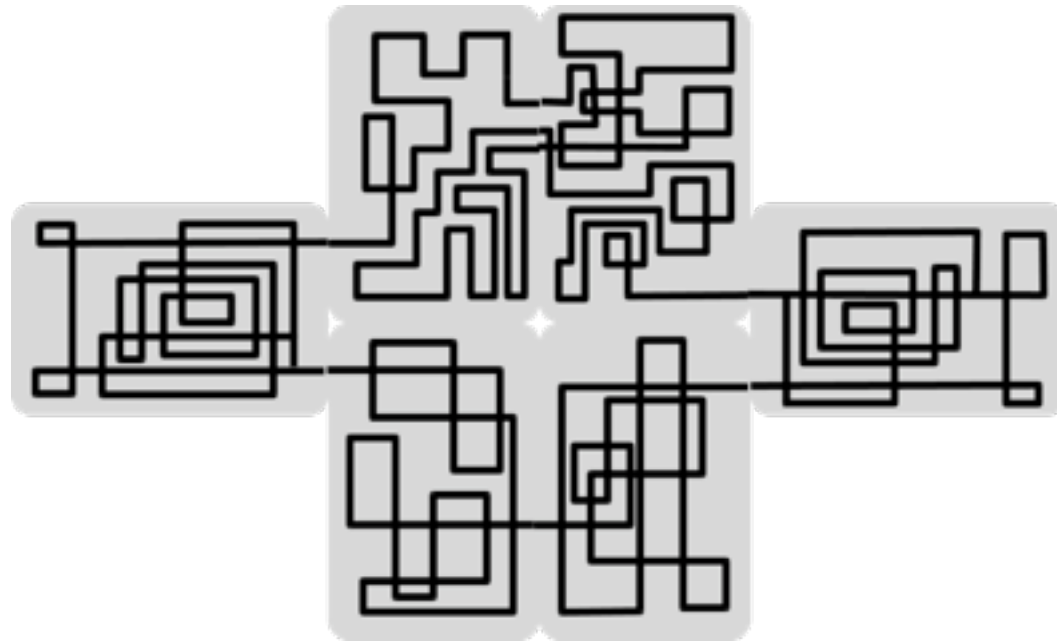


Abstraction Mechanisms for Modular Software Construction

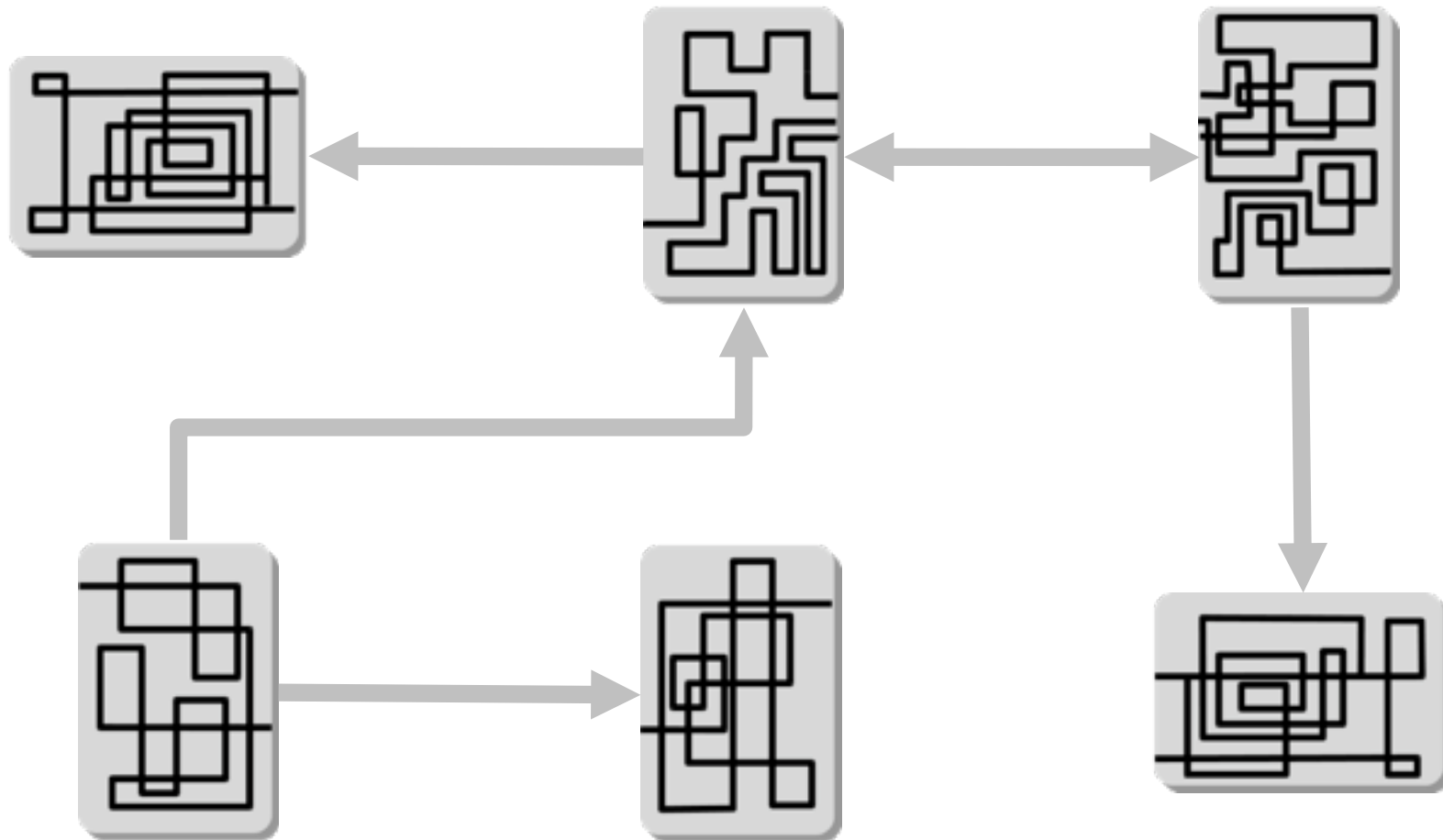
Shan Shan Huang

Georgia Institute of Technology

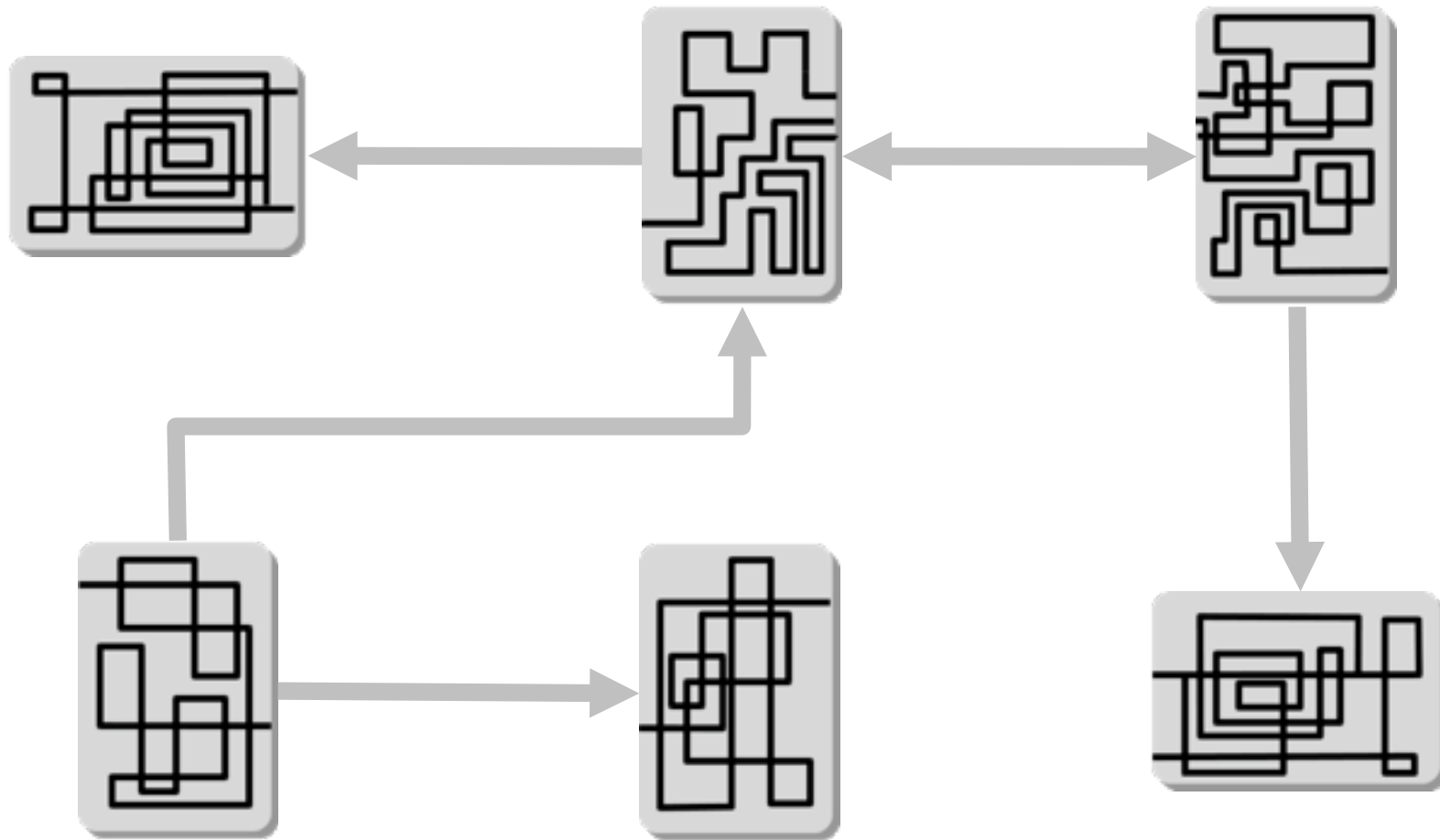
Modular Software Construction



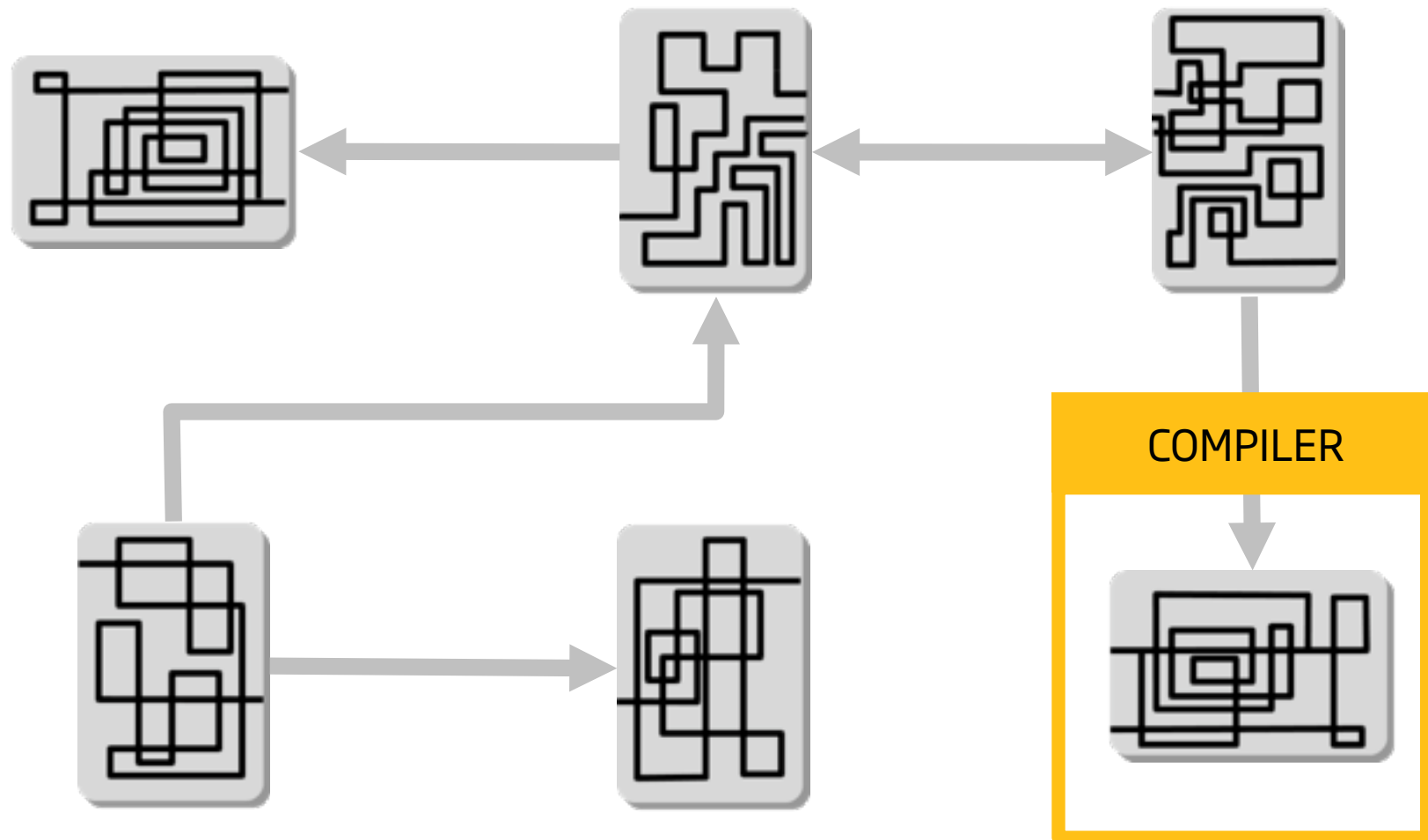
Modular Software Construction



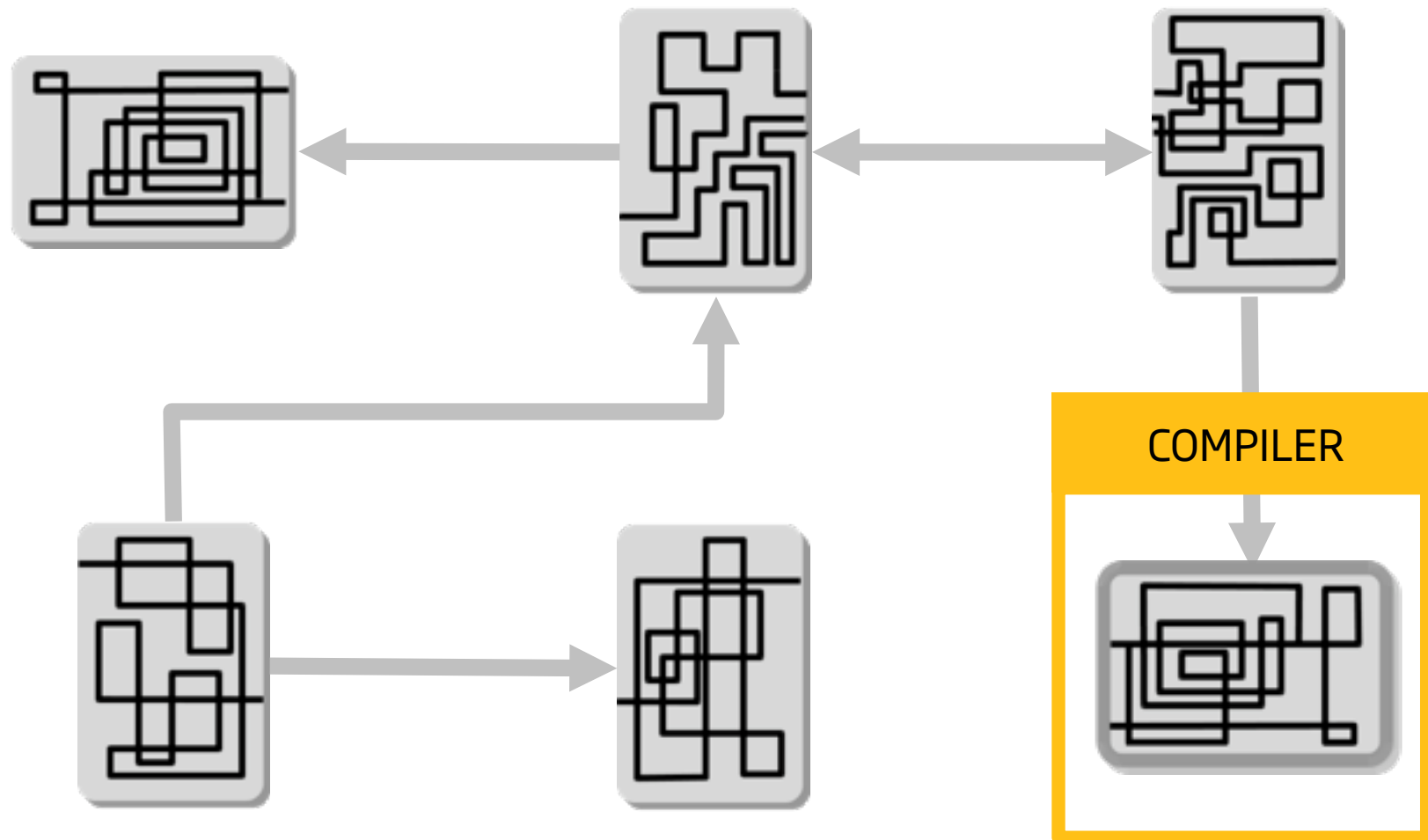
Safety: Separate Reasoning



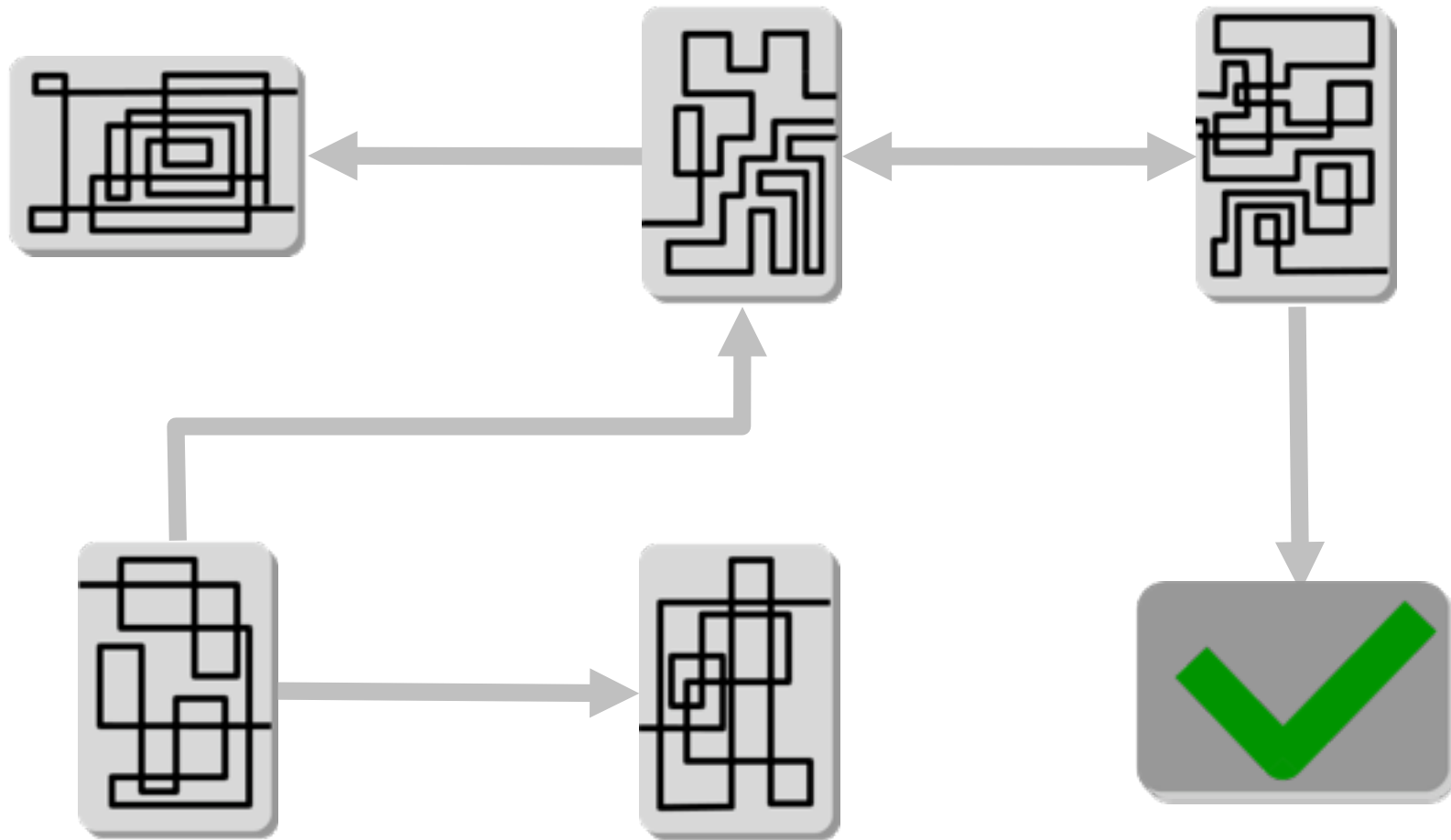
Safety: Separate Reasoning



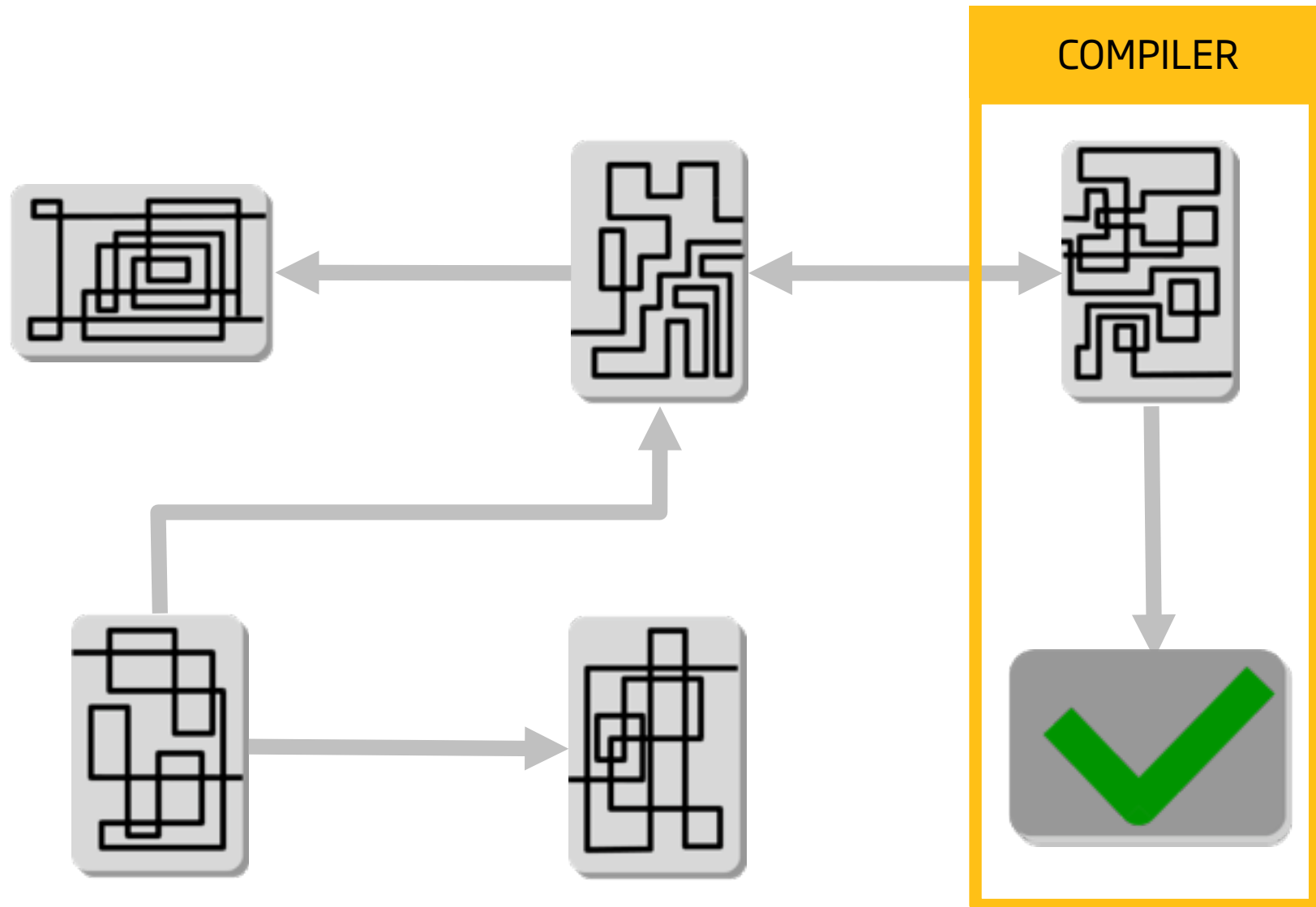
Safety: Separate Reasoning



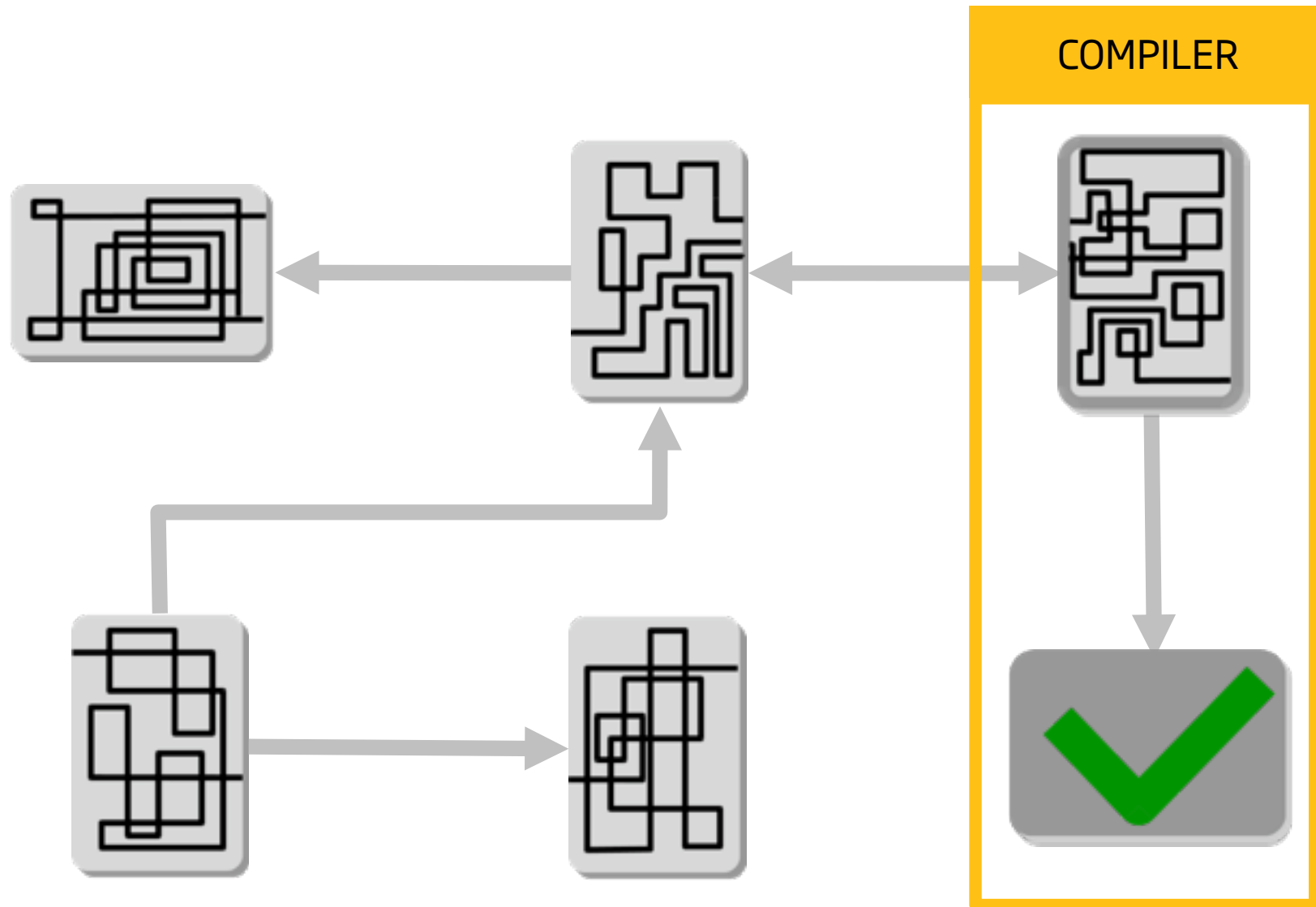
Safety: Separate Reasoning



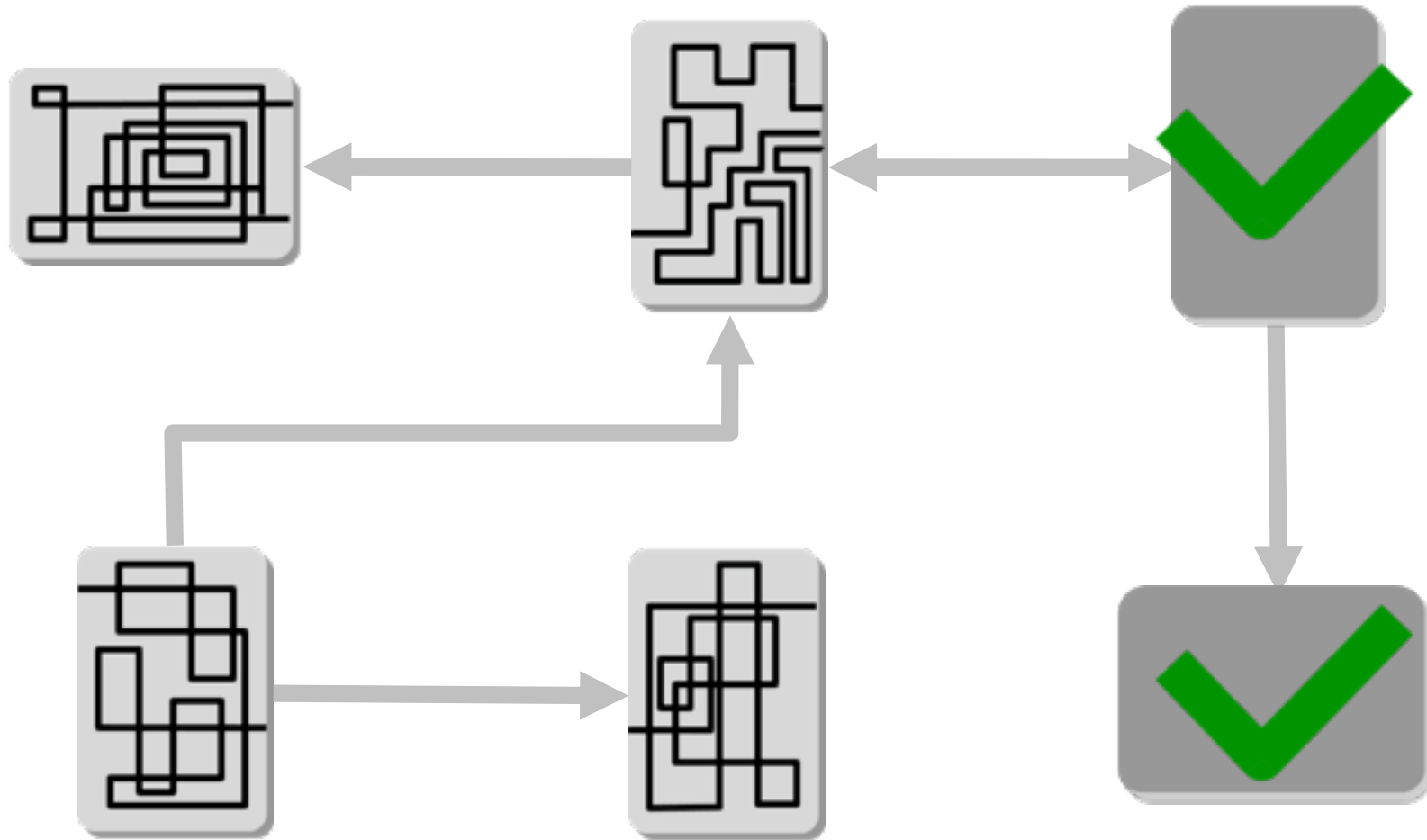
Safety: Separate Reasoning



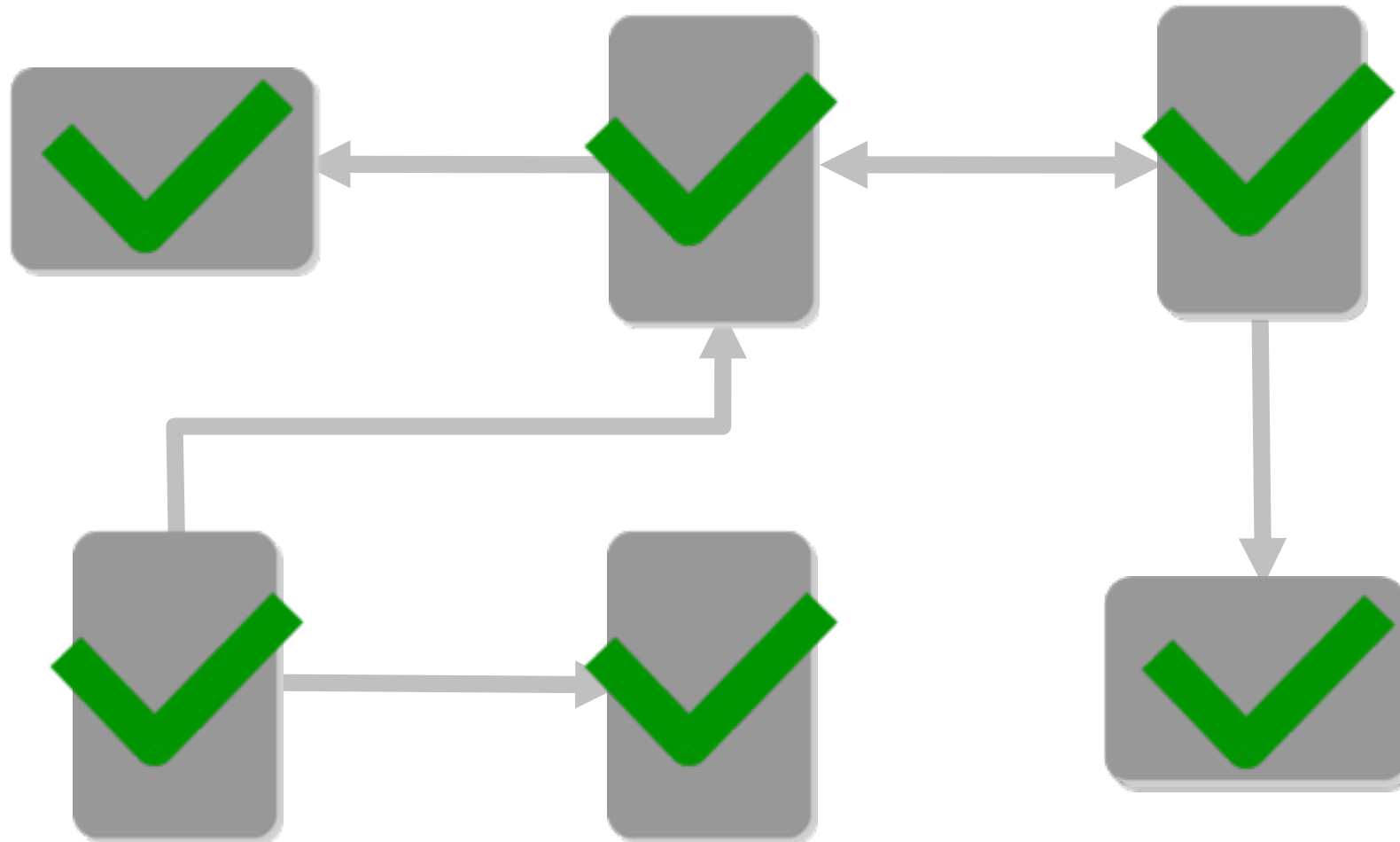
Safety: Separate Reasoning



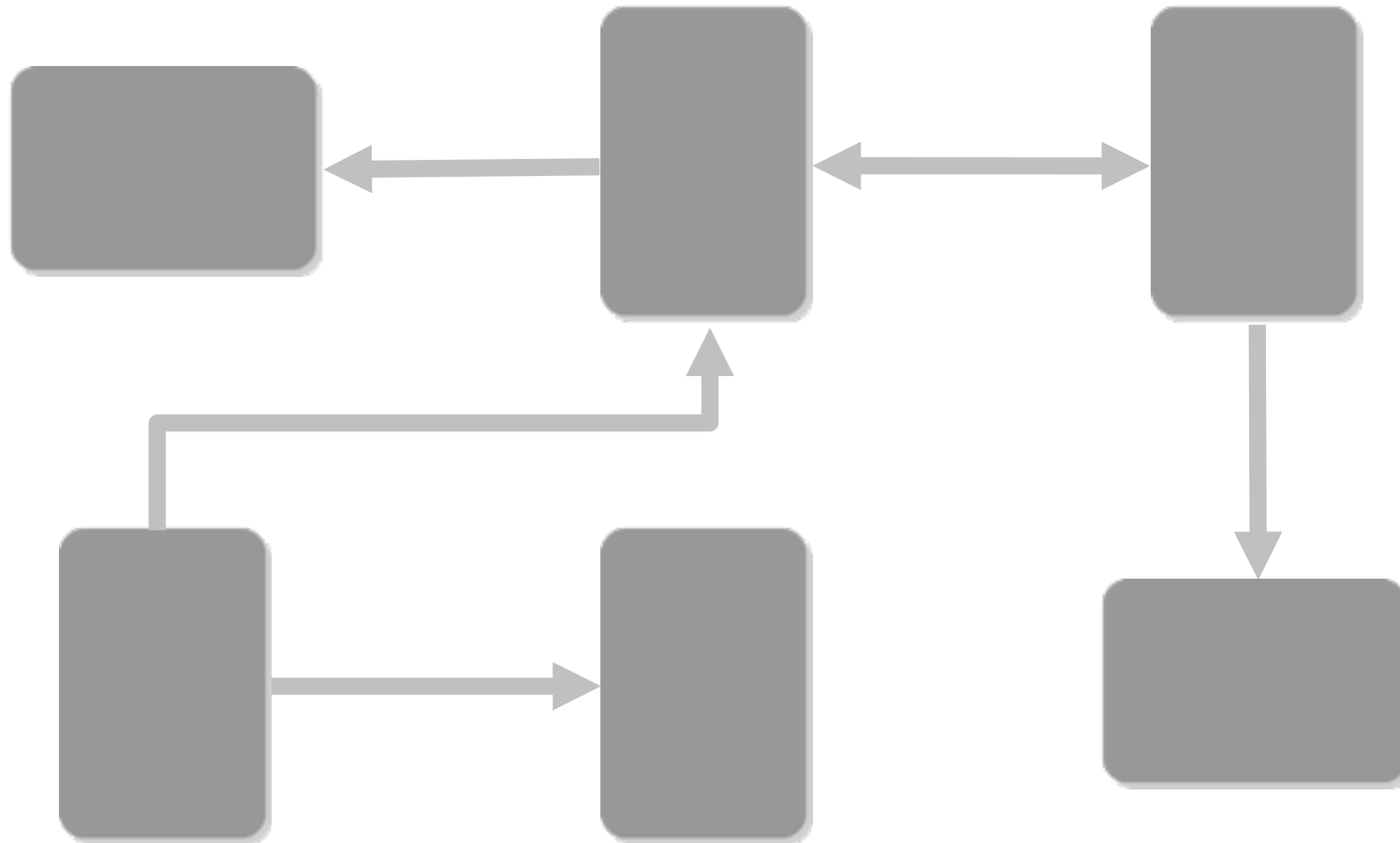
Safety: Separate Reasoning



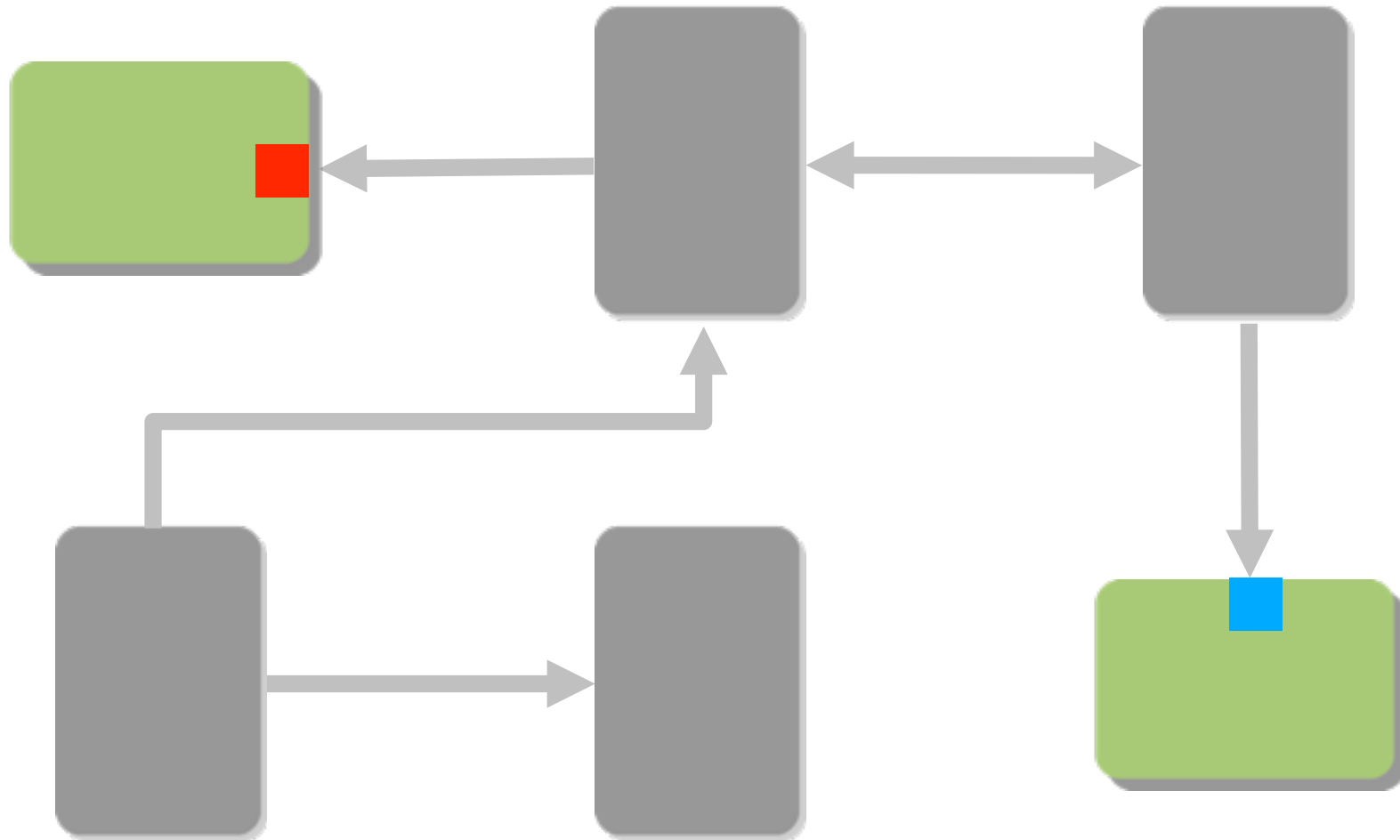
