Static and Dynamic Structure in Design Patterns

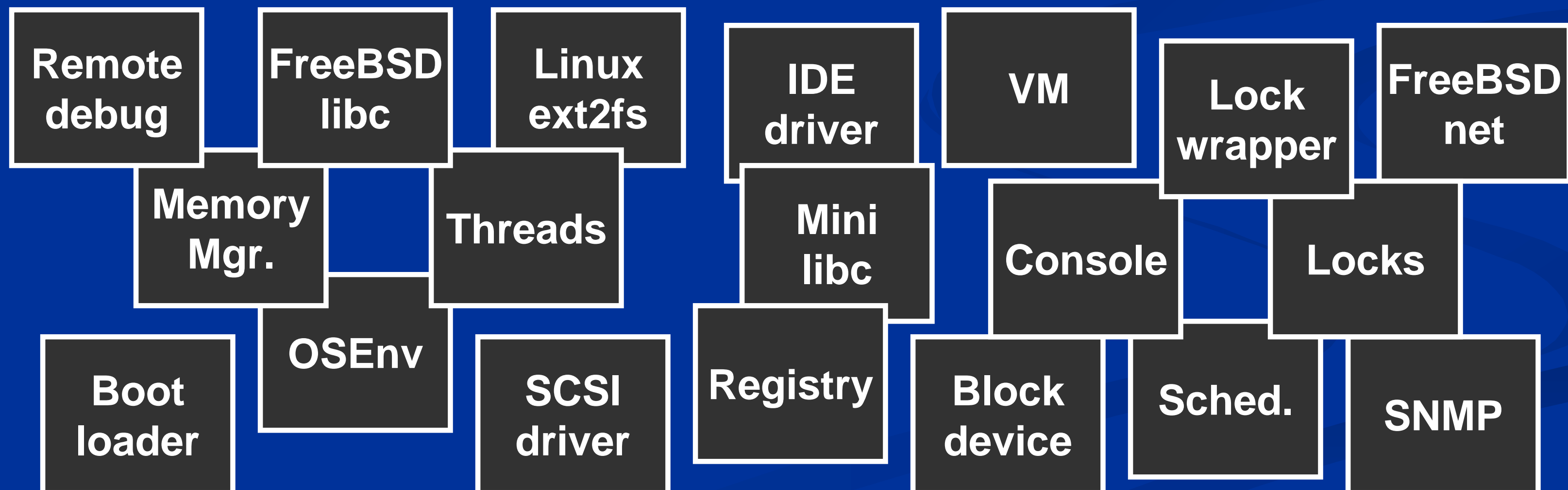
Eric Eide Alastair Reid
John Regehr Jay Lepreau

University of Utah, School of Computing

May 22, 2002

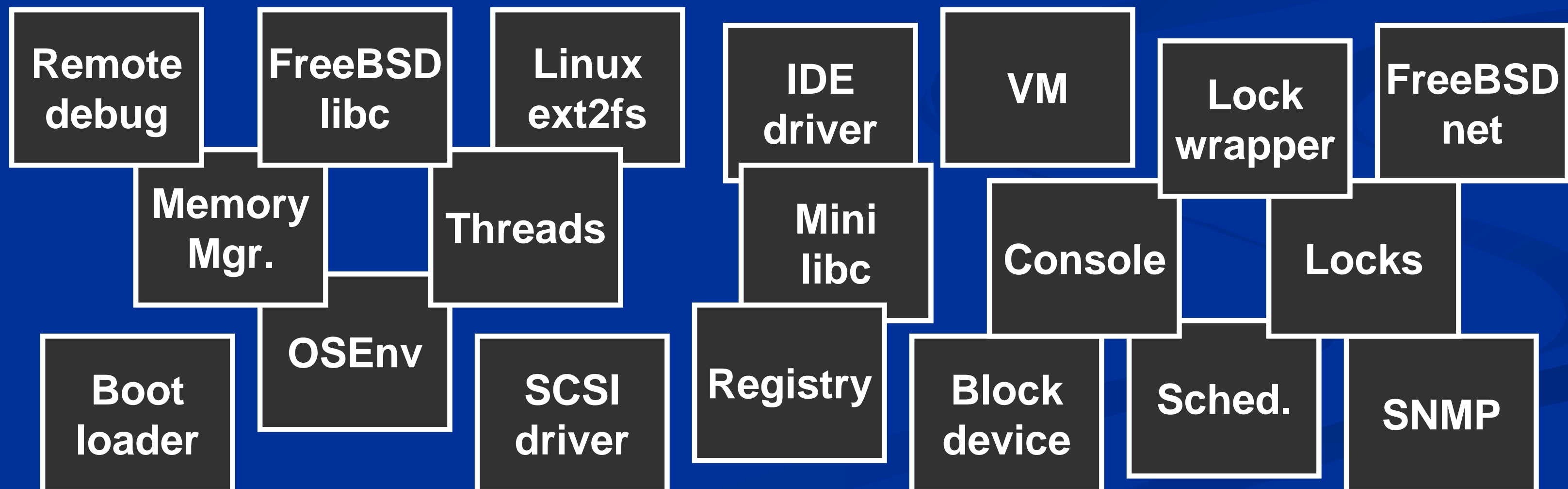
OSKit Components

- Systems software components
- Many taken from Linux, FreeBSD, …
- Largely written in C; 1M+ LOC



