Toward a Formal Theory of Extensible Software

Shriram Krishnamurthi
Matthias Felleisen
Rice University

Background

- Major goals of software engineering include
  - Code reuse
  - Code structure
- There are many informal claims about reusability and patterns of code
- We would like to do better!

Focus: Extensibility

An extensible program can be adapted to new tasks without access to its sources

- no editing
- no copying

(This is an inversion of “black box” terminology)

Why Does Extensibility Matter?

- Reflects important principles:
  - rely on interfaces, not implementation
  - maintain a single point of control
- Captures principles of "software components"
- Represents an important delivery medium

Design Patterns Are Not Enough

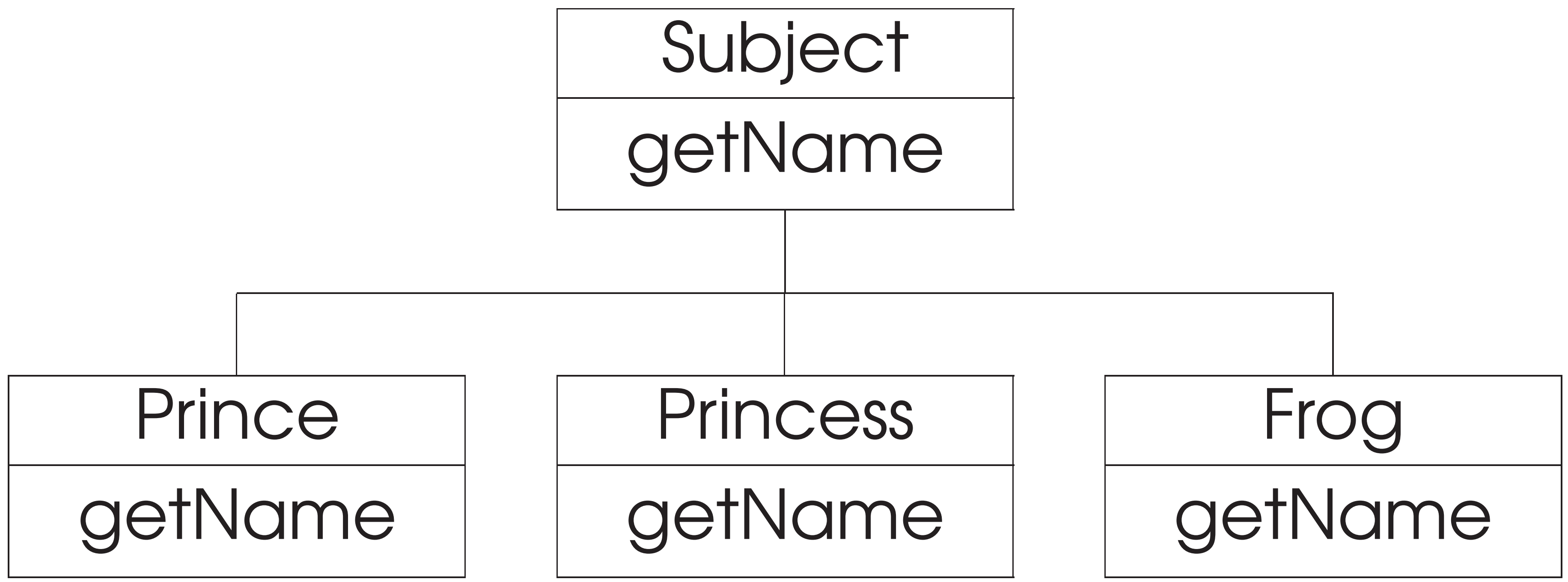
Patterns are primarily a syntactic characterization:

- excessive dependence on language features
- not formalized:
  - difficult to compare
  - cannot be sure of capabilities