Safety: Separate Reasoning



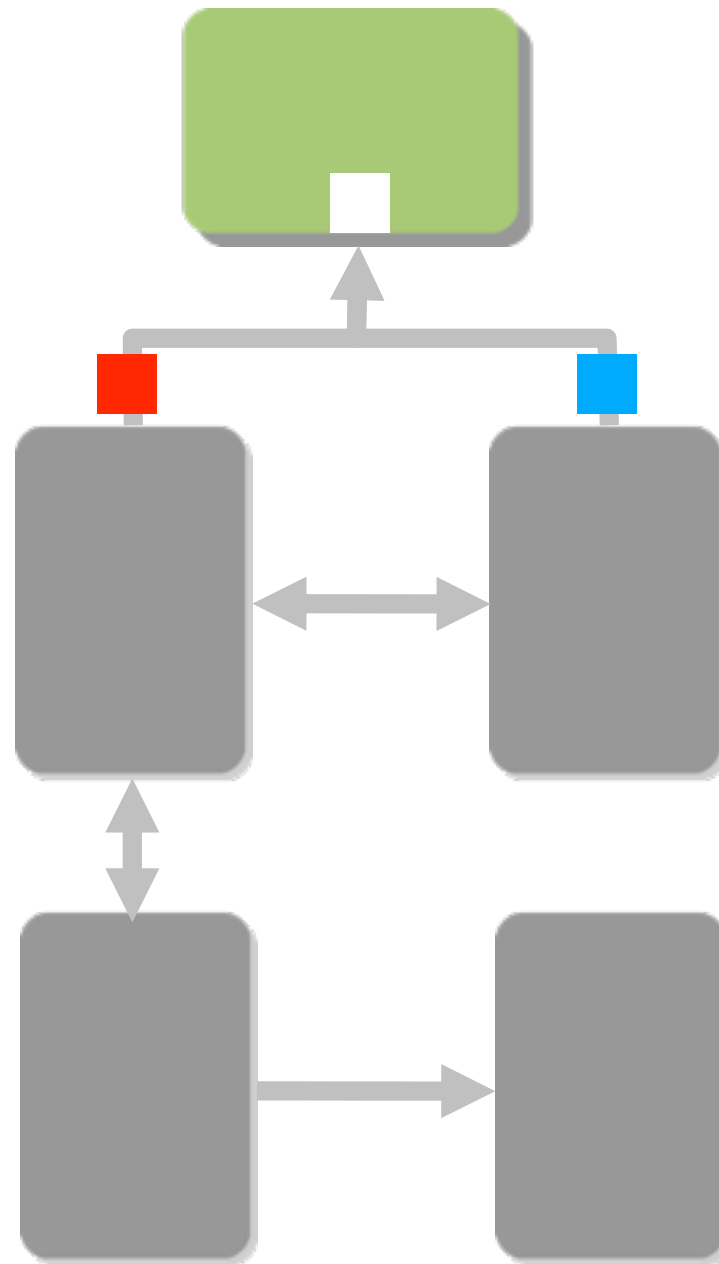
Expressiveness: Maximize Reuse



Expressiveness: Maximize Reuse

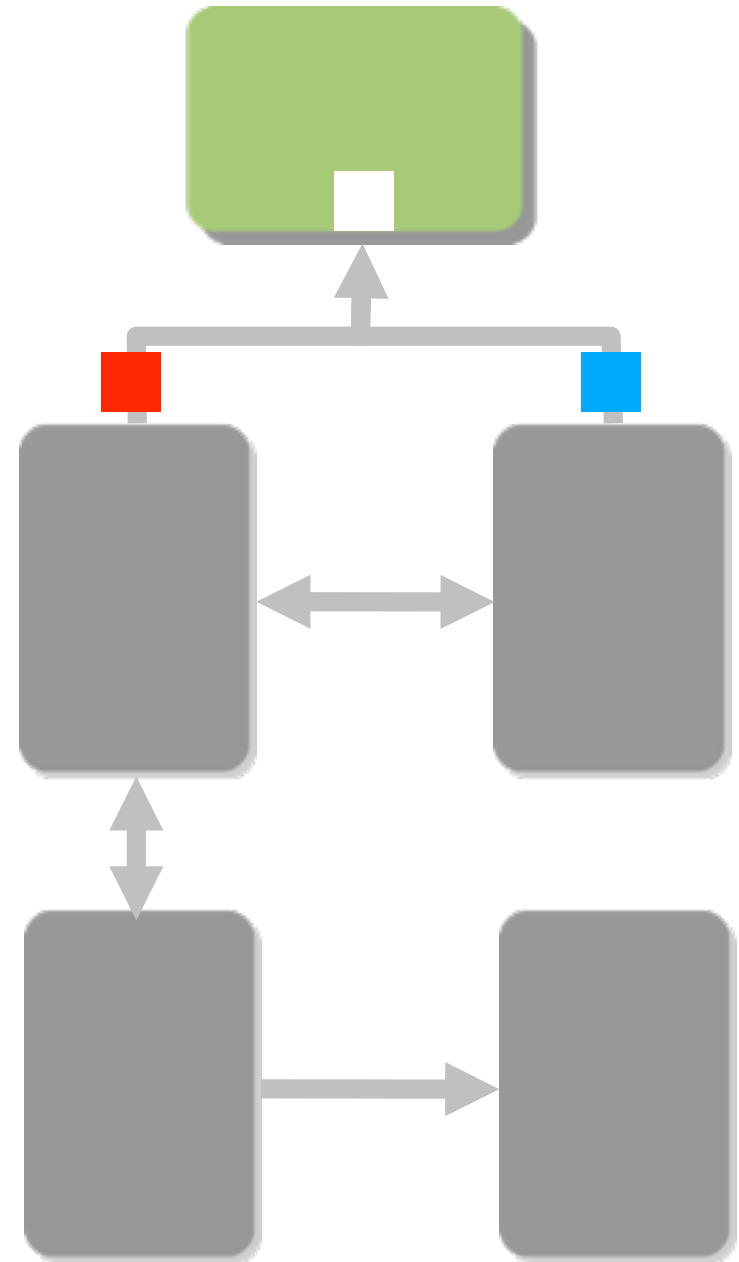


Expressiveness: Maximize Reuse



Expressiveness: Maximize Reuse

Abstraction Mechanisms!

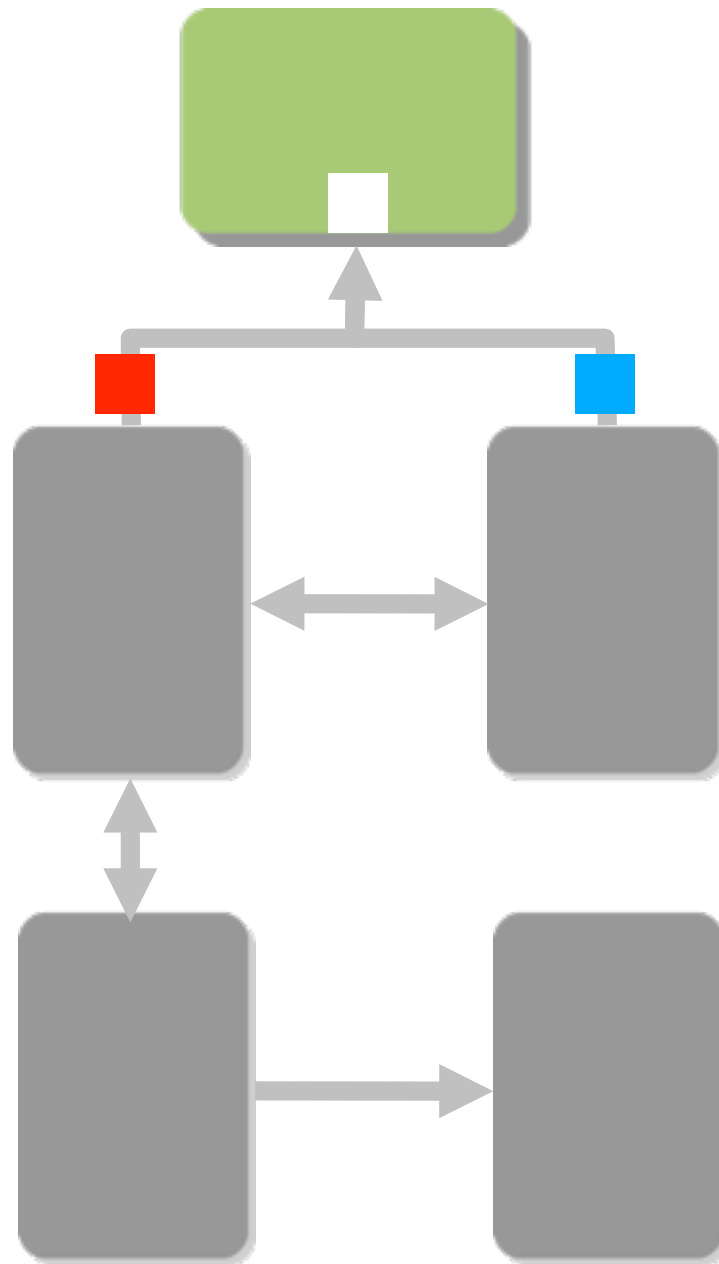


My Research

*Develop **expressive** and **safe** abstraction mechanisms for programming languages.*

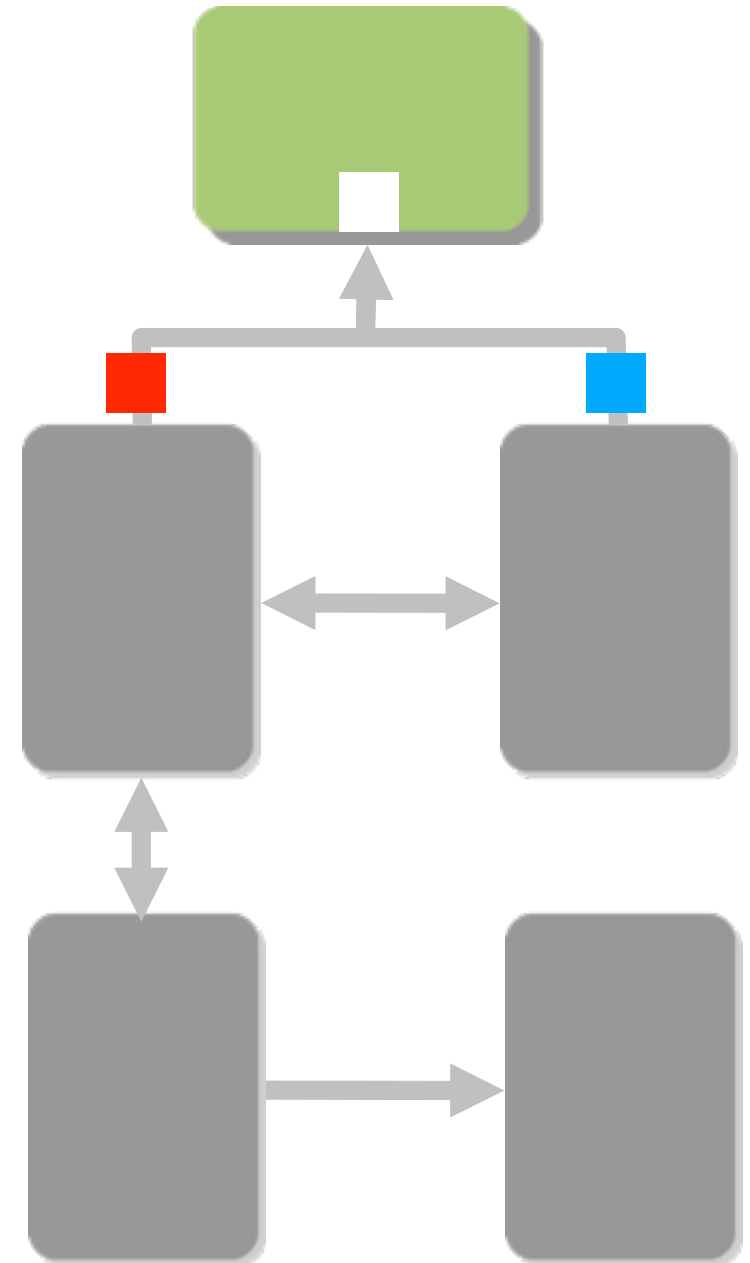
A Brief History of Abstraction Mechanisms

1960's: Procedural Abstraction



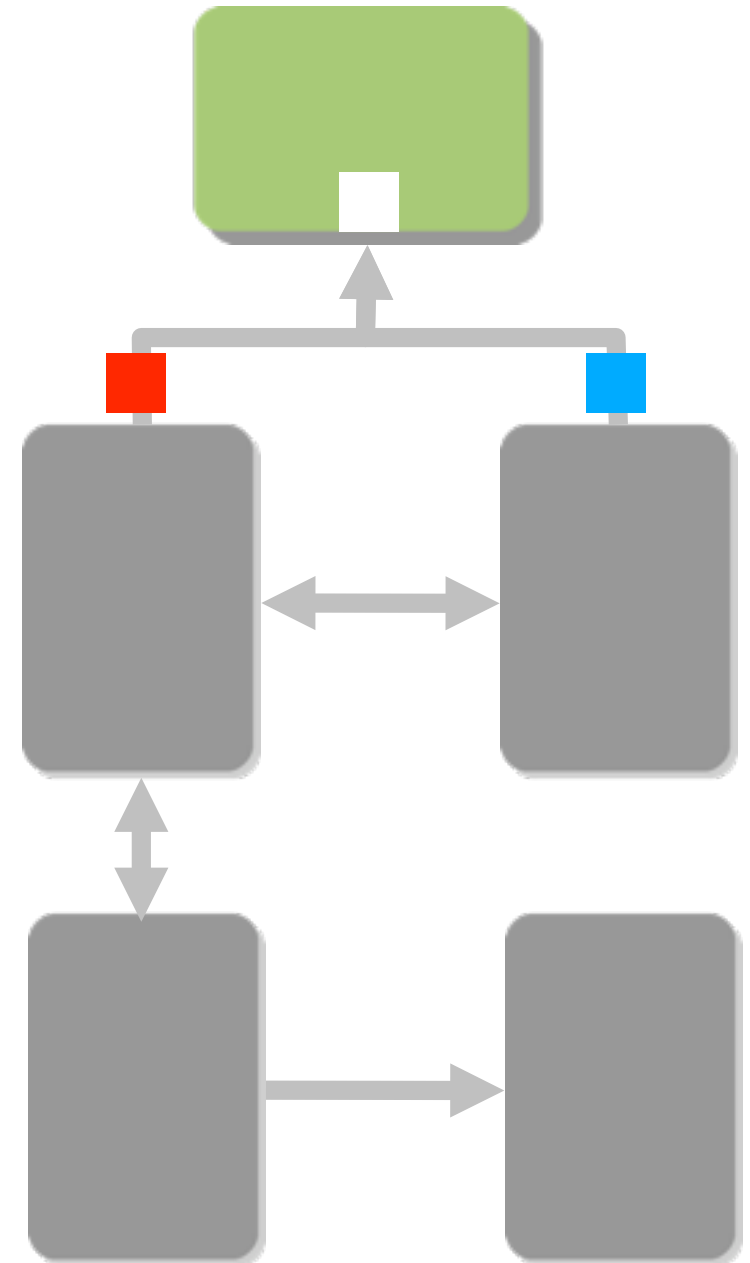
1960's: Procedural Abstraction

```
void fill(Square thing, Color c)
{
  thing.fillColor := c
}
```



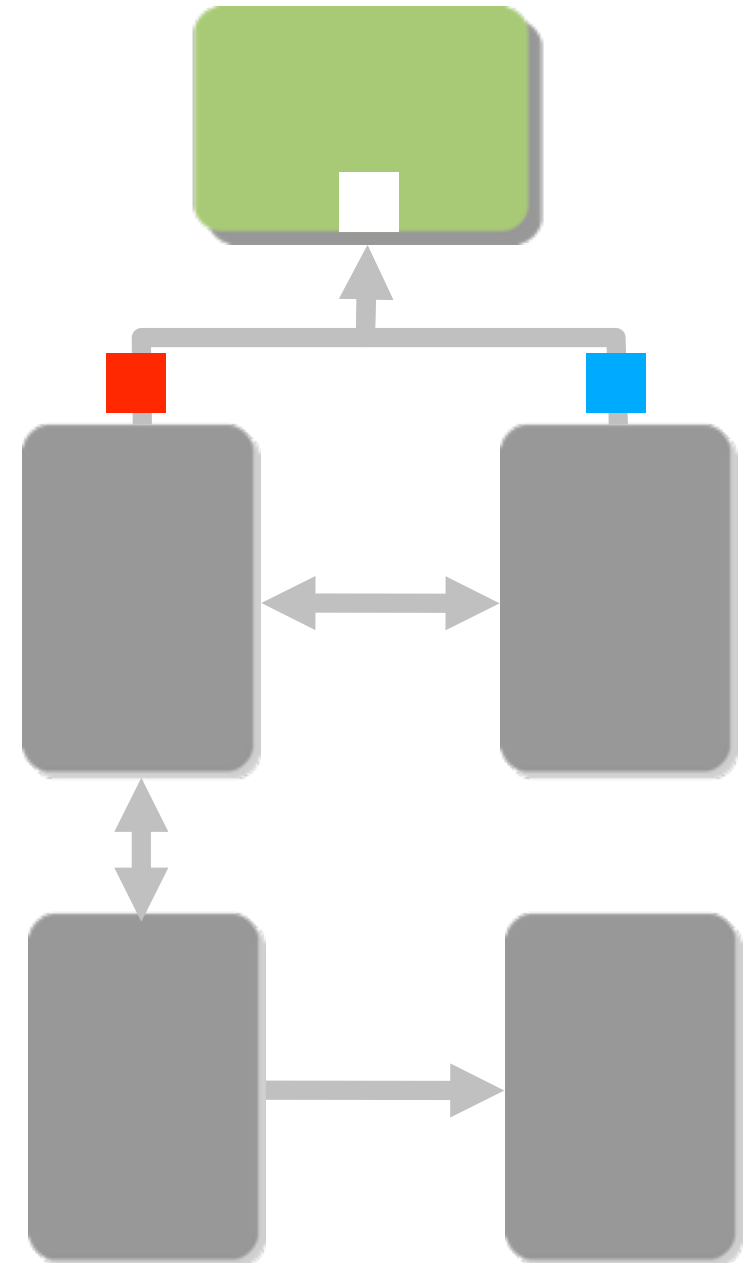
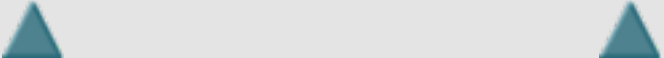
1960's: Procedural Abstraction

```
void fill(Square thing, Color c)
{
  thing.fillColor := c
}
```