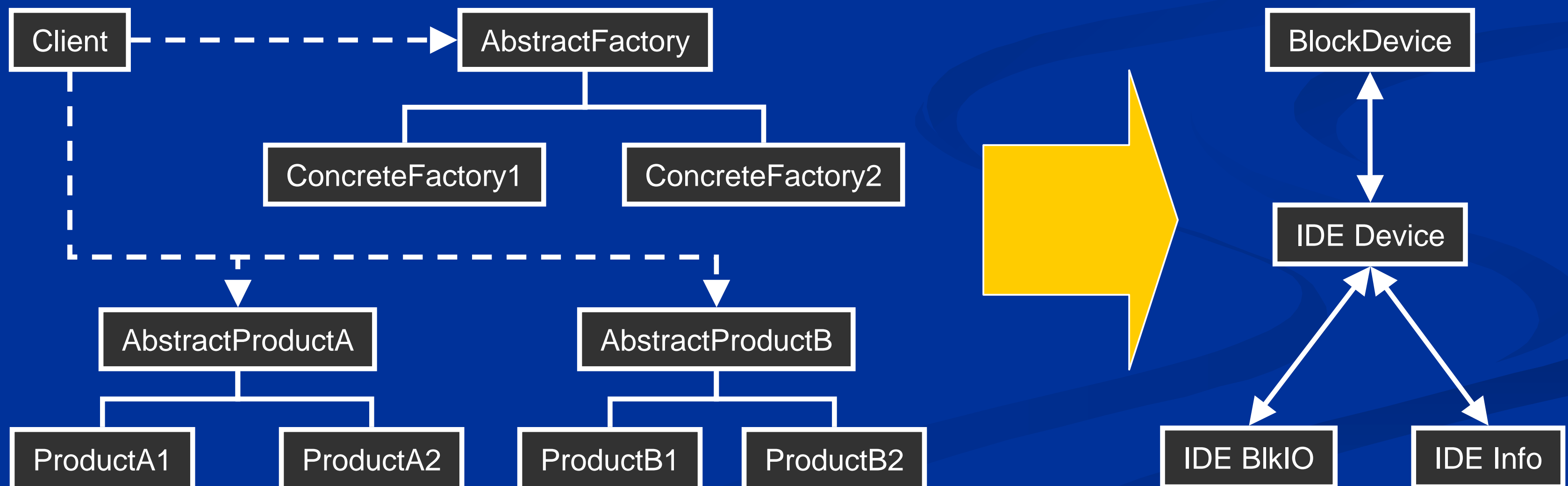
OSKit Components

- Want reuse, with minimal modification
- Want to combine in myriad ways
- Want to understand the resulting systems



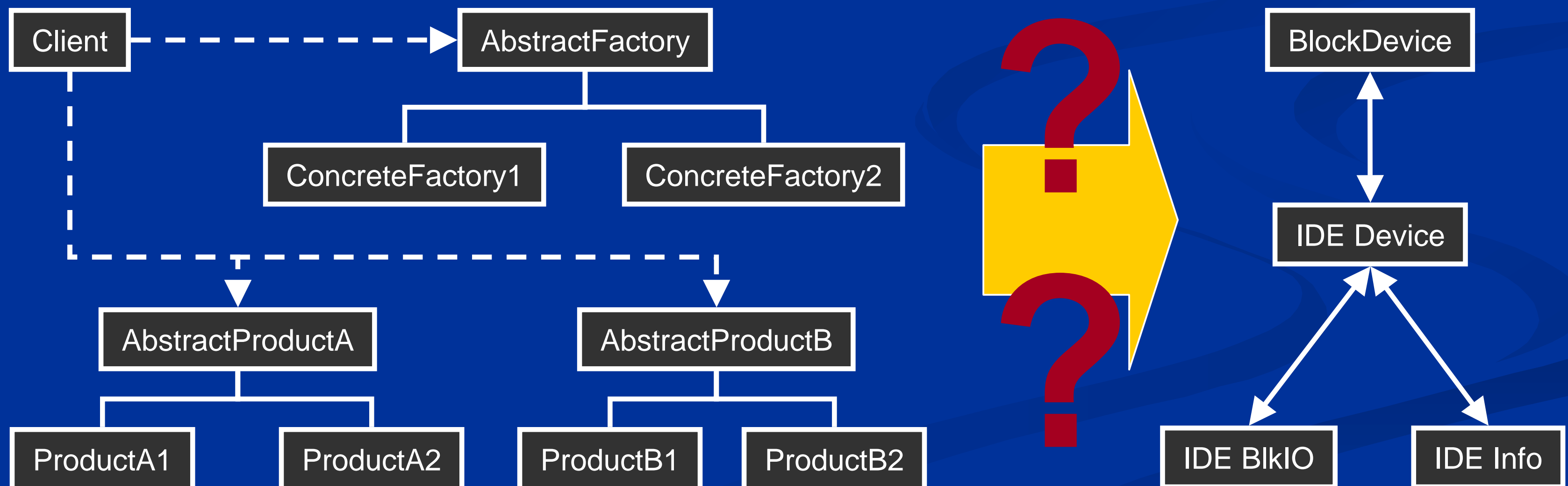
Idea: Design Patterns

- Capture systems design expertise
- Leverage shared knowledge base
- Useful throughout software lifecycle



Idea: Design Patterns?

- "Legacy" C code
- No language-supported classes, objects
- Conventional OO approach: too dynamic



Motivation

- Apply design patterns in context of CBD
- Make patterns explicit/obvious at the level of components
- Avoid changing the components' code

Key Ideas

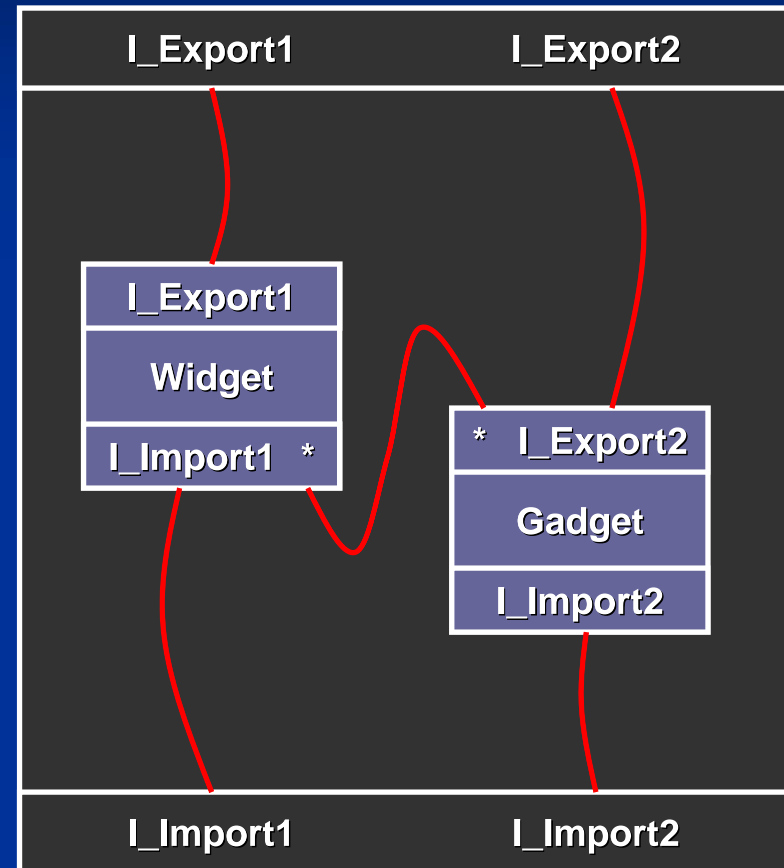
- Separate the *static* and *dynamic* parts of design patterns
  - “lift” static parts to level of components
  - realize dynamic parts with objects
- Leverage unit component model
  - [Flatt and Felleisen, PLDI ’98]
  - Implemented for C, Java, Scheme

