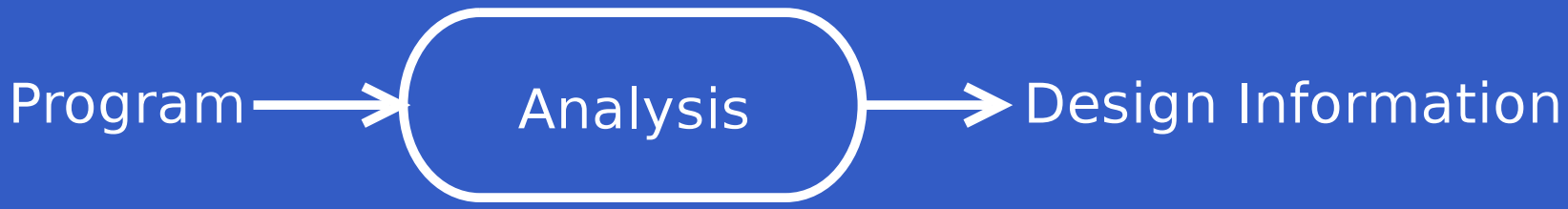
A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information

Patrick Lam and Martin Rinard

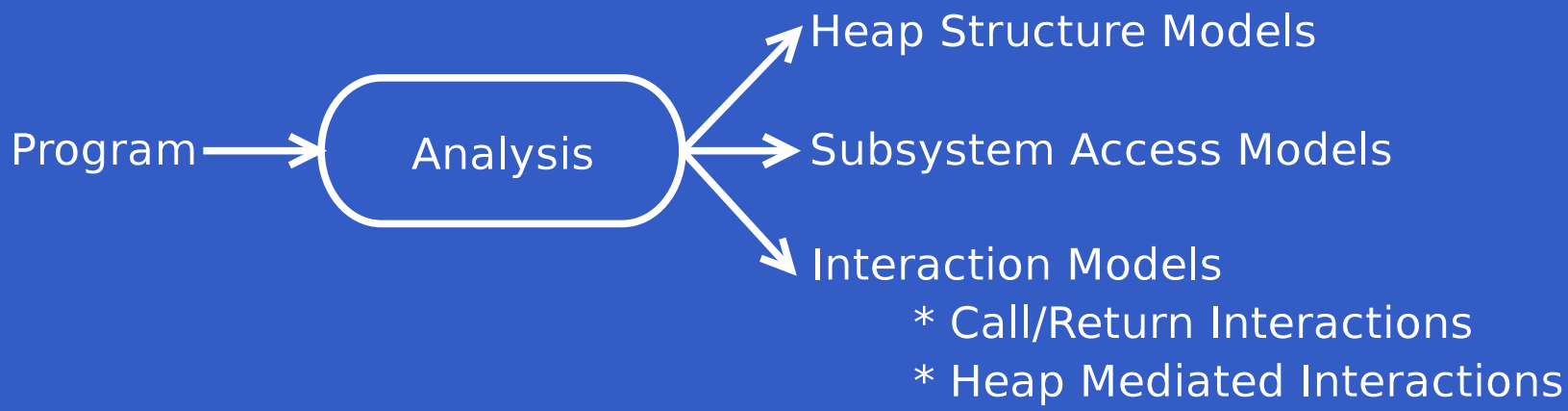
Computer Science and Artificial Intelligence Laboratory

Massachusetts Institute of Technology

Research Goal



Research Goal



Key Issue: Naming Code and Objects

- Objects
  - Unbounded number of objects
  - But we want finite model

  $\Rightarrow$ Need finite abstraction for objects
- Code
  - Huge number of code elements (procedures, statements, etc.)
  - Want comprehensible model

  $\Rightarrow$ Need comprehensible abstraction for code

Outline

- Example
- Experience
- Analysis and Model Extraction
- Related Work & Conclusion

Example: Drawing Program

Subsystems:

- Front-end subsystem
  (processes mouse clicks)
- Engine subsystem
  (builds and displays shapes)

Both use lists to represent data

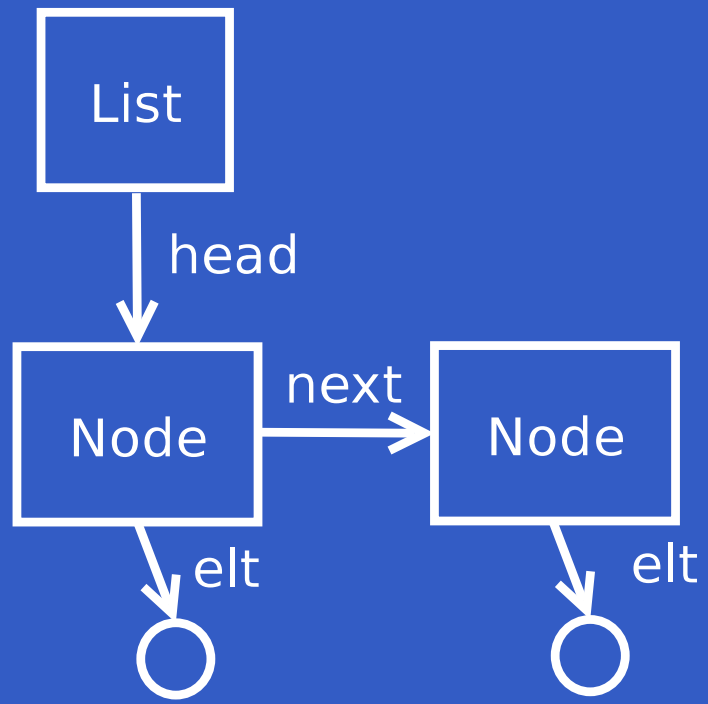
- Front-end subsystem:
  - List of mouse clicks
- Engine subsystem:
  - List of polygons
  - List of polygon vertices

Example: Data Layout

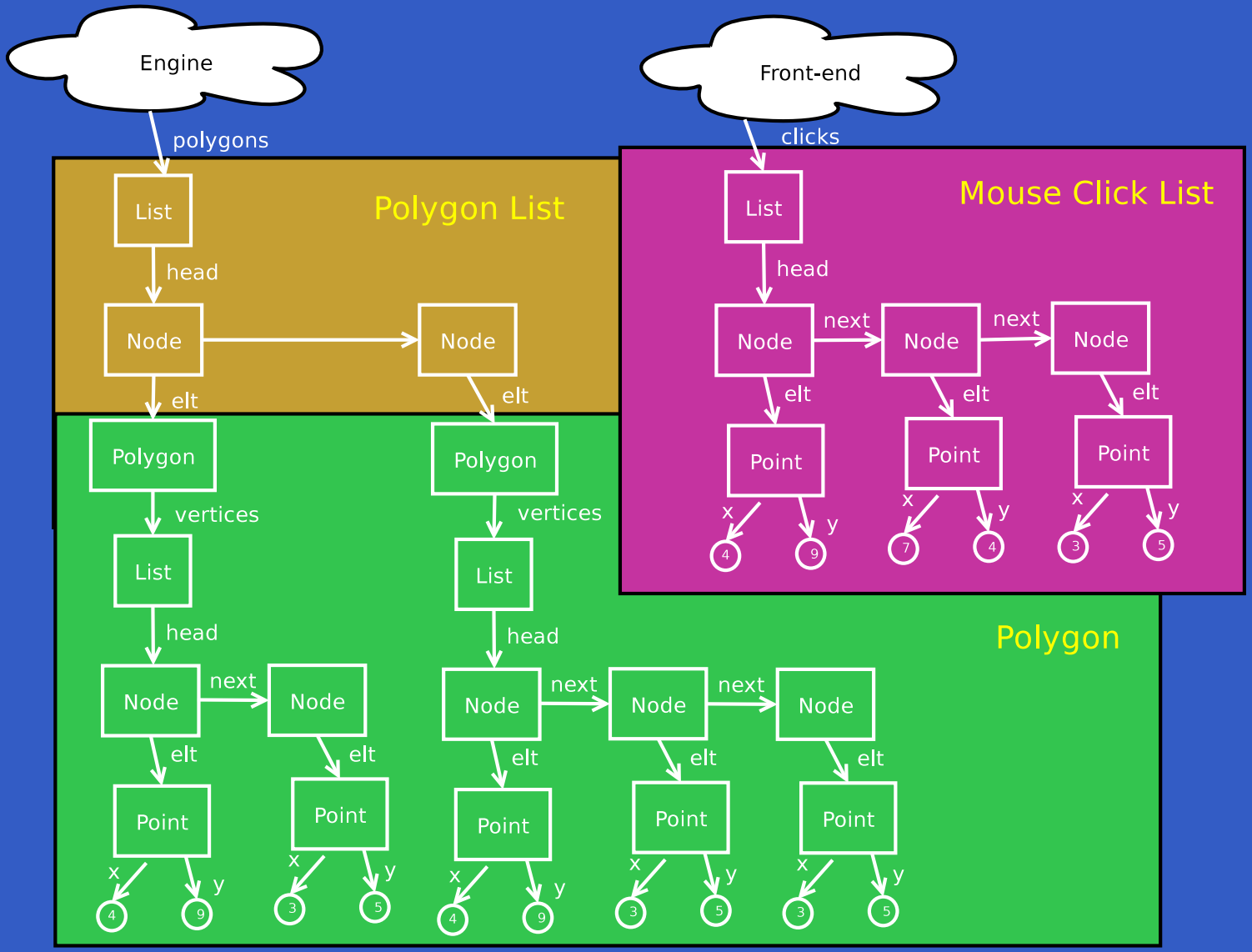
Consider a `List` data structure.

```
class List {
  Node head;
}

class Node {
  Node next;
  Object elt;
}
```