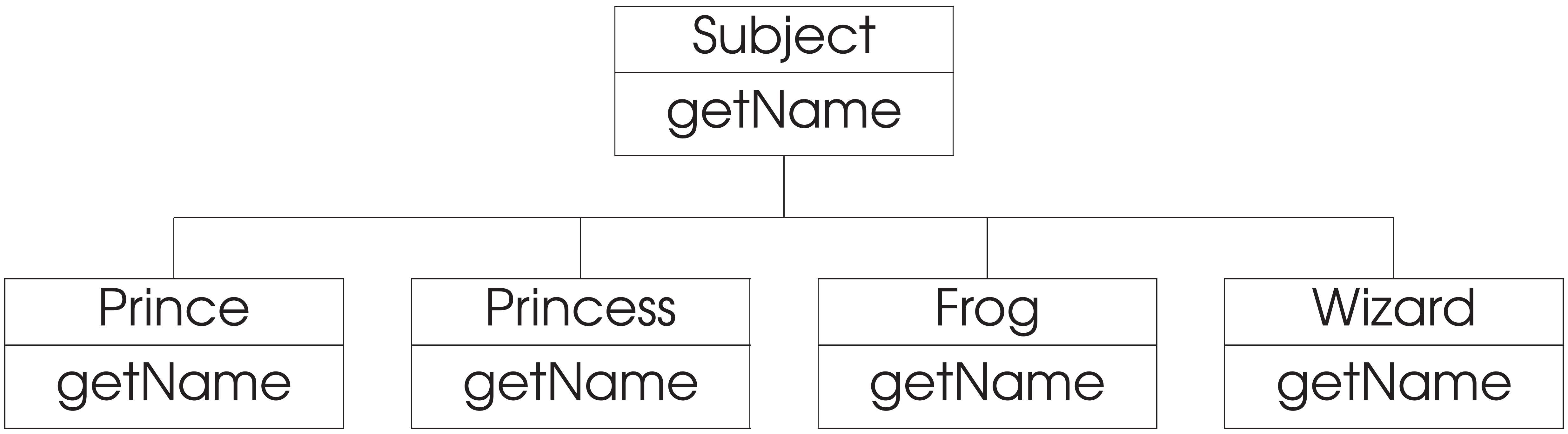
Goals of This Talk

- Clarify what it means to be extensible
- Provide a semantic definition for the concept
- Explore its dependence on classes of programming languages