Contributions

- Describe our approach to realizing patterns
- Define a method for realizing existing patterns via our approach, applicable to:
  - …imperative, functional, and OO languages
  - …many existing (GoF) patterns
- Demonstrate with examples from the OSKit
- Evaluate benefits and costs of our approach
  - increased opportunities for reuse
  - verification of architectural constraints
  - performance optimizations

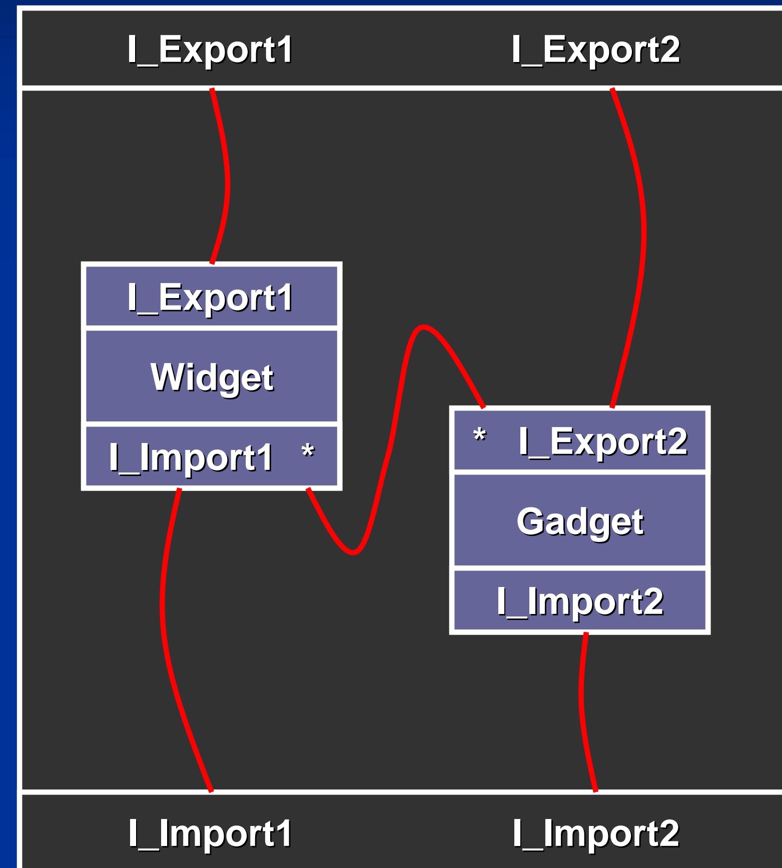
Units

- Imports
- Exports
- Interfaces
- Connections
- Hierarchy
- Multiple instances
- Separate language
- C, Java, Scheme



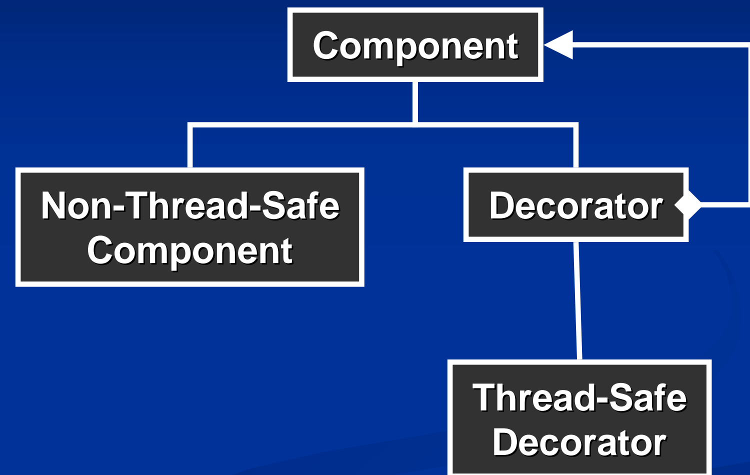
Units

- Parts of a static assembly, not run-time values
- Import/export types and classes
- Constraints, build-time constraint checking
- Optimization via cross-component inlining



Expressing Patterns with Units

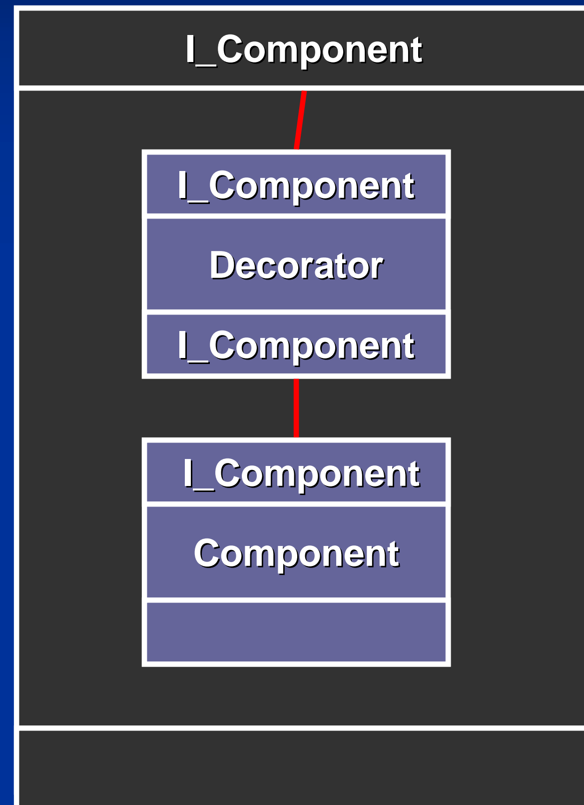
- Example: protect a non-thread-safe component with a lock
- Solution: apply Decorator pattern
- OO approach:
  - one or two abstract classes
  - two derived classes
  - run-time: create objects, links
- OO approach hides static properties of the system




```
obj = new TSDecorator(
          new NTSComponent(…))
res = obj->op(…)
```