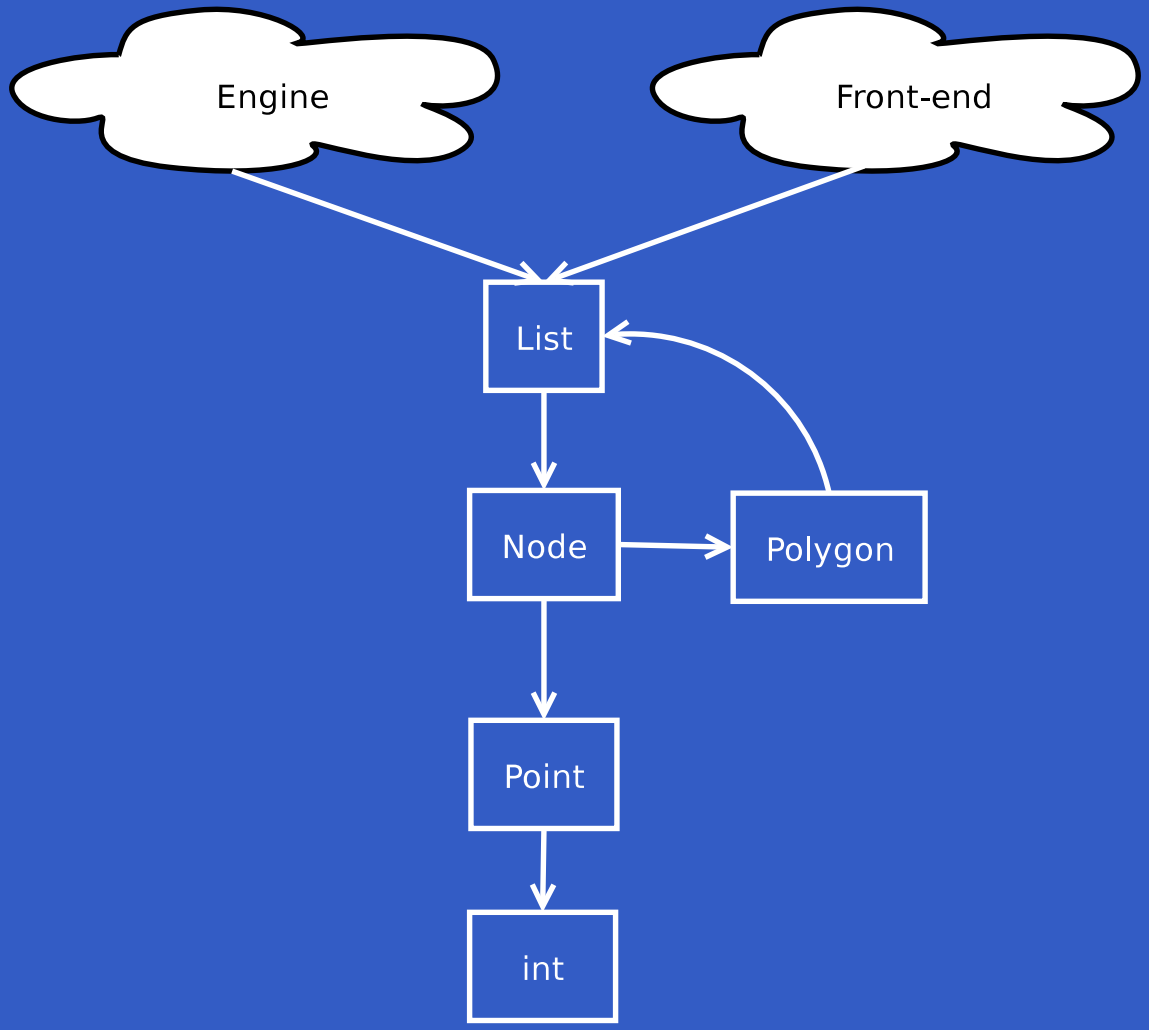


Example: Heap Structure



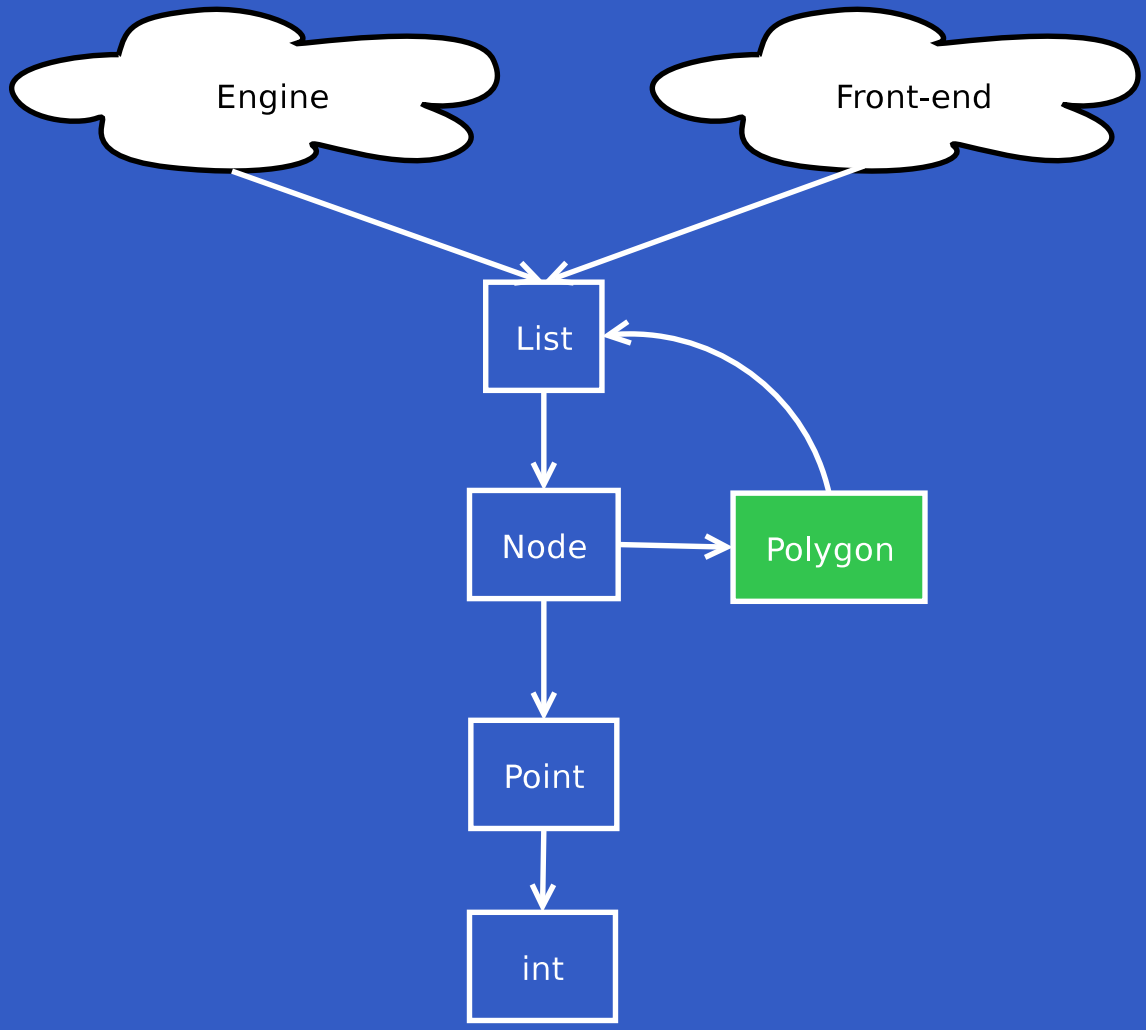
Modelling the Heap Structure

Using classes to abstract objects:



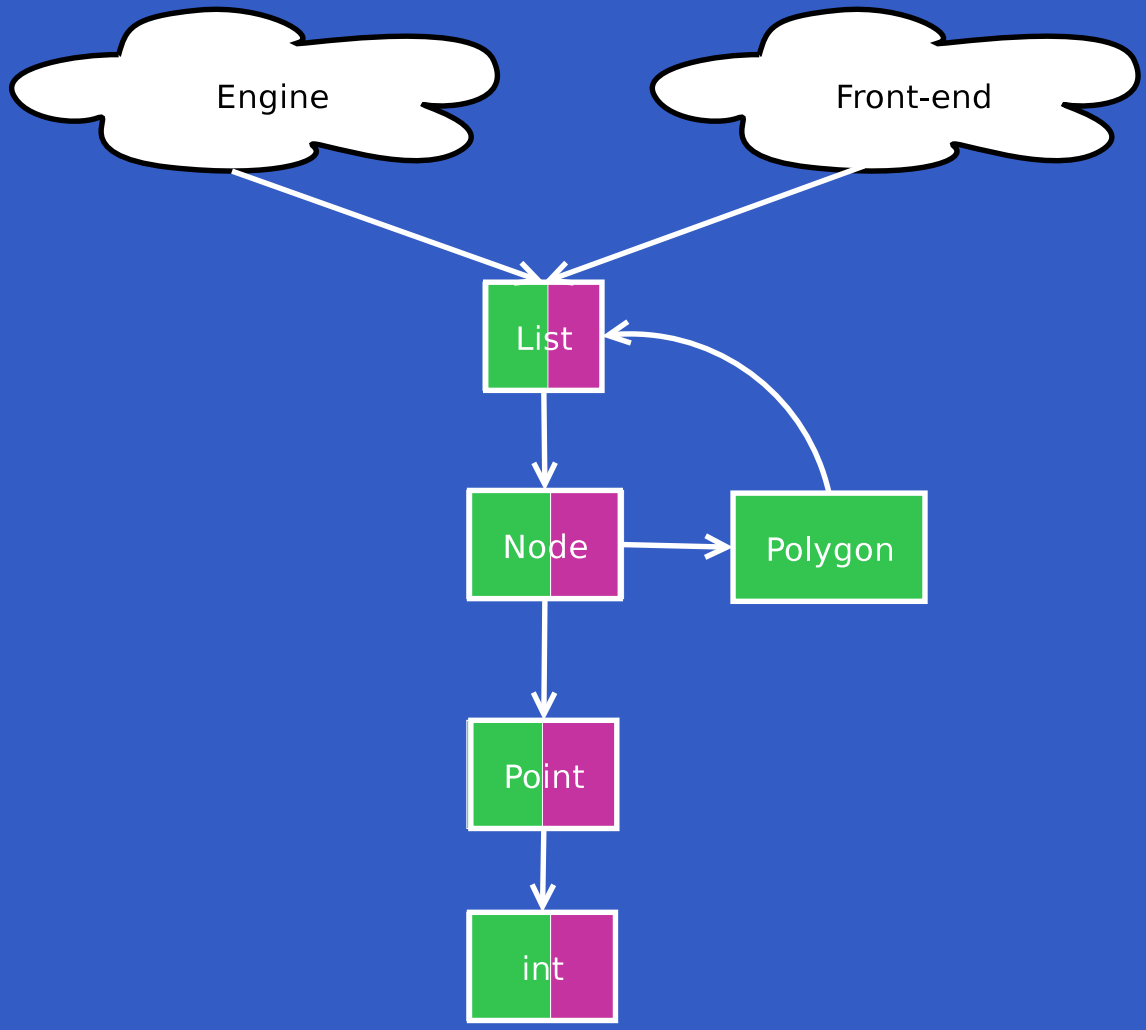
Modelling the Heap Structure

Using classes to abstract objects:



Modelling the Heap Structure

Using classes to abstract objects:

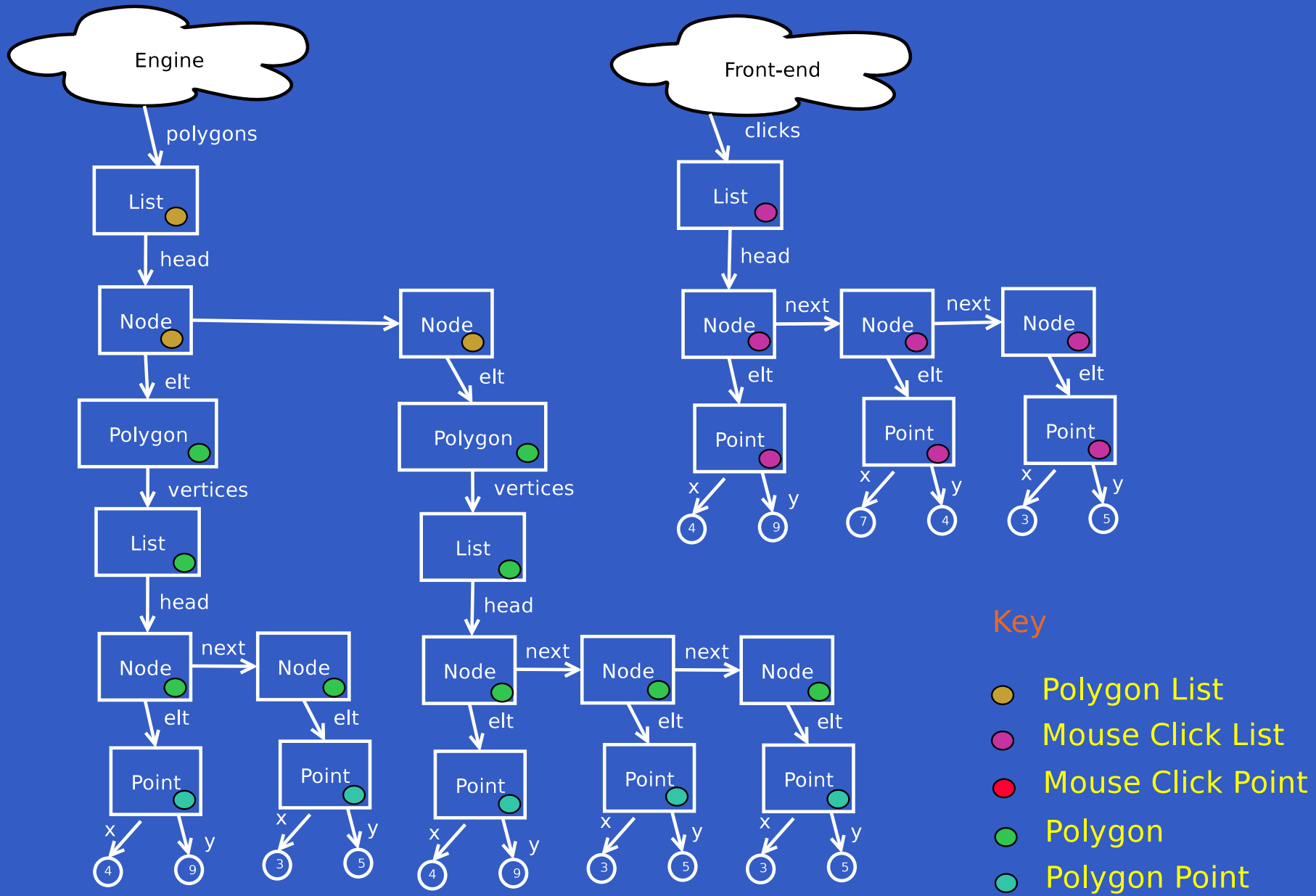


Key Insight

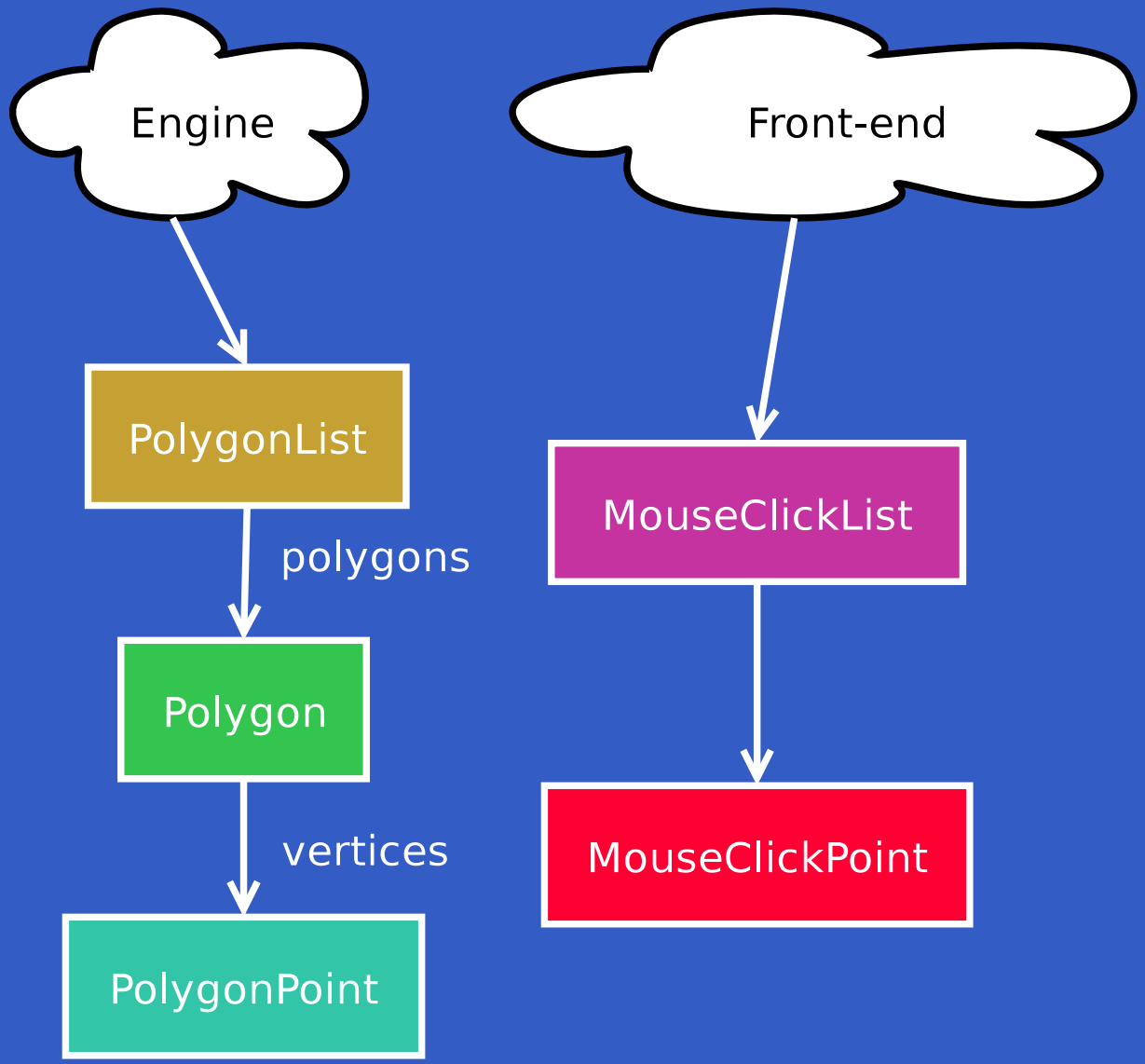
Let the programmer guide abstraction:

- Tokens abstract objects
- Programmer explicitly places tokens onto objects
- Tokens represent objects in the model