1960's: Procedural Abstraction

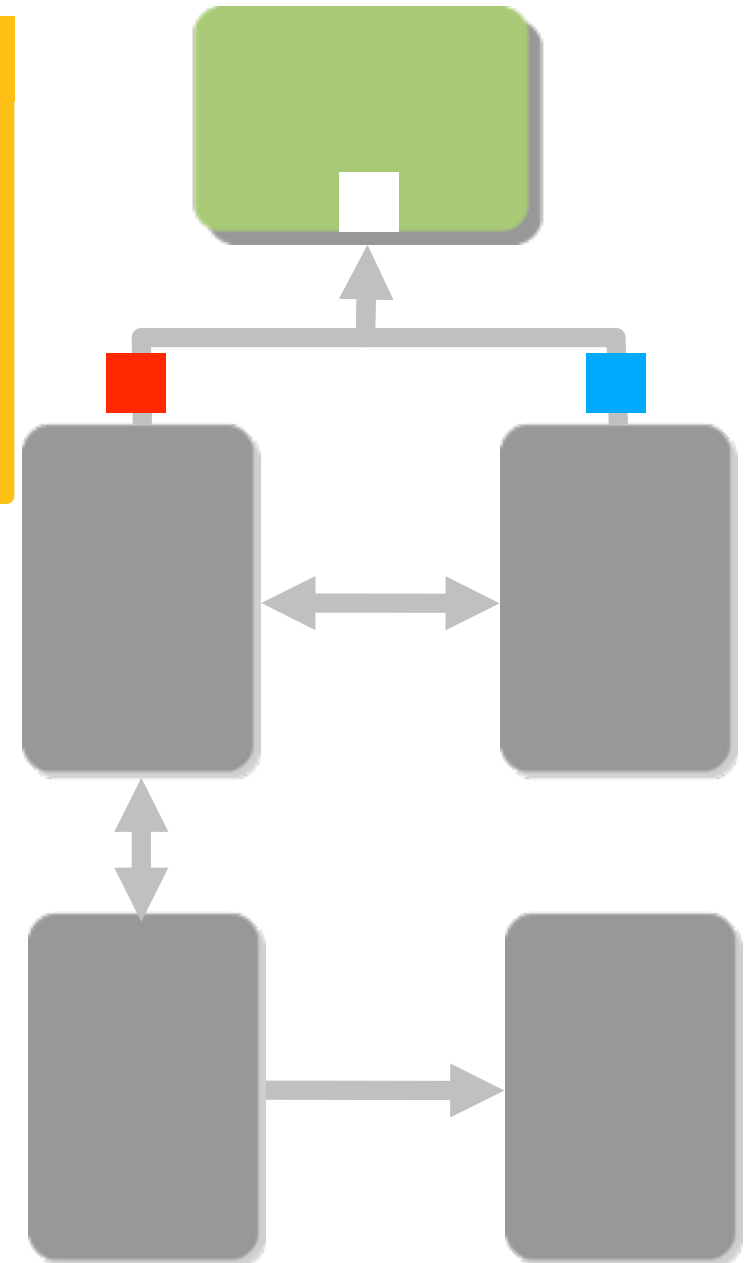
```
void fill(Square thing, Color c)
{
  thing.fillColor := c
}
```



1960's: Procedural Abstraction

COMPILER

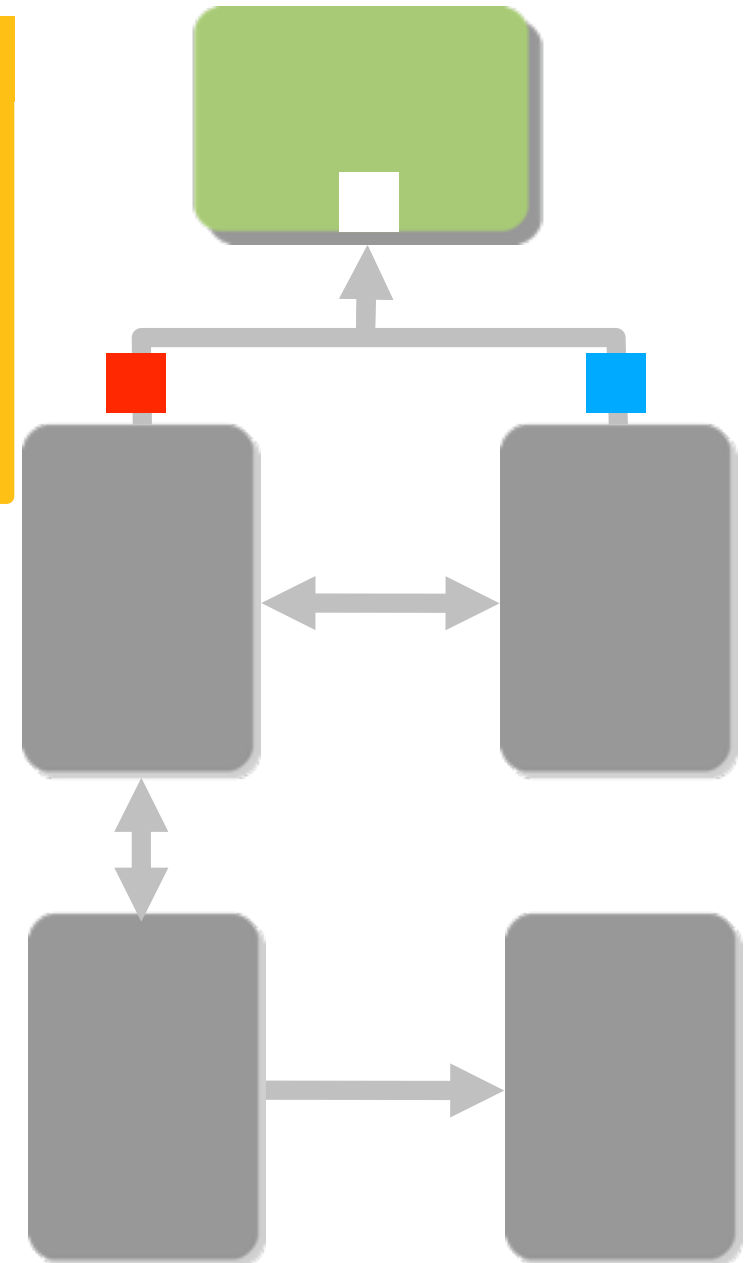
```
void fill(Square thing, Color c)
{
  thing.fillColor := c
}
```



1960's: Procedural Abstraction

COMPILER

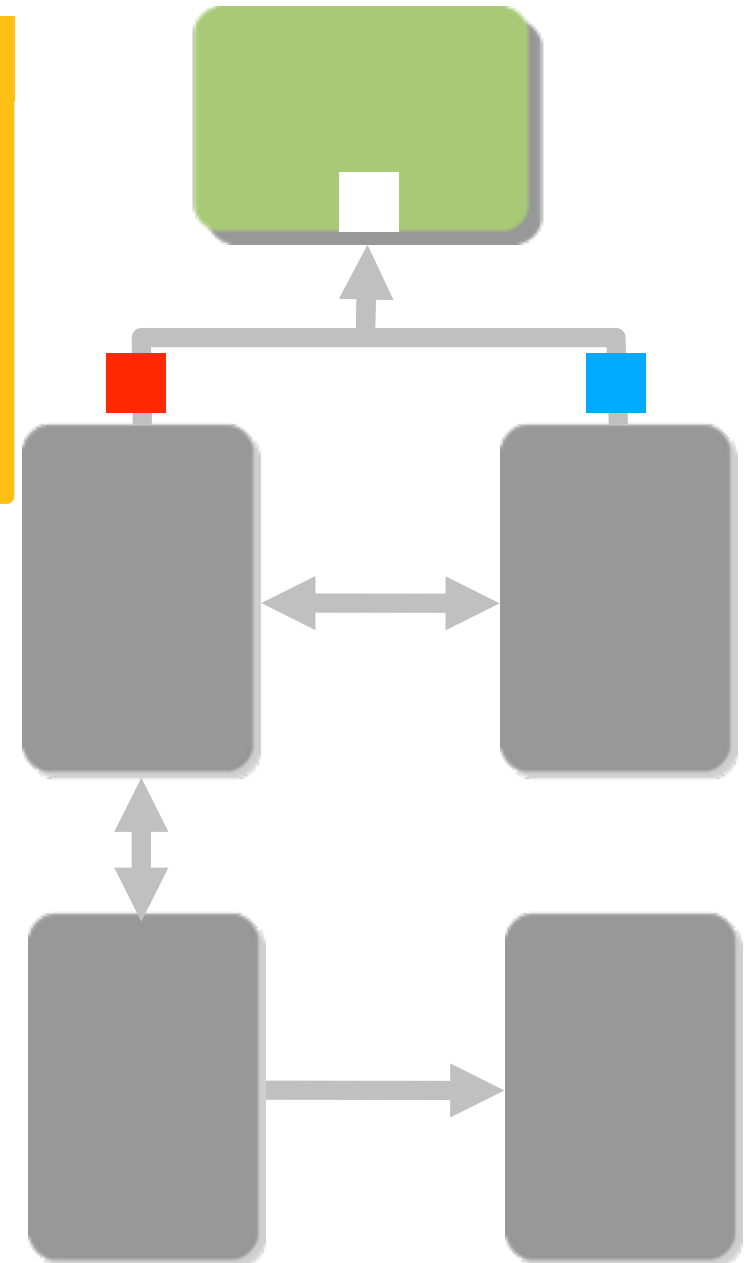
```
void fill(Square thing, Color c)
{
  thing.fillColor := c
}
```



1960's: Procedural Abstraction

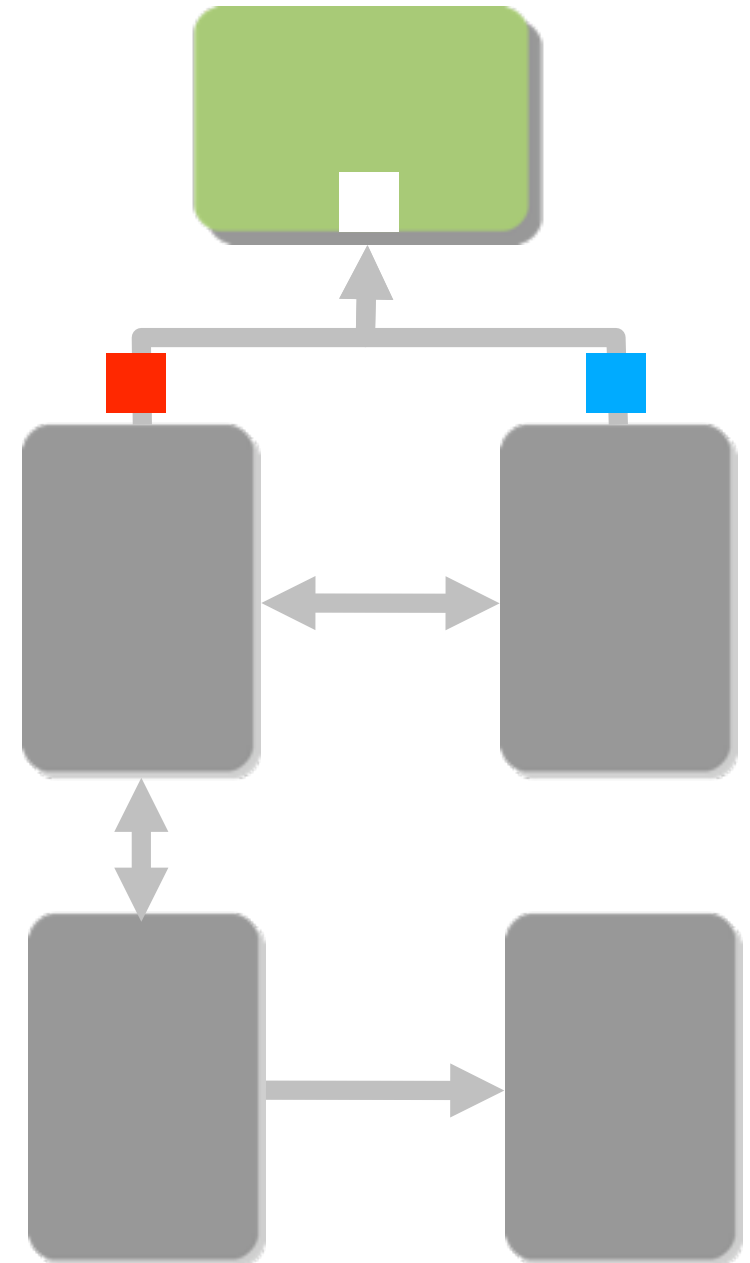
COMPILER

```
void fill(Square thing, Color c)
{
  thing.fillColor := c
}
```



1960's: Procedural Abstraction

```
void fill(Square thing, Color c)
```

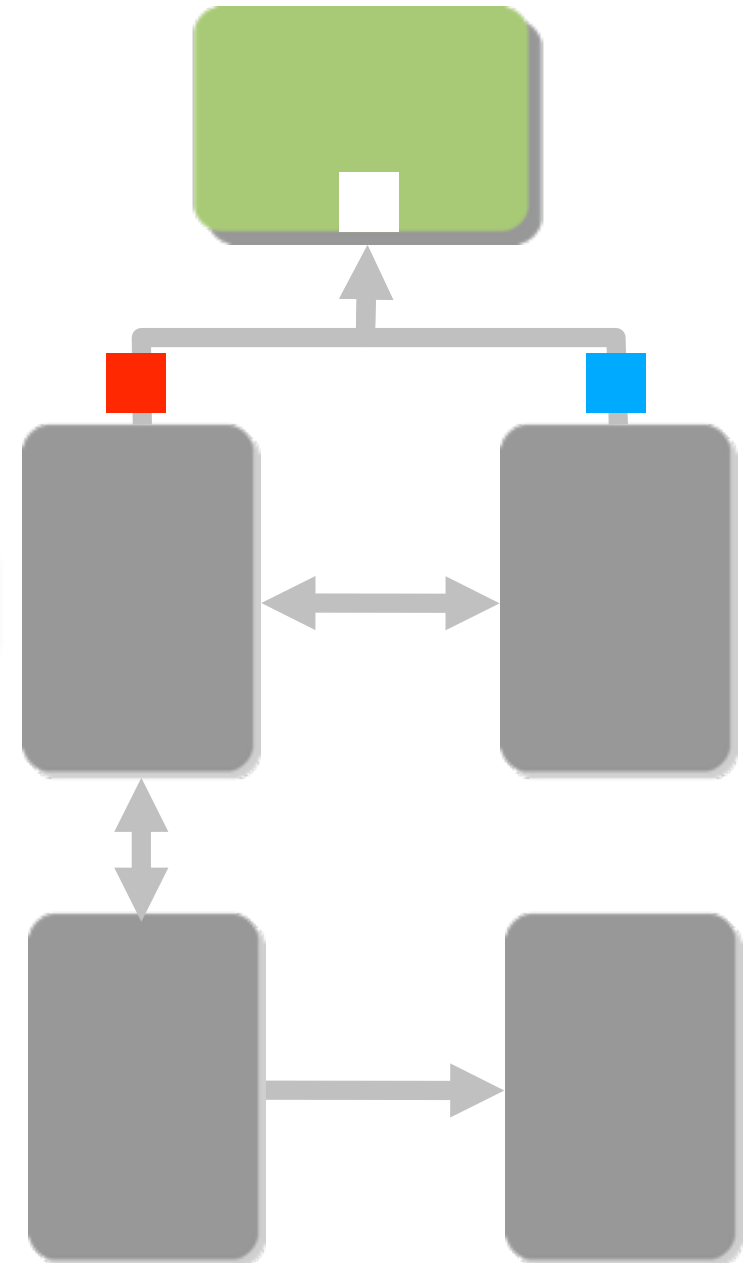


1960's: Procedural Abstraction

```
void fill(Square thing, Color c)
```



```
fill(aSquare, "foo")
```



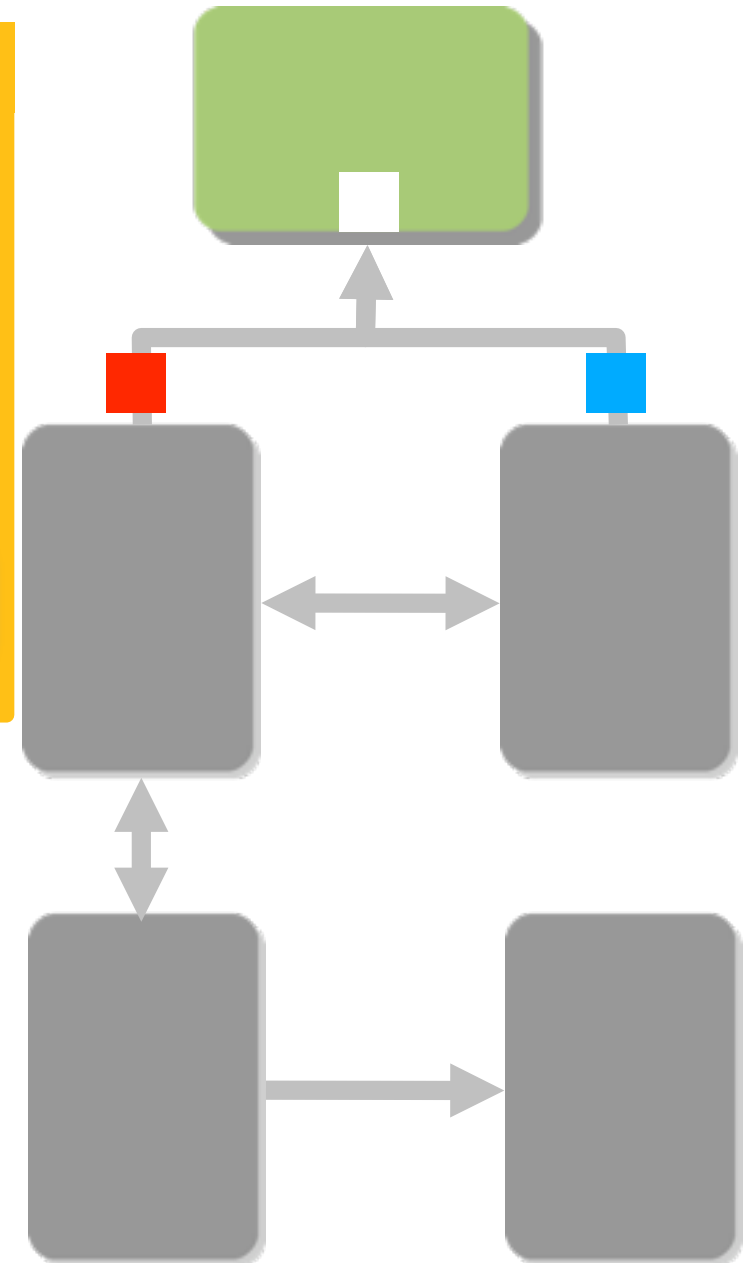
1960's: Procedural Abstraction

COMPILER

```
void fill(Square thing, Color c)
```



```
fill(aSquare, "foo")
```



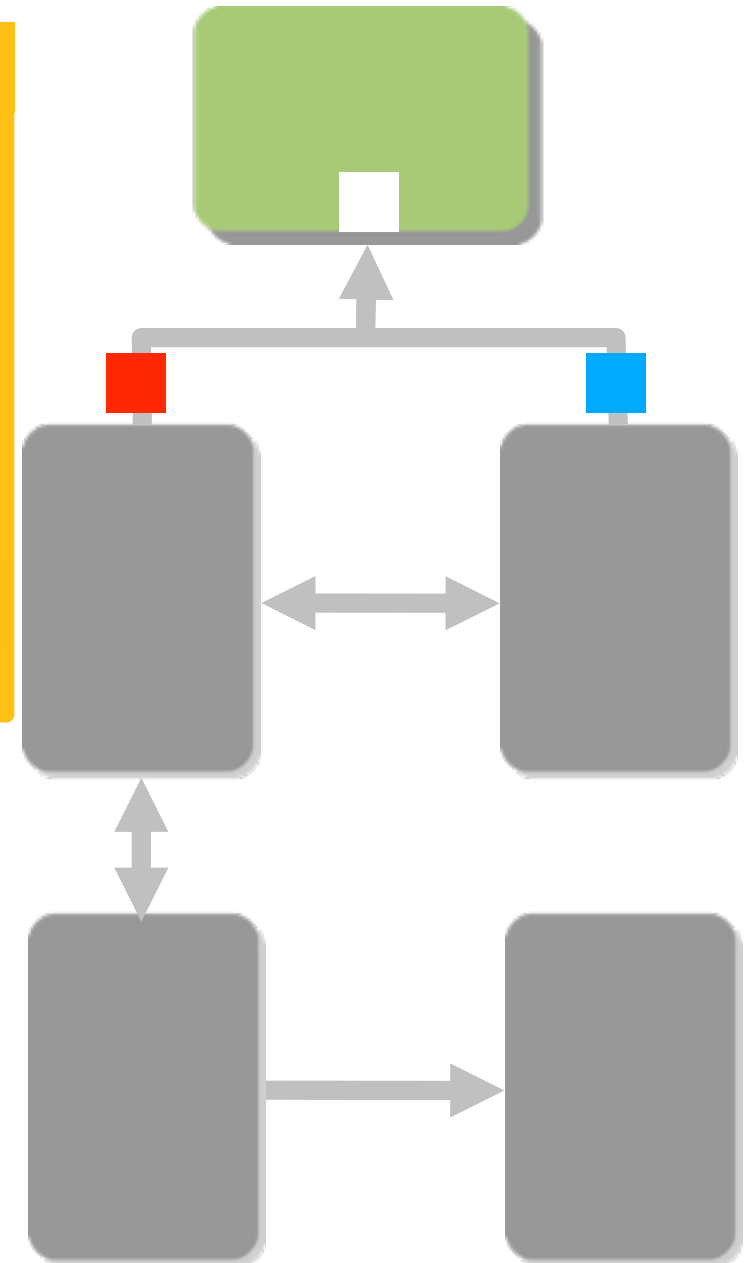
1960's: Procedural Abstraction

COMPILER

```
void fill(Square thing, Color c)
```



```
fill(aSquare, "foo")
```



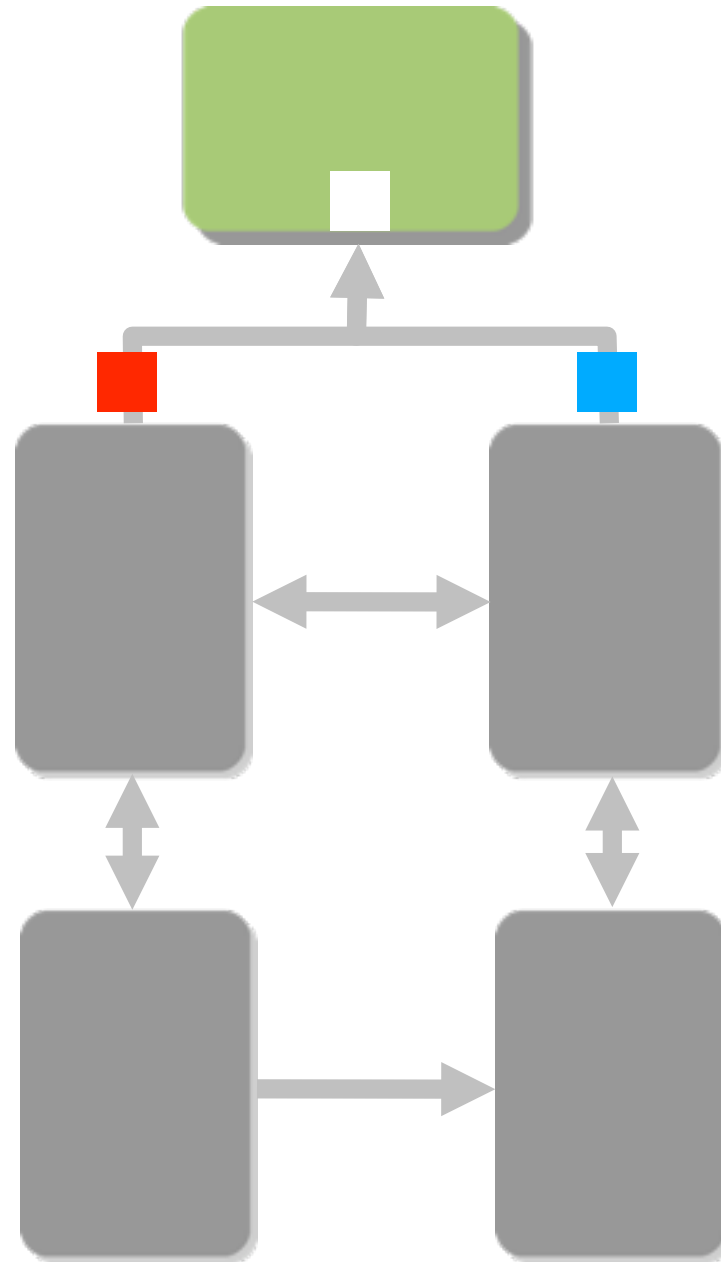
Abstraction Mechanism Design Space



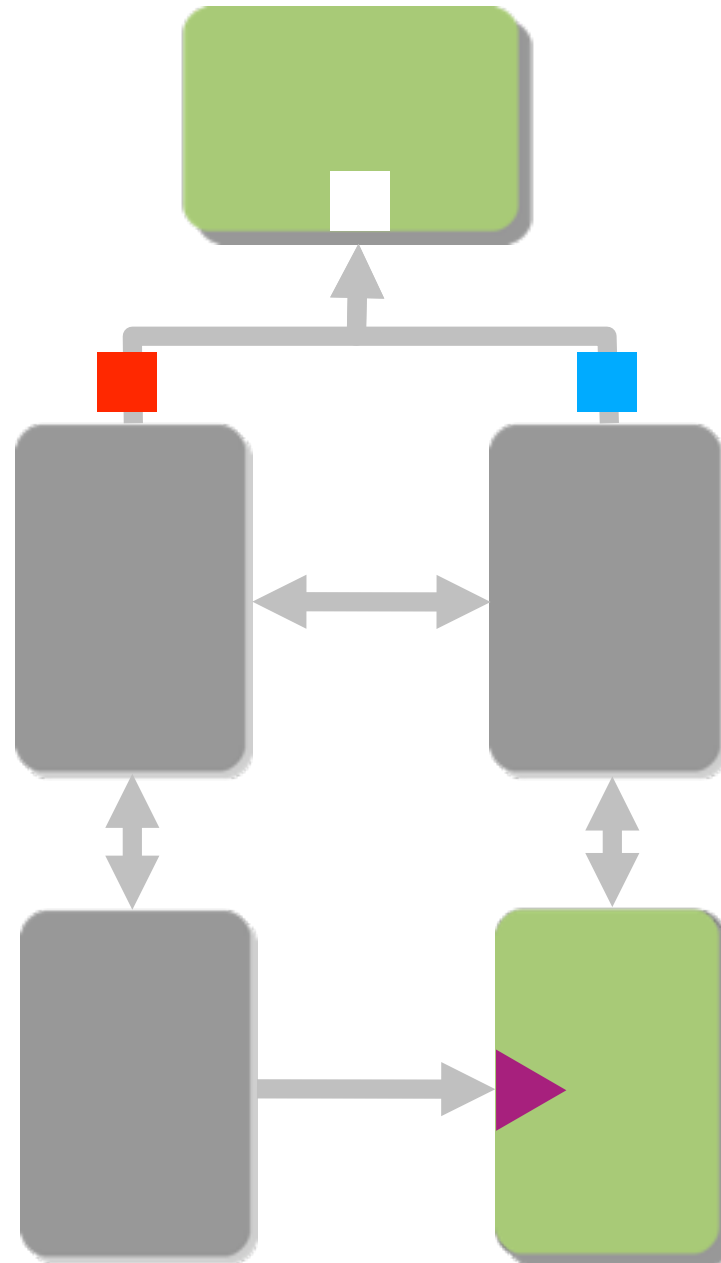
Abstraction Mechanism Design Space



What's Next?

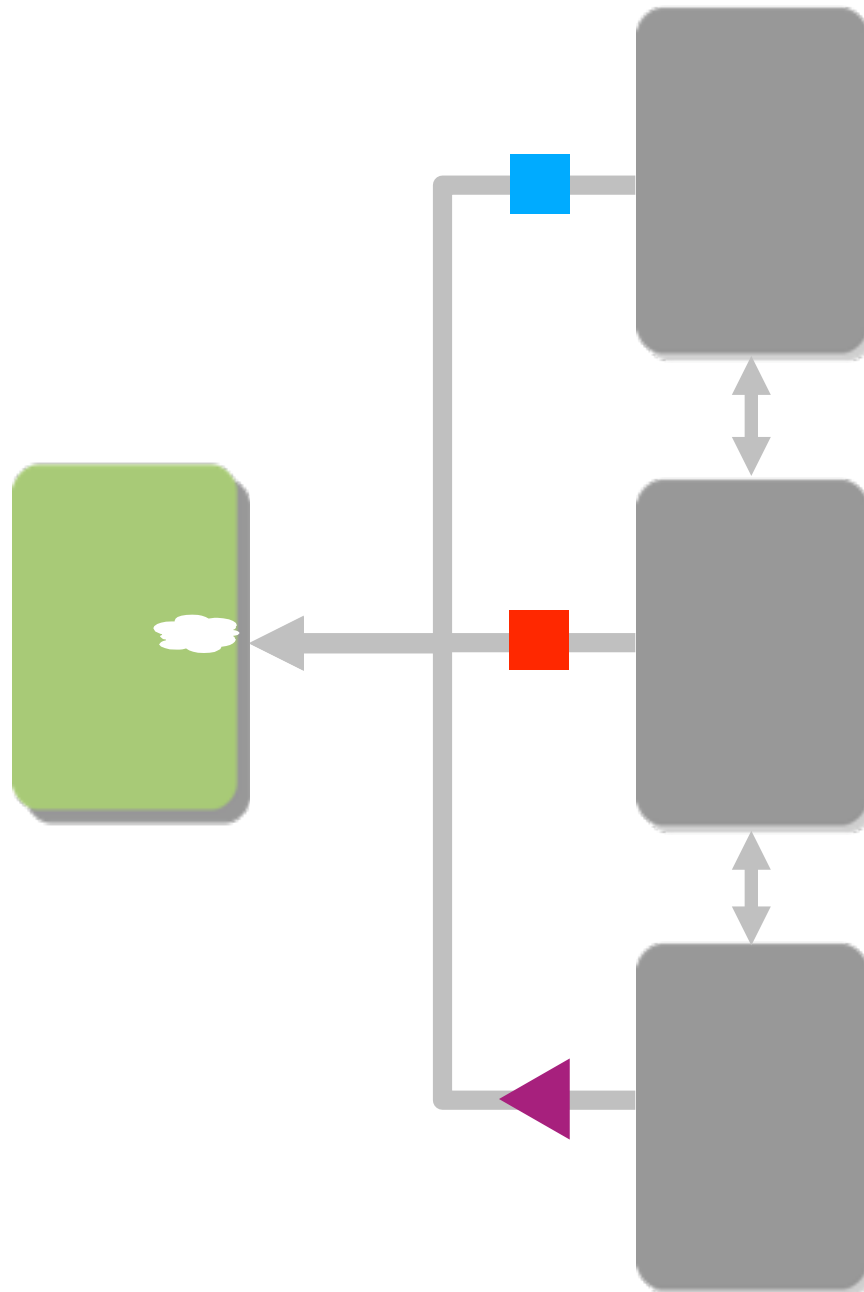


What's Next?