Expressing Patterns with Units

- Our solution: apply Decorator at the level of units
  - one interface
  - three components
  - build-time instantiation, connection, and encapsulation
  - build-time constraint checking
- Architecture is clear, enforced, and localized at the component level

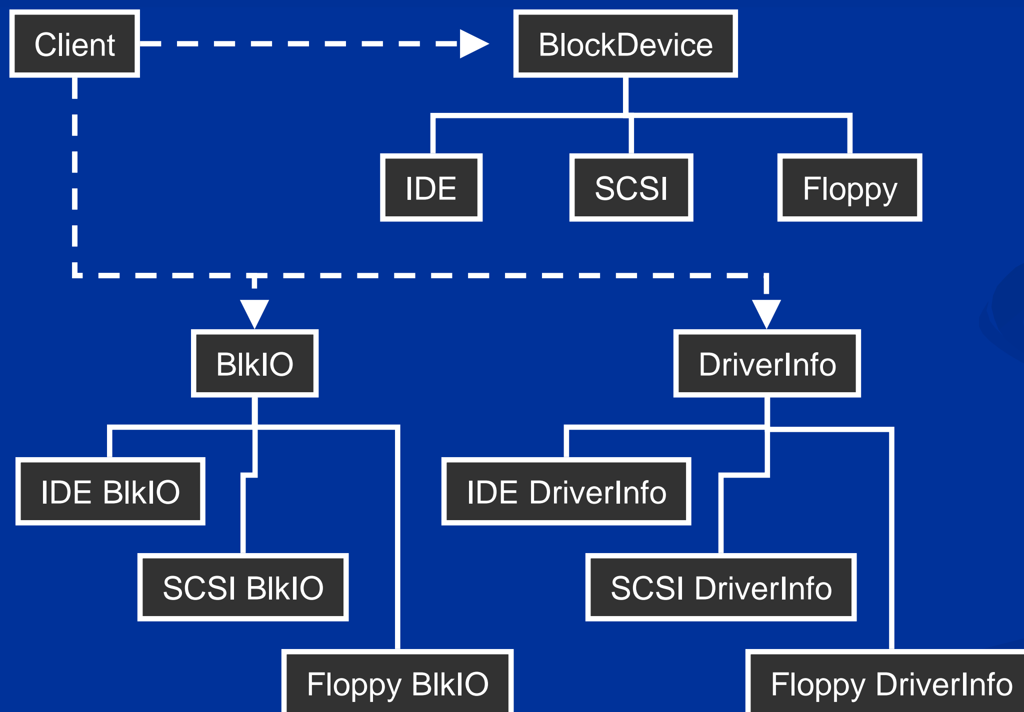


Method for Expressing Patterns

- General task:
  - identify parts of the pattern that correspond to static knowledge
  - "lift" that knowledge out of code
  - realize via unit definitions and connections
- Necessarily specific to individual uses of a pattern
- But, general process can be described as translation from OO description to units

