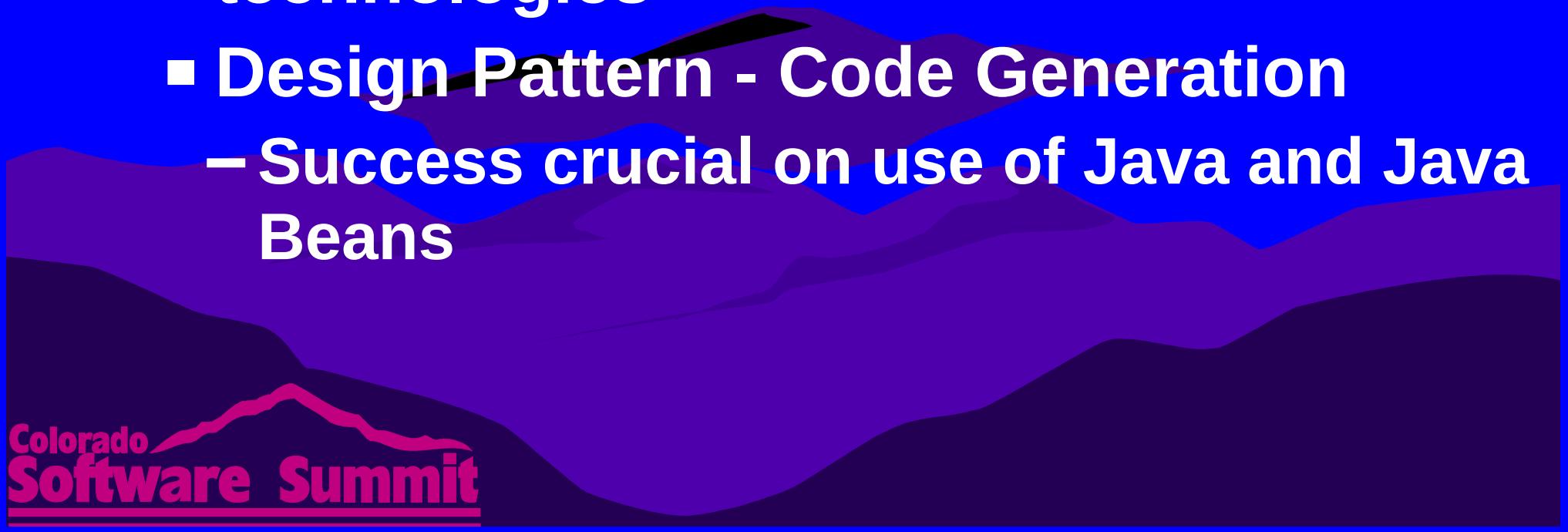
Generating Java Code from Design Patterns

Ray Grimmond
Christie Whitesides
Threshold Computer Systems, Inc.
ray@thresholdobjects.com