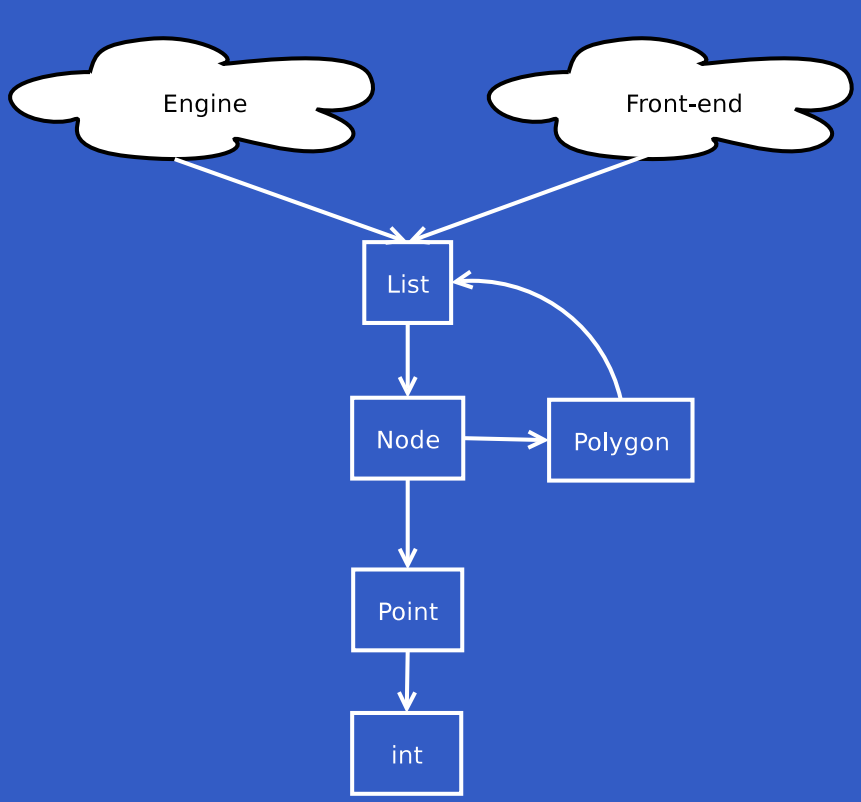
Tokenization of Example

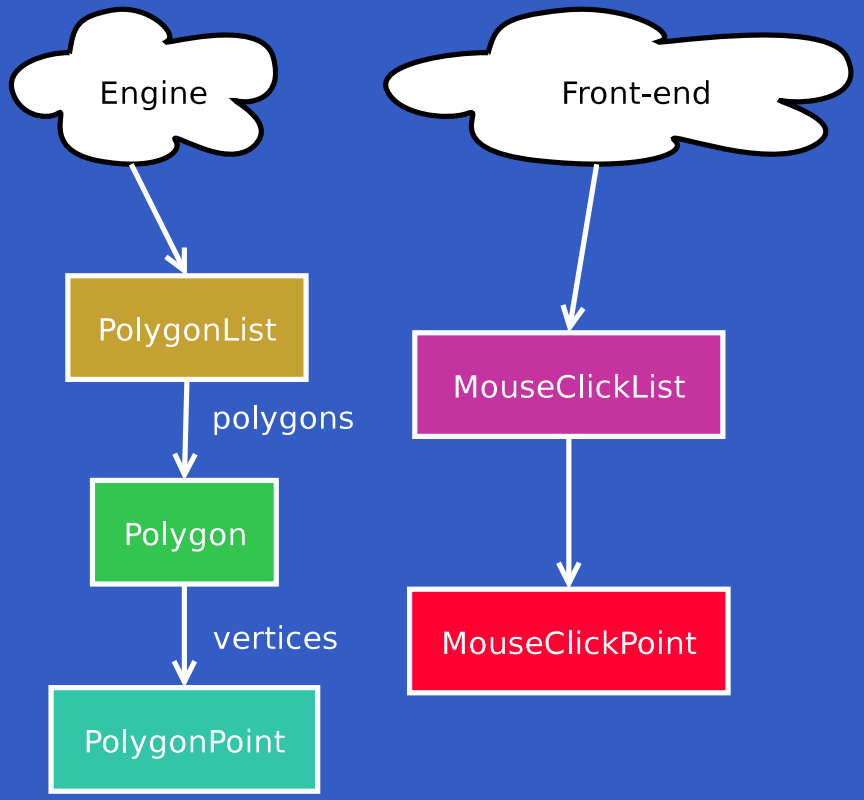


Heap Structure Model with Tokens



Comparing Models





Expressing Model Information in Source

```
class Point<P> { int x, y; }

class List<T,P> { Node<T,P> head; }
class Node<T,P>
  { Node<T,P> next; Object<P> elt; }

class Polygon<S>
  { List<S,PolygonPt> vertices; }

class Engine {
  List<PolygonList,Polygon> polygons;
}
class EventHandler {
  List<ClickList,ClickPt> clicks;
}
```

Mismatch Between Code and Model Elements

Multiple code elements for one model element.

- Caused by implementation complexity
- Model needs to abstract away from complexity

Multiple model elements for one code element.

- Caused by (desirable!) reuse
- Different instances of same class in different contexts
- Model needs to capture conceptual distinctions, not implementation distinctions

Mismatches are inevitable and desirable:

Code and model have different concerns.

Problems modelling code by method

An obvious abstraction is to model code at the granularity of methods.
Too fine-grained:

- We don't want to see every method in every subsystem; need coarser granularity.

Too coarse-grained:

- Shared libraries, and other shared code, execute in multiple contexts;
- Should be part of the context that invokes them, not their own context.

Code in Example

```
class FrontEnd {
 List clicks;

 receiveClick(Point p) {
  if (relevant(p))
   clicks.insert(p);
  ...   }

 processClickQueue() {
  ...
  while (l.canIterate()) {
   x = clicks.iterate();
   doClick(x);
  } } }
```

```
class Engine {
 List polys;
 List genPoly() {
  ...
 polys.insert();
}

 displayPoly(List vx) {
  Point p, oldP;
  while (vx.canIterate()) {
  p = vx.iterate();
  if (oldP) drawLine(p, oldP);
} } }
```

```
class List {
    Node head,ptr;
    insert(Object o) {...}
    startIteration() {ptr=head;}
    canIterate() {return ptr == null;}

    Object iterate() { ptr=ptr.next; }
}
```

Code in Example

```
class FrontEnd {
 List clicks;

 receiveClick(Point p) {
  if (relevant(p))
   clicks.insert(p);
  ...   }

 processClickQueue() {
  ...
  while (l.canIterate()) {
   x = clicks.iterate();
   doClick(x);
  } } }
```

```
class Engine {
 List polys;
 List genPoly() {
  ...
 polys.insert();
}

 displayPoly(List vx) {
  Point p, oldP;
  while (vx.canIterate()) {
  p = vx.iterate();
  if (oldP) drawLine(p, oldP);
} } }
```

```
class List {
    Node head,ptr;
    insert(Object o) {...}
    startIteration() {ptr=head;}
    canIterate() {return ptr == null;}

    Object iterate() { ptr=ptr.next;}
}
```

Code in Example

```
class FrontEnd {
 List clicks;

 receiveClick(Point p) {
  if (relevant(p))
   clicks.insert(p);
  ...   }

 processClickQueue() {
  ...
  while (l.canIterate()) {
   x = clicks.iterate();
   doClick(x);
  } } }
```

```
class Engine {
 List polys;
 List genPoly() {
  ...
 polys.insert();
}

 displayPoly(List vx) {
  Point p, oldP;
  while (vx.canIterate()) {
  p = vx.iterate();
  if (oldP) drawLine(p, oldP);
} } }
```

```
class List {
    Node head,ptr;
    insert(Object o) {...}
    startIteration() {ptr=head;}
    canIterate() {return ptr == null;}

    Object iterate() { ptr=ptr.next;}
}
```

Subsystems

Concept of subsystem

- Set of method invocations that serve same conceptual purpose in computation

Concept of subsystem entry point

- Some classes identified as subsystem entry points
- Subsystem does not change until invocation of method in another subsystem entry point class

Invocation of different methods may be in same subsystem
Different invocations of one method may be in different subsystems.

Mediating the Mismatch

Mismatch between code and model elements.

- Tokens and component identifiers allow developer to guide abstraction:
  - Can merge objects (with tokens)
  - Can merge methods (with subsystems)
- Token parameterization enables model to separate instances of shared classes.

  Subsystem entry points enable model to separate different invocations of shared methods.

Our Models

We use our program element abstraction to produce the following models:

- Heap Structure Model
- Subsystem Access Model
- Call/Return Interaction Model
- Heap Interaction Model

Outline

- Example
- Experience
- Analysis and Model Extraction
- Related Work & Conclusion

Experience

Implemented as extension to Polyglot
(Andrew Myers, Cornell University)

Tested on Tagger application

- Text formatting system (Daniel Jackson)
- Translates Tagger documents into Quark
- 1721 lines of code
- 14 classes in application

Development overhead

- Changed 201 lines of code
- Final version 1755 lines of code

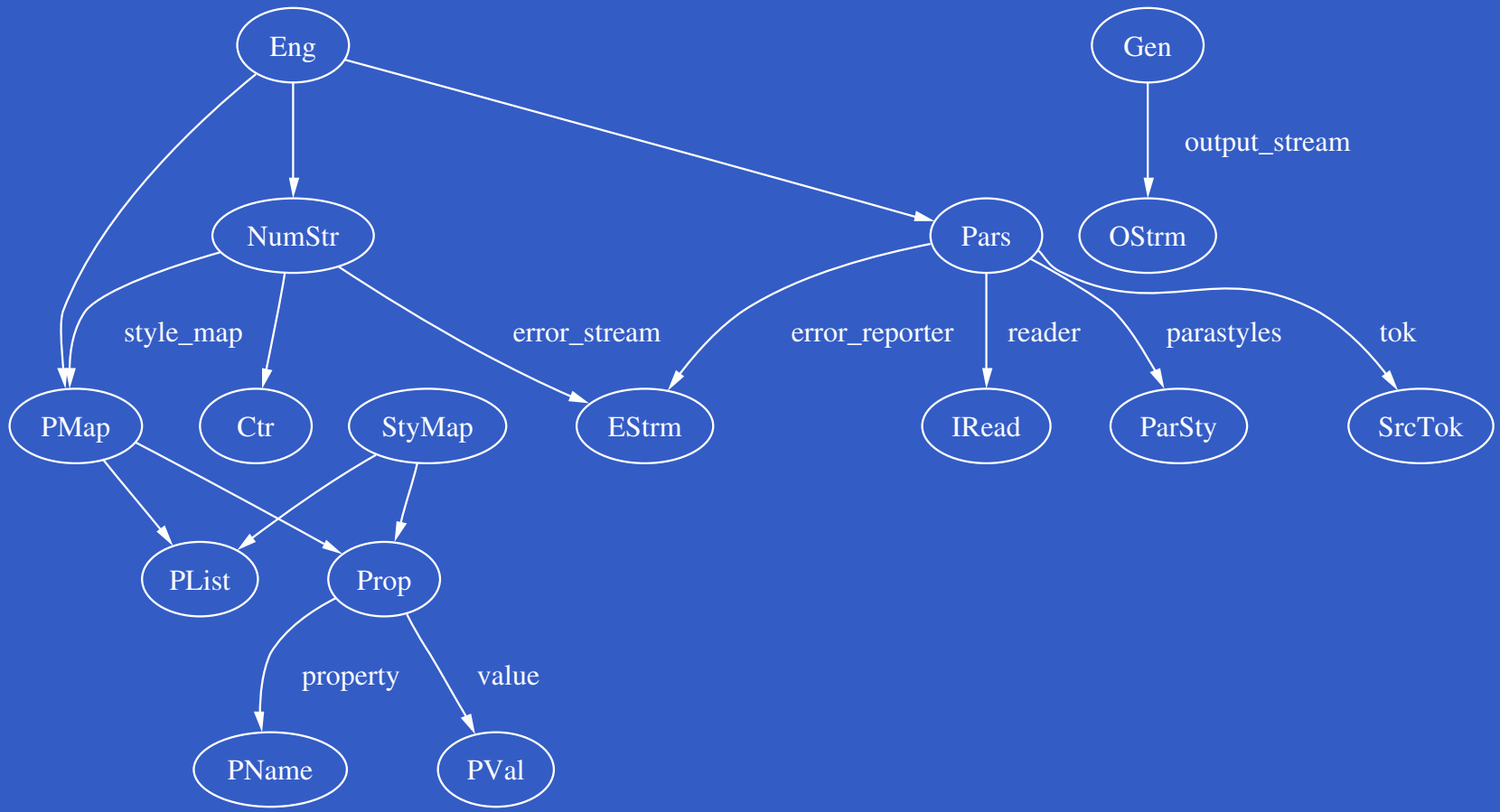
Subsystems

Pars	Parser for Tagger document
PMap	Property Management
Act	Translates Tagger commands to Quark
Gen	Generates Quark document
Eng	Dispatches Tagger commands to Act
Main	Initializes, connects subsystems