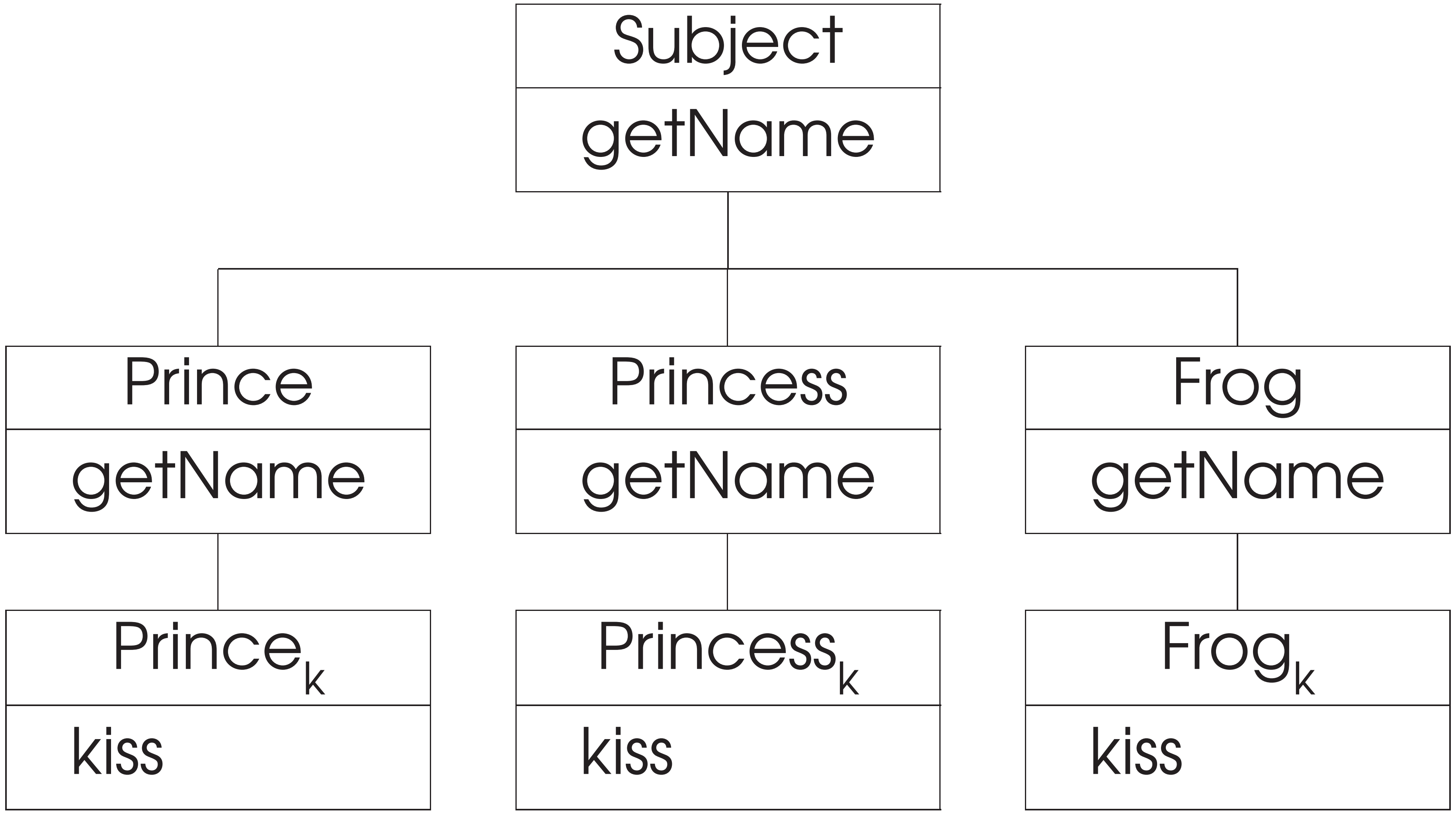
A Fairy Tale Example



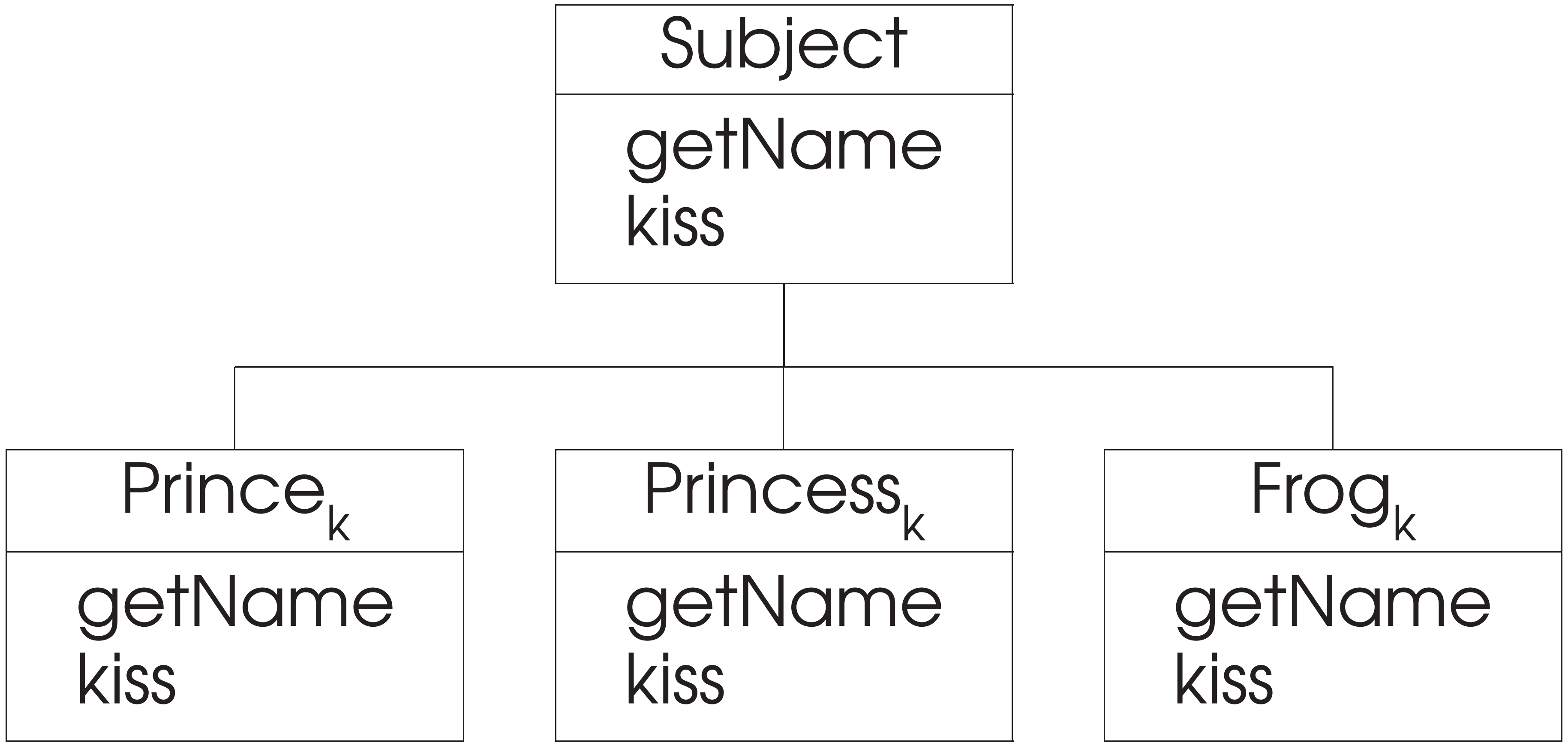
Fairy Tale:
Adding Wizards



Fairy Tale:
Adding Passion Using the Interpreter Pattern



Fairy Tale: Adding Passion by Editing Code



Consolidation

These two versions are syntactically different, but behaviorally similar

The earlier version has a structure that we would like to encourage

How can we compare these versions?

Terminology

- A declaration is a class or interface definition that is well-formed
- A repertoire is a set of declarations
- A program is a (closed) composition of a repertoire with an expression

Operational Semantics, I

- Describes meaning through reductions
- Expressions reduce $\langle e_1, \mathcal{S}_1 \rangle \rightarrow \langle e_2, \mathcal{S}_2 \rangle \rightarrow \cdots$ where $\rightarrow$ represents a "machine" and $\mathcal{S}$ is some "state"
- $\rightarrow$ defines the evaluator Eval

Operational Semantics, II

The key definition is observational equivalence:

$$e_1 \sim e_2 \iff \forall C[], Eval(C[e_1]) \cong Eval(C[e_2])$$

where $\cong$ is Kleene equality (both sides are undefined or both are the same constant)

How This Helps

- $\sim$ is a good notion of program equivalence
- Definitions are independent of syntax
- Possible to compare across languages

Definitions: Extending $\sim$