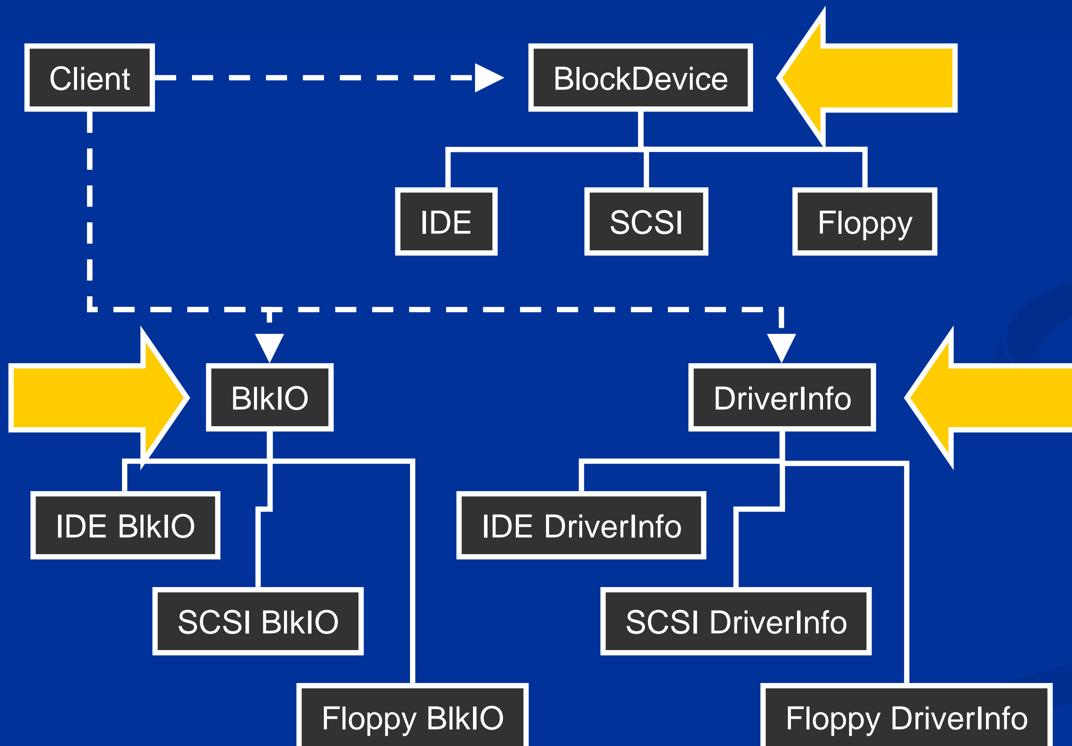
Method for Expressing Patterns

- Example: OSKit block I/O device drivers



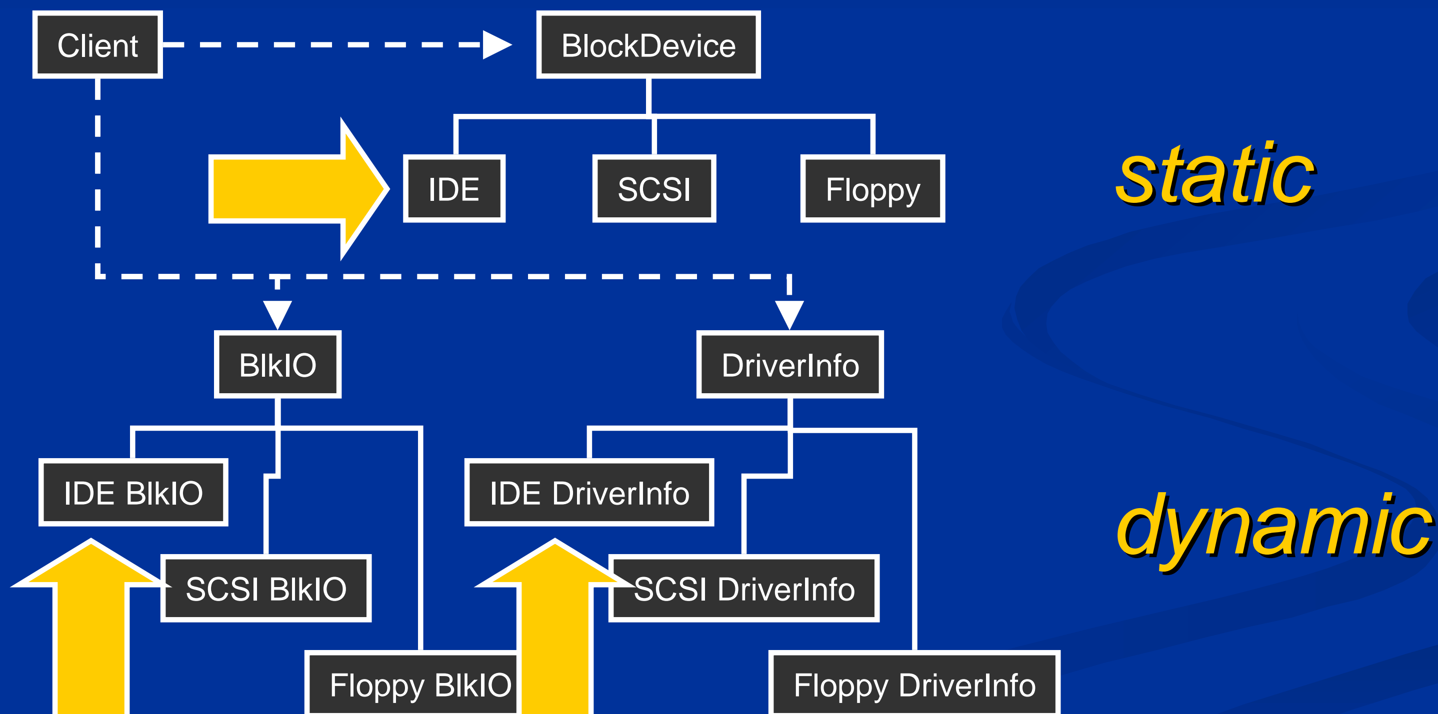
Method for Expressing Patterns

1. Identify the abstract classes/interfaces.



Method for Expressing Patterns

2. Identify *static* and *dynamic* participants.