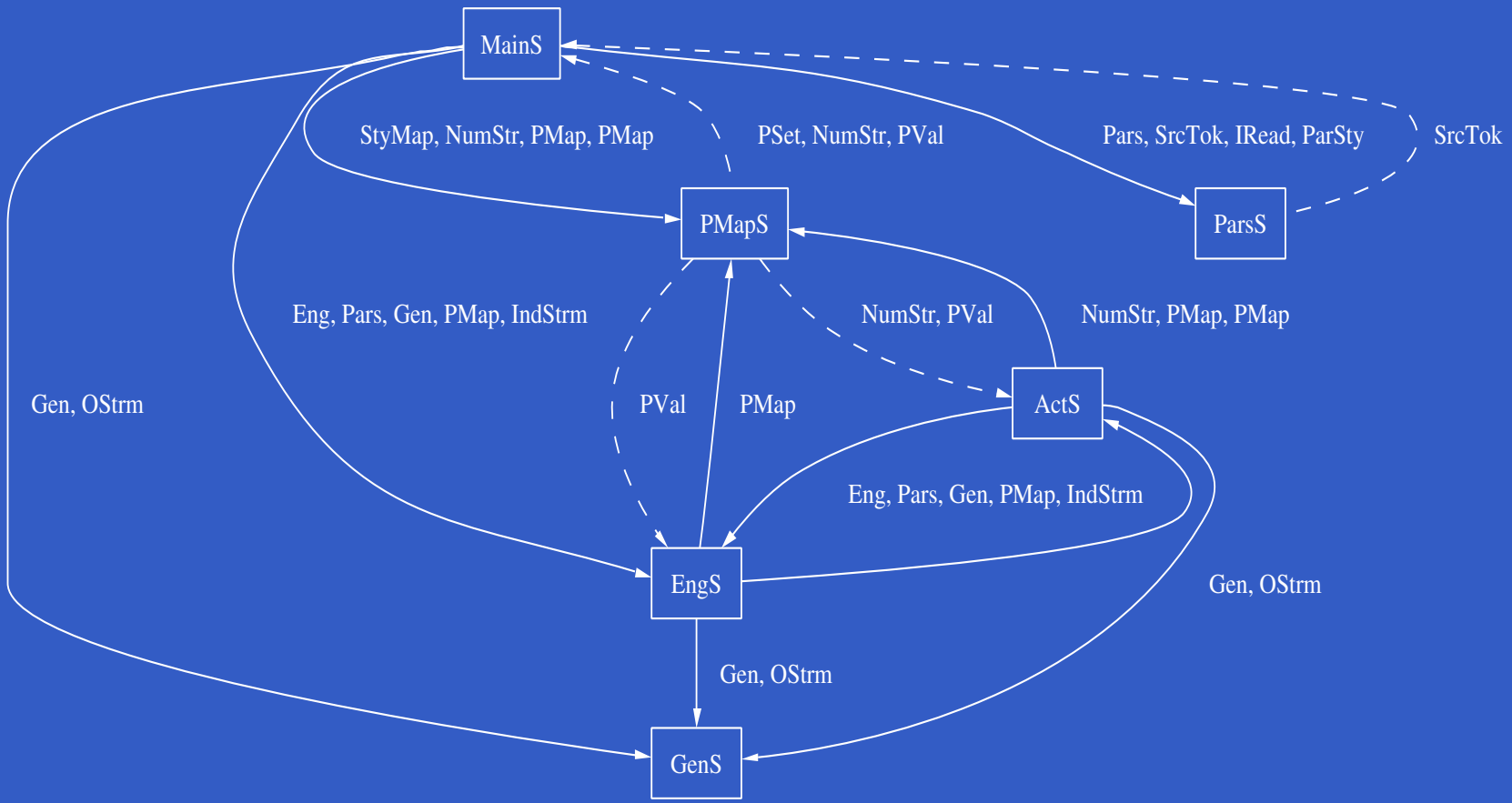
Tokens

Gen, Eng, Pars	(objects for subsystems)
NumStr, Ctr	(list counters in Tagger source)
SrcTok	(parser tokens)
PMap, PList	(text properties
PName, PVal	and maps thereof)
StyMap, ParSty	(style information)
IRead	(input stream object)
EStrm, OStrm	(error and system output)