Colorado Software Summit

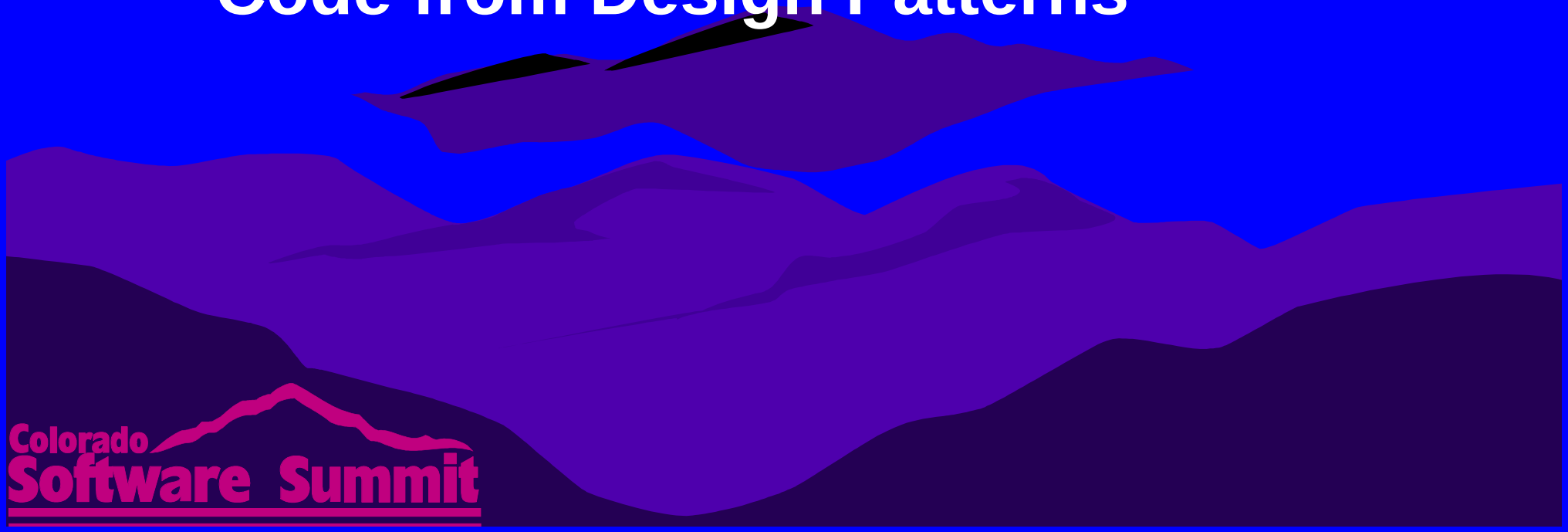
Introduction

- Generating Java Code from Design Patterns
- General Interest in Code Generation
- Tools are crucial in SUCCESSFUL technologies
- Design Pattern - Code Generation
  - Success crucial on use of Java and Java Beans



Agenda

- Code Generation - Background
- Problems with Design Patterns
- How Java helps solves our problems
- Putting it all together - Generating Java Code from Design Patterns

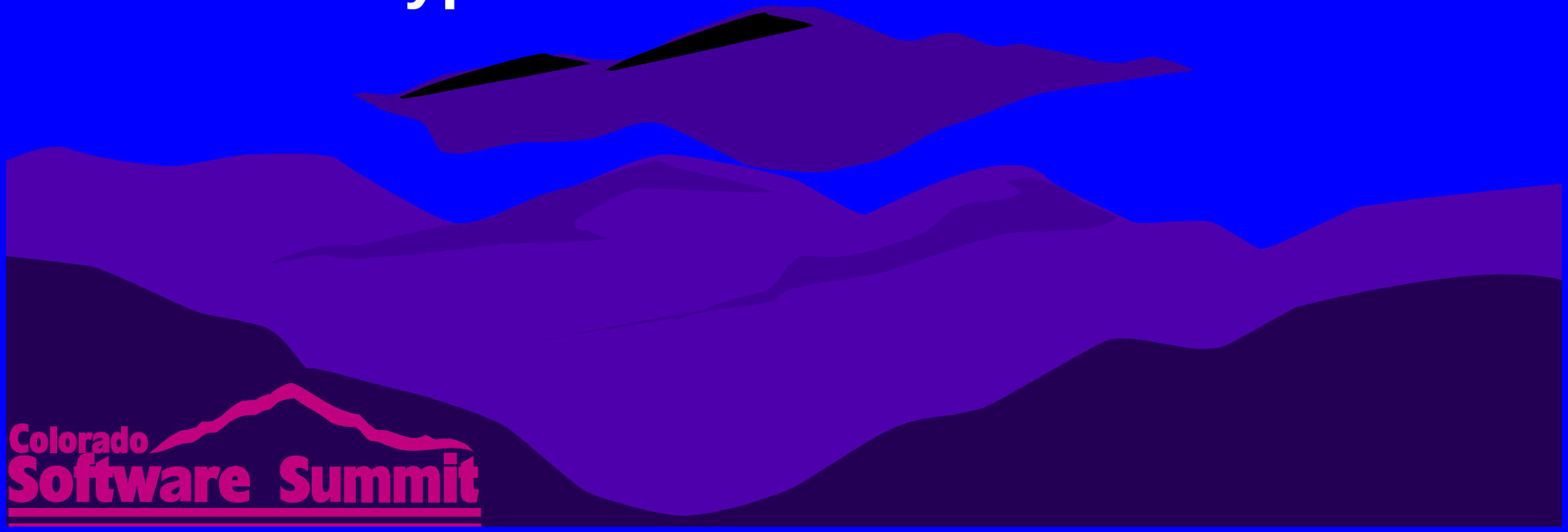


Early Experience - 1994/1995

- Code generation using IBM SOM emitter framework.
  - Generated code from CORBA IDL definitions using specialized emitters.
  - Well defined finite set of types in CORBA - produced templates based on CORBA types.
  - Generated PC-based C++ classes for View classes, Memory model, Database access schema and host message formats.
  - Successful re-generation of system in minutes for IDL changes.

Emitters

- Works well when perfected.
  - System re-generation is straight forward.
- Conceptually simple
  - Coding is to the implementation of finite IDL types and structures.



Emitters (Continued)

- Disadvantages
  - Hard to code
    - Takes a long time to produce the template files and the emitters.
  - Hard to maintain
    - Knowledge of SOM Parser and AST's required.
    - Virtually unreadable by non-author.
  - Monolithic
    - Re-use of existing code is difficult by non-author.
  - Limited audience
    - Dependent on IBM's SOM (System Object Model).