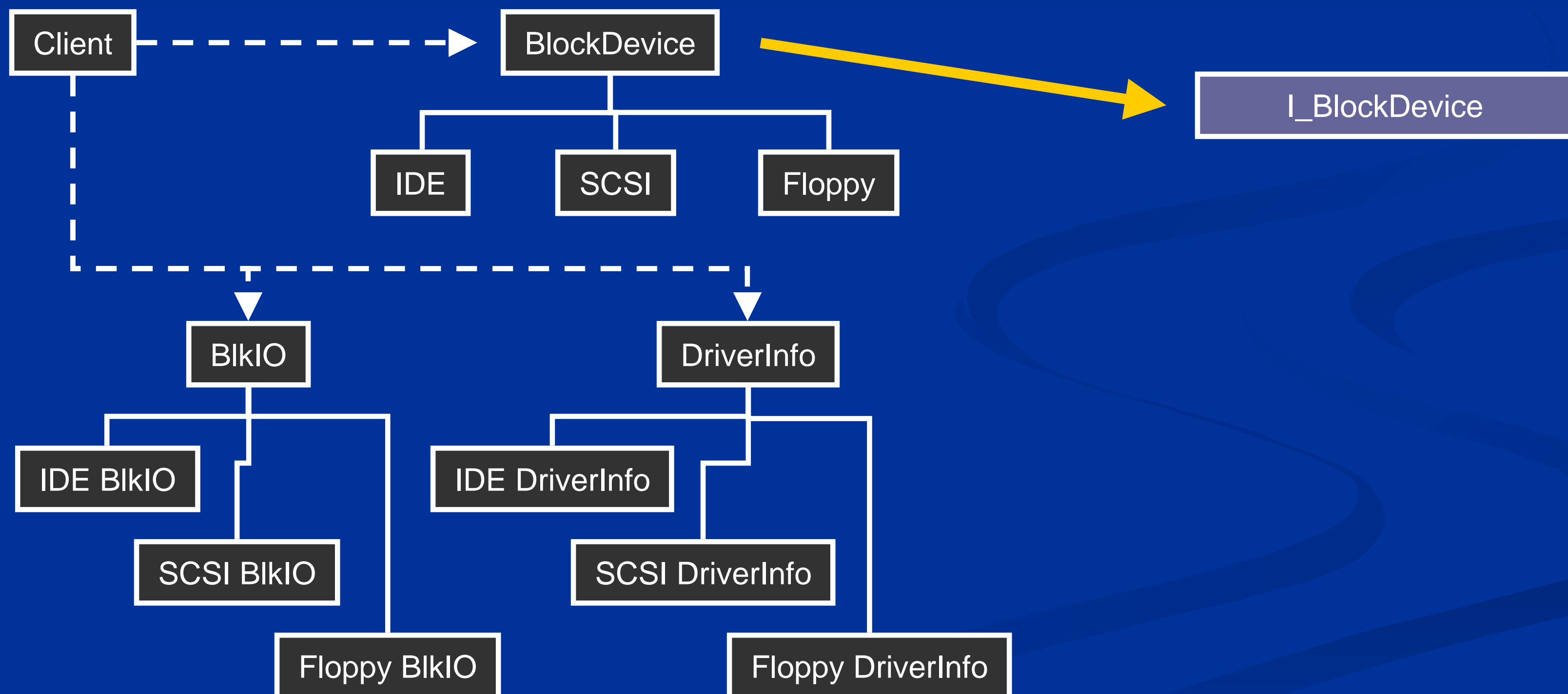


static

dynamic

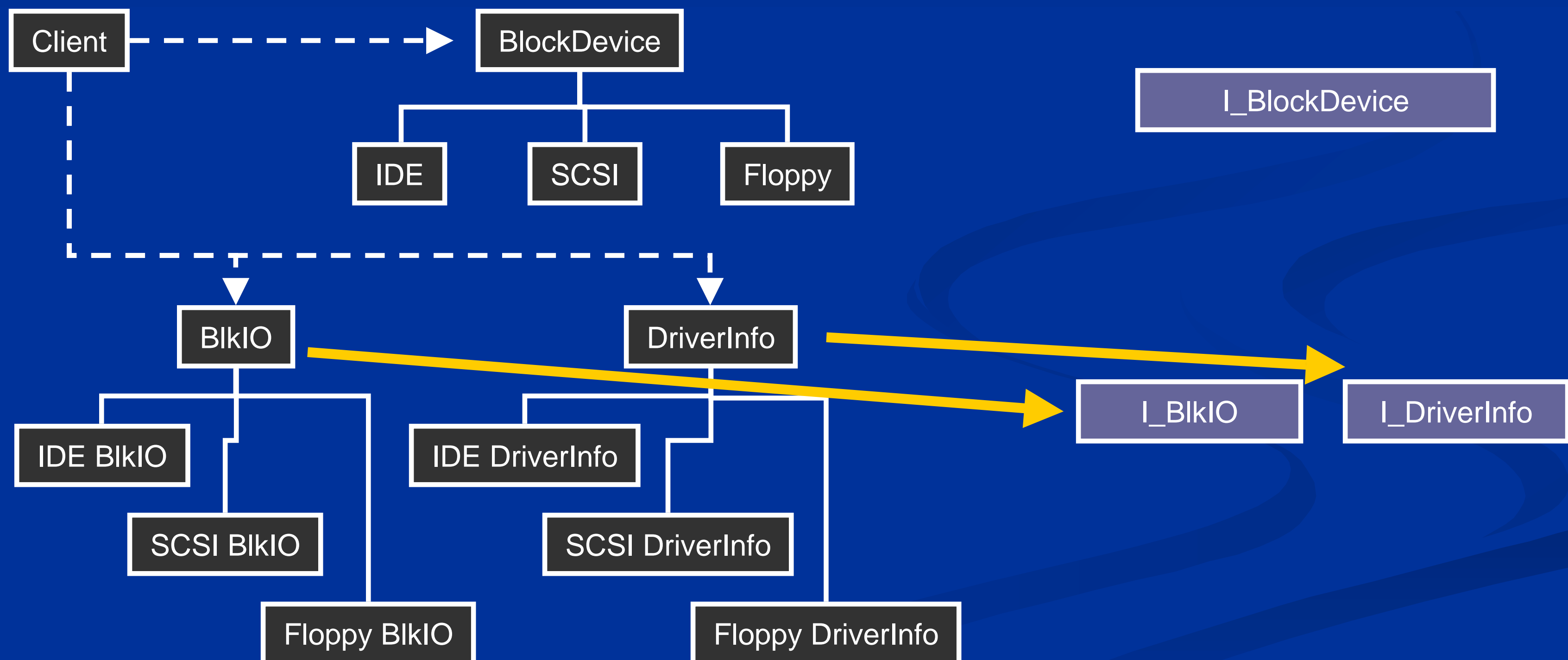
Method for Expressing Patterns

3. Define interfaces for static participants.



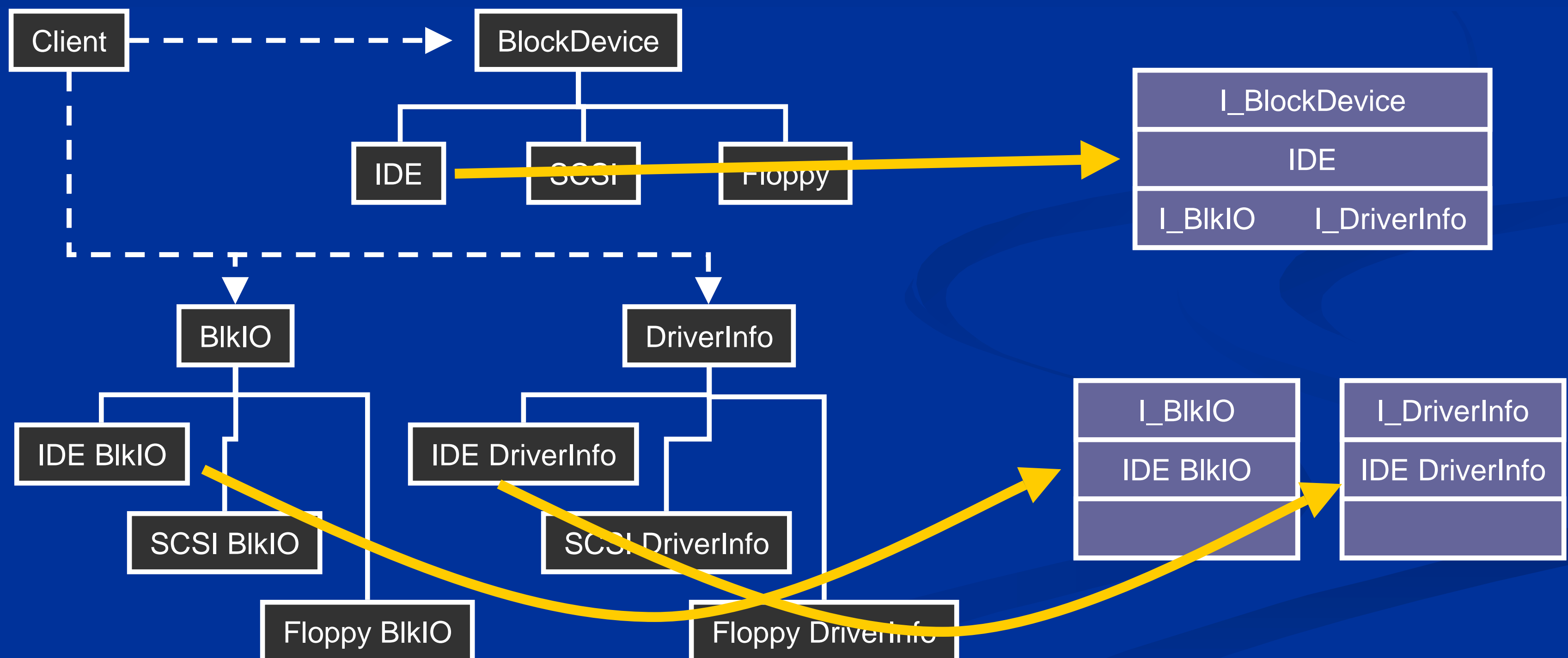
Method for Expressing Patterns

4. Define interfaces for dynamic participants.