- Extend $\sim$ to programs by comparing termination behavior
- This in turn induces $\sim$ on repertoires:

  $r_1 \sim r_2$ iff, for all expressions e,
  - $r_1.e$ is a program iff $r_2.e$ is a program, and
  - $r_1.e \sim r_2.e$

Definitions: Syntax vs Semantics

- Containment is a textual, ie, a syntactic notion
- Approximation: the second halts whenever the first one does
- Behavioral Extension: when one repertoire is contained in another and approximates its behavior

Definitions: The Focal Point

Relative Extensibility:

$$\begin{array}{rcccl} & r & \sim & x & \\ \text{“edits”} & \Downarrow & & \downarrow & \text{extension} \\ & r' & \sim & x' & \end{array}$$

Definitions: The Focal Point

Relative Extensibility:

$$\begin{array}{rcccl} & r & \sim & x & \\ \text{“edits”} & \Downarrow & & \downarrow & \text{extension} \\ & r' & \sim & x' & \end{array}$$

For x, r and r', x is an extensible version of r wrt r' if there exists an x' such that

- x' behaviorally extends x
- $x' \sim r'$

Implications

- No constraint on r and r'
- For some x, r and r', there may not exist an x'
- "Extensibility" is meaningless as an "absolute"
- Extensibility is a property of programs, not languages

Application of Definitions

Using these definitions, we can formalize our intuitions about the initial set of programs