Background

- Patterns
  - Type-based emitter template programming yields recurring "patterns" in code.
- Frustration
  - Unable to formalize ,capture, or re-use these "Patterns".
- *Design Patterns*, 1995
  - Landmark OO-Book by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Rest is History.

Background (Continued)

- *IBM Systems Journal* - Vol 35. No 2 1996
- Automatic code generation from design patterns
  - John Vlissides, Marilyn Finnie, Frank Budinsky and Patsy Yu
- Written before Java
  - Oriented towards Perl and HTML.
  - Home grown mapper and code generator

Problems with Design Patterns

- They are not code
  - They must be implemented each time they are applied
- Most patterns are in hard to read textbooks
  - Real-world pattern implementations different from simple uncombined form found in most text books.
- Most examples are in either Smalltalk or C++
- Code cannot be easily reused
- Minimal or non-existent Visual composition tools
- Problems of authoring and dissemination

Problems with Design Patterns (Continued)

- File System 101 example had implementations of the following patterns - Can you find them??
  - Composite
  - Proxy
  - Visitor
  - TemplateMethod
  - Singleton
  - Mediator

Problems with Design Patterns (Continued)

- File System 101 Example in previous talk
  - Dense pattern implementation - 6 Patterns within fairly small code example.
  - Pattern Paradox - example shows both the power and problems with Design Patterns.
    - A misguided or inexperienced individual who misinterprets the problem the pattern solves, or the context of when to apply the pattern ,or is unable to recognize the patterns or participants within the design, may initiate re-applications, or re-combinations of the patterns, or introduce new patterns that may break the integrity of the existing Patterns and the solution.
    - On the other hand , with guidance and experience , re-application of old Patterns, or the introduction of new Patterns may enhance the integrity of the existing Patterns and the clarity and maintainability of the solution.

The Problems Worsen

- GofF patterns are only the tip of the iceberg
- Writers are writing "Descriptive" Patterns
  - (Those never intended to have code generated from them) Patterns exist for Training, for Organizations, for Education, ... *etc.*
- Pattern Languages - structured collections of patterns that are themselves patterns

Pattern Languages

- From a mathematical point of view, the simplest kind of language system is a system which contains 2 sets.
  - Set of elements, or symbols.
  - Set of rules for combining these symbols.

    "Ordinary language and pattern languages are finite combinatory systems which allow us ot create an infinite variety of unique combinations, appropriate to different circumstances, at will..."

    Christopher Alexander, *The Timeless Way of Building,* pp.187
  - Natural language <=> Pattern language
  - Words <=> Patterns
  - Rules of grammar <=> Patterns which specify connections
  - Sentences <=> Building ..

Patterns and OO Design

- Doug Lea's article "Christopher Alexander: Introduction for OO Designers" makes an insightful connection between patterns and classes.
  - "Patterns extend the definition of OO Classes. Classes are analogous to patterns in the following ways".
  - "External, problem-space view: Description of properties , responsibilites, capabilities and supported services as seen by software clients or the outside world".
  - "The Internal, solution-space view: Static and dynamic descriptions, constraints, and contracts among other components, delegate, collaborators, and helpers, each of which is known only with respect to a possibly incomplete external view". (*i.e.*, a class, but where the actual member may conform to a stronger subclass)."

Goal - Design Patterns to Java Code

- Given the infinite combinations of Patterns and Pattern Languages, how can we even consider generating Code from Design Patterns?
- The following problems need to be solved
  - How to Discover and Recognize Design Patterns
  - How to Represent, Apply and Combine Design Patterns
- The following conditions inhibit code generation
  - Inadequate abstractions
  - Inadequate visualization Tools to examine , explore, and experiment with these missing abstractions.
  - Lack of a generative solution.
  - (*i.e.* generative solution is one that should be self-generating - similar to the Bootstrap process, use the tools and the output of tools to build the tools.)

"Simple" Model Solution

- Don't worry about finding patterns
- Invent new template language
- Build some panels to make user selections for implementation trade-offs
- Do some more symbol substitution and editing
- Cut and paste results into your favorite application

Bad Solution !!!

- Solution is inadequate
  - Too time consuming
    - We will spend more time building tools than applying patterns.
  - Building a one-off solution.
    - Need easily adaptable solution with design re-use in mind.
  - Cannot use other peoples work.
    - Need to capture, communicate, and apply design knowledge.