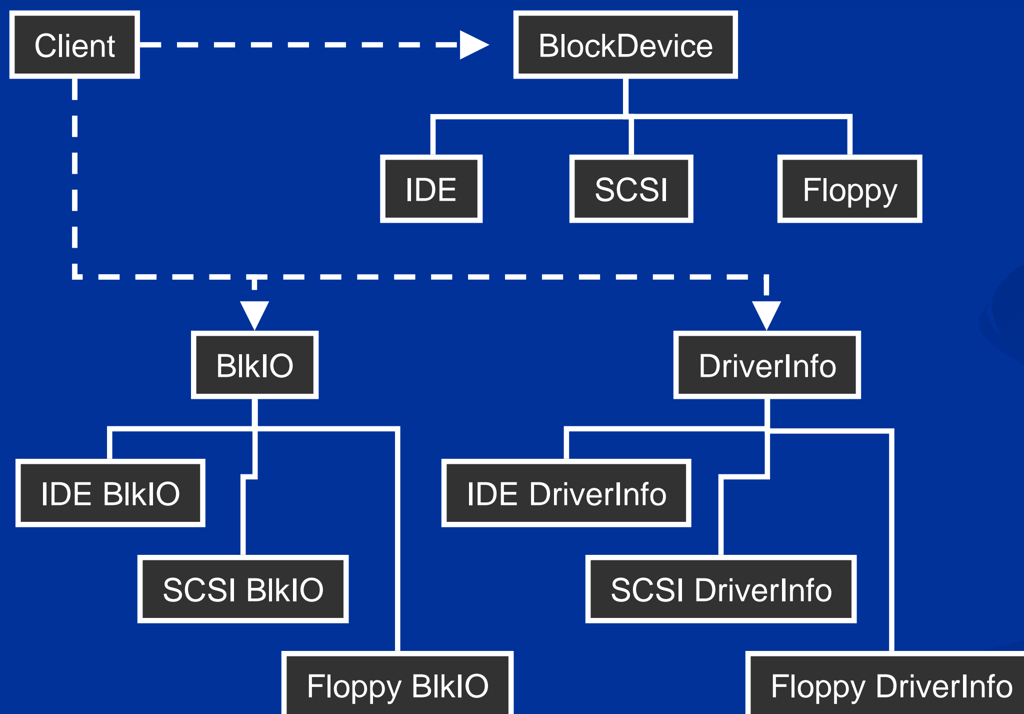
Method for Expressing Patterns

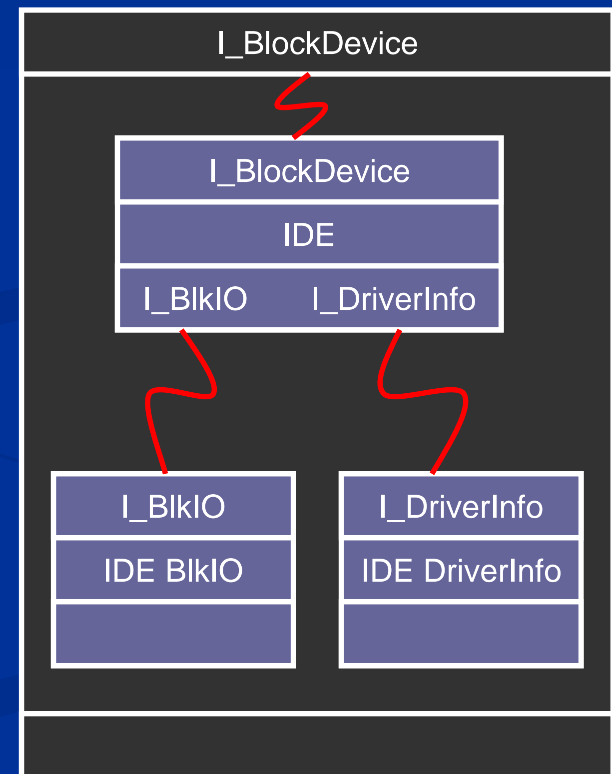
5. Unit defn. for each concrete participant.



Method for Expressing Patterns

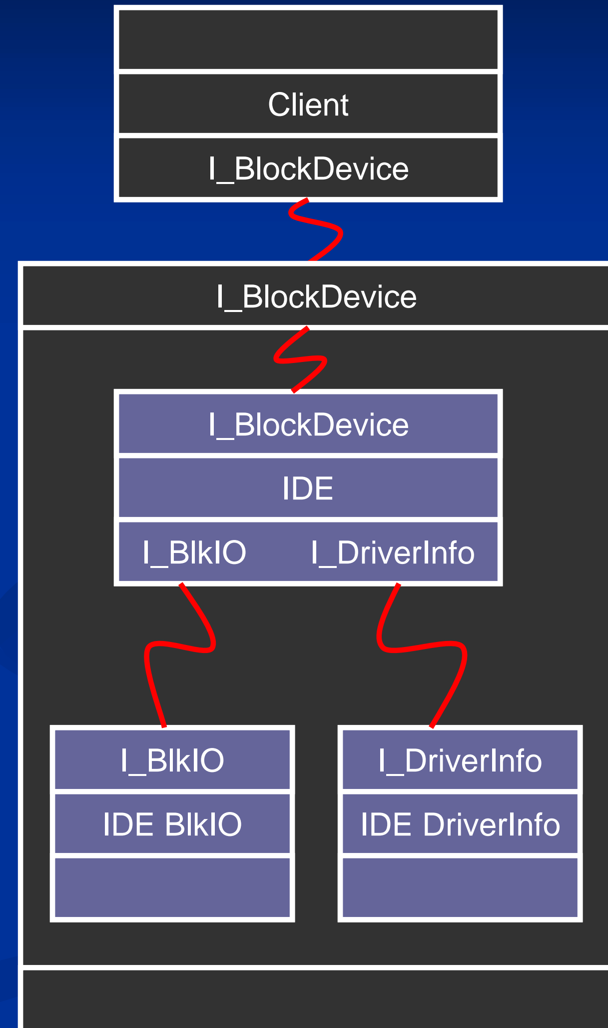
6. Instantiate and connect participant units.