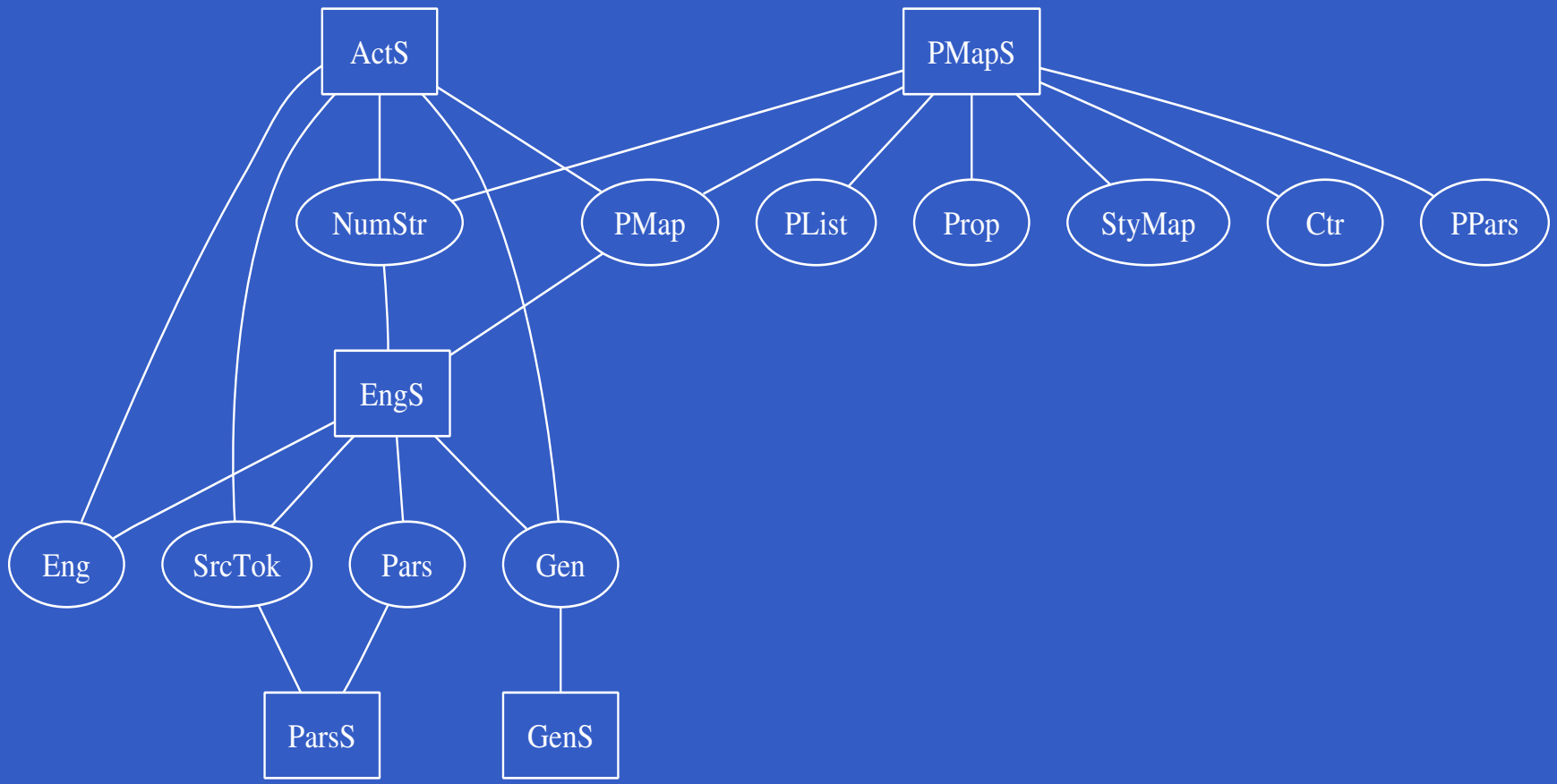
Tagger: Heap Structure Model



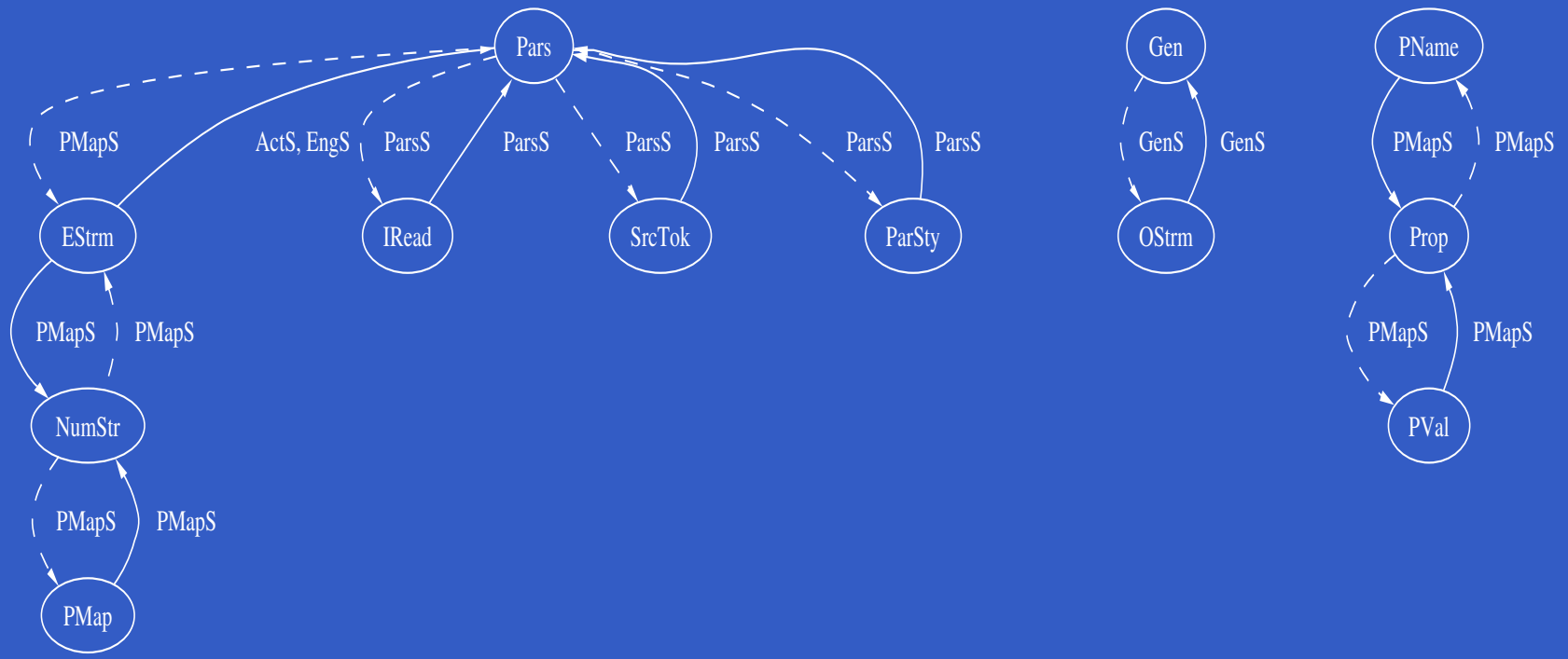
Tagger: Call/Return Interaction Model



Tagger: Subsystem Access Model



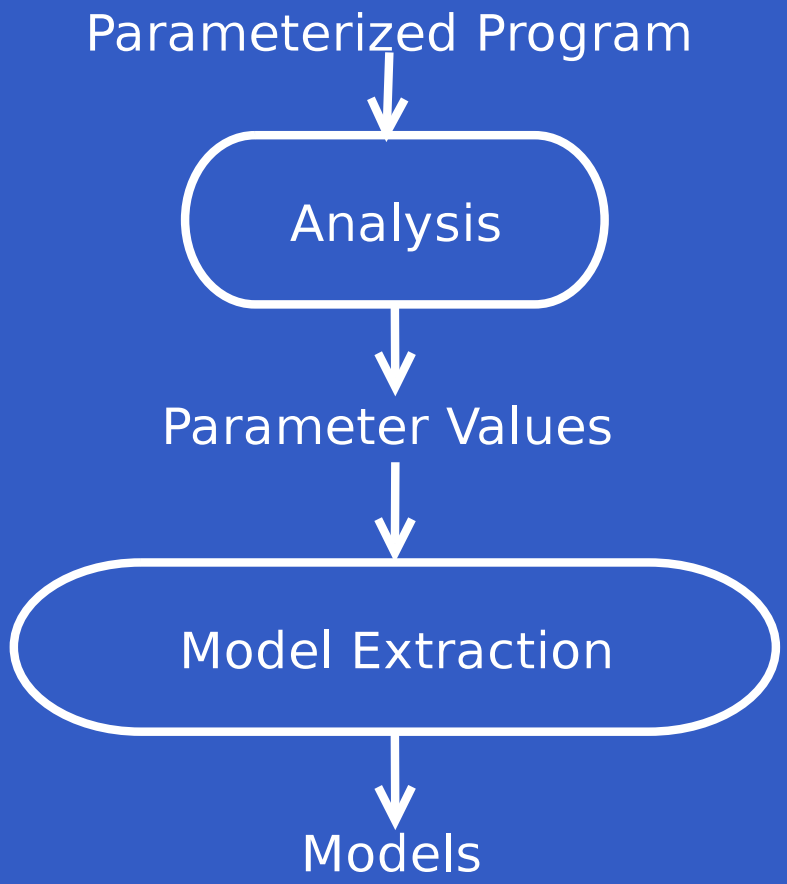
Tagger: Heap Interaction Model



Outline

- Example
- Experience
- Analysis and Model Extraction
- Related Work & Conclusion

Analysis and Model Extraction



Analysis Algorithm

Algorithm traverses call graph; for each call site:

- Compute contexts—combinations of parameter values.

Properties of analysis:

- Top-down, context-sensitive algorithm.
- Analyzes callers before callees
- Each context specifies combination of parameter values for instantiation
- One context for each distinct use

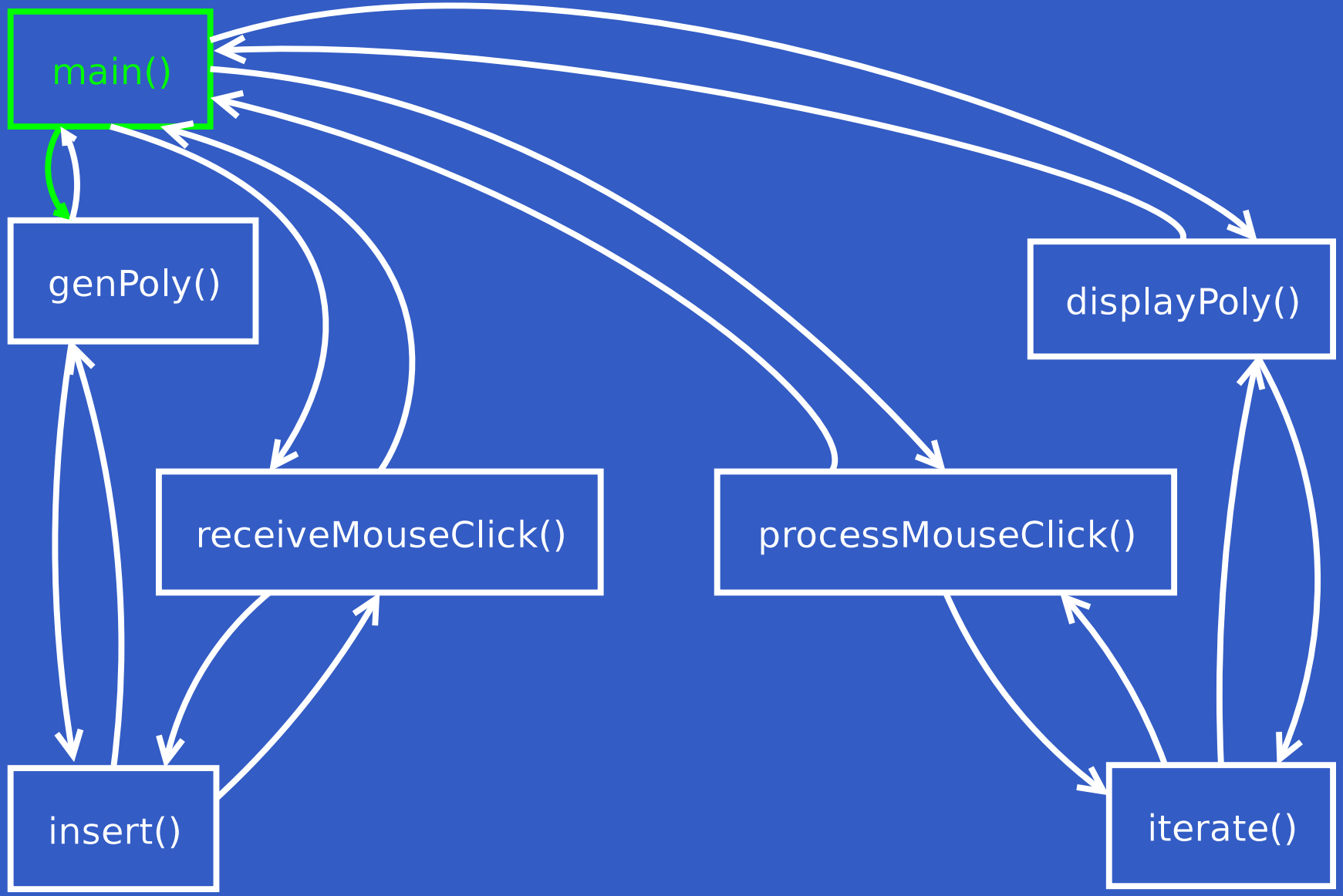
What is a Context?

For each code element (method), we track a set of tuples:

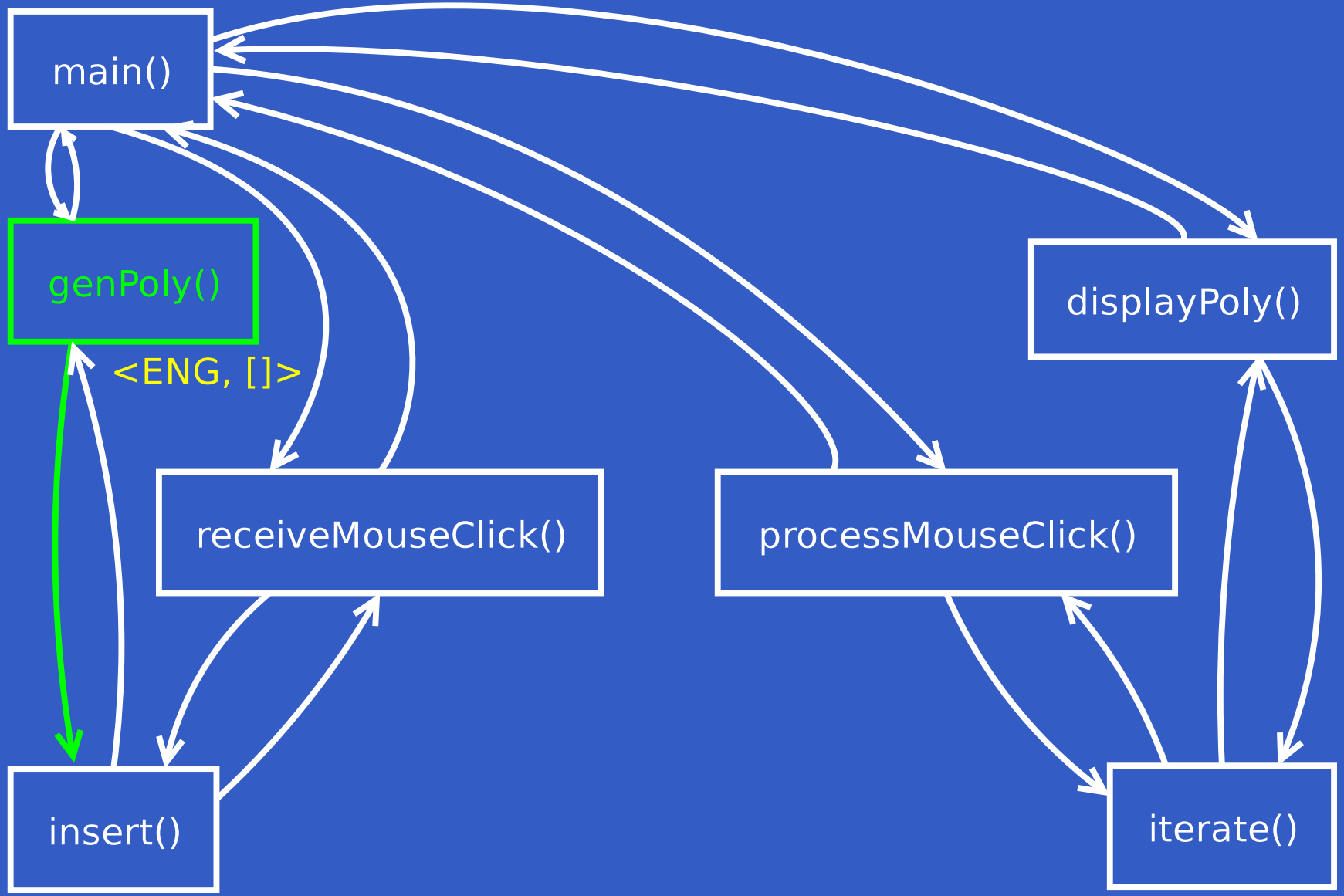
$$\langle \mathsf{Sub}, \mathsf{TMap} \rangle$$