Tools Strategy

- Need a Tools Strategy
  - Identify our requirements for the tools
  - Solve the following problems
    - Discovering and Recognizing Patterns.
    - Representing Patterns.
    - Applying and Combining Patterns.
  - Develop our strategy

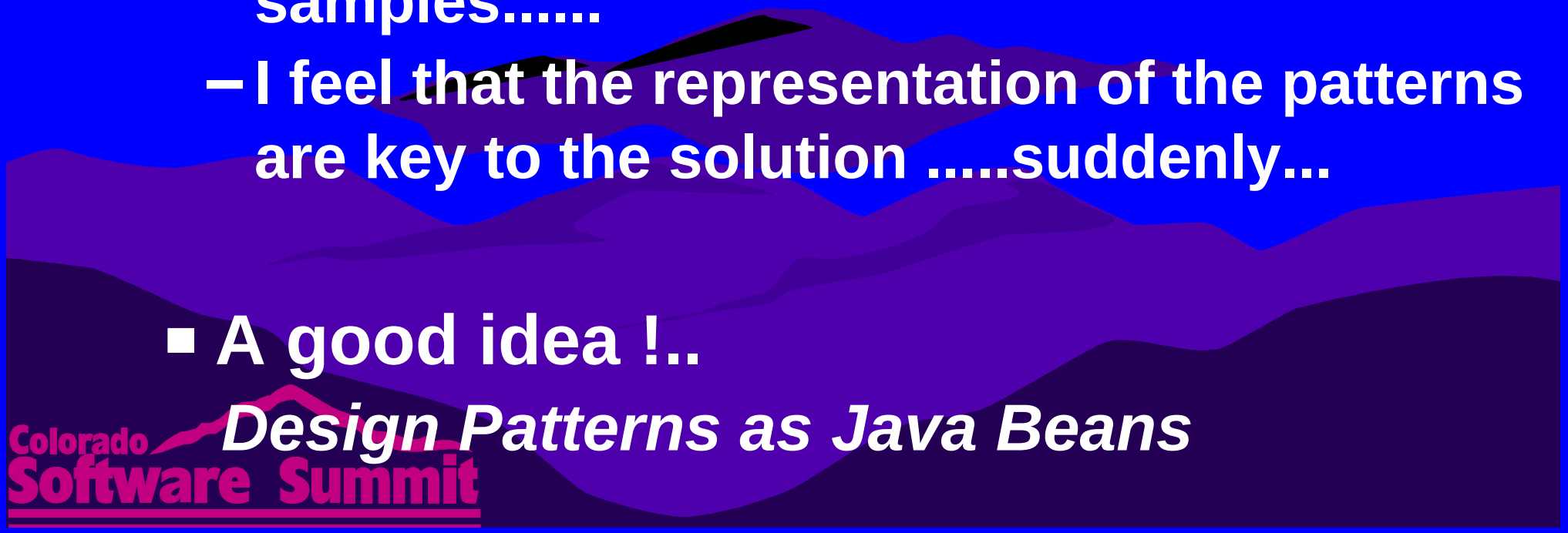
Requirements for Design Pattern Tools

- Design requirements
  - Rapid prototyping
  - Flexible and extensible
  - Easy specification
  - Symmetry
- Ease of use
  - Utility
  - Seamless Integration - Plug-in, other people can use or add to
  - Wide audience - Internet - (Java Applet)

HELP!

- Now what?
  - I have all this information, but no way to use it. I have these lofty goals, some requirements, and a strategy in mind. But what can I do? All I can do is cut and paste samples.....
  - I feel that the representation of the patterns are key to the solution .....suddenly...
- A good idea !..

Design Patterns as Java Beans



The Benefits of Beans

- Java Beans Component Model
  - Discrete
  - Reusable
  - Visually configurable
  - Can interact with other beans
  - Can be combined to form complex applications
- Works with builder tools
- Persistent
- Dynamic loading
- Object Model
- Reflection

Patterns as Beans

- Can we extend the JavaBean model to support a Pattern Bean?
  - Bean Definition:
    - "A reusable software component that can be manipulated by a builder tool"
  - Application Builders:
    - None exist that recognize the Pattern Bean. "One can be built"
  - Auxiliary Information:
    - Can be provided by extending BeanInfo class to PatternBeanInfo
    - New Descriptor classes for Patterns can be added to the BeanInfo
- The answer to our question is "YES".

Patterns as Beans (Continued)

- Pattern Bean - More details
  - Which pieces of a Pattern can be made into Beans?
    - Patterns and Participants (i.e. their Class representations) could be beans. Even the connections between Patterns or within a Pattern could be expressed as Beans.
  - Containers
    - Beans support the notion of containers - Beans within Beans, that fits our model of Patterns within Patterns
  - Serialization
    - Specialized forms of serialization can be used to indicate particular applications of a Pattern or a Participant.