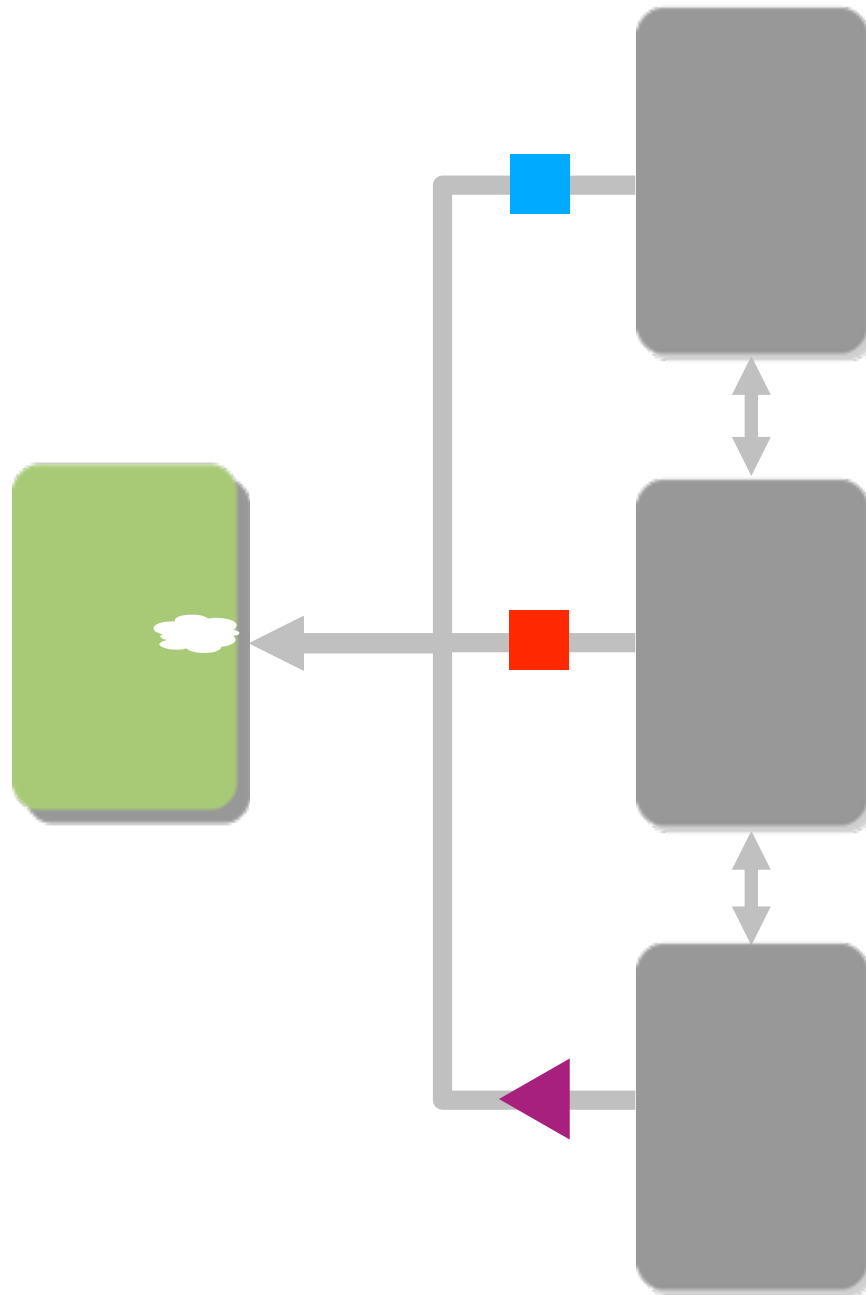


Shan Shan Huang

What's Next?

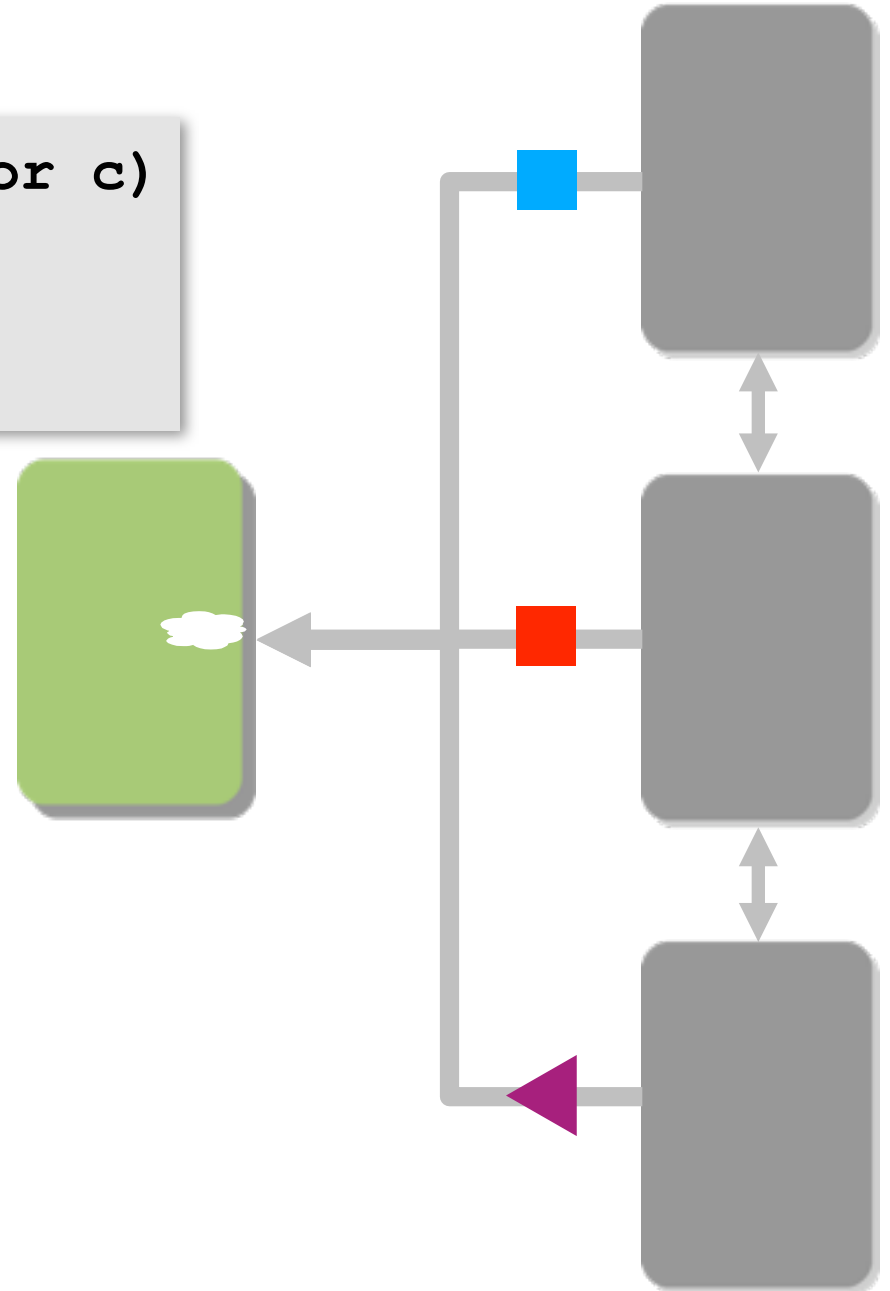


1970's: Type Abstraction



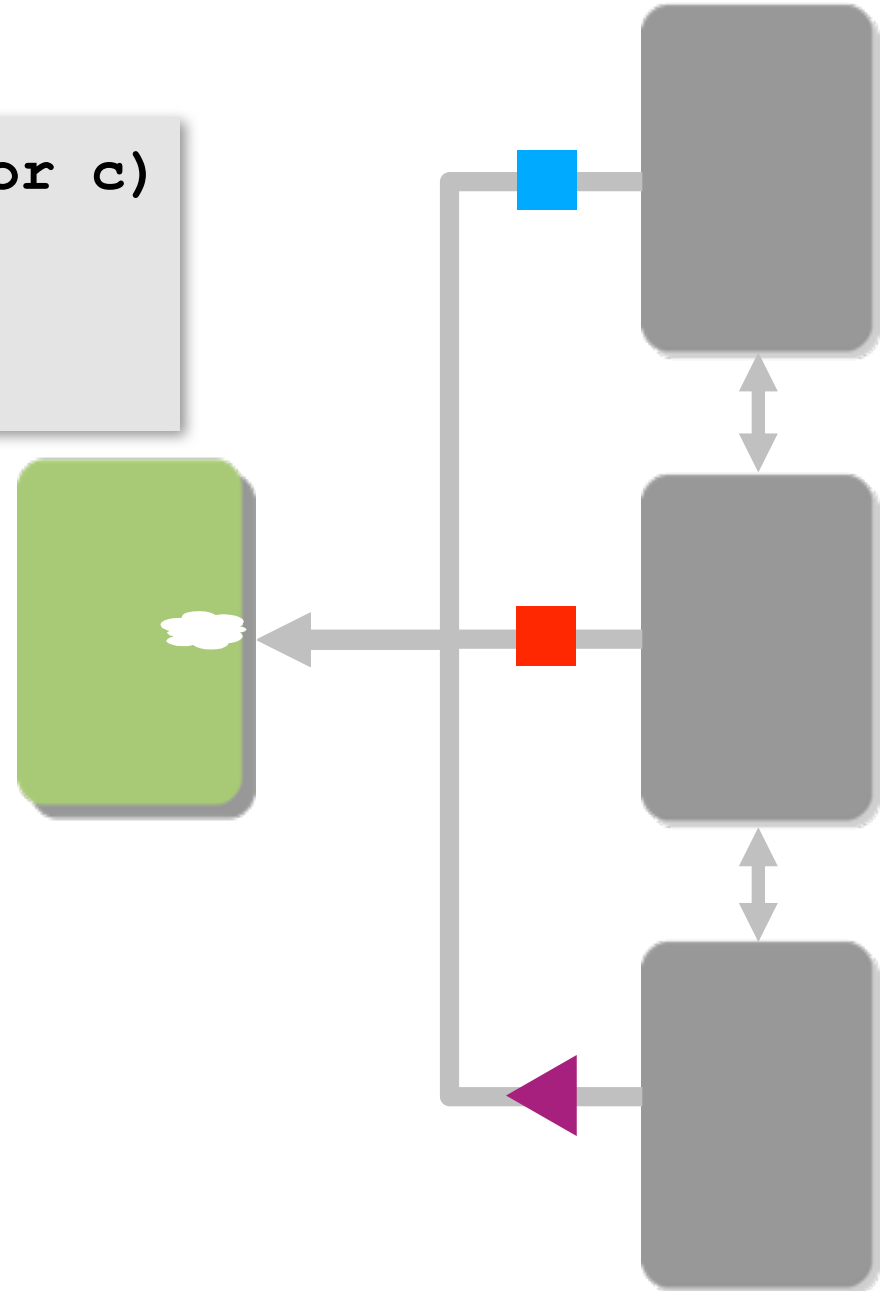
1970's: Type Abstraction

```
void fill<S:Shape>(S thing, Color c)
{
  thing.fillColor := c
}
```



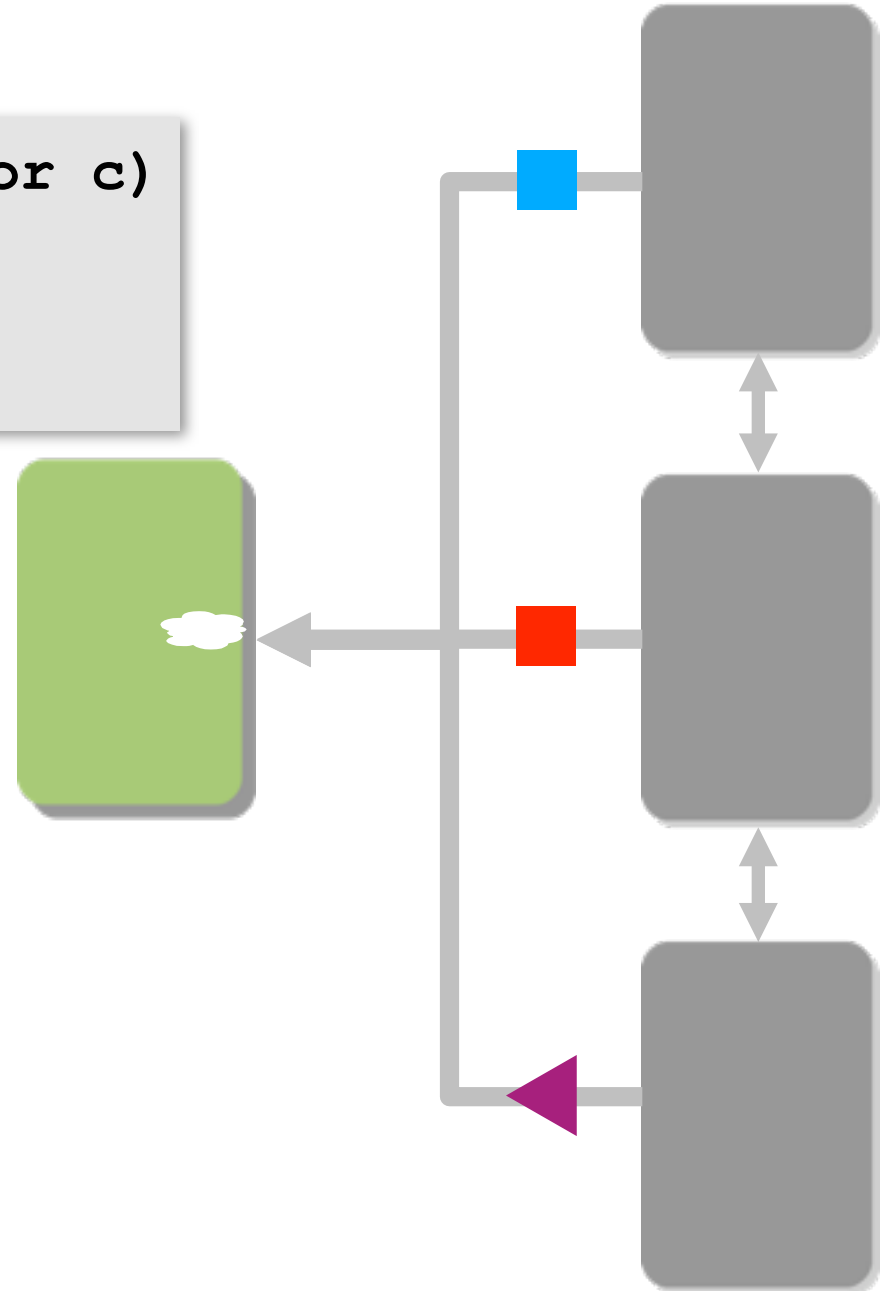
1970's: Type Abstraction

```
void fill<S:Shape>(S thing, Color c)
{
  thing.fillColor := c
}
```



1970's: Type Abstraction

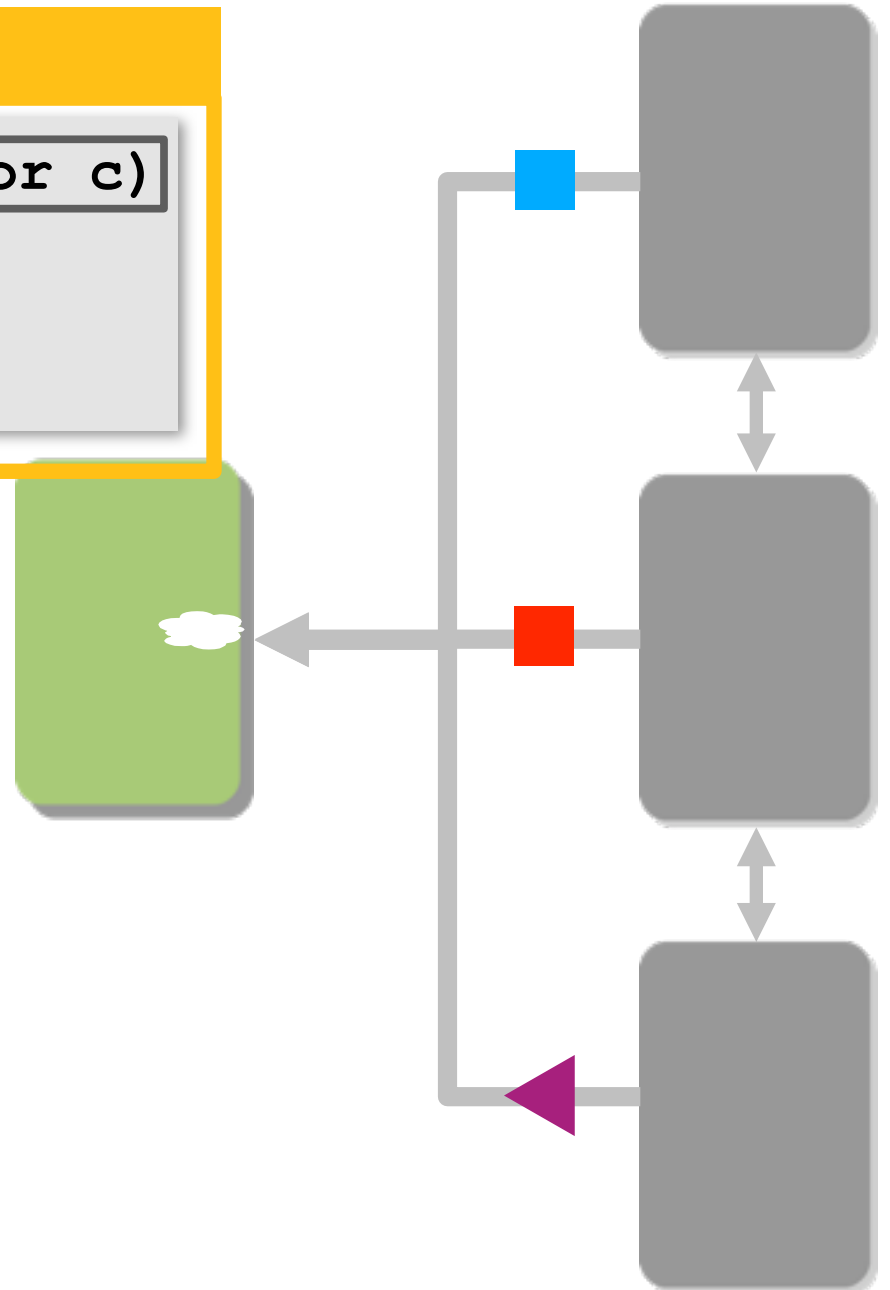
```
void fill<S:Shape>(S thing, Color c)
{
  thing.fillColor := c
}
```



1970's: Type Abstraction

COMPILER

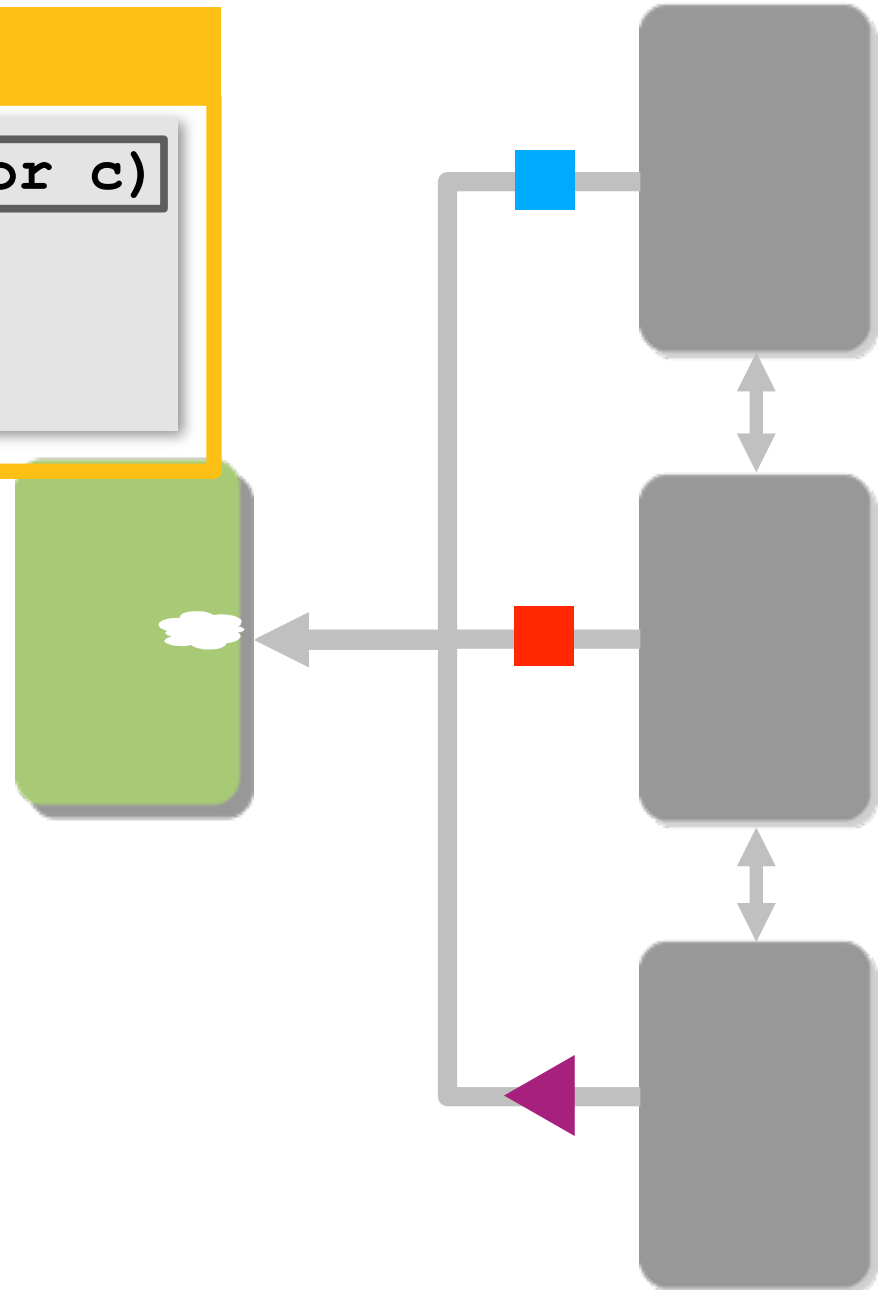

```
void fill<S:Shape>(S thing, Color c)  
{  
  thing.fillColor := c  
}
```



1970's: Type Abstraction

COMPILER

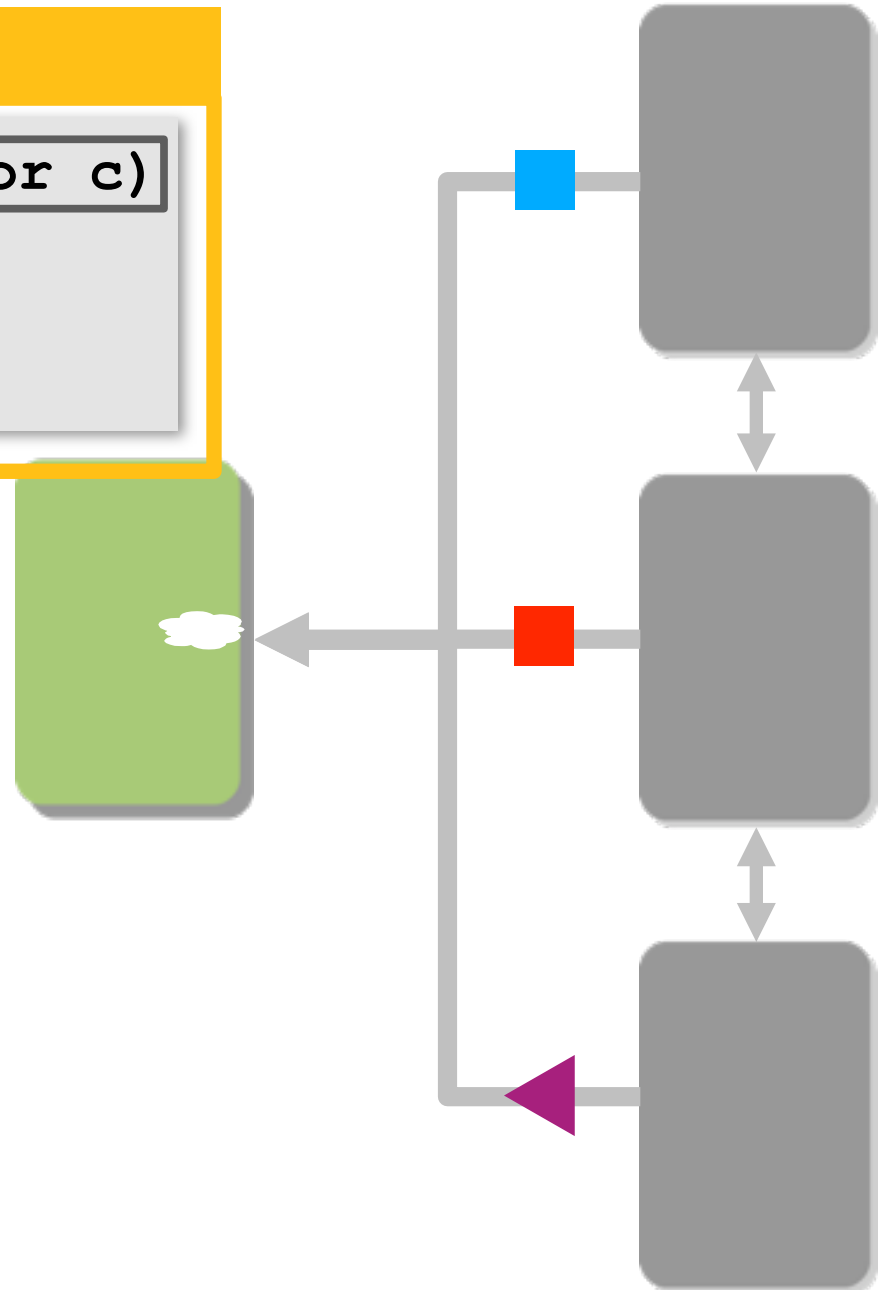
```
void fill<S:Shape>(S thing, Color c)
{
  thing.fillColor := c
}
```



1970's: Type Abstraction

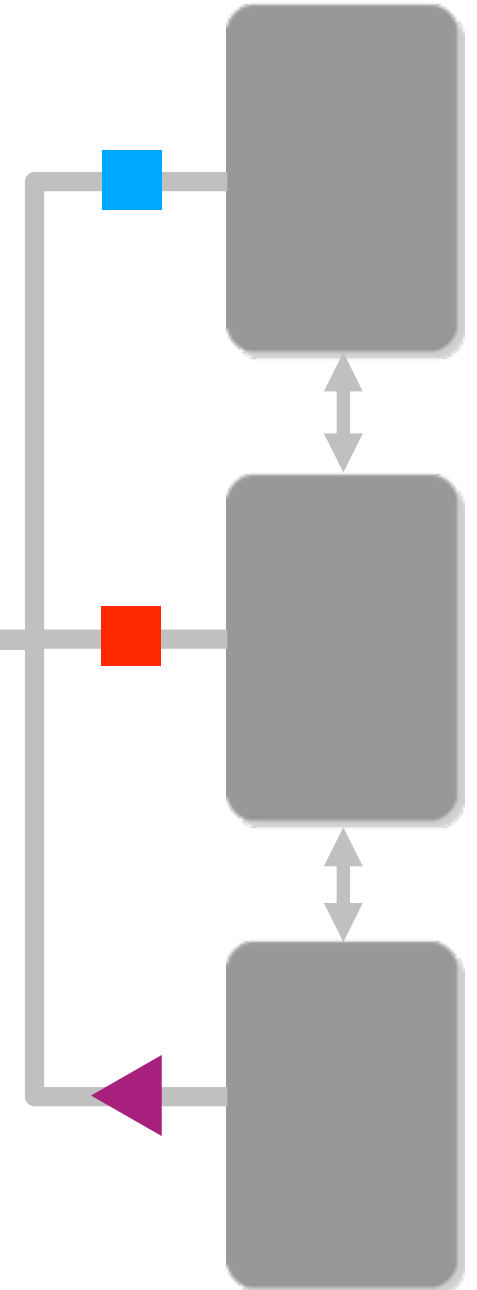
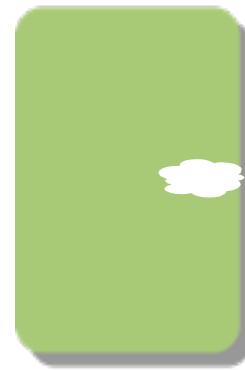
COMPILER

```
void fill<S:Shape>(S thing, Color c)
{
  thing.fillColor := c
}
```



1970's: Type Abstraction

```
void fill<S: Shape>(S thing, Color c)
```

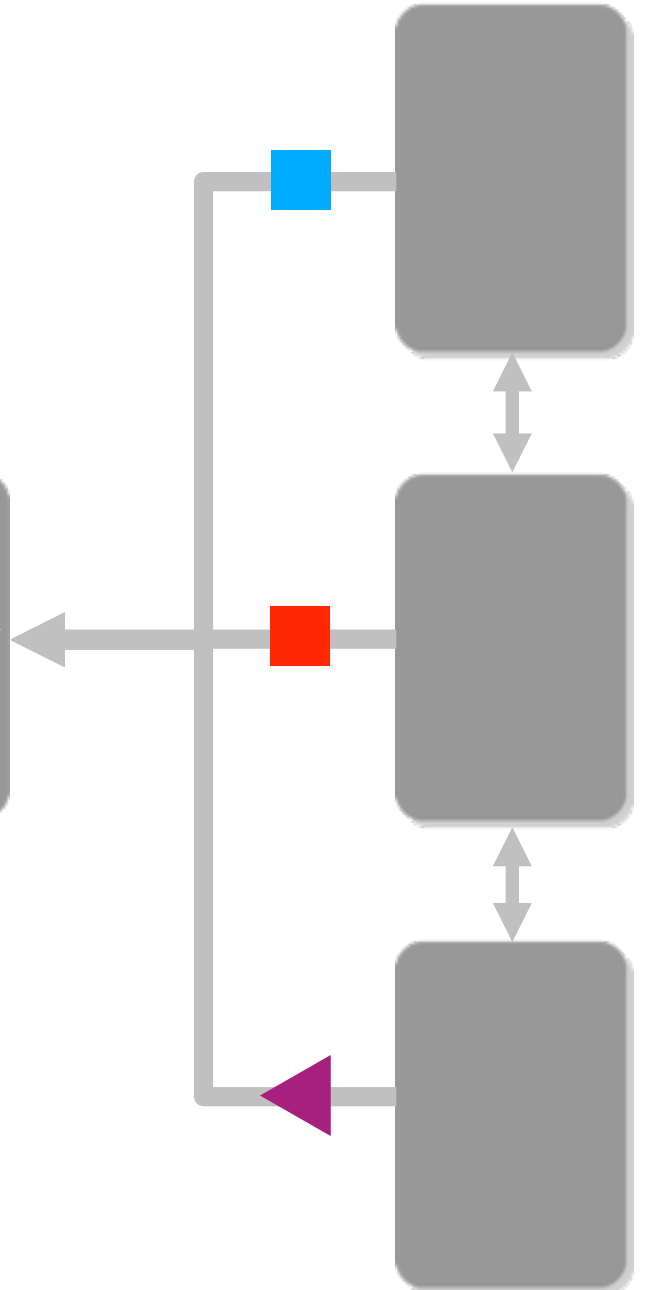
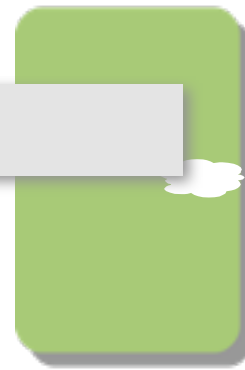


1970's: Type Abstraction

```
void fill<S: Shape>(S thing, Color c)
```



```
fill<String>("foo", red)
```