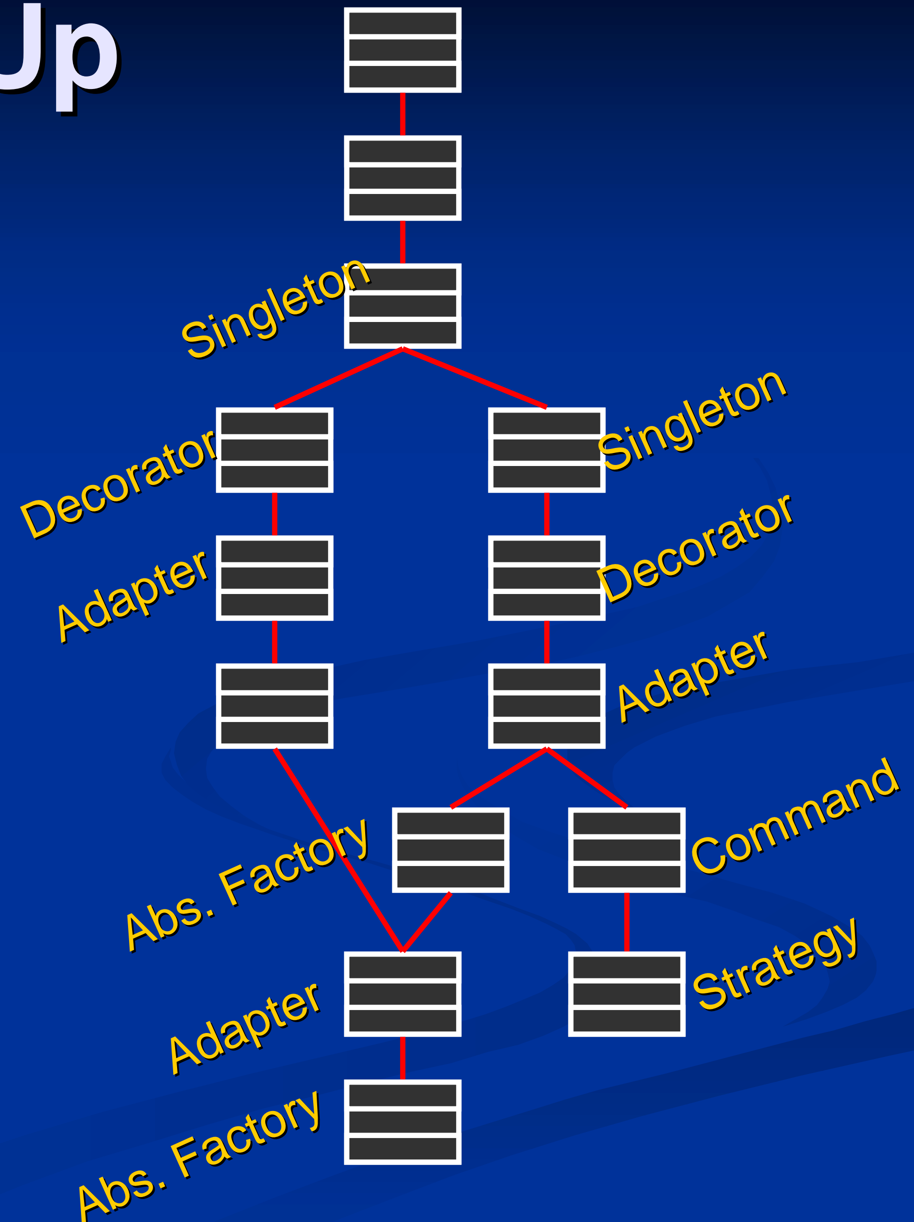
Method Wrap-Up

- Modify as needed for other considerations
  - match existing code, e.g., static/dynamic decision
  - remove participants
  - participants in multiple patterns
  - aggregate parts to simplify



Method Wrap-Up

- More complicated FS example in paper
- Applicable to many (GoF) pattern uses; see paper
- Commonly, much pattern knowledge is static



Analysis

- Static pattern information is located in a single place
  - unit-based specification of the system
  - “resolved” when the system is built
- Unit language is designed specifically for describing components and their linkages
- Pattern realization can be moved out of components’ implementations

Benefits

- Increased opportunities for code reuse
  - disentangled “pattern role” code; multiple patterns
  - applicable when code cannot be changed (legacy)
- Ability to check architectural constraints
  - global, high-level, domain-specific checks
  - checker need not understand the base language
- Enabled performance optimizations
  - cross-component inlining…
  - …enables more significant optimizations

Costs

- Only the static parts of a pattern are specified by our approach
- Participants are committed to being static or dynamic
- Unit descriptions can obscure the differences between patterns
- To achieve full benefits, our approach requires support for the unit model

Related Work

- Language-based approaches
  - LayOM [Bosch, JOOP ’98]
- Metaprogramming-, template-, and macro-based approaches
  - [Marcos et al., ’99]
  - [Alexandrescu, ’01]
  - [Krishnamurthi et al., ESOP ’99]
- ADLs and MILs

Conclusions

- Common pattern applications contain a great deal of exploitable static knowledge
- Separating the static and dynamic parts of design patterns can yield significant benefits
  - improved opportunities for reuse
  - ability to check architectural constraints
  - enabled performance optimizations
- Paper presents a method for obtaining these benefits via the unit model, applicable to:
  - …many existing patterns
  - …imperative, functional, and OO languages

Thanks!

- http://www.cs.utah.edu/flux/