Patterns as Beans (Continued)

- Pattern Bean - More details
  - Property Editors
    - Could be used to set pattern properties such as Gang of Four's; Name, Intent, Motivation, Applicability, Also Known As, Known Uses, Related Patterns...
  - Customizers
    - A better mechanism to view and customize the overall pattern. Presents pages for simple properties listed above, as well a implementation option selections, tree view of participant classes and an imbedded source code editor.
  - Editor Kits
    - Better than a standard source code editor. Specialized with pattern intelligence built into Customizer as an editing environment.

From Design Patterns to Java Code

- We need to build the tools required to solve each of the following five problems with Design Patterns
  - Recognition
  - Representation
  - Application
  - Combination
  - Generation

Discovering Patterns

- Discovering Patterns in existing applications
  - Detect and Identify Structure
    - Examine the Java Classes for overall structure , fields, methods, uses, constructor, inheritance, implementations, *etc.*
  - Detect and Identify Participants
    - Check the Java classes, look for reponsibilities and collaborations between classes.
  - Detect and Identify Patterns
    - Look for inter-related classes

Discovering Patterns (Continued)

- Examine relationships between participants in the same Design Pattern
  - Detect and Identify
    - Relationships
      - inheritance
      - aggregation - containment, reference
      - uses - method parameter types
    - Polymorphic uses
      - protected methods and overriding methods
    - Interfaces/Abstract Classes

Remember relationships that are NOT there are equally important.

Discovering Patterns (Continued)

- Examine relationships between design patterns
  - Large
    - Regions (architecture) <=> Frameworks
  - Medium
    - Buildings <=> Programs
  - Small
    - Bricks <=> idioms (2/3 lines of code)

Remember relationships that do not exist are equally as important.

Pattern Recognition

- Look for patterns in existing code sources
  - Existing .java files
  - Examine .class classfiles
  - Use Java core reflection
  - Read the documentation and comments for hints !

Pattern Recognition (Continued)

- Perform rule-based decomposition of classes
  - Examine relationships between classes, packages, and interfaces.
  - Examine relationships between classes, fields, methods, constructors, and parameters.
  - Examine class and method modifiers, look for use of protected, private, public and final modifiers.
  - Further structural analysis, including aggregation, scope, assignment, overriding methods, *etc.*

Pattern Recognition - Tools

- JFC Tree
  - Easy representation and visualization of a classes ASTs (Abstract Syntax Trees).
  - Allows visualization of internal/external pattern relationships.
- Magelang ANTLR Tool
  - Magelang Institute provide a free full-source Java Lexer and Java Parser generator. Allows for the construction of ASTs.
- Magelang X-Ref Tool
  - Able to examine relationships between Java Classes; any number of classes in any number of packages.

Pattern Recognition - Tools (Continued)

- Java Core Reflection
  - Another way of examining Java Classes and their internal structure - limited however to classes, fields, methods and their parameters. (Security check problems with Applets and some Browsers.)
- Javap
  - Another approach is to use disassemblers similar to the SUN *javap*.
- Classfile analyzers
  - Java *.class* classfile format stable and well documented.

Pattern Representation

- How do we represent a Pattern in a Tool?

  Relationships between Patterns are Patterns themselves - Challenge is how do we represent this?

  Here are just a few possibilites:

  - Create implementation based sample Java code.
  - Collect HTML based Pattern implementations.
  - Define a Pattern using a program and data structures.
  - Plug-in Design Patterns CD's HTML.