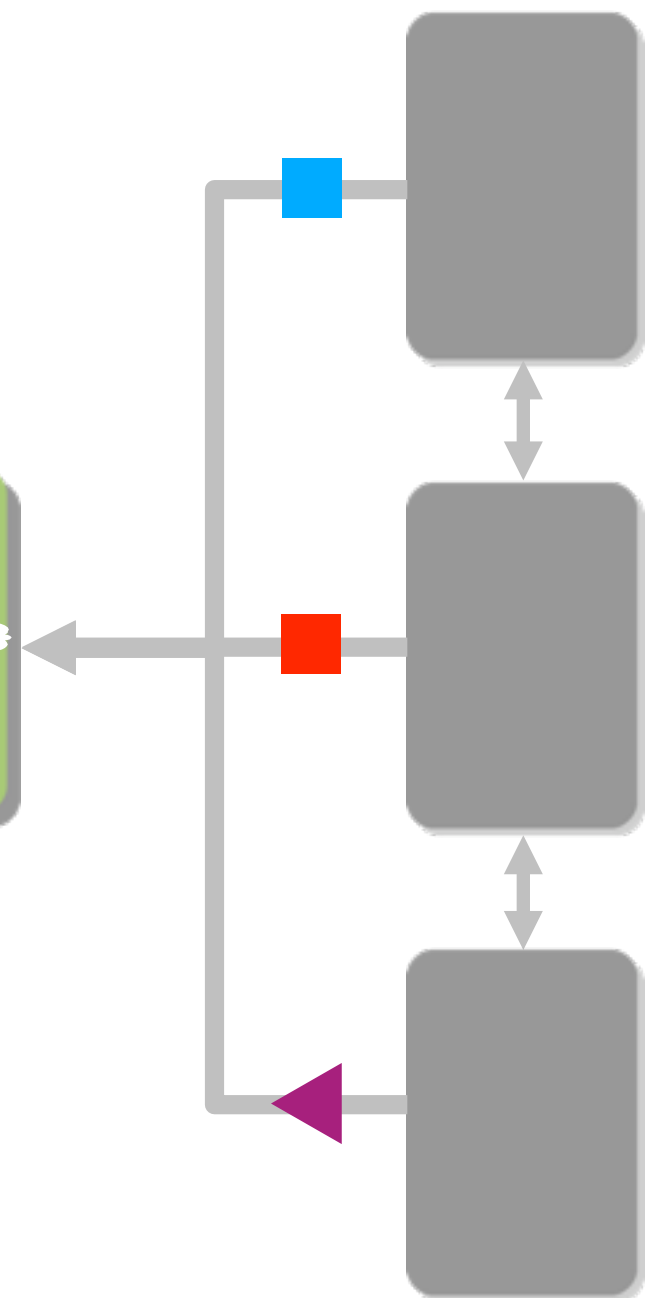
1970's: Type Abstraction

COMPILER

```
void fill<S:Shape>(S thing, Color c)
```



```
fill<String>("foo", red)
```