- Sub is the subsystem which called this instance of the method.
- TMap is a map from token formals to token actuals e.g. $[t \mapsto \mathsf{Polygon}, p \mapsto \mathsf{PolygonPt}]$

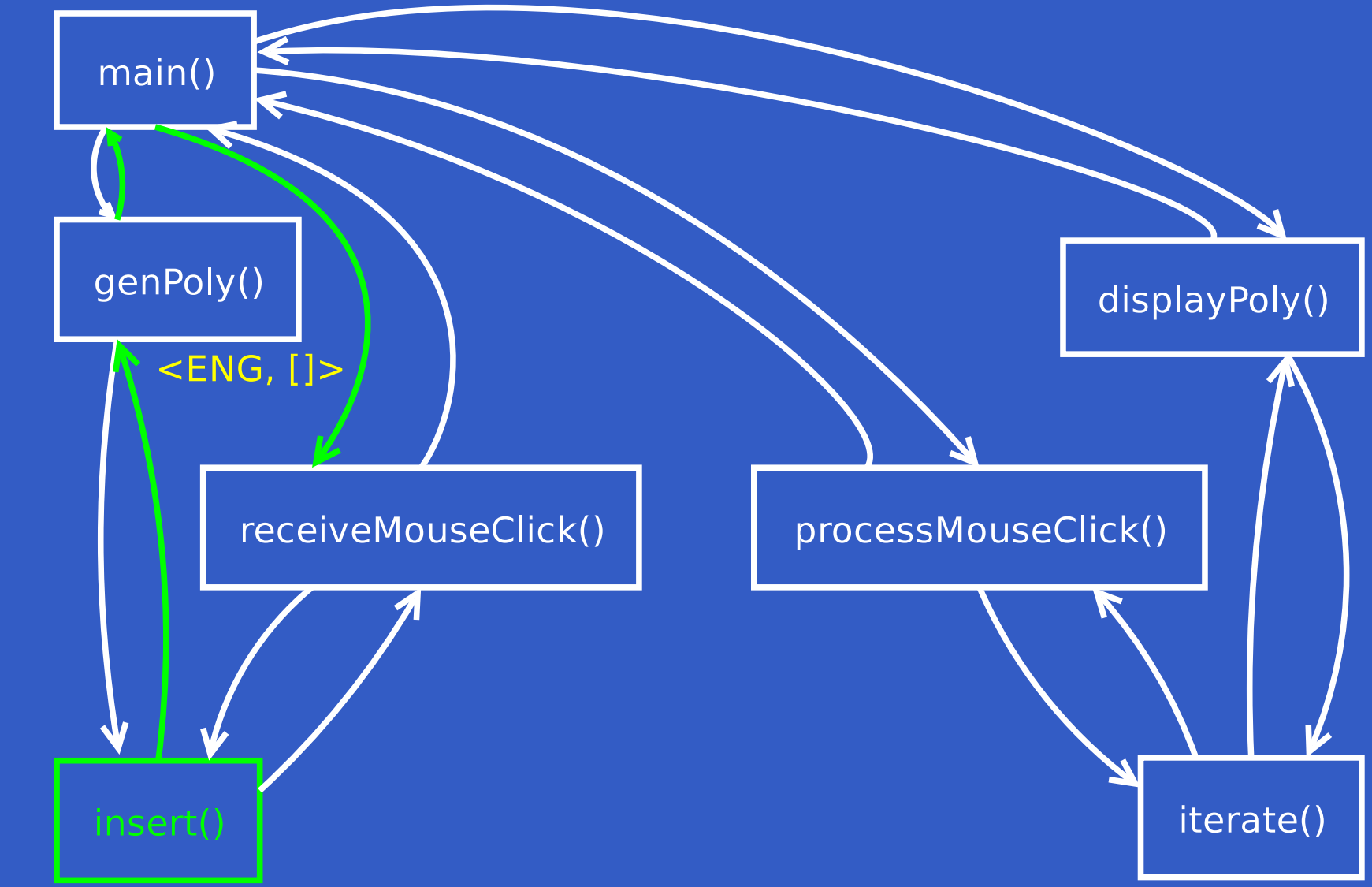
Analysis in Action



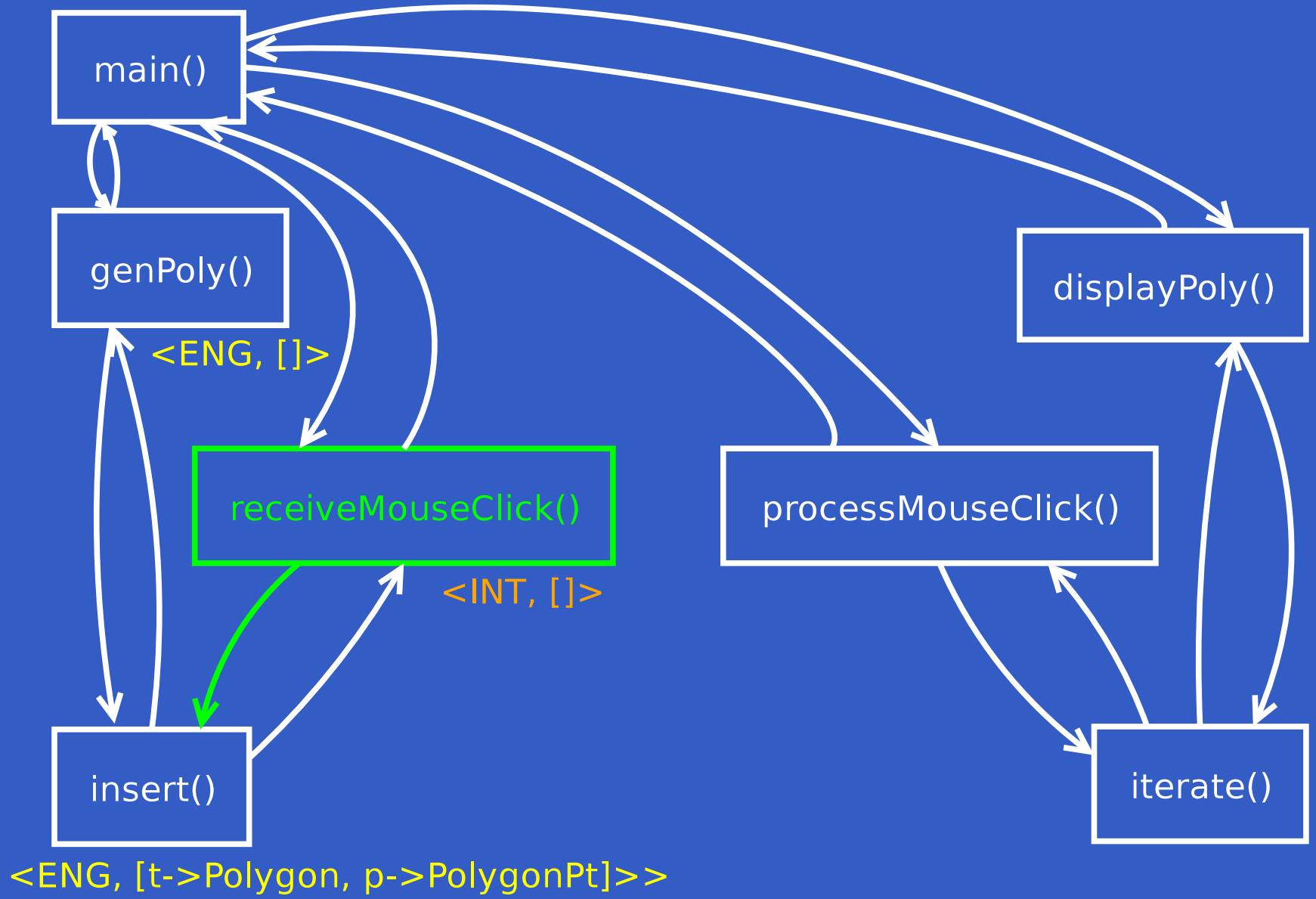
Analysis in Action



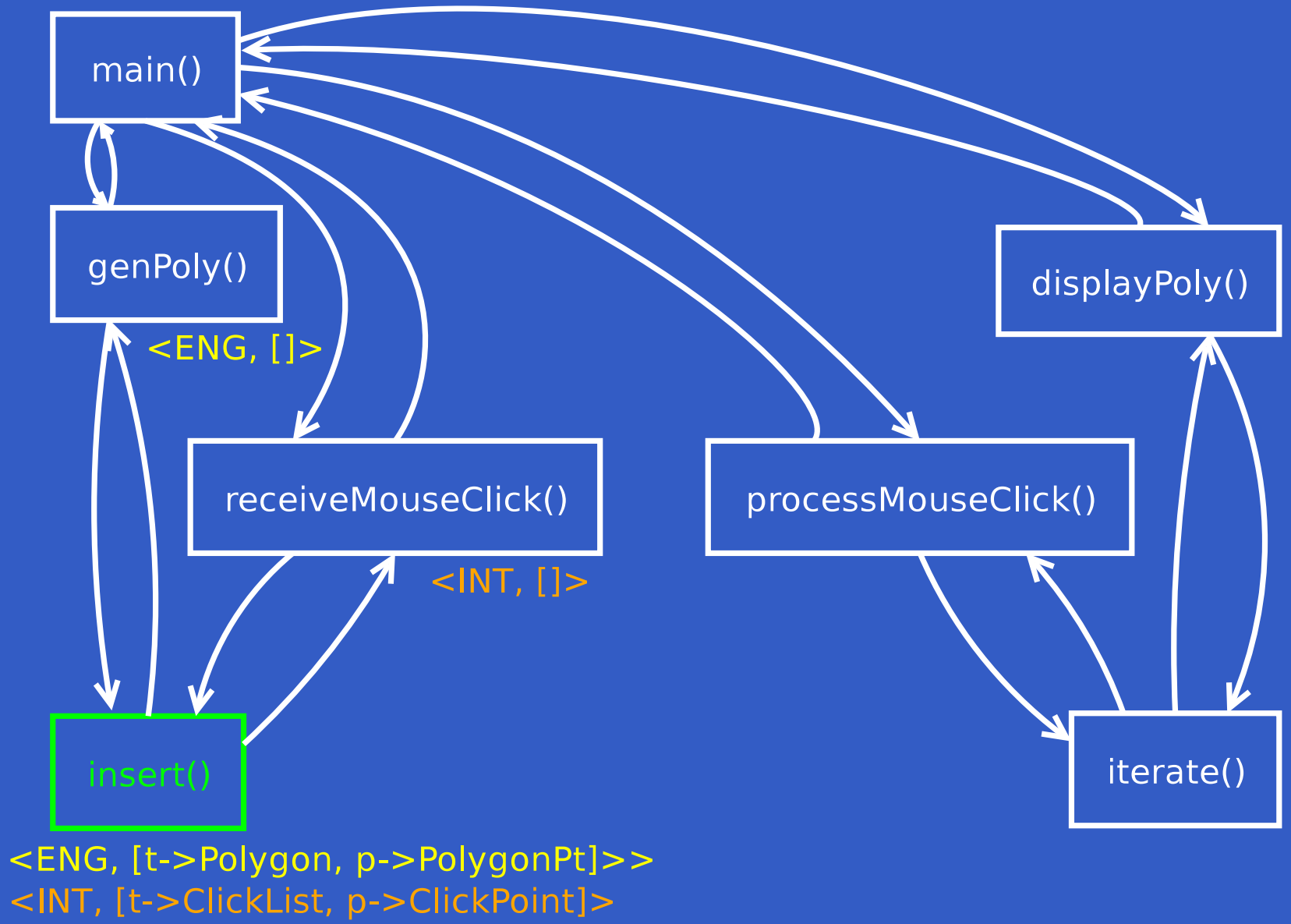
Analysis in Action



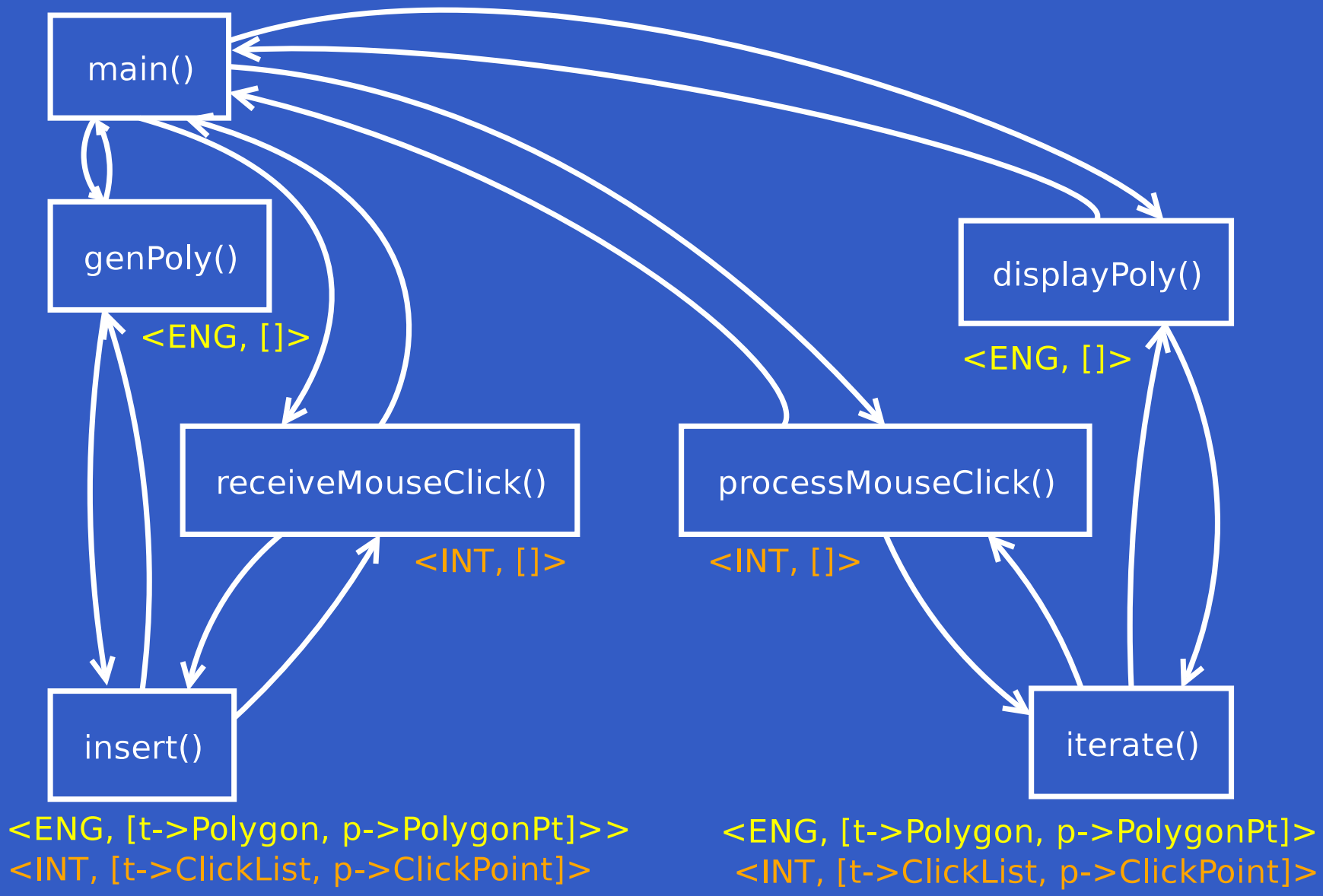
Analysis in Action



Analysis in Action



Analysis in Action



Model Extraction: Heap Structure Model

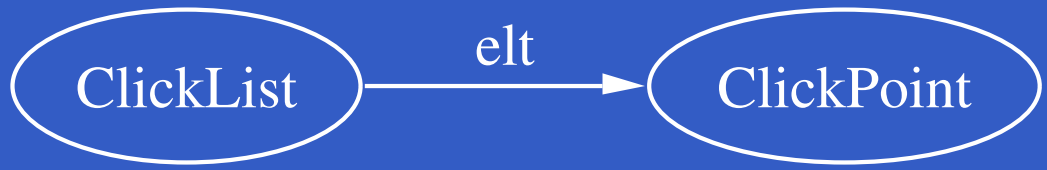
Iterate over all instantiation sites in program and record which tokens can point to which.

```
class List<t,p> { Node<t,p> next; Obj <p> elt; }
new List<ClickList, p>();
```

Possible context:

$$\langle \texttt{EventHandler}, \{p \mapsto \texttt{ClickPoint}\}\rangle$$

computed by analysis algorithm gives HSM edge



Model Extraction: Subsystem Access Model

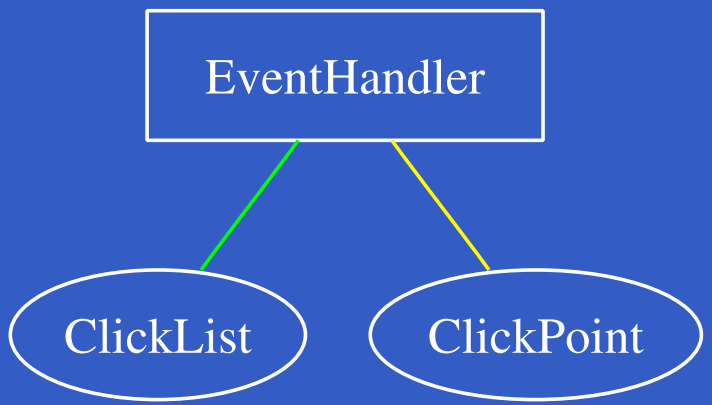
Iterate over field accesses and determine which subsystem accesses which tokens from context.

```
class List<t,p> { Node<t,p> first;
  Obj getFirst() {
    Node<t,p> n = this.first; return n.elt; }}
```

Context

$\langle$**EventHandler**, $\{t \mapsto$ **ClickList**, $p \mapsto$ **ClickPoint**$\}\rangle$

gives subsystem access edges



Extraction: Call/Return Interaction Model

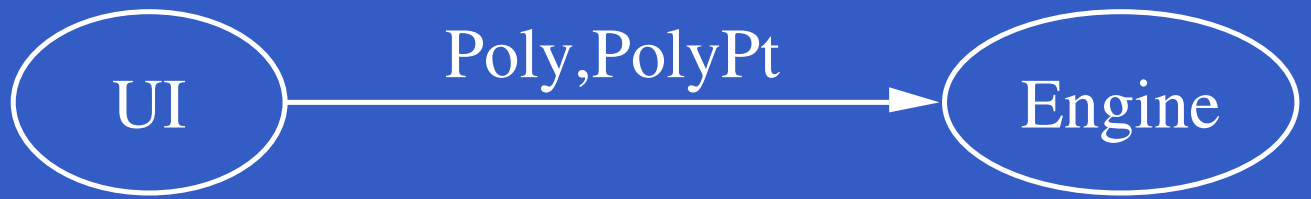
Iterate over method invocations and record transfer of control between subsystems.

```
class E entry Engine {
 <p, pt> insertPoint(Polygon<p,pt> poly, Point<pt> point);
}
class U<pt> entry UI { Polygon<Poly,pt> p;
 receivePoint(Point<pt> userPt) {
  s: eng.insertPoint<Poly,pt>(userPoly, userPt);
}}
```

Context at `s`:

$$\langle \mathbf{UI}, \{pt \mapsto \mathbf{PolyPt}\}\rangle$$

gives subsystem interaction edge:



Model Extraction: Heap Interaction Model

Iterate over program actions, adding edge for each program action, labelled by subsystem.

```
class E<p,pt> entry Engine {
 insertPoint(Polygon<p,pt> poly, Point<pt> point) {
   poly.vertices.insert(point); }
}
class List<t,p> {
 insert(Obj<p> pt) { Node<t,p> n; ...
   s:  n.elt = pt;
}
```

Context at s:

$$\langle \mathbf{Engine}, \{t \mapsto \mathbf{Polygon}, p \mapsto \mathbf{PolyPt}\}\rangle$$

Related Work

UML:

- Our models designed to be embeddable inside program code.

Automatic Model Extraction:

- Commercial tools do UML models $\Longleftrightarrow$ code [TogetherSoft, Rational Rose]
- Womble extracts UML models from code. [Waingold and Jackson]
- ArchJava expresses and enforces software architecture constraints embedded in code [Aldrich et al.]

Pointer Analysis:

- Unlike pointer analysis, we allow developer to choose most suitable abstraction of heap for program.
- Standard pointer analysis is allocation-site based.
- Our approach dodges need for flow-sensitivity.

Ownership Types [Boyapati and Rinard]:

- We focus on inter-subsystem communication.

Conclusion

Type system for extracting design information

- Heap Structure Models
- Object Access Models
- Interaction Models

Key Issue: Mediating Granularity Mismatch

Key Concepts: Tokens, Subsystems, Polymorphism

- Tokens abstract objects (multiple code elements,
- Subsystems abstract code single design element)
- Polymorphism supports reuse
  (single code element, multiple design elements)

Implemented and used on Tagger program

- Models provide useful insight into design of program
- Development overhead quite small