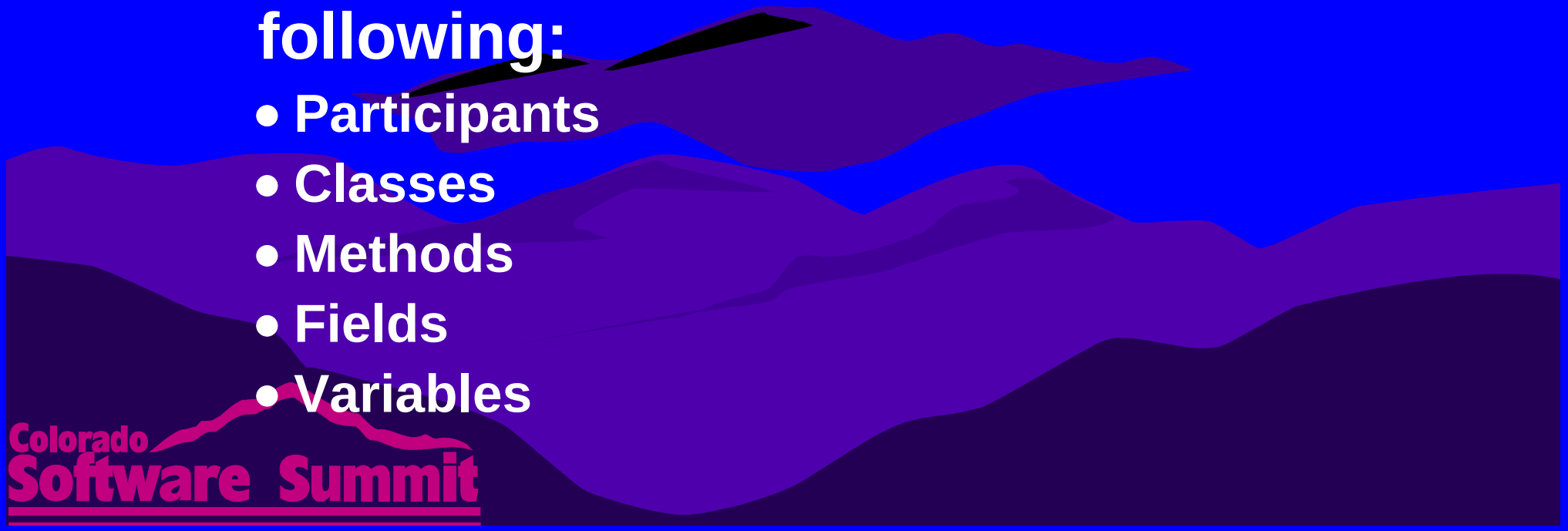
  Answer is to initially allow all these forms to represent Patterns and their relationships. What forms are viable or necessary for code generation will be discovered later.

Pattern Representation - Tools

- HTML Plug-in.
  - Allow simple HTML pattern descriptions and samples to be included as part of the pattern representation.
- GofF Design Patterns CD Plug-in.
  - Allow local or network use of Design Patterns CD-ROM.
- Pattern Bean Interface API.
  - Settable properties allow inclusion of other authors patterns into the environment.
- Internal Pattern representation.
  - Common intermediate pattern form used by all tools.

Pattern Application

- Information required to implement design patterns
  - Choices for implementation trade-offs and code generation options
  - Application specific names for the following:
    - Participants
    - Classes
    - Methods
    - Fields
    - Variables



Pattern Application - Tools

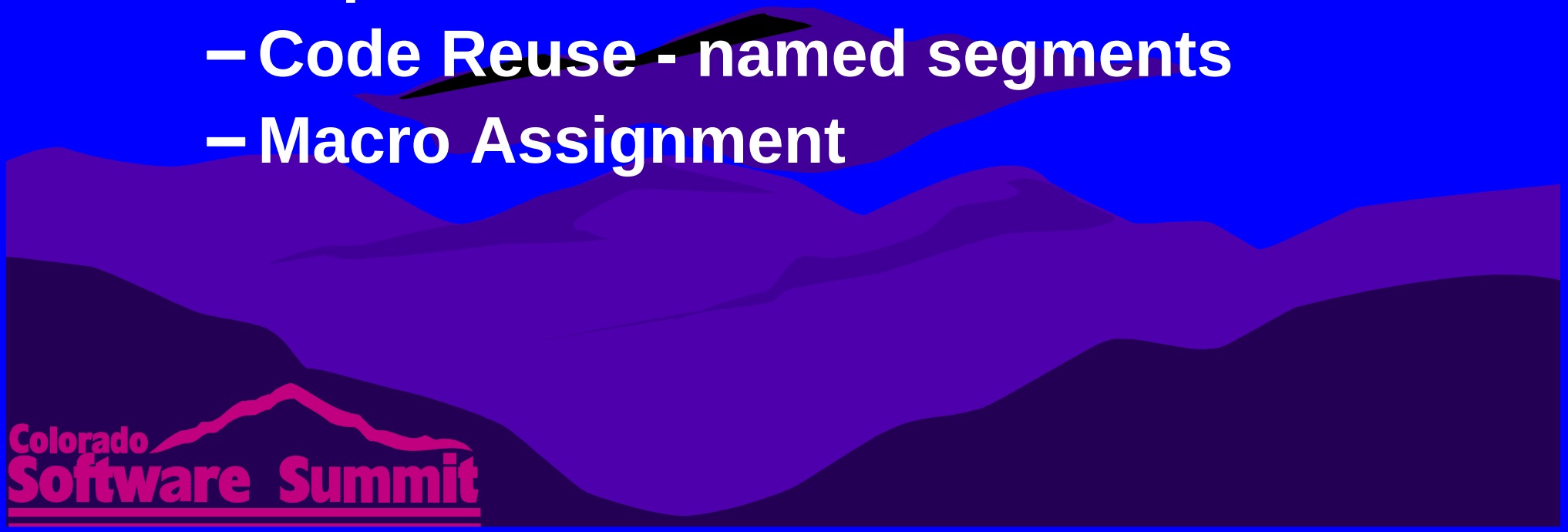
- Bean Tools
  - Customiziers
    - Series of panels that allows user to customize a particular pattern implementation.
  - Property Editors
    - Allows editing and setting of properties with Pattern Bean.
  - Bean Serialization
    - Allows user changes to be serialized and saved.
  - Swing - Text Package
    - Specialized Editor Kits.