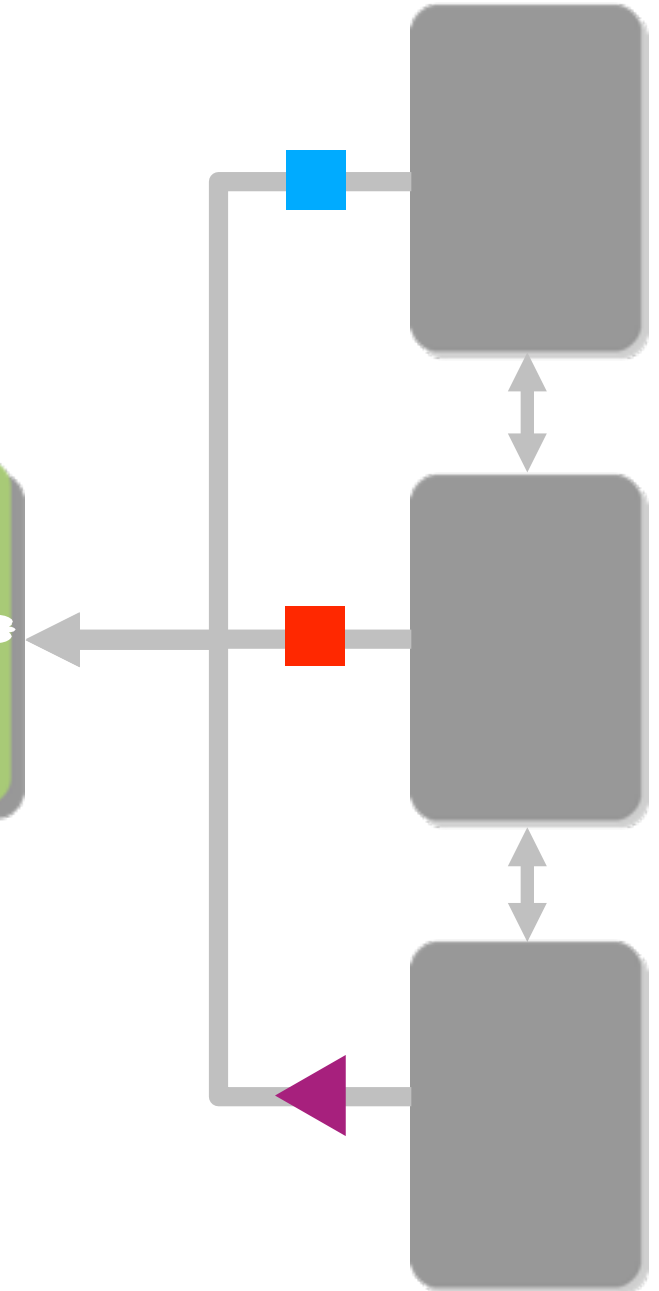
1970's: Type Abstraction

COMPILER

```
void fill<S:Shape>(S thing, Color c)
```



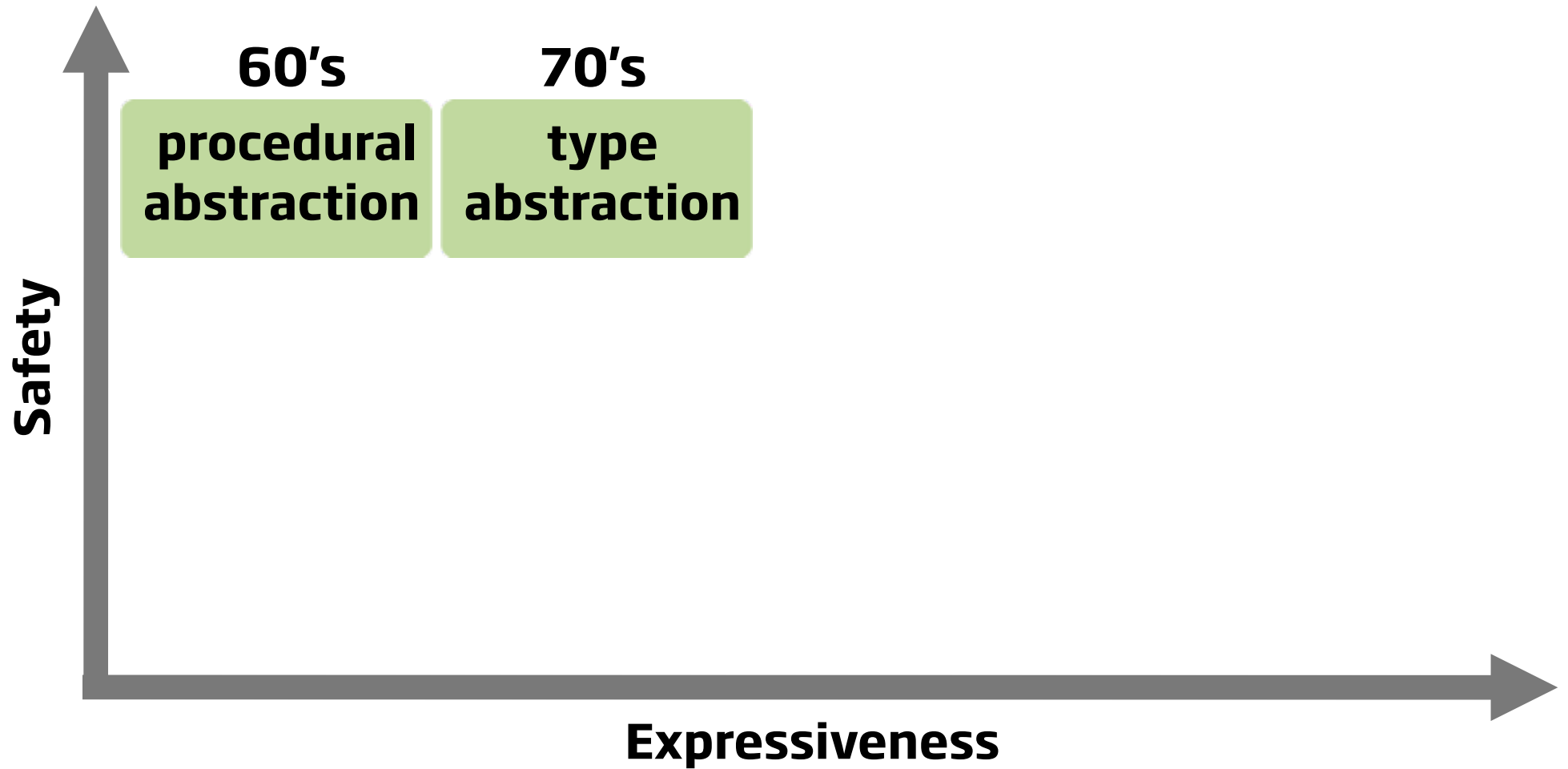
```
fill<String>("foo", red)
```



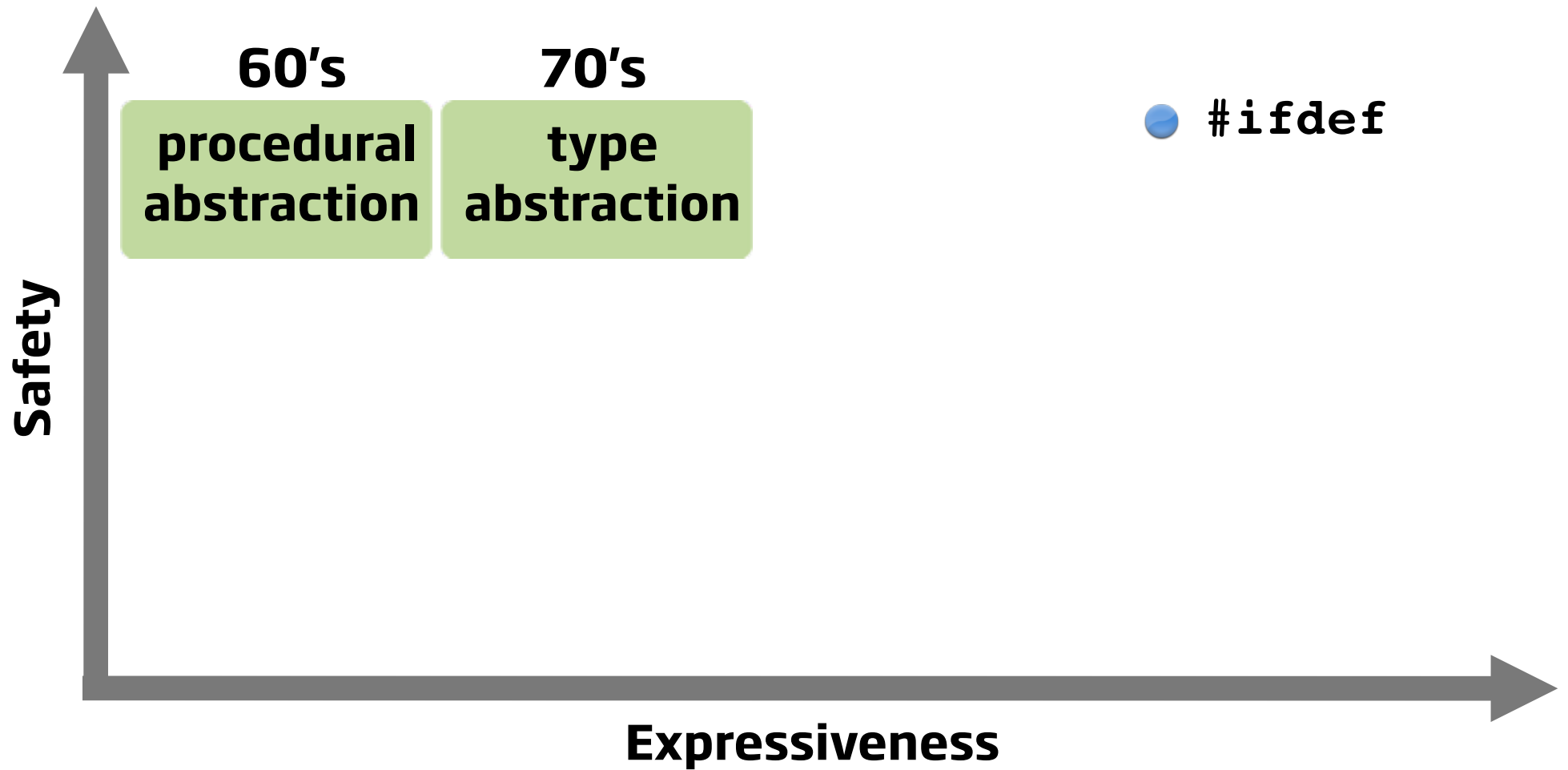
Abstraction Mechanism Design Space



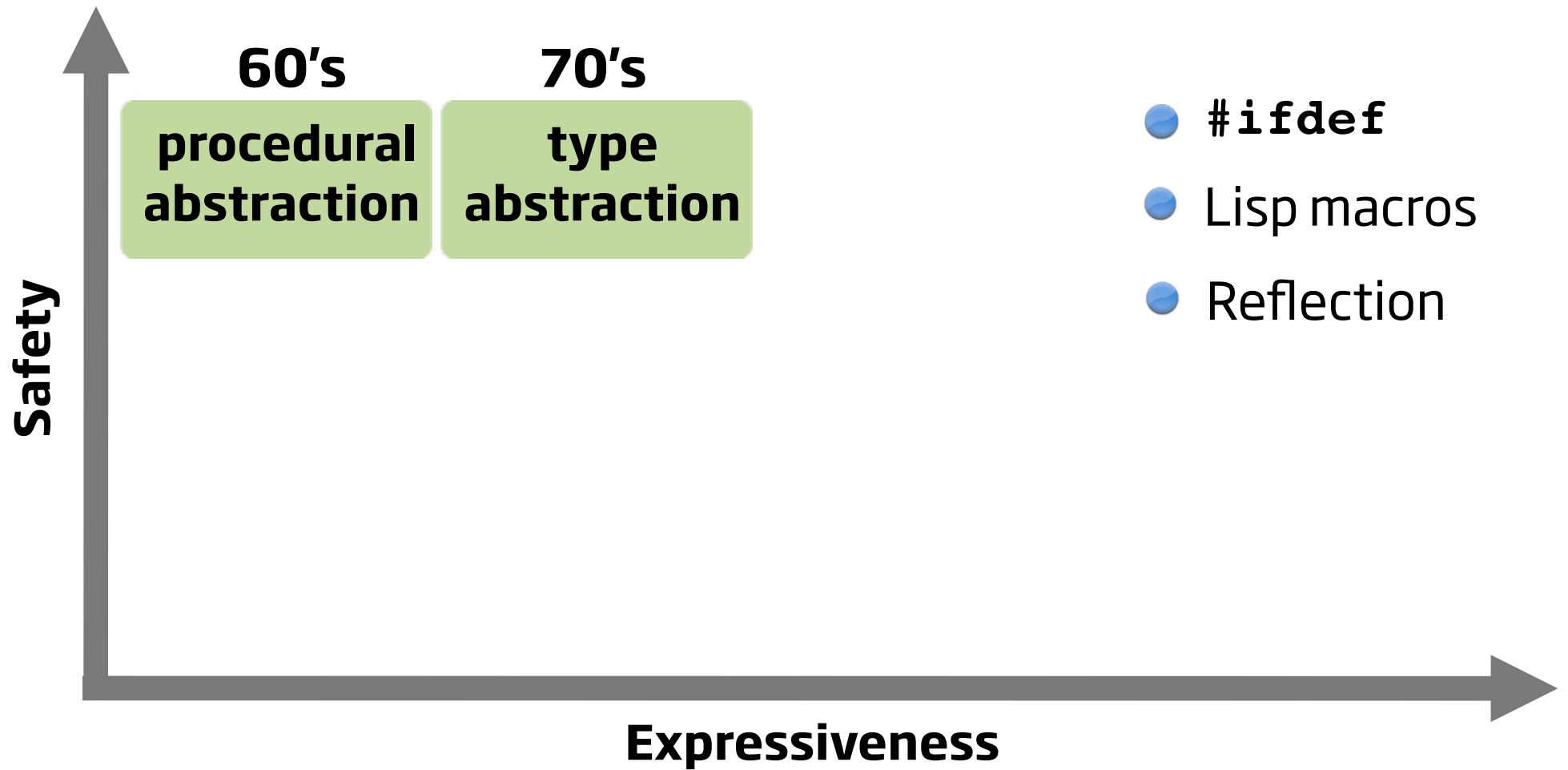
Abstraction Mechanism Design Space



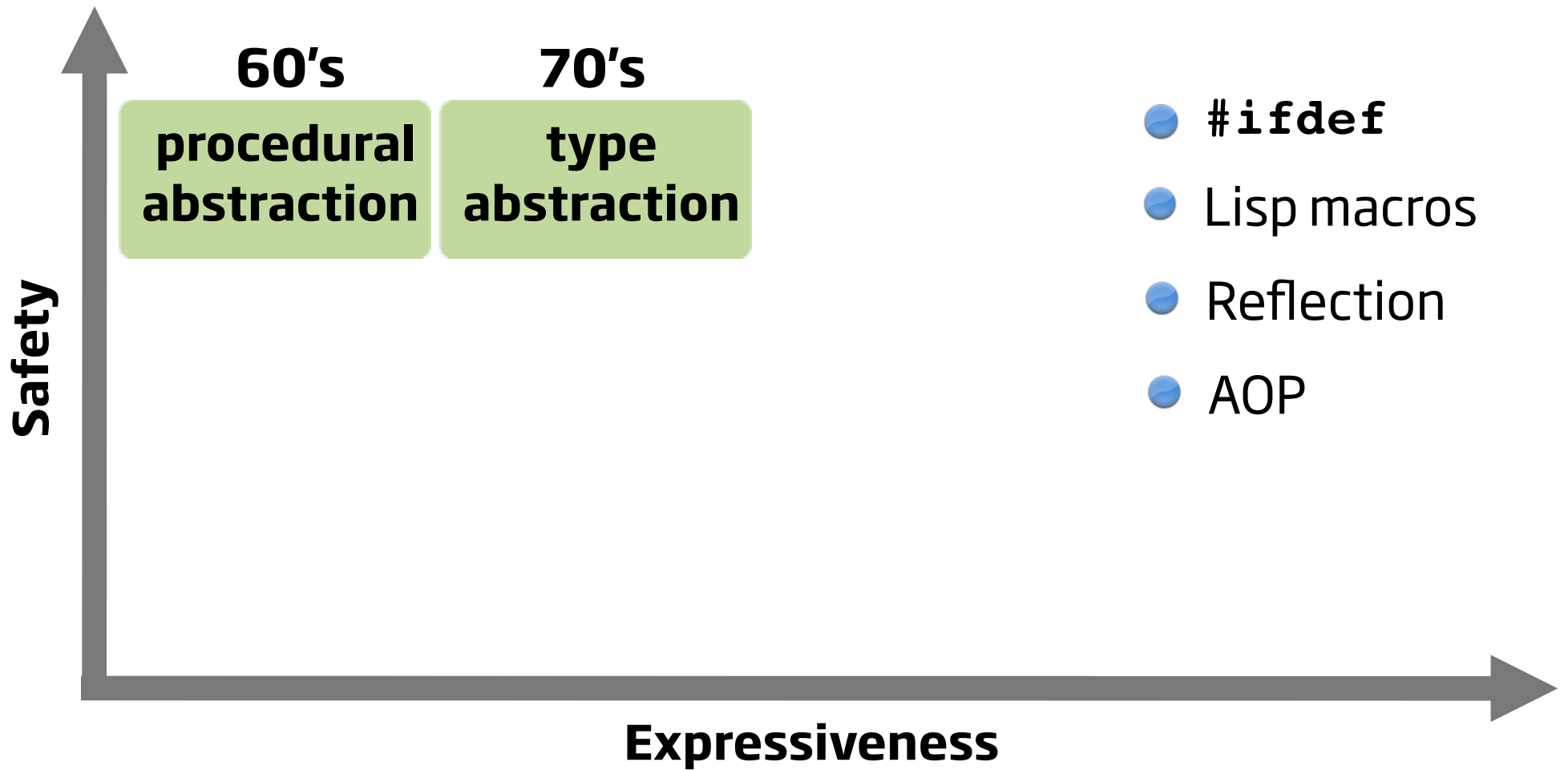
Abstraction Mechanism Design Space



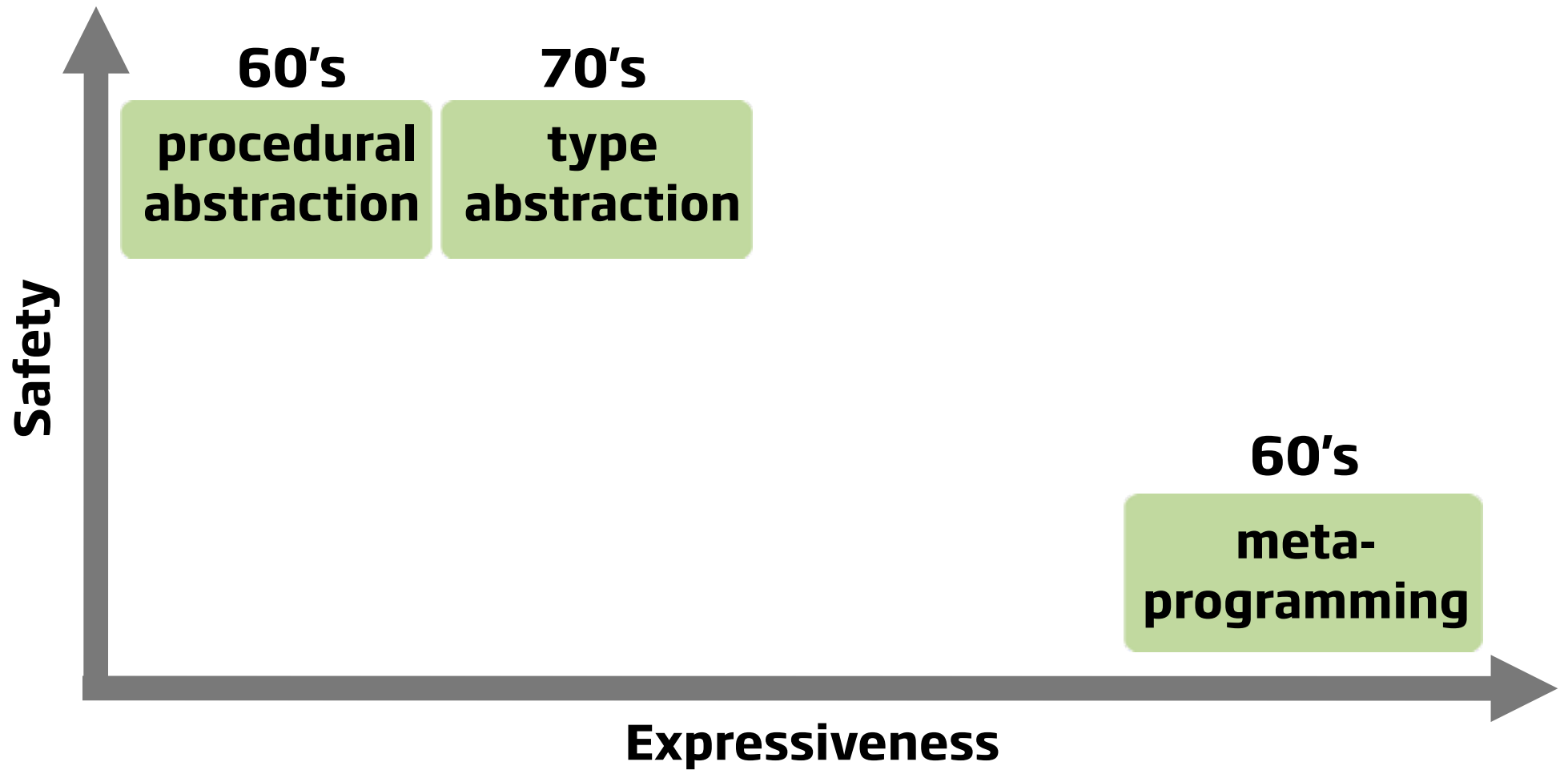
Abstraction Mechanism Design Space



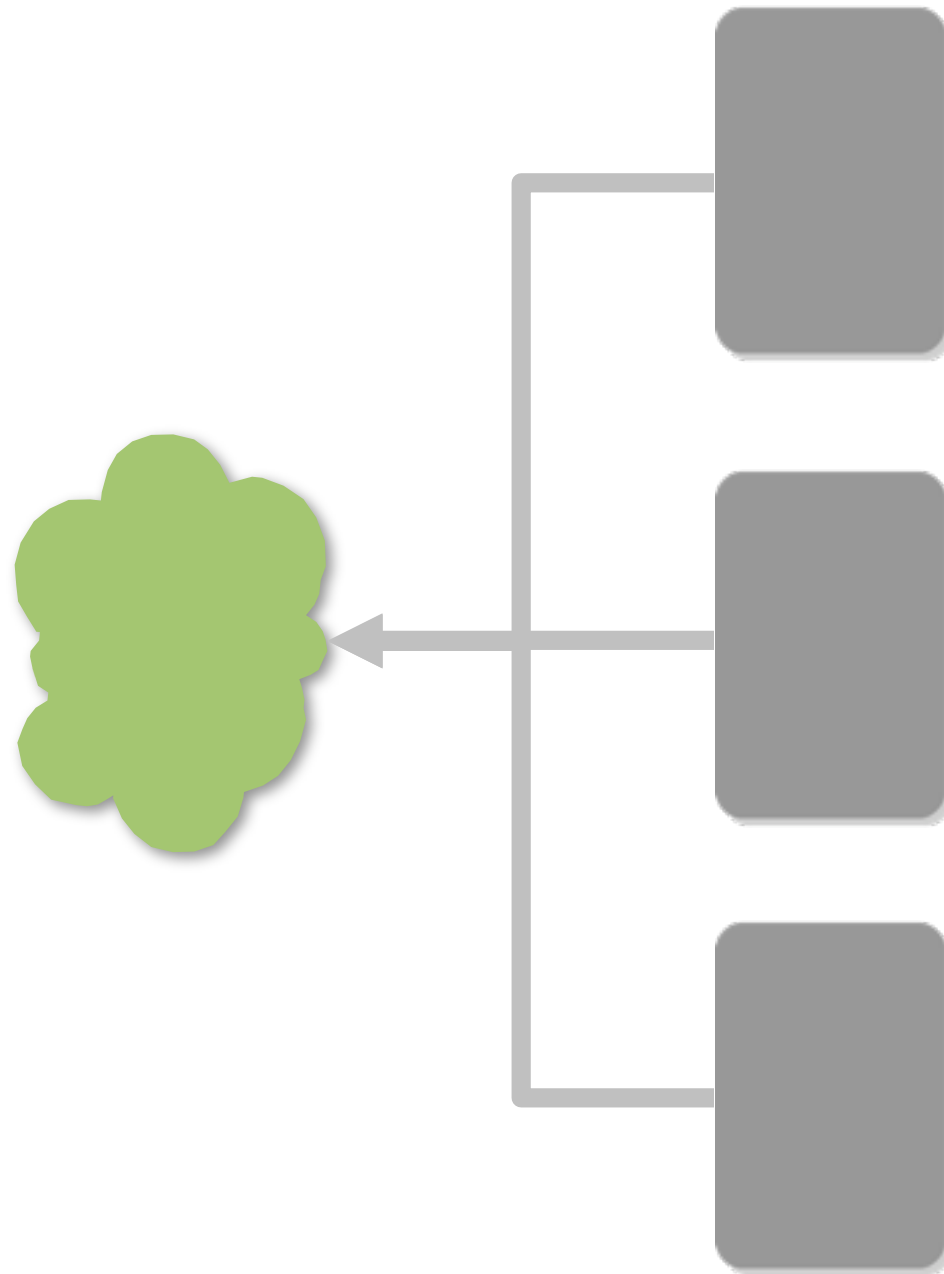
Abstraction Mechanism Design Space



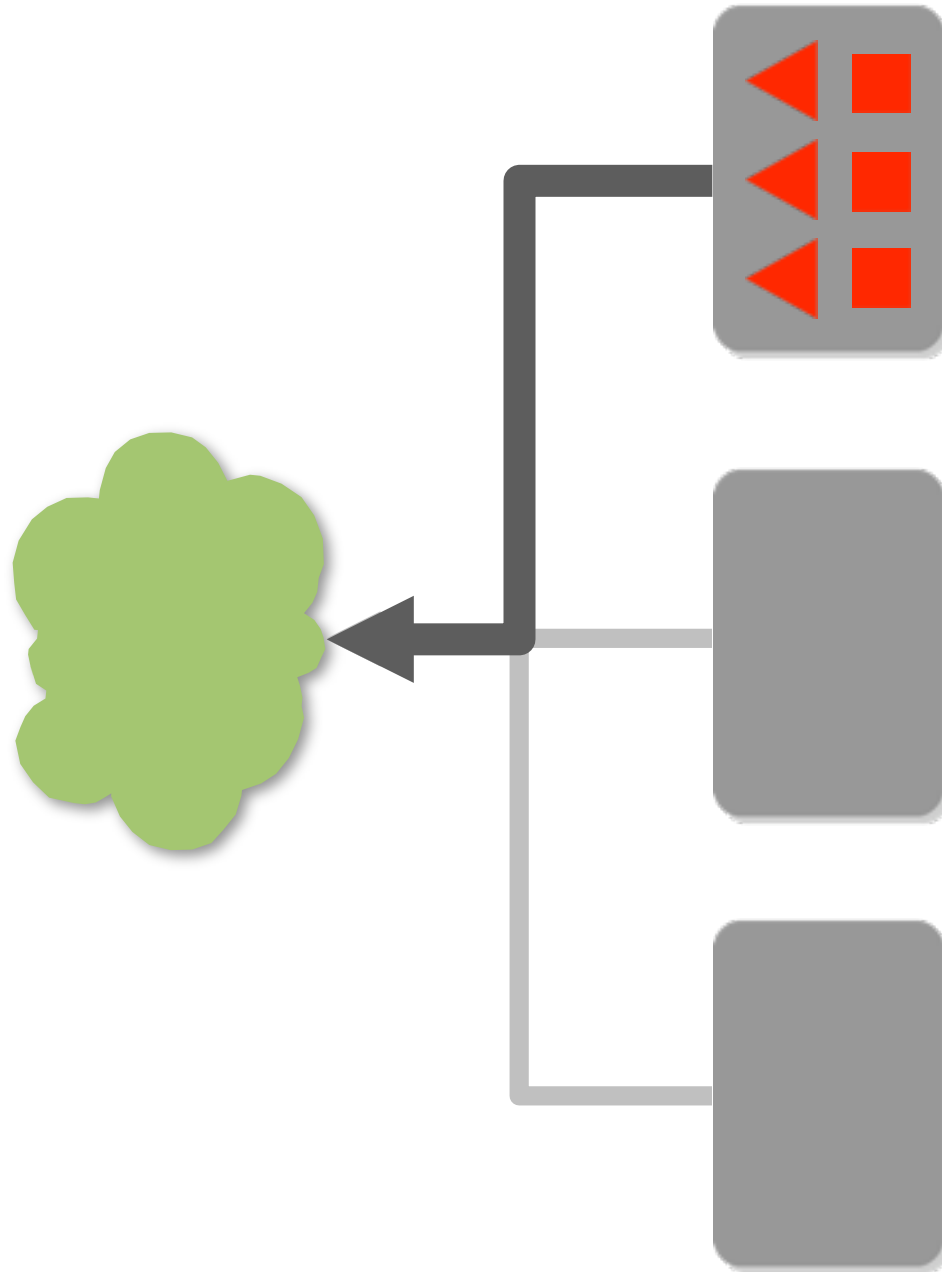
Abstraction Mechanism Design Space



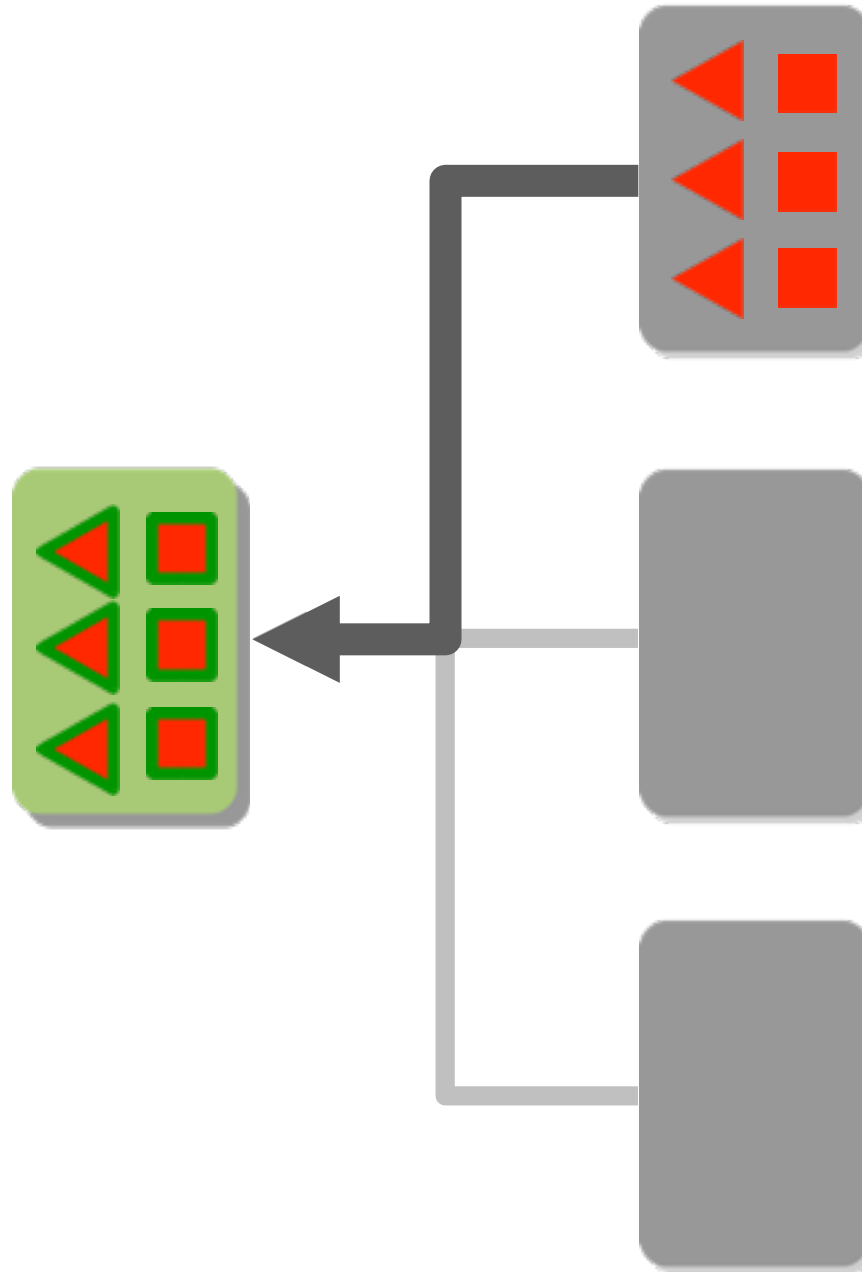
Really Soft Software



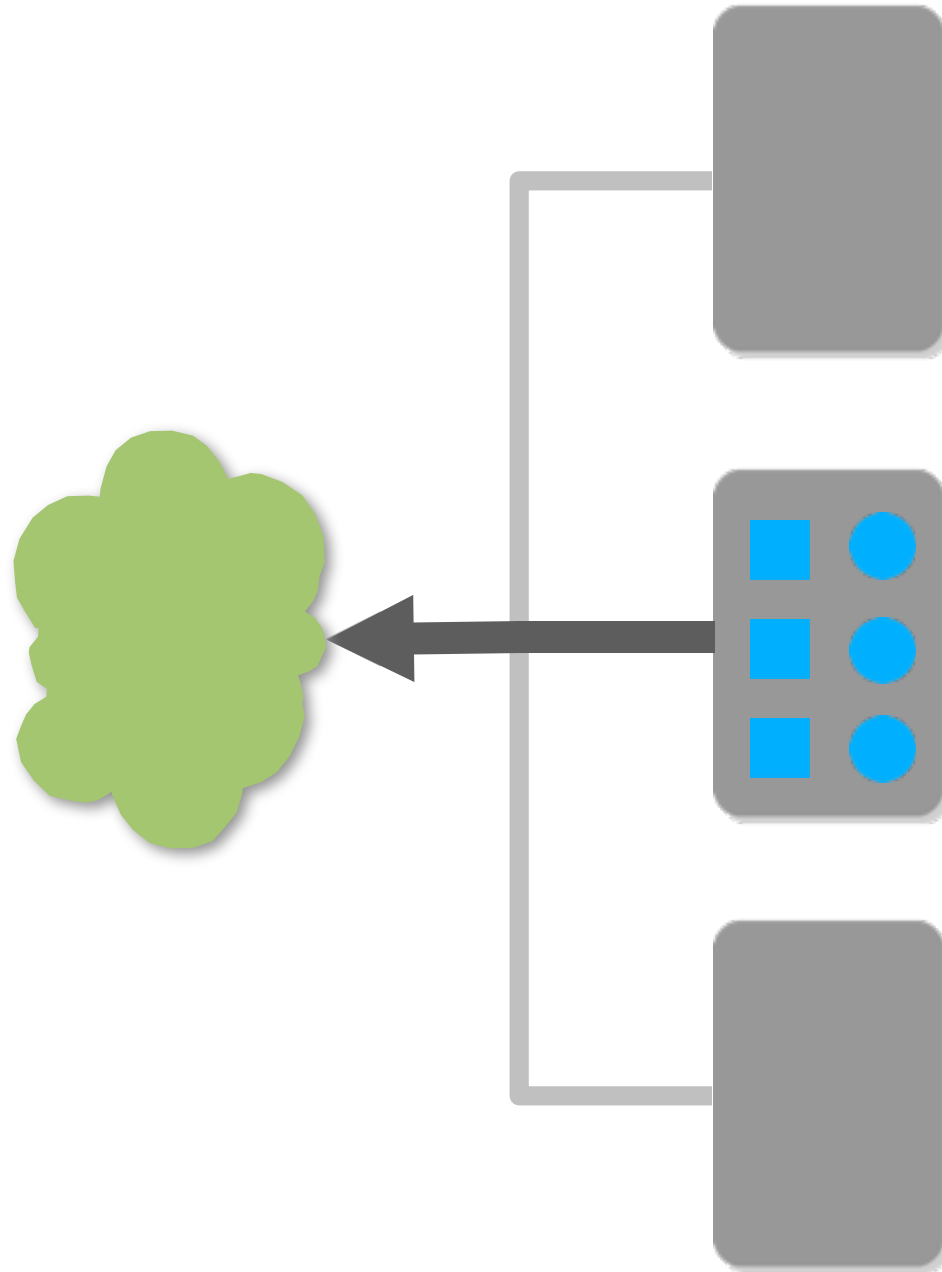
Really Soft Software



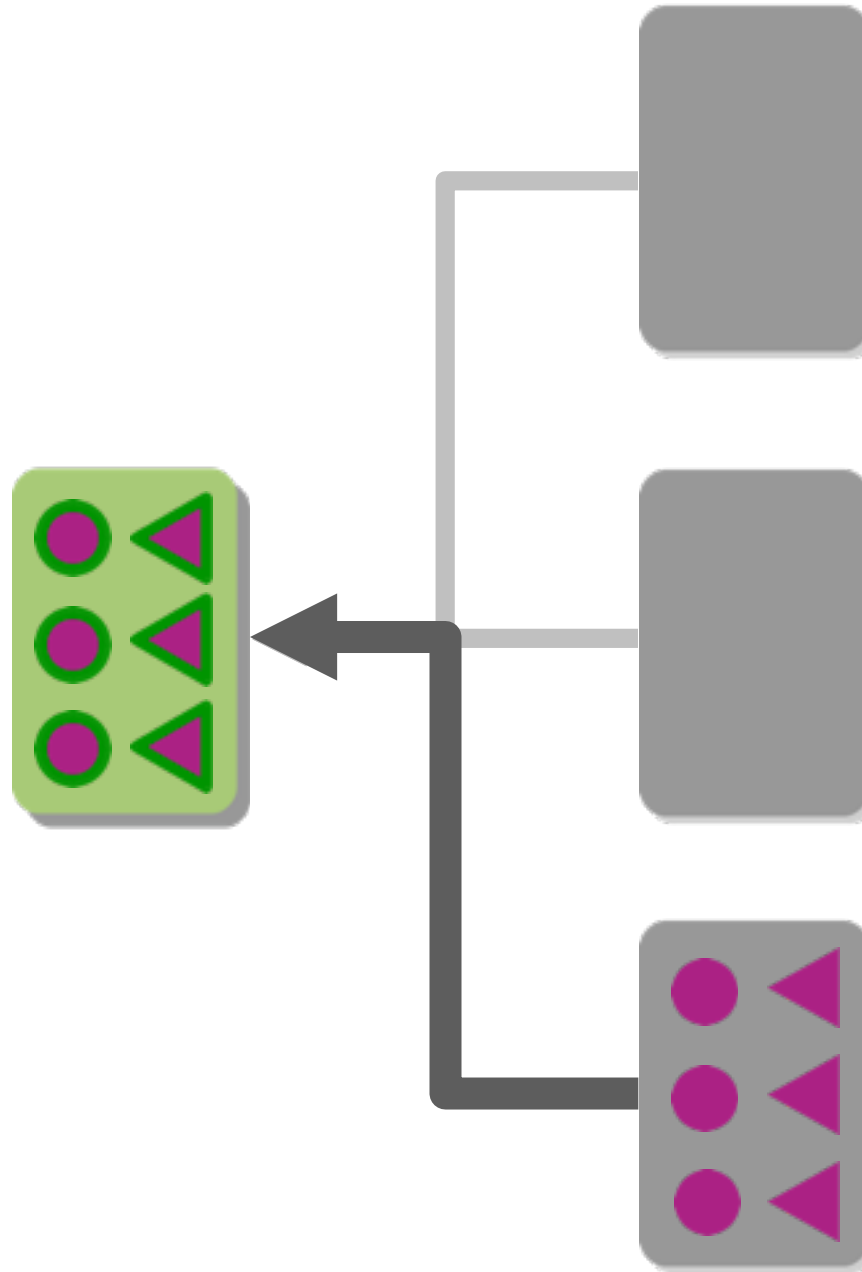
Really Soft Software



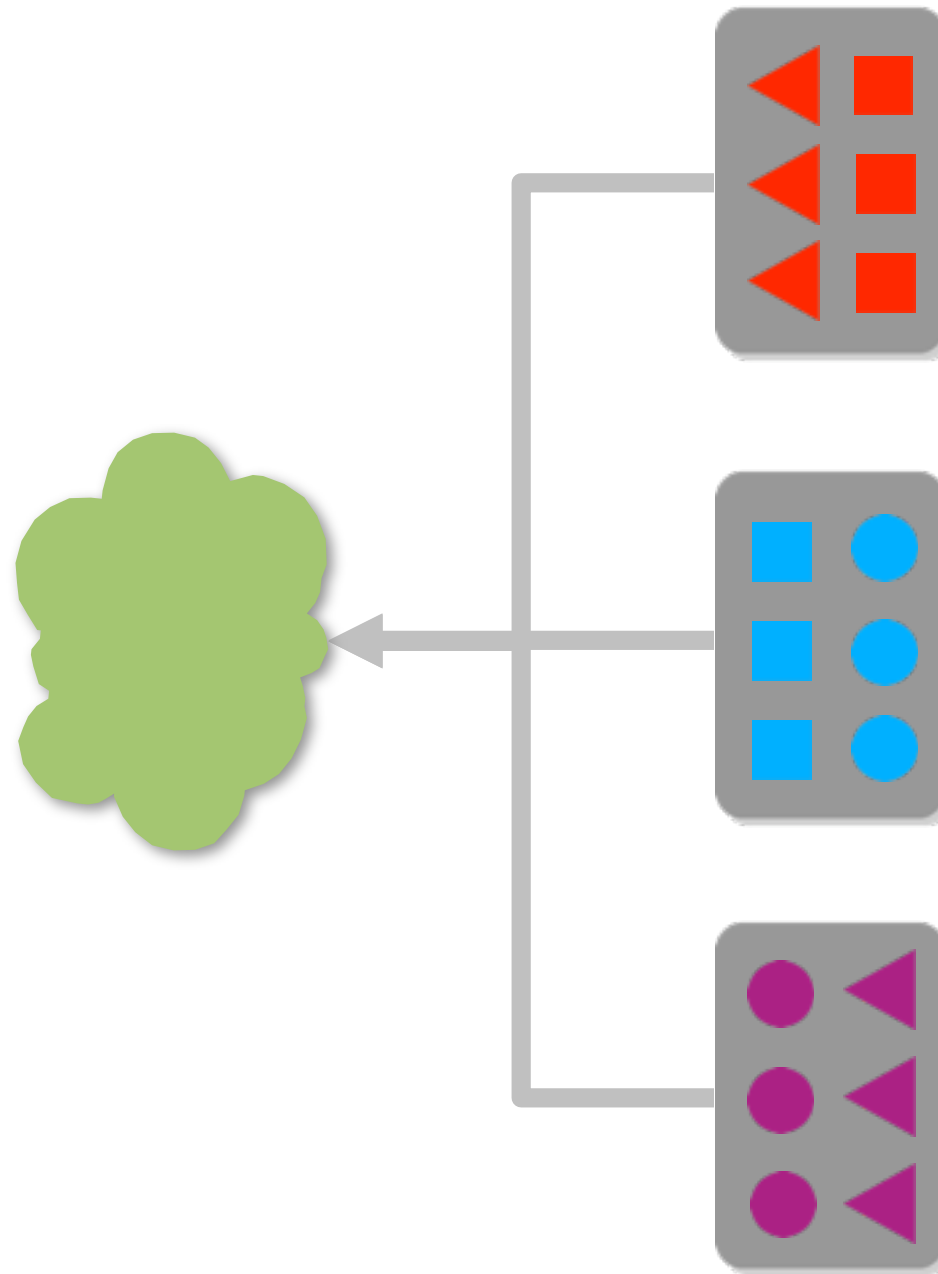
Really Soft Software



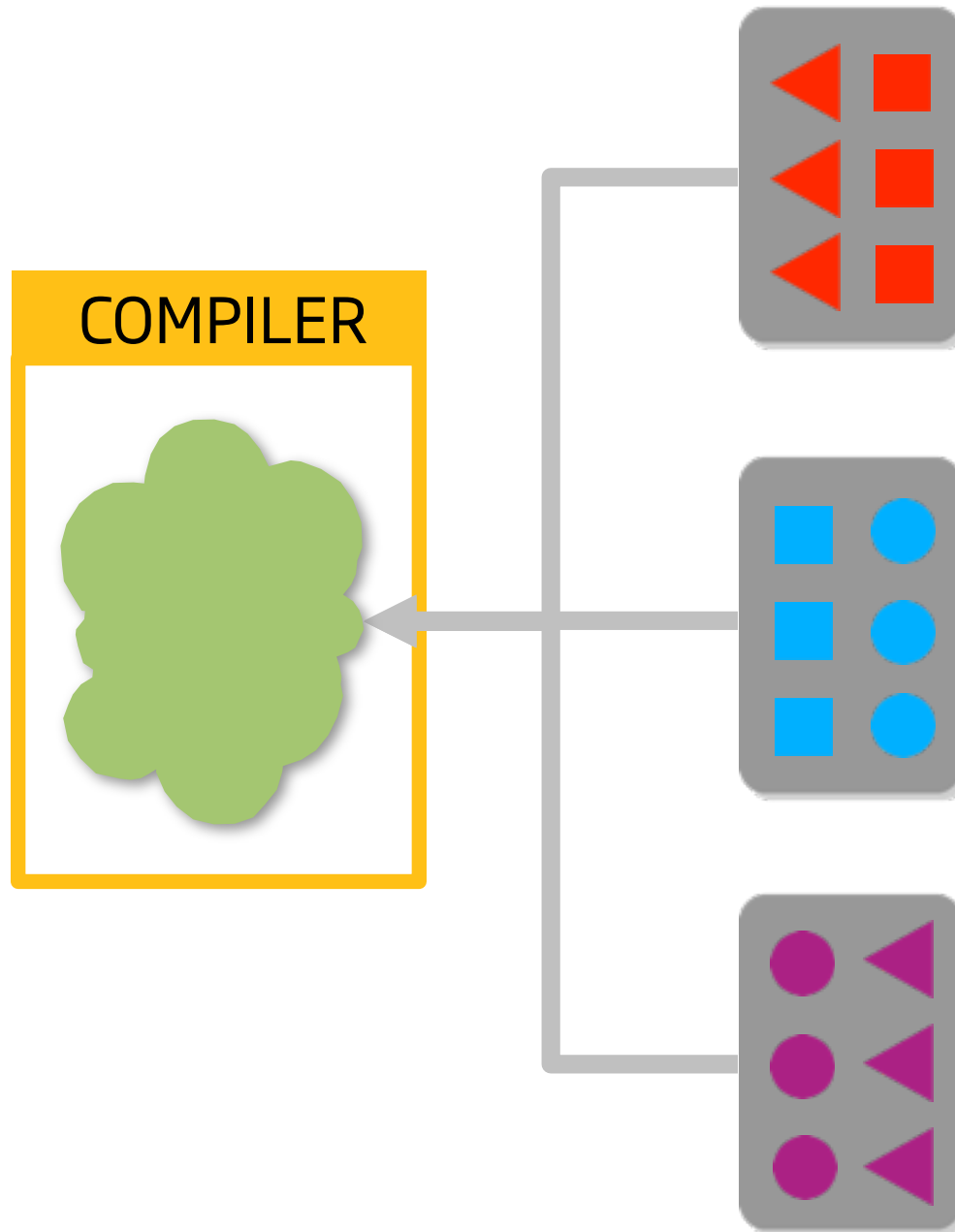
Really Soft Software



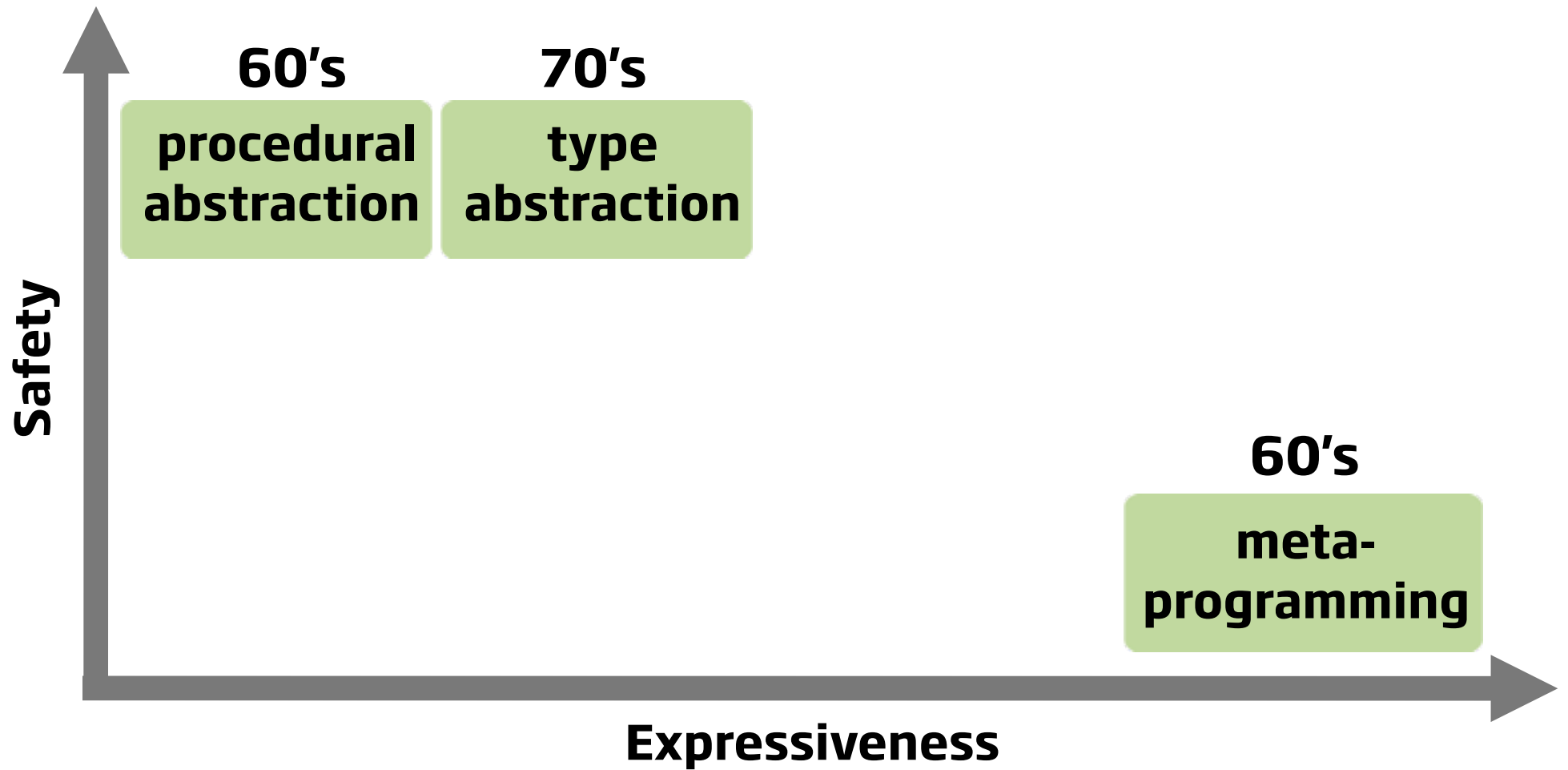
Really Soft Software



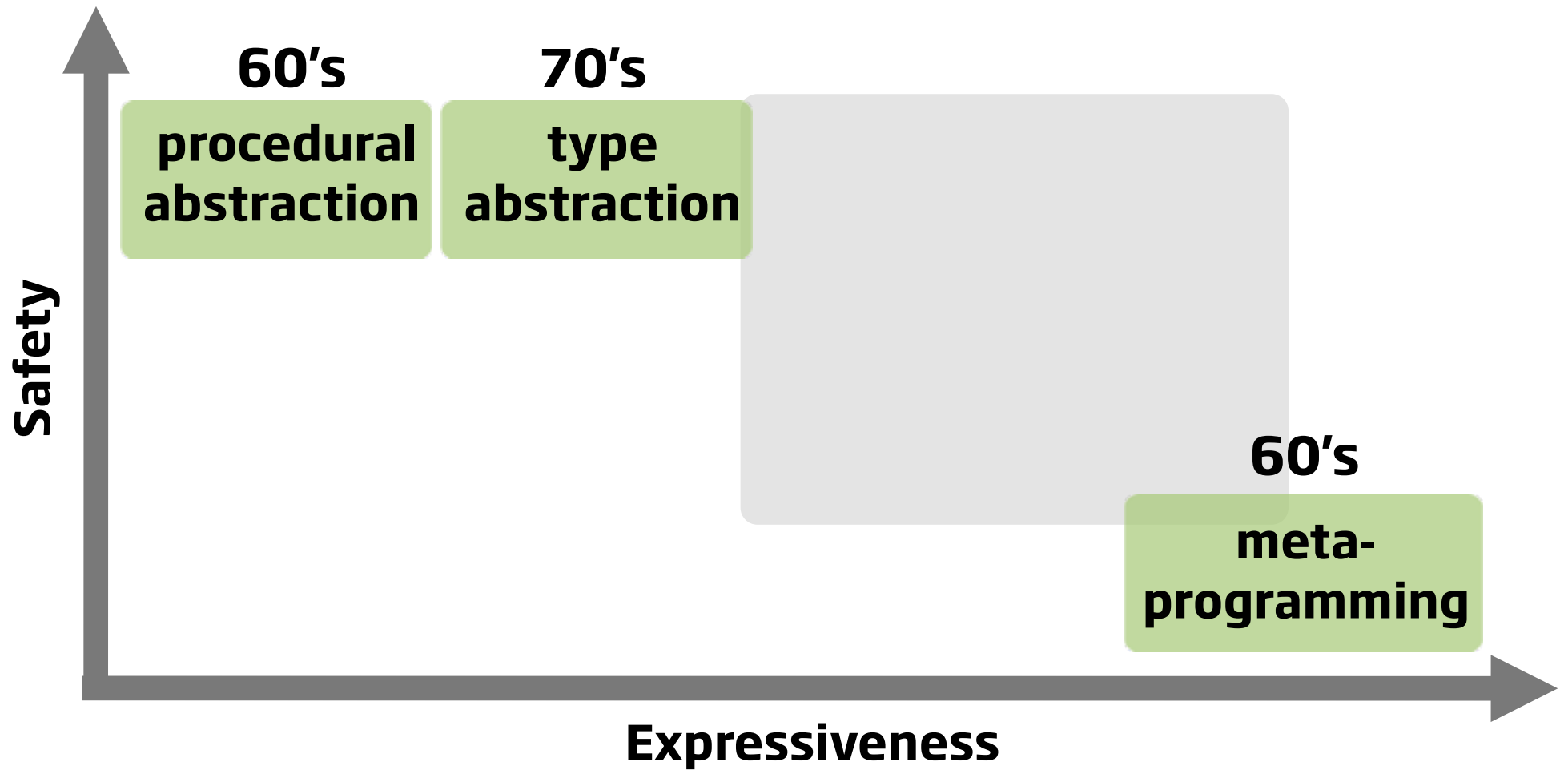
Really Soft Software



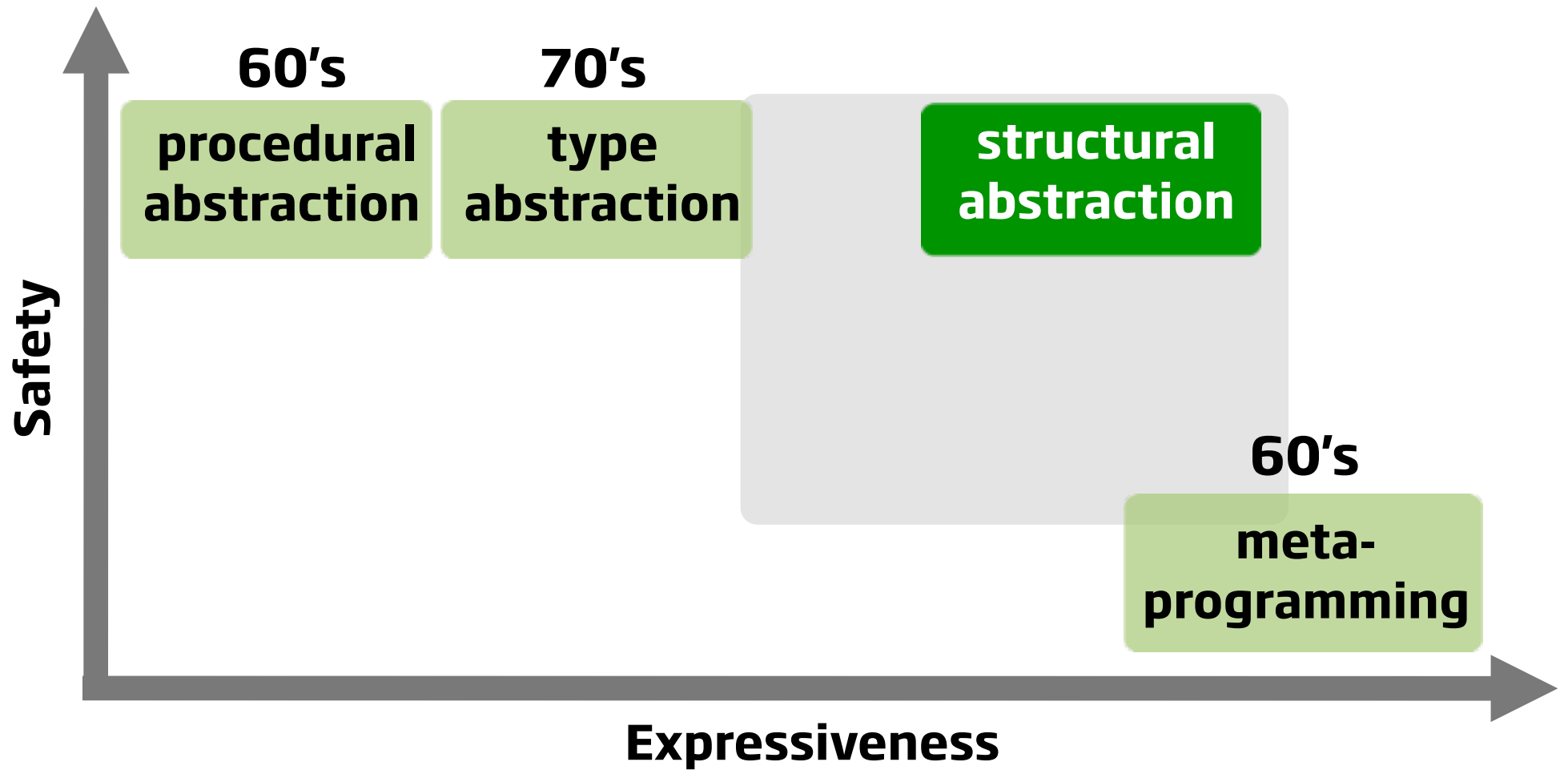
Structural Abstraction



Structural Abstraction



Structural Abstraction



My Research

*Develop **expressive** and **safe** abstraction mechanisms for programming languages.*

My Research

Develop **expressive** and **safe** abstraction mechanisms for programming languages.

- Morphing - statically safe reflection

My Research

Develop **expressive** and **safe** abstraction mechanisms for programming languages.

- Morphing - statically safe reflection
- Static type conditions - configurable libraries
 - **#ifdef** done right!