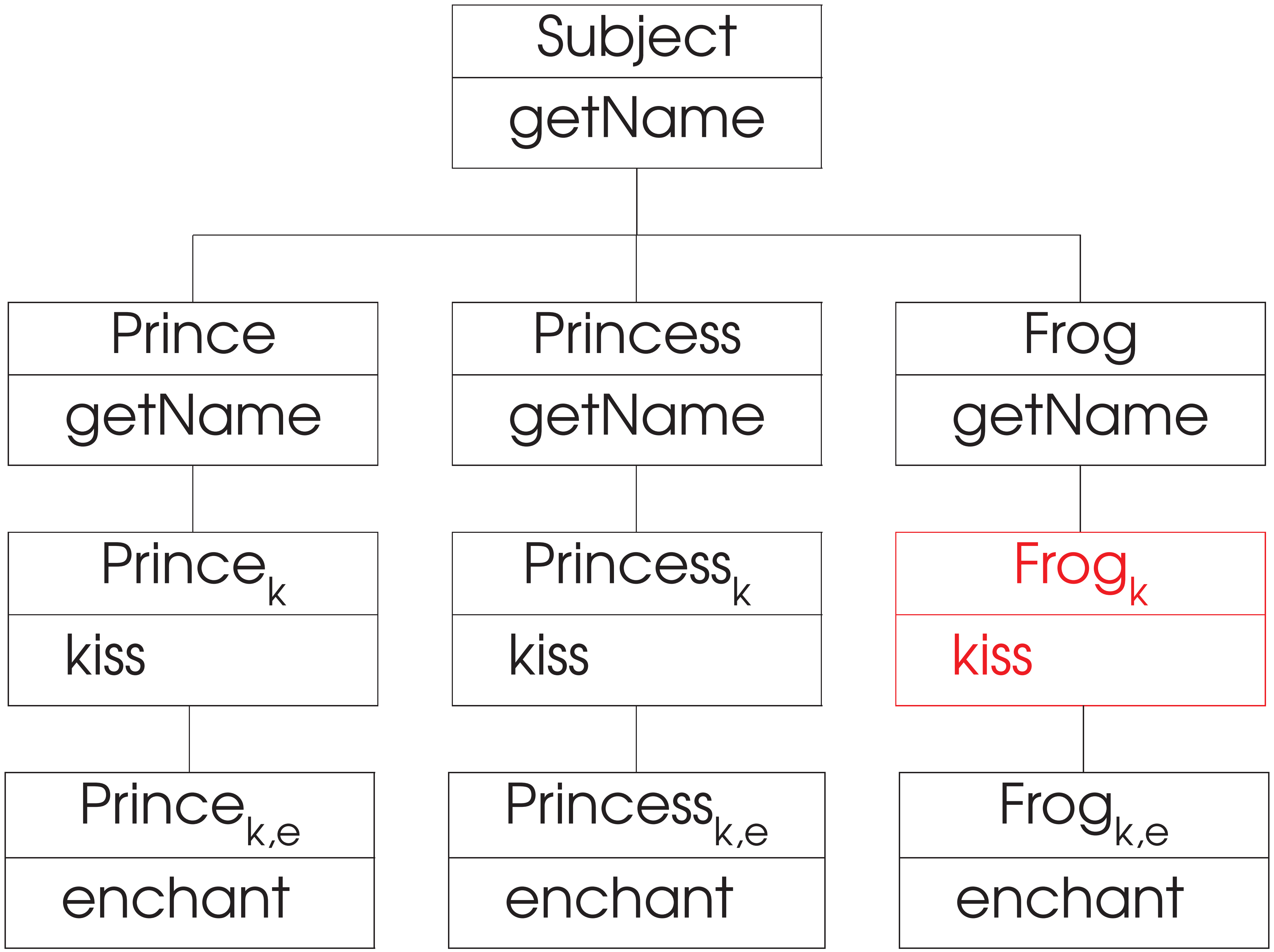
We also find that some intended extensions aren't!

Generalization to Design Patterns

Implications for design patterns:

- given a set of characteristics (such as variants), we can prove that patterns support extensibility along those dimensions
- we can find where patterns break down

Where Extensions Fail



Other Programming Languages

The same definitions carry over to other languages

For functional languages, substitute function and type declarations for classes and interfaces; all other definitions remain unchanged

The First Interpreter

$$\begin{array}{lcl} L & = & \langle \mathrm{num} \rangle \\ & | & L + L \\ & | & L - L \end{array}$$

A standard interpreter for arithmetic cannot accommodate new operators, since the "else" case is fixed

The Second Interpreter

Fix the problem by passing a function that is invoked in the fall-through case; now it seems to handle

$$\begin{array}{rcl} L & = & \langle \mathsf{num} \rangle \\ & | & L + L \\ & | & L - L \\ & | & L * L \end{array}$$

This still has a problem similar to the Interpreter pattern (the same sort of expression, for the same reasons)

The Third Interpreter

The solution to this is related to the OO solution: in this case, code in a variant of the Y combinator (ie, this is a fixed-point problem)

The resulting systems can handle

- new variants (using this pattern)
- new tools (by defining new families of functions)

Implications

Extensibility

- is not the sole province of OO languages
- takes effort with both OOD and FD

Our definitions are stable, not whimsical

Moral: concentrate on the programming pattern!

Contributions

- Formal definition of extensibility
- Semantic footing for future work
- Analysis of some ubiquitous patterns
- Definition independent of classes of languages

Future Work

- Classify "extensions"
  - adaptive programming
  - aspect-oriented programming
- Devise extensibility certificates
- Build program analysis tools