Pattern Application - Tools
(Continued)

- Swing Text Package - Editor Kits
  - Specialized types of Pattern Editor Kits
    - Single Pattern Editor Kit.
    - Pattern Identifier Editor Kit.
    - Pattern Application Editor Kit.
  - Editor Kits allow multiple views.
    - Pattern View.
    - Participant View.
    - Document View.
    - classfile View.

Pattern Application - Tools
(Continued)

- Template Editor Kit
  - Simple Macro replacement with optional transformation
  - Conditional Inclusion
  - Repetitive Inclusion
  - Code Reuse - named segments
  - Macro Assignment



Pattern Combination

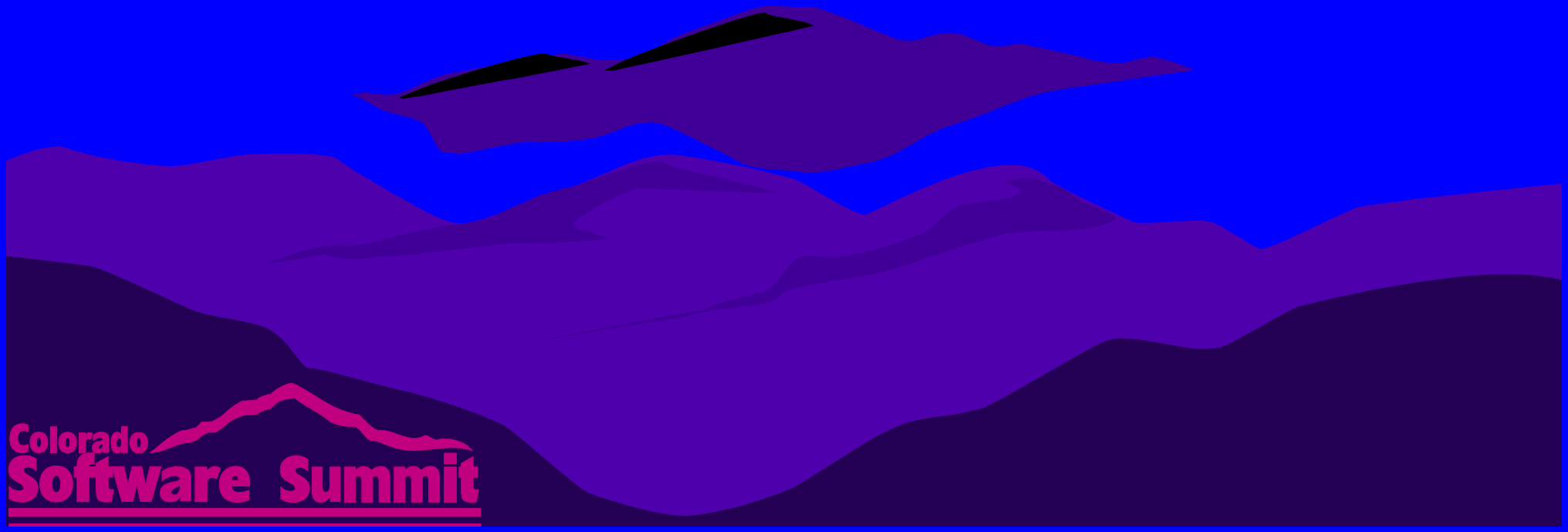
- Problems with Combining Multiple Patterns
  - Easy to apply changes to 1 isolated pattern, and generate the code.
  - Strategy needed to combine individual Patterns into larger Patterns and Pattern Languages
  - Need to resolve implementation conflicts between individual patterns in regard to:
    - Merging of Structure
    - Merging of Program Logic
    - Merging of Names

Pattern Combination - Tools

- Visual Builders, BeanBoxes - Interim Solution
  - Current Paradigm - Components and Parts , need revised concept for Pattern Beans.
  - Links between Beans - connections between beans rely on tying of Events, Properties, and Methods together.
  - Links between Patterns are Patterns - concept not well formalized or understood.
  - Current family of Visual Builders seen as interim measure to gain familiarity with Pattern Tool Concepts.