My Research

Develop **expressive** and **safe** abstraction mechanisms for programming languages.

- Morphing - statically safe reflection
- Static type conditions - configurable libraries
 - `#ifdef` done right!
- Other work
 - Object-Oriented abstraction for hardware programming

My Research

Develop **expressive** and **safe** abstraction mechanisms for programming languages.

- Morphing - statically safe reflection
- Static type conditions - configurable libraries
 - `#ifdef` done right!
- Other work
 - Object-Oriented abstraction for hardware programming

Motivation: Java Collections Framework (JCF)

- A Synchronization Proxy:

```
class SynchronizedList implements List {  
    final List l;  
    final Object mutex;  
  
}
```

Motivation: Java Collections Framework (JCF)

- A Synchronization Proxy:

```
class SynchronizedList implements List {  
    final List l;  
    final Object mutex;  
  
}
```

Motivation: Java Collections Framework (JCF)

- A Synchronization Proxy:

```
class SynchronizedList implements List {  
    final List l;  
    final Object mutex;  
  
    // repeat for all methods of List  
  
}
```

Motivation: Java Collections Framework (JCF)

- A Synchronization Proxy:

```
class SynchronizedList implements List {
    final List l;
    final Object mutex;

    // repeat for all methods of List
    public int size () {
        synchronized(mutex) { return l.size(); }
    }
    public boolean remove (int i) {
        synchronized(mutex) { return l.remove(i); }
    }
    ...
}
```

Motivation: Java Collections Framework (JCF)

- A Synchronization Proxy:

```
class SynchronizedList implements List {
    final List l;
    final Object mutex;

    // repeat for all methods of List
    public int size () {
        synchronized(mutex) { return l.size(); }
    }
    public boolean remove (int i) {
        synchronized(mutex) { return l.remove(i); }
    }
    ...
}
```

Motivation: Java Collections Framework (JCF)

- A Synchronization Proxy:

```
class SynchronizedList implements List {
    final List l;
    final Object mutex;

    // repeat for all methods of List
    public int size () {
        synchronized(mutex) { return l.size(); }
    }
    public boolean remove (int i) {
        synchronized(mutex) { return l.remove(i); }
    }
    ...
}
```

Motivation: Java Collections Framework (JCF)

- A Synchronization Proxy:

```
class SynchronizedCollection implements Collection {
    final Collection c;
    final Object mutex;

    // repeat for all methods of Collection
    public boolean add(Object o) {
        synchronized(mutex) { return c.add(o); }
    }
    public boolean clear() {
        synchronized(mutex) { return c.clear(); }
    }
    ...
}
```

Motivation: Java Collections Framework (JCF)

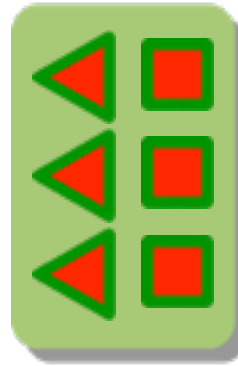
- A Synchronization Proxy:

```
class SynchronizedSet implements Set {
    final Set s;
    final Object mutex;

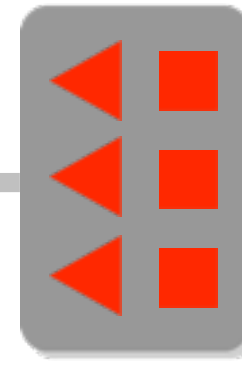
    // repeat for all methods of Set
    public boolean contains(Object o) {
        synchronized(mutex) { return s.contains(o); }
    }
    public boolean isEmpty() {
        synchronized(mutex) { return s.isEmpty(); }
    }
    ...
}
```

Hardcoded Proxies - Not Soft Software

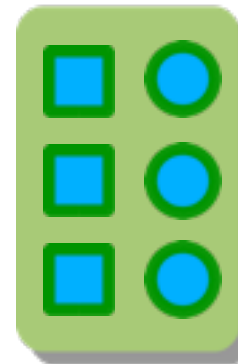
SynchronizedList



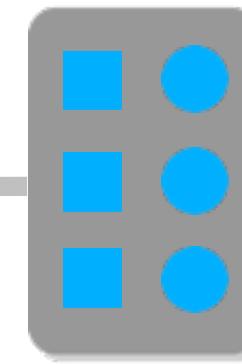
List



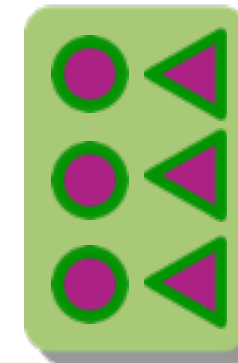
SynchronizedCollection



Collection



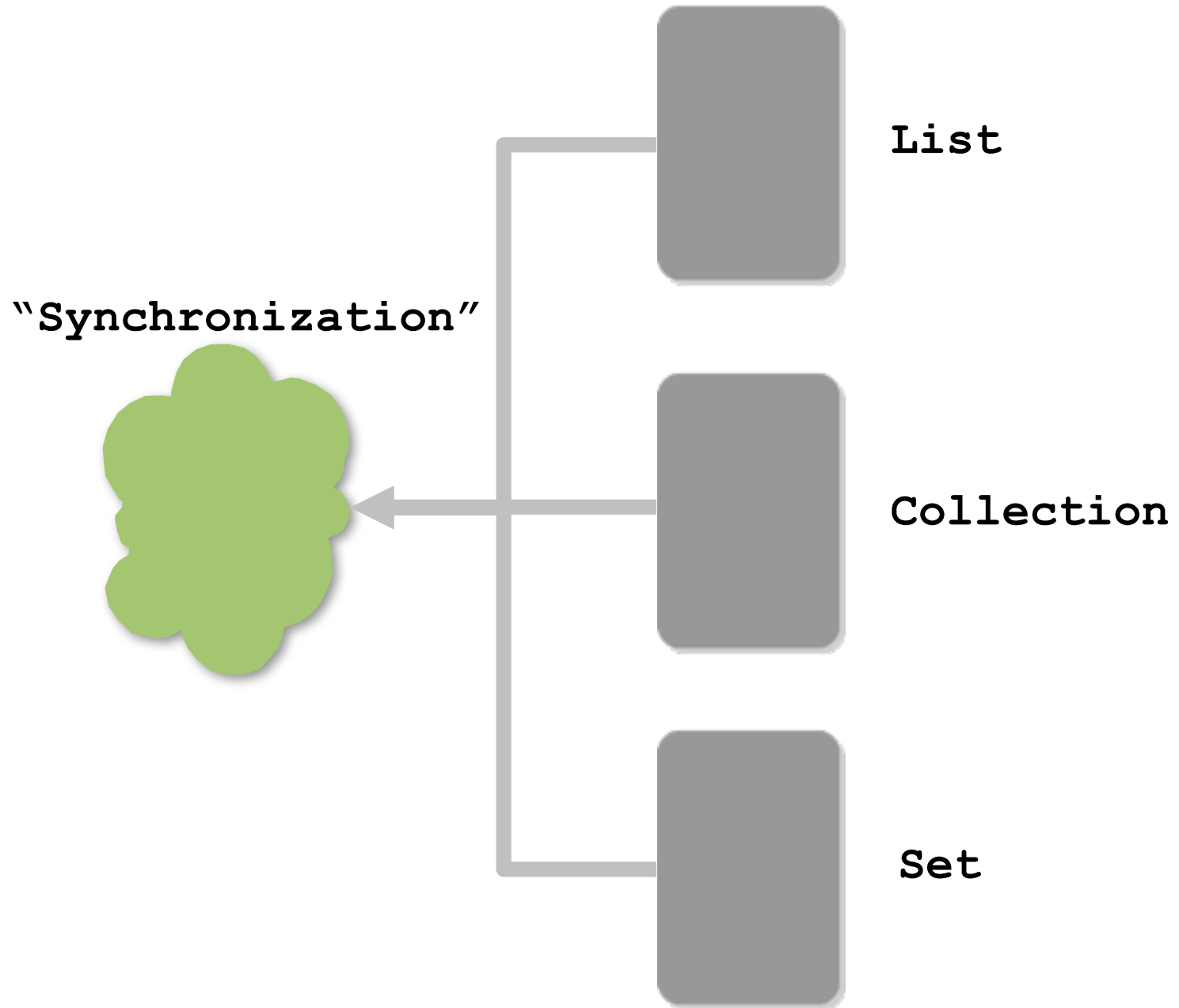
SynchronizedSet



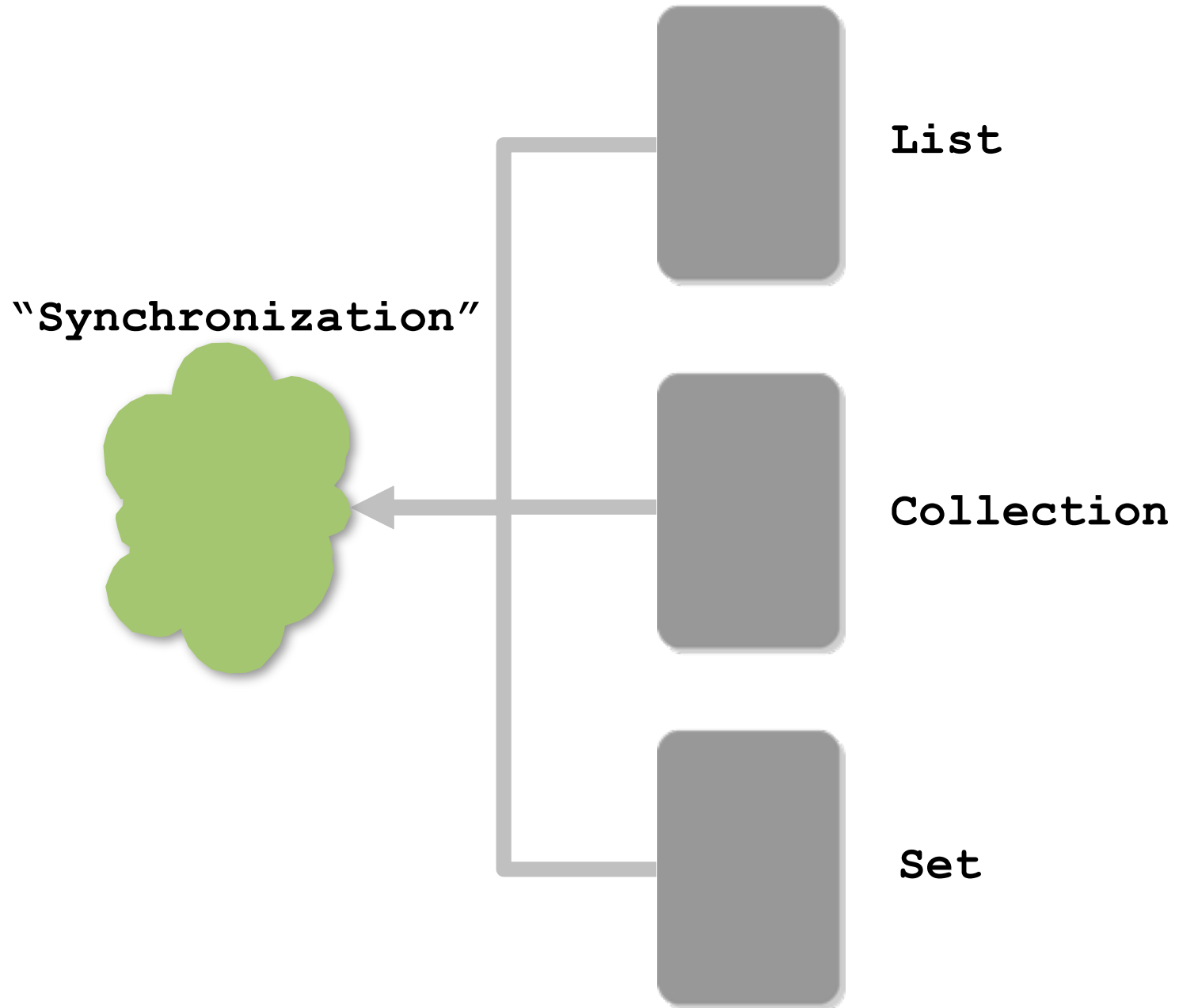
Set



Really Soft Software



Really Soft Software



What I Really Want To Say Is....

```
"Synchronization" for any type T {
```

```
  for each method of T
```

```
    declare a method with the same signature {
```

```
      synchronize on a mutex
```

```
      delegate the call to the real object of T
```

```
    }
```

```
  }
```

What I Really Want To Say Is....

```
"Synchronization" for any type T {
```

```
▶ for each method of T  
  declare a method with the same signature {  
    synchronize on a mutex  
    delegate the call to the real object of T  
  }  
}
```

What I Really Want To Say Is....

```
"Synchronization" for any type T {
```

```
  for each method of T
```

```
    ▶ declare a method with the same signature {
```

```
      synchronize on a mutex
```

```
      delegate the call to the real object of T
```

```
    }
```

```
  }
```

What I Really Want To Say Is....

```
"Synchronization" for any type T {
```

```
  for each method of T
```

```
    declare a method with the same signature {
```

```
      synchronize on a mutex
```

```
      delegate the call to the real object of T
```

```
    }
```

```
  }
```

What I Really Want To Say Is....

```
"Synchronization" for any type T {
```

```
  for each method of T
```

```
    declare a method with the same signature {
```

```
      synchronize on a mutex
```

```
      delegate the call to the real object of T
```

```
    }
```

```
  }
```

What Do People Do In Practice?


- Meta-programming techniques
 - Reflection
 - Code generation using string templates, bytecode engineering
 - AOP
 - etc.

What Do People Do In Practice?

- Meta-programming techniques
 - Reflection
 - Code generation using string templates, bytecode engineering
 - AOP
 - etc.
- Problem: ***No separate type checking!***

Morphing: Safe Static Reflection

Synchronization Proxy using MorphJ



```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;  
  
}
```


Synchronization Proxy using MorphJ

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;  
  
}
```

Synchronization Proxy using MorphJ

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;  
  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) {  
        synchronized(mutex) { return me.m(args); }  
    }  
}
```

Synchronization Proxy using MorphJ

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;  
  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) {  
        synchronized(mutex) { return me.m(args); }  
    }  
}
```

Synchronization Proxy using MorphJ

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) {  
        synchronized(mutex) { return me.m(args); }  
    }
```

```
}
```

Synchronization Proxy using MorphJ

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;  
  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) {  
        synchronized(mutex) { return me.m(args); }  
    }  
}
```

Synchronization Proxy using MorphJ

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;  
  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) {  
        synchronized(mutex) { return me.m(args); }  
    }  
}
```

Morphing Your Code

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;  
  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) {  
        synchronized(mutex) { return me.m(args); }  
    }  
}
```

Morphing Your Code

```
class Synchronized<interface I> {  
    final I me;  
    final Object mutex;
```

```
interface List {  
    int size();  
    Object get (int i);  
    void clear();  
    ...  
}
```

```
<R,A*>[m] for ( R m (A) : I.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}
```

```
}
```

Morphing Your Code

```
class Synchronized<List> implements  
    final List me;  
    final Object mutex;
```

```
interface List {  
    int size();  
    Object get (int i);  
    void clear();  
    ...  
}
```

```
<R,A*>[m] for ( R m (A) : List.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}  
}
```

Morphing Your Code

```
class Synchronized<List> implements  
    final List me;  
    final Object mutex;
```

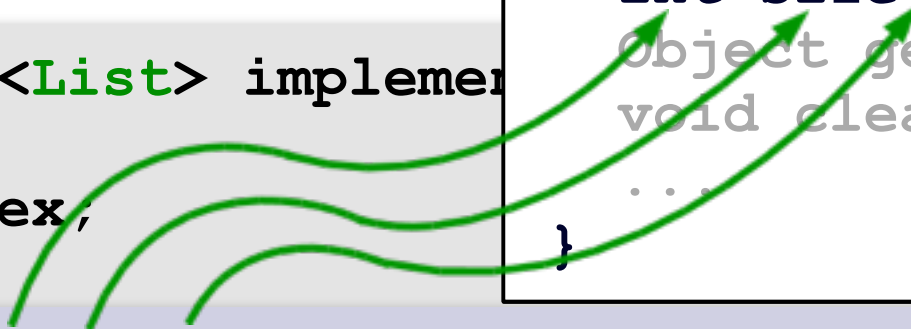
```
interface List {  
    int size();  
    Object get (int i);  
    void clear();  
    ...  
}
```

```
<R,A*>[m] for ( R m (A) : List.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}  
}
```

Morphing Your Code

```
class Synchronized<List> implements  
    final List me;  
    final Object mutex;
```

```
interface List {  
    int size();  
    Object get (int i);  
    void clear();  
    ...  
}
```



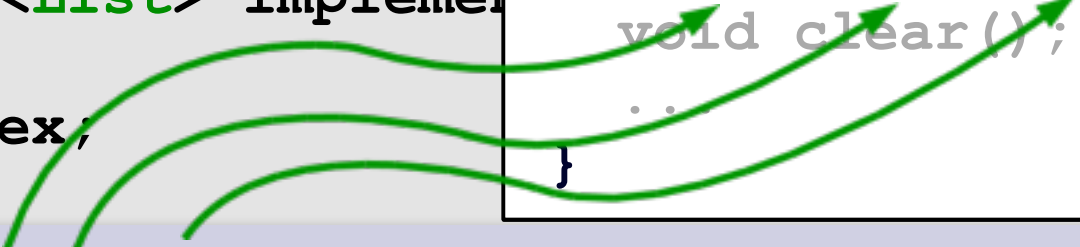
```
<R,A*>[m] for ( R m (A) : List.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}  
}
```

Morphing Your Code

```
class Synchronized<List> implements  
    final List me;  
    final Object mutex;
```

```
<R,A*>[m] for ( R m (A) : List.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}
```


```
interface List {  
    int size();  
    Object get (int i);  
    void clear();  
    ..  
}
```



Morphing Your Code

```
class Synchronized<List> implements  
    final List me;  
    final Object mutex;
```

```
interface List {  
    int size();  
    Object get (int i);  
    void clear();  
    ...  
}
```



```
<R,A*>[m] for ( R m (A) : List.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}
```

```
}
```

Morphing Your Code

```
class Synchronized<List> implements  
    final List me;  
    final Object mutex;
```

```
interface List {  
    int size();  
    Object get (int i);  
    void clear();  
    ...  
}
```

```
<R,A*>[m] for ( R m (A) : List.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}  
}
```

Morphing Your Code

```
class Synchronized<List> implements  
    final List me;  
    final Object mutex;
```

```
public int size () {  
    synchronized(mutex) { return me.size(); }  
}
```

```
public Object get(int args) {  
    synchronized(mutex) { return me.get(args); }  
}
```

```
}
```

```
interface List {  
    int size();  
    Object get (int i);  
    void clear();  
    ...  
}
```

Morphing Your Code

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;  
  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) {  
        synchronized(mutex) { return me.m(args); }  
    }  
}
```

Morphing Your Code

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
<R,A*>[m] for ( R m (A) : I.methods )
```

```
publ  
    sy  
}
```

Hardcoded Proxies: 314 LOC

```
}
```

Morphing Your Code

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
<R,A*>[m] for ( R m (A) : I.methods )
```

```
publ  
    sy  
}
```

Hardcoded Proxies: 314 LOC

MorphJ: 26 LOC

```
}
```

Morphing: Safe Static Reflection

Separate Type Checking

Checking Validity of Reference

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;  
  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) {  
        synchronized(mutex) { return me.m(args); }  
    }  
}
```

Checking Validity of Reference

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;  
  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) {  
        synchronized(mutex) { return me.m(args); }  
    }  
}
```

Checking Validity of Reference

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
<R,A*>[m] for ( R m (A) : I.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}
```

```
}
```

Checking Validity of Reference

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
<R,A*>[m] for ( R m (A) : Foo<I>.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}
```

```
}
```

Checking Validity of Reference

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
<R,A*>[m] for ( R m (A) : Foo<I>.methods )  
public R m (A args) {  
    synchronized(mutex) { return  
        (new Bar<I>()) .m(args); }  
    }  
}
```

Checking Validity of Reference

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
<R,A*>[m] for ( R m (A) : Foo<I>.methods )  
public R m (A args) {  
    synchronized(mutex) { return  
        (new Bar<I>()) .m(args); }  
    }  
}
```

Checking Validity of Reference

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
<R,A*>[m] for ( R m (A) : Foo<I>.methods )  
public List<R> m (A args) {  
    synchronized(mutex) { return  
        (new Bar<I>()).m(args); }  
}  
}
```

Type Safety Is Theorem Proving

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;  
  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) {  
        synchronized(mutex) { return me.m(args); }  
    }  
}
```

Type Safety Is Theorem Proving

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) {  
        synchronized(mutex) { return me.m(args); }  
    }
```

```
}
```

Type Safety Is Theorem Proving

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
<R,A*>[m] for ( R m (A) : I.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}
```

$\forall I \mid \text{interface}(I) \wedge$

Type Safety Is Theorem Proving

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
<R,A*>[m] for ( R m (A) : I.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}
```

$$\forall I \mid \text{interface}(I) \wedge$$
$$(\forall m1, R, A, m \mid \text{methodOf}(m1, I) \wedge \text{nameOf}(m1, m) \wedge$$
$$\text{retType}(m1, R) \wedge \text{argTypes}(m1, A)$$

Type Safety Is Theorem Proving

```
class Synchronized<interface I> implements I {  
    final I me;  
    final Object mutex;
```

```
<R,A*>[m] for ( R m (A) : I.methods )  
public R m (A args) {  
    synchronized(mutex) { return me.m(args); }  
}
```

$$\forall I \mid \text{interface}(I) \wedge$$
$$(\forall m1, R, A, m \mid \text{methodOf}(m1, I) \wedge \text{nameOf}(m1, m) \wedge$$
$$\text{retType}(m1, R) \wedge \text{argTypes}(m1, A)$$
$$\Leftrightarrow \exists m2 \mid \text{methodOf}(m2, \text{Synchronized}) \wedge \text{nameOf}(m2, m) \wedge$$
$$\text{retType}(m2, R) \wedge \text{argTypes}(m2, A))$$

Type Safety Is Theorem Proving

```
class Synchronized<interface I> implements I {
```

$\forall I \mid \text{interface}(I) \wedge$

$(\forall m1, R, A, m \mid \text{methodOf}(m1, I) \wedge \text{nameOf}(m1, m) \wedge$
 $\text{retType}(m1, R) \wedge \text{argTypes}(m1, A)$

$\Leftrightarrow \exists m2 \mid \text{methodOf}(m2, \text{Synchronized}) \wedge \text{nameOf}(m2, m) \wedge$
 $\text{retType}(m2, R) \wedge \text{argTypes}(m2, A))$

\Rightarrow

$(\forall m3 \mid \text{methodOf}(m3, \text{Synchronized}))$

Type Safety Is Theorem Proving

```
class Synchronized<interface I> implements I {
```

$\forall I \mid \text{interface}(I) \wedge$

$(\forall m1, R, A, m \mid \text{methodOf}(m1, I) \wedge \text{nameOf}(m1, m) \wedge$
 $\text{retType}(m1, R) \wedge \text{argTypes}(m1, A)$

$\Leftrightarrow \exists m2 \mid \text{methodOf}(m2, \text{Synchronized}) \wedge \text{nameOf}(m2, m) \wedge$
 $\text{retType}(m2, R) \wedge \text{argTypes}(m2, A))$

\Rightarrow

$(\forall m3 \mid \text{methodOf}(m3, \text{Synchronized})$

$\Rightarrow \nexists m4 \mid \text{methodOf}(m4, \text{Synchronized}) \wedge$

Type Safety Is Theorem Proving

```
class Synchronized<interface I> implements I {
```

$\forall I \mid \text{interface}(I) \wedge$

$(\forall m1, R, A, m \mid \text{methodOf}(m1, I) \wedge \text{nameOf}(m1, m) \wedge$
 $\text{retType}(m1, R) \wedge \text{argTypes}(m1, A)$

$\Leftrightarrow \exists m2 \mid \text{methodOf}(m2, \text{Synchronized}) \wedge \text{nameOf}(m2, m) \wedge$
 $\text{retType}(m2, R) \wedge \text{argTypes}(m2, A))$

\Rightarrow

$(\forall m3 \mid \text{methodOf}(m3, \text{Synchronized})$

$\Rightarrow \nexists m4 \mid \text{methodOf}(m4, \text{Synchronized}) \wedge$

$\text{name}(m3) = \text{name}(m4) \wedge$

$\text{argTypes}(m3) = \text{argTypes}(m4) \wedge$

$\text{retType}(m3) \neq \text{retType}(m4))$

Morphing: Core Solution

$$\begin{aligned} & \forall \mathbf{I} \mid \text{interface}(\mathbf{I}) \wedge \\ & \quad (\forall m1, \mathbf{m}, \mathbf{R}, \mathbf{A} \mid \text{methodOf}(m1, \mathbf{I}) \wedge \text{nameOf}(m1, \mathbf{m}) \wedge \\ & \quad \quad \text{retType}(m1, \mathbf{R}) \wedge \text{argType}(m1, \mathbf{A}) \\ & \quad \Leftrightarrow \exists m2 \mid \text{methodOf}(m2, \text{Synchronized}) \wedge \text{nameOf}(m2, \mathbf{m}) \wedge \\ & \quad \quad \text{retType}(m2, \mathbf{R}) \wedge \text{argTypes}(m2, \mathbf{A})) \\ & \Rightarrow \\ & (\forall m3 \mid \text{methodOf}(m3, \text{Synchronized}) \\ & \quad \Rightarrow \nexists m4 \mid \text{methodOf}(m4, \text{Synchronized}) \wedge \\ & \quad \quad \text{name}(m3) = \text{name}(m4) \wedge \\ & \quad \quad \text{argTypes}(m3) = \text{argTypes}(m4) \wedge \\ & \quad \quad \text{retType}(m3) \neq \text{retType}(m4) \end{aligned}$$

Morphing: Core Solution

- Reflectively Declared Code \Leftrightarrow Abstract Sets

Morphing: Core Solution

- Reflectively Declared Code \Leftrightarrow Abstract Sets
 - Validity of Reference \Leftrightarrow Set Containment

Morphing: Core Solution

- Reflectively Declared Code \Leftrightarrow Abstract Sets
 - Validity of Reference \Leftrightarrow Set Containment
 - Uniqueness of Declarations \Leftrightarrow Set Disjointness

Morphing: Core Solution

- Reflectively Declared Code \Leftrightarrow Abstract Sets

- Validity of Reference \Leftrightarrow Set Containment

- Uniqueness of Declarations \Leftrightarrow Set Disjointness

- Set Relationships \Leftrightarrow Unification of Patterns and Signatures

Morphing: Core Solution

- Reflectively Declared Code \Leftrightarrow Abstract Sets

- Validity of Reference \Leftrightarrow Set Containment

- Uniqueness of Declarations \Leftrightarrow Set Disjointness

- Set Relationships \Leftrightarrow Unification of Patterns and Signatures

- Set Containment \Leftrightarrow One-way Unification

Morphing: Core Solution

- Reflectively Declared Code \Leftrightarrow Abstract Sets

- Validity of Reference \Leftrightarrow Set Containment

- Uniqueness of Declarations \Leftrightarrow Set Disjointness

- Set Relationships \Leftrightarrow Unification of Patterns and Signatures

- Set Containment \Leftrightarrow One-way Unification

- Set Disjointness \Leftrightarrow Two-way Unification

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```


Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

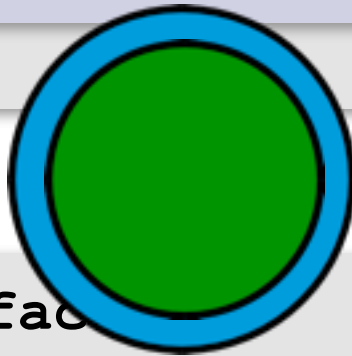
Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```



```
class UseSynched<interface J> extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

• I <: J

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

• I <: J

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

• I <: J

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

• $I <: J$

• $R \rightarrow \text{String}$

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

- $I <: J$

- $R \rightarrow \text{String}$

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

- $I <: J$

- $R \rightarrow \text{String}$

- $m \rightarrow n$

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

• $I <: J$

• $R \rightarrow \text{String}$

• $m \rightarrow n$

```
class UseSynched<interface J>  
    extends Synchronized<J> {  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Reference Validity \Leftrightarrow Set Containment

```
class Synchronized<interface I> implements I {  
    ...  
    <R,A*>[m] for ( R m (A) : I.methods )  
    public R m (A args) { ... }  
}
```

• $I <: J$

• $R \rightarrow \text{String}$

• $m \rightarrow n$

• $A \rightarrow B$

```
class UseSynched<interface J>  
    extends Synchronized  
    <B*>[n] for ( String n (B) : J.methods )  
    public String n (B args) {  
        return super.n(args);  
    }  
}
```

Morphing: Safe Static Reflection

A Case Study

Case Study: DSTM2

- Software transactional memory framework (Herlihy et al.)

Case Study: DSTM2

- Software transactional memory framework (Herlihy et al.)
 - Input → Any Java class or interface \mathbf{T}
 - Output → Transactional implementation of \mathbf{T}

Case Study: DSTM2

- Software transactional memory framework (Herlihy et al.)
 - Input → Any Java class or interface \mathbf{T}
 - Output → Transactional implementation of \mathbf{T}
 - Implemented using Reflection, BCEL (Bytecode engineering)

Case Study: DSTM2

- Software transactional memory framework (Herlihy et al.)
 - Input → Any Java class or interface \mathbf{T}
 - Output → Transactional implementation of \mathbf{T}
 - Implemented using Reflection, BCEL (Bytecode engineering)

Reflection+BCEL: 1,484 LOC

Case Study: DSTM2

- Software transactional memory framework (Herlihy et al.)
 - Input → Any Java class or interface \mathbf{T}
 - Output → Transactional implementation of \mathbf{T}
 - Implemented using Reflection, BCEL (Bytecode engineering)

Reflection+BCEL: 1,484 LOC

- MorphJ reimplementation of DSTM2:

Case Study: DSTM2

- Software transactional memory framework (Herlihy et al.)
 - Input → Any Java class or interface \mathbf{T}
 - Output → Transactional implementation of \mathbf{T}
 - Implemented using Reflection, BCEL (Bytecode engineering)

Reflection+BCEL: 1,484 LOC

- MorphJ reimplementation of DSTM2:
 - High level - no need to study bytecode standards

Case Study: DSTM2

- Software transactional memory framework (Herlihy et al.)
 - Input → Any Java class or interface \mathbf{T}
 - Output → Transactional implementation of \mathbf{T}
 - Implemented using Reflection, BCEL (Bytecode engineering)

Reflection+BCEL: 1,484 LOC

- MorphJ reimplementations of DSTM2:
 - High level - no need to study bytecode standards
 - Statically guaranteed to produce well-typed code

Case Study: DSTM2

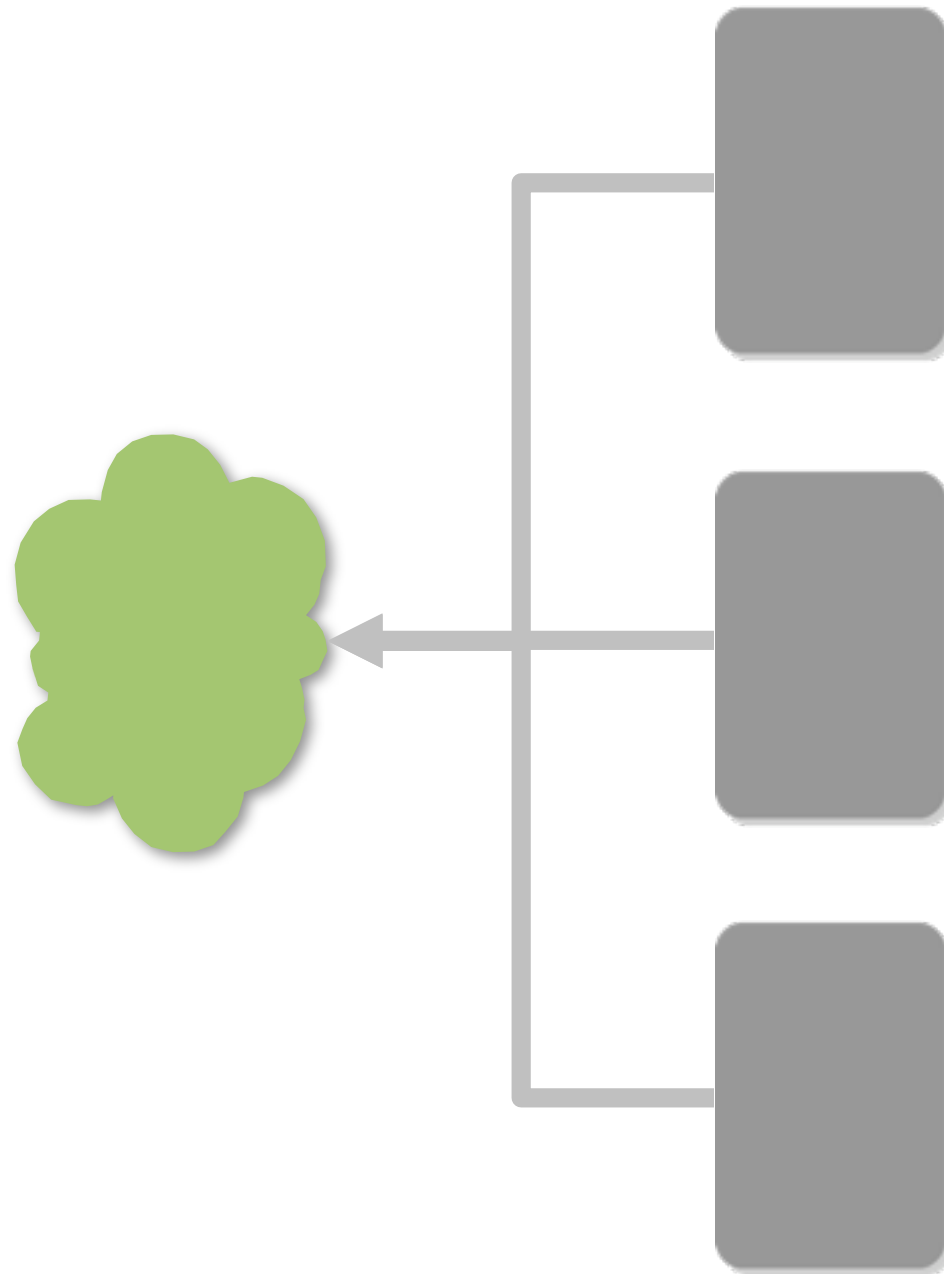
- Software transactional memory framework (Herlihy et al.)
 - Input → Any Java class or interface \mathbf{T}
 - Output → Transactional implementation of \mathbf{T}
 - Implemented using Reflection, BCEL (Bytecode engineering)

Reflection+BCEL: 1,484 LOC

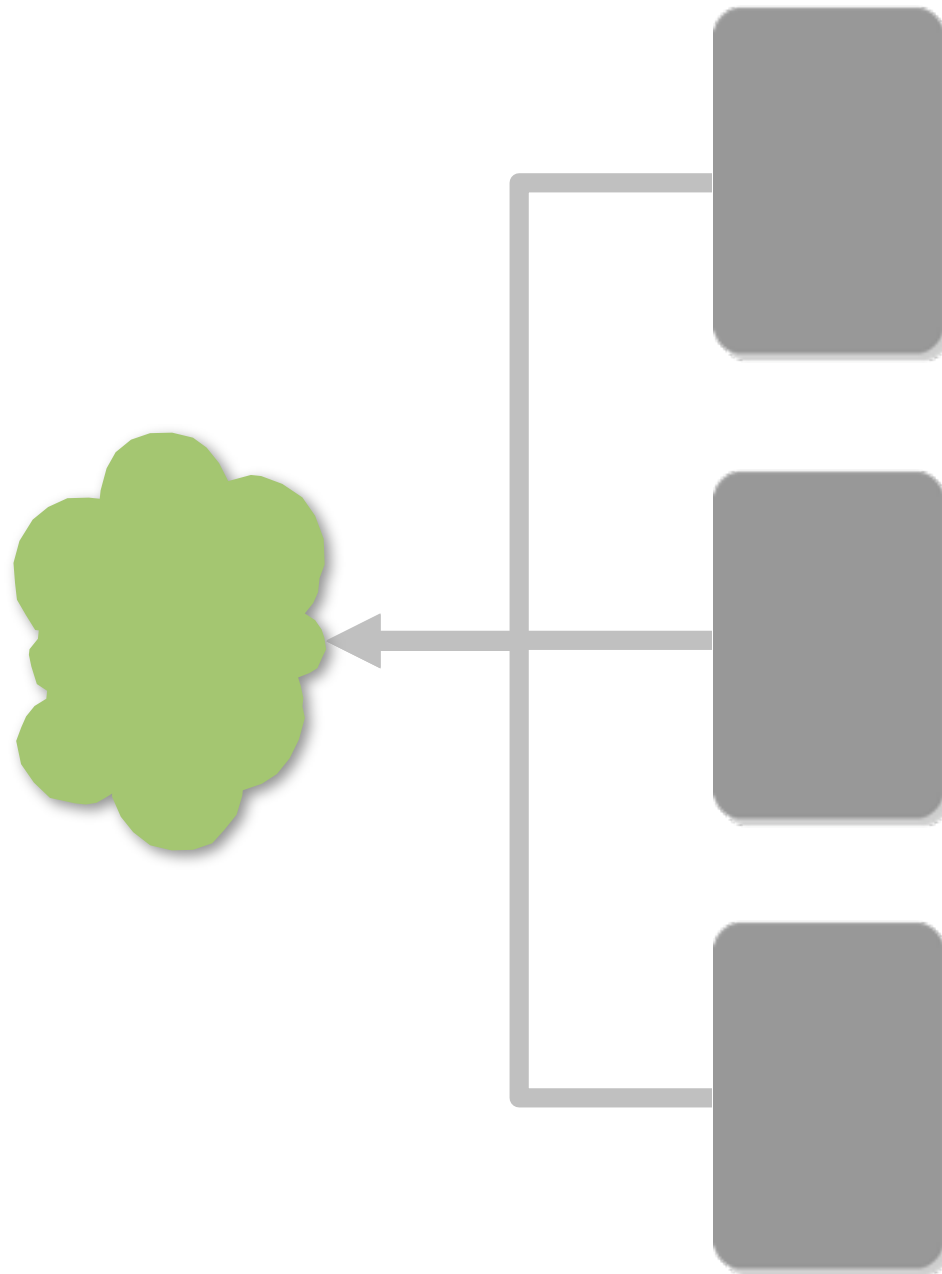
- MorphJ reimplementaion of DSTM2:
 - High level - no need to study bytecode standards
 - Statically guaranteed to produce well-typed code

MorphJ: 576 LOC

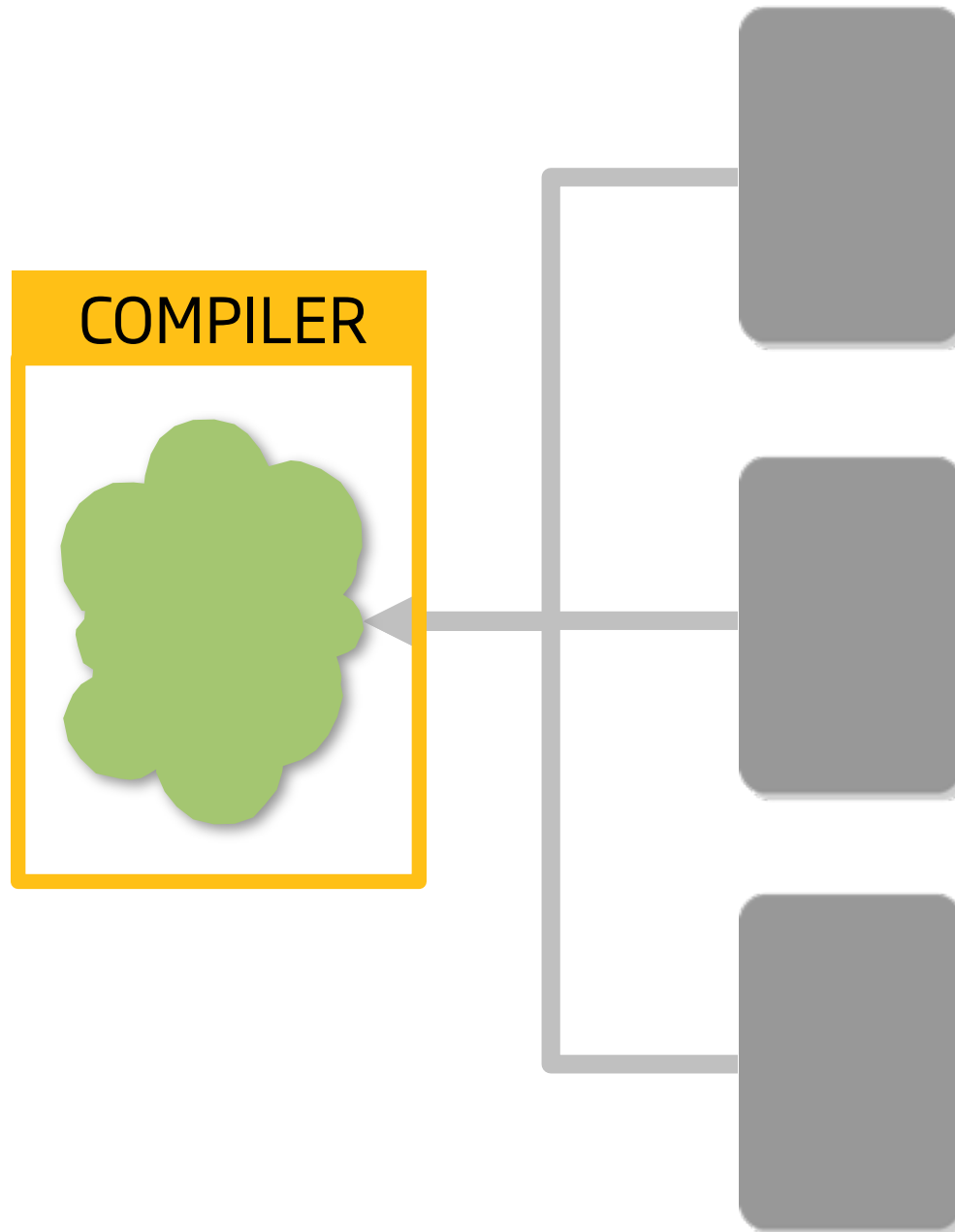
Morphing: Really Soft Software



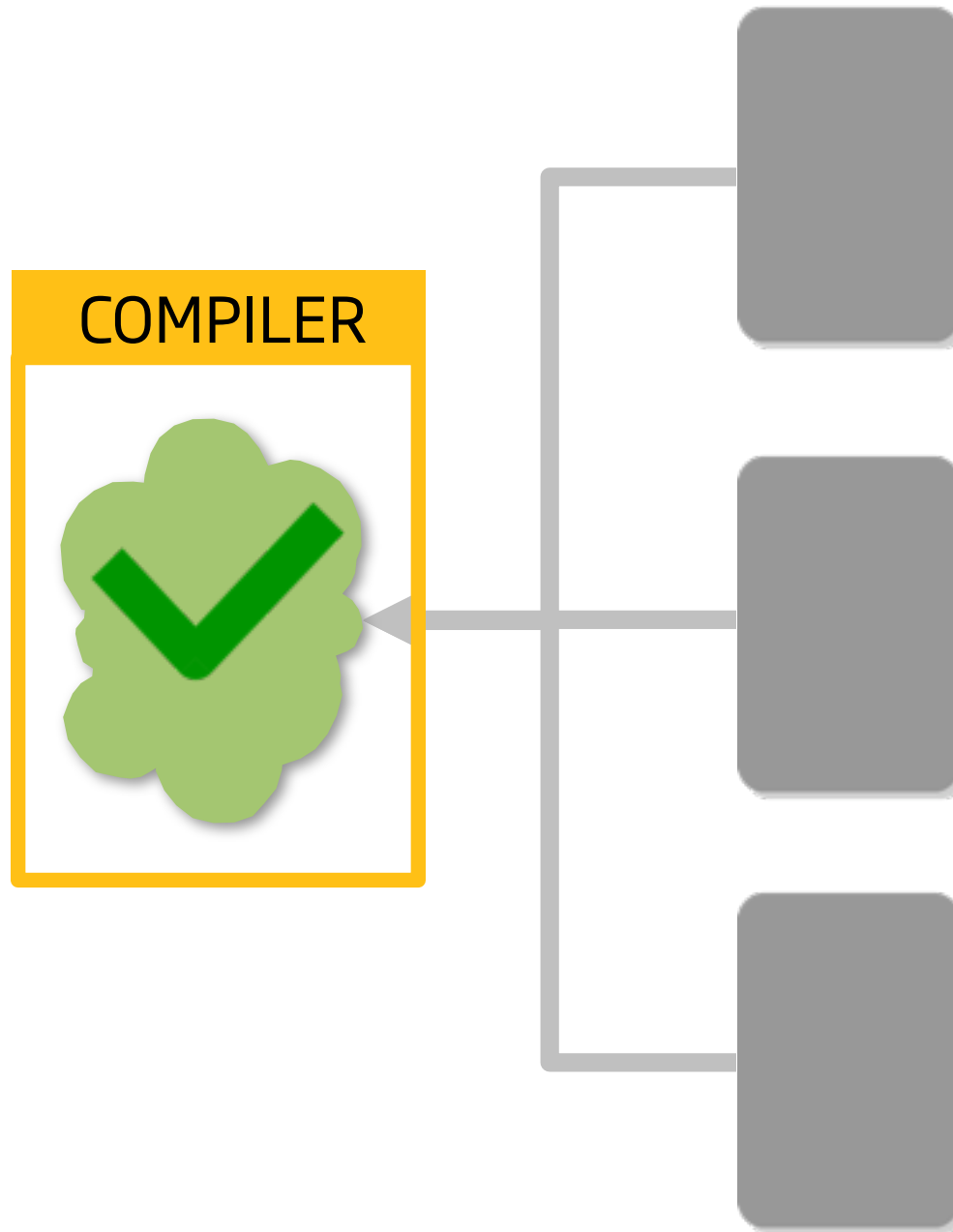
Morphing: Really Soft Software



Morphing: Really Soft Software



Morphing: Really Soft Software



My Research

Develop **expressive** and **safe** abstraction mechanisms for programming languages.

- Morphing - statically safe reflection
- Static type conditions - configurable libraries
 - `#ifdef` done right!
- Other work
 - Object-Oriented abstraction for hardware programming

My Research

Develop **expressive** and **safe** abstraction mechanisms for programming languages.

- Morphing - statically safe reflection
- Static type conditions - configurable libraries
 - `#ifdef` done right!
- Other work
 - Object-Oriented abstraction for hardware programming

My Research

Develop **expressive** and **safe** abstraction mechanisms for programming languages.

- Morphing - statically safe reflection
- Static type conditions - configurable libraries
 - **#ifdef** done right!
- Other work
 - Object-Oriented abstraction for hardware programming

Problem: Conditionally Declared Code

```
class ArrayList {  
    boolean add (Object elmt) { ... }  
}
```

Problem: Conditionally Declared Code

```
class ArrayList {  
▶ boolean add (Object elmt) { ... }  
}
```

Problem: Conditionally Declared Code

- The `#ifdef` "solution"

```
class ArrayList {  
▶ boolean add (Object elmt) { ... }  
}
```

Problem: Conditionally Declared Code

- The `#ifdef` “solution”

```
class ArrayList {  
    #ifdef VariableSize  
    ▶ boolean add (Object elmt) { ... }  
    #endif  
}
```

Problem: Conditionally Declared Code

- The `#ifdef` “solution”

```
class ArrayList {  
    #ifdef VariableSize  
    ▶ boolean add (Object elmt) { ... }  
    #endif  
  
    boolean populate ( ArrayList src ) {  
        for ( Object elmt : src )  
            add(elmt);  
    }  
}
```

Problem: Conditionally Declared Code

- The `#ifdef` “solution”

```
class ArrayList {  
    #ifdef VariableSize  
    boolean add (Object elmt) { ... }  
    #endif  
  
    boolean populate ( ArrayList src ) {  
        for ( Object elmt : src )  
            add(elmt);  
    }  
}
```

Problem: Conditionally Declared Code

- The `#ifdef` “solution”
 - “Correctness” of code verified by exhaustive testing.

```
class ArrayList {  
    #ifdef VariableSize  
    boolean add (Object elmt) { ... }  
    #endif  
  
    boolean populate ( ArrayList src ) {  
        for ( Object elmt : src )  
            add(elmt);  
    }  
}
```

Problem: Conditionally Declared Code

- The `#ifdef` “solution”
 - “Correctness” of code verified by exhaustive testing.
 - e.g. Linux kernel has 120 `#ifdef` flags → 2^{120} combinations to test.

Problem: Conditionally Declared Code

- The `#ifdef` “solution”
 - “Correctness” of code verified by exhaustive testing.
 - e.g. Linux kernel has 120 `#ifdef` flags → 2^{120} combinations to test.
- Problem: no ***separate type checking*** of conditional code

Static Type Conditions

Conditional Code with `cj`

```
class ArrayList {
  #ifdef VariableSize
  boolean add (Object elmt) { ... }
  #endif

  boolean populate ( ArrayList src ) {
    for ( Object elmt : src )
      add(elmt);
  }
}
```

Conditional Code with `cj`

```
interface VariableSize {}
```

```
class ArrayList {  
    #ifdef VariableSize  
    boolean add (Object elmt) { ... }  
    #endif  
  
    boolean populate ( ArrayList src ) {  
        for ( Object elmt : src )  
            add(elmt);  
    }  
}
```

Conditional Code with cj

```
interface VariableSize {
```

```
class ArrayList<M> {  
    <M extends VariableSize>?  
    boolean add (Object elmt) { ... }  
  
    boolean populate ( ArrayList src ) {  
        for ( Object elmt : src )  
            add(elmt);  
    }  
}
```

Conditional Code with cj

```
interface VariableSize {
```

```
class ArrayList<M> {  
▶ <M extends VariableSize>?  
  boolean add (Object elmt) { ... }  
  
  boolean populate ( ArrayList src ) {  
    for ( Object elmt : src )  
      add(elmt);  
  }  
}
```

Conditional Code with cj

```
interface VariableSize {
```

```
class ArrayList<M> {  
▶ <M extends VariableSize>  
  boolean add (Object elmt) { ... }  
  
  boolean populate ( ArrayList src ) {  
    for ( Object elmt : src )  
      add(elmt);  
  }  
}
```

```
ArrayList<VariableSize>
```

Conditional Code with cj

```
interface VariableSize {
```

```
class ArrayList<M> {
```

```
▶ <M extends VariableSize>?
```

```
boolean add (Object elmt) { ... }
```

```
boolean populate ( ArrayList src ) {  
    for ( Object elmt : src )  
        add(elmt);  
}
```

```
}
```

```
ArrayList<Object>
```

Separate Type Checking Conditional Code

```
class ArrayList<M> {  
    <M extends VariableSize>?  
    boolean add (Object elmt) { ... }  
  
    boolean populate ( ArrayList src ) {  
        for ( Object elmt : src )  
            add(elmt);  
    }  
}
```

Separate Type Checking Conditional Code

cj COMPILER

```
class ArrayList<M> {  
    <M extends VariableSize>?  
    boolean add (Object elmt) { ... }  
  
    boolean populate ( ArrayList src ) {  
        for ( Object elmt : src )  
            add(elmt);  
    }  
}
```

Separate Type Checking Conditional Code

cj COMPILER

```
class ArrayList<M> {  
    <M extends VariableSize>?  
    boolean add (Object elmt) { ... }  
  
    boolean populate ( ArrayList src ) {  
        for ( Object elmt : src )  
            add(elmt);  
    }  
}
```

Separate Type Checking Conditional Code

cj COMPILER

```
class ArrayList<M> {  
    <M extends VariableSize>?  
    boolean add (Object elmt) { ... }  
  
    boolean populate ( ArrayList src ) {  
        for ( Object elmt : src )  
            add(elmt);  
    }  
}
```



Method `add` is not visible in `populate`.

Separate Type Checking Conditional Code

```
class ArrayList<M> {
  <M extends VariableSize>?
  boolean add (Object elmt) { ... }

  boolean populate ( ArrayList src ) {
    for ( Object elmt : src )
      add(elmt);
  }
}
```

Separate Type Checking Conditional Code

```
class ArrayList<M> {  
    <M extends VariableSize>?  
    boolean add (Object elmt) { ... }  
  
    <M extends VariableSize>?  
    boolean populate ( ArrayList src ) {  
        for ( Object elmt : src )  
            add(elmt);  
    }  
}
```

Separate Type Checking Conditional Code

cj COMPILER

```
class ArrayList<M> {  
  <M extends VariableSize>?  
  boolean add (Object elmt) { ... }  
  
  <M extends VariableSize>?  
  boolean populate ( ArrayList src ) {  
    for ( Object elmt : src )  
      add(elmt);  
  }  
}
```



Static Type Conditions

A Case Study

Java Collections Framework (JCF)

```
public interface List extends Collection {  
    public int size();  
    public Object get(int index);  
  
    ...  
}
```

Java Collections Framework (JCF)

```
public interface List extends Collection {  
    public int size();  
    public Object get(int index);  
  
    public Object add(int index, Object elmt);  
  
    ...  
}
```

Java Collections Framework (JCF)

```
public interface List extends Collection {  
    public int size();  
    public Object get(int index);  
  
    // Size modifying operations: OPTIONAL METHODS  
    public Object add(int index, Object elmt);  
  
    ...  
}
```

Java Collections Framework (JCF)

```
public interface List extends Collection {  
    public int size();  
    public Object get(int index);  
  
    // Size modifying operations: OPTIONAL METHODS  
    public Object add(int index, Object elmt);  
  
    public Object set(int index, Object elmt);  
    ...  
}
```

Java Collections Framework (JCF)

```
public interface List extends Collection {
    public int size();
    public Object get(int index);

    // Size modifying operations: OPTIONAL METHODS
    public Object add(int index, Object elmt);

    // Content modifying operations: OPTIONAL METHODS
    public Object set(int index, Object elmt);
    ...
}
```

Java Collections Framework (JCF)

```
public interface List extends Collection {  
    public int size();  
    public Object get(int index);  
  
    // Size modifying operations: OPTIONAL METHODS  
    public Object add(int index, Object elmt);  
  
    // Content modifying operations: OPTIONAL METHODS  
    public Object set(int index, Object elmt);  
    ...  
}
```

- Throws **UnsupportedOperationException** at ***RUNTIME***

A Real Piece of Code from JCF

```
class UnmodifiableList implements List ... {  
    ...  
    public Object add(int index, Object elmt) {  
        throw new UnsupportedOperationException();  
    }  
    public Object set(int index, Object elmt) {  
        throw new UnsupportedOperationException();  
    }  
}
```

A Real Piece of Code from JCF

```
class UnmodifiableList implements List ... {  
    ...  
    public Object add(int index, Object elmt) {  
        throw new UnsupportedOperationException();  
    }  
    public Object set(int index, Object elmt) {  
        throw new UnsupportedOperationException();  
    }  
}
```

- **10 out of 25** methods of `List` are **OPTIONAL**.

The Official Rationale

- The first question on FAQ for JCF:

“Why don’t you support immutability directly in the core collection interfaces so that you can do away with optional operations (and `UnsupportedOperationException`)?”

The Official Rationale

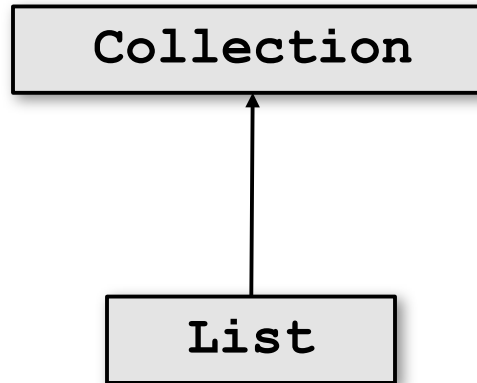
- The first question on FAQ for JCF:

“Why don’t you support immutability directly in the core collection interfaces so that you can do away with optional operations (and `UnsupportedOperationException`)?”

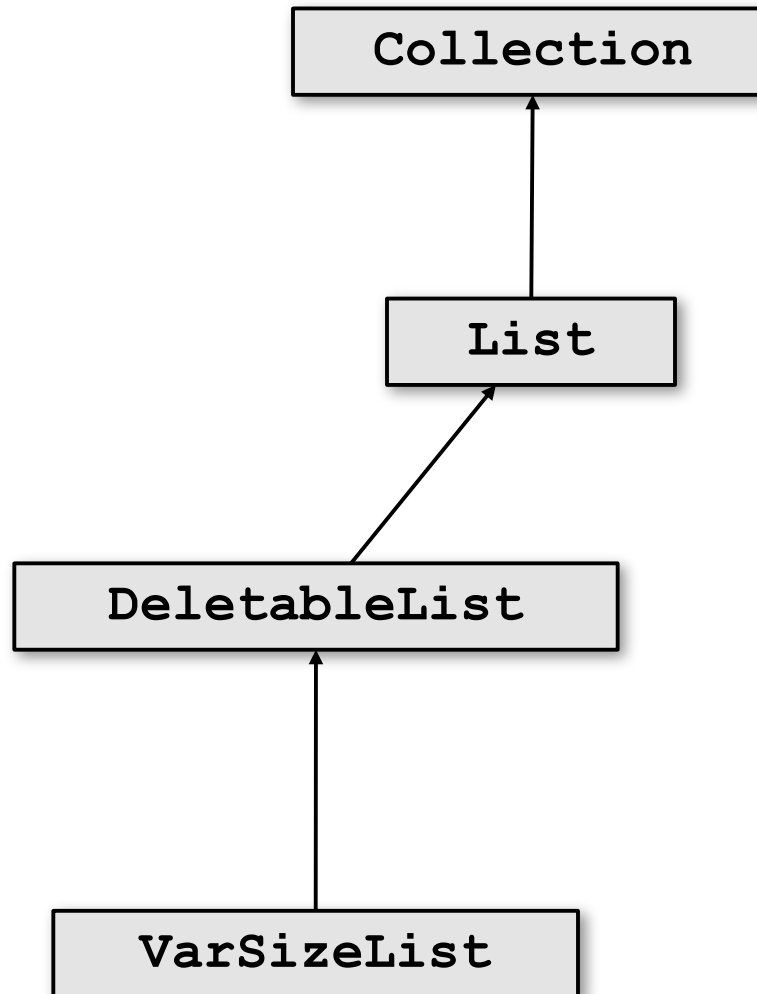
- The answer (much abbreviated):

“Clearly, static type checking is highly desirable... Unfortunately, attempts to achieve this goal cause an explosion in the size of the interface hierarchy.”

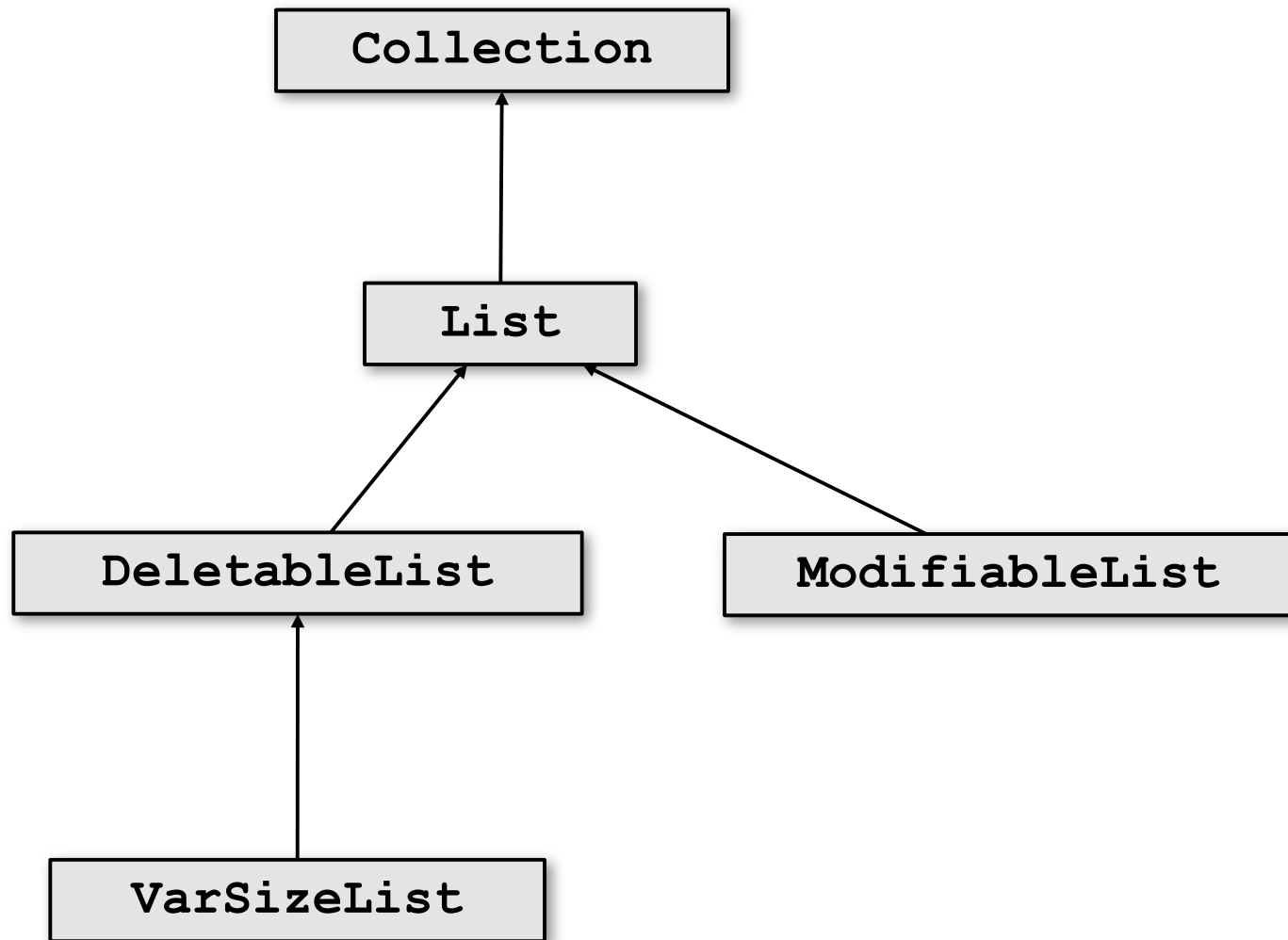
A Type-safe Implementation?