Pattern Combination - Tools
(Continued)

- Long term solution
  - New generation of Visual Builders
  - Advanced Editor Kits
  - Graphical visualization Tools



Pattern Combination - Tools (Continued)

- Swing Text - Document Interface
  - Document - Holds lot of potential for Pattern Combination
    - Allows arbitary complex element structures to be built within a single Document (uses Composite pattern)
    - Multiple element structures could include structures for classfiles, sourcefile, ASTs , databases, *etc.*
    - Multiple views are supported at the element level.
    - Different views (including graphical) can be created for different elements and element structures.

Code Generation - Requirements

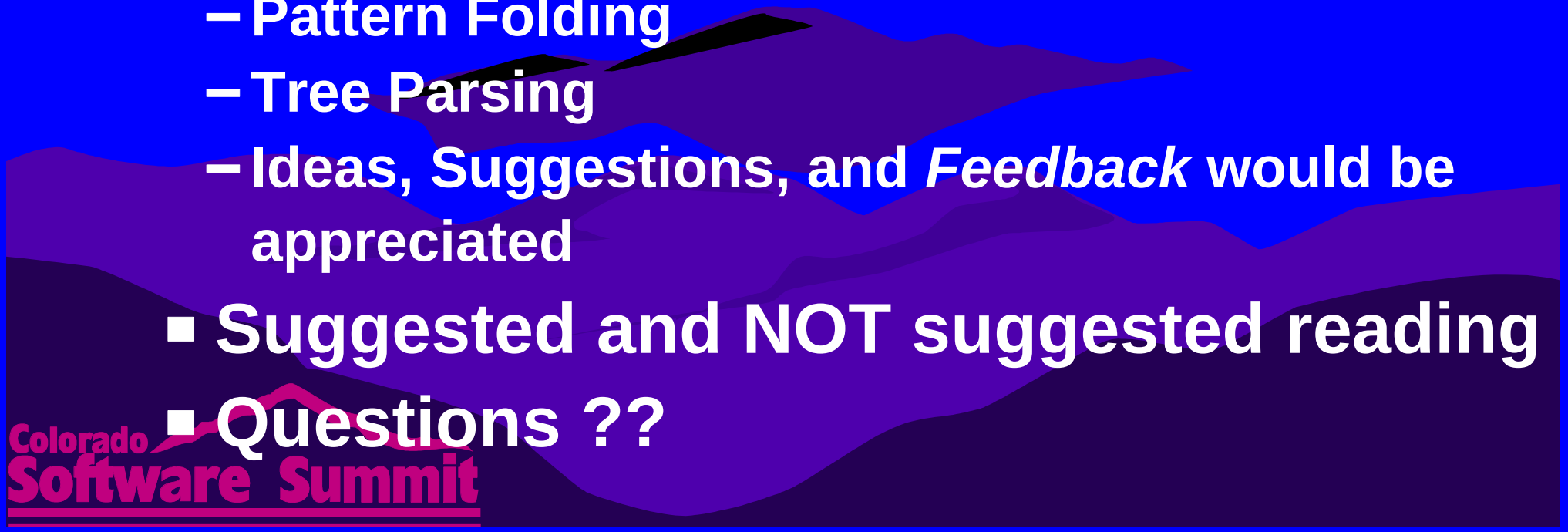
- Convenient
  - Solution has to be usable and easily understood.
- Non-Invasive
  - The solution should be comprehensive and complete. The user should not have to spend ages tweaking the output.
- Non-Irreversible
  - Incorporation of user changes after the code has been generated, need to be preserved or re-applied after one of the underlying patterns have changed, or a new pattern has been added.

Code Generation - Solution

- Symmetrical
  - Leverage use and development of symmetrical Tools, those that can recognize patterns and create the Pattern intermediate form can also generate code from the Pattern intermediate form.
- Solution
  - Key to the solution is the Pattern Intermediate form
- Work - in - Progress
  - Work is still underway when this presentation was assembled.

Where Do We Go from Here?

- What's Next ??
  - More on Pattern Languages
  - UML
  - Formal specification languages - 'Z' and VDM
  - Lambda Calculus
  - Pattern Folding
  - Tree Parsing
  - Ideas, Suggestions, and *Feedback* would be appreciated
- Suggested and NOT suggested reading
- Questions ??



References

- Threshold Computers Systems - Contact Web Site

  www.thresholdobjects.com
- Threshold Pattern Tools

  www.qwan.com
- Books
  - *The Timeless Way of Building,* Christopher Alexander, OUP, ISBN 0195024028
  - *A Pattern Language,* Christopher Alexander, OUP, ISBN 0195019199
  - *The Patterns Handbook*, Linda Rising, SIGS, ISBN 0521648181
  - *Design Patterns*, Gamma, Helm, Johnson, Vlissides, AW, ISBN 0201633612
  - *Pattern Hatching*, Vlissides, AW, ISBN 0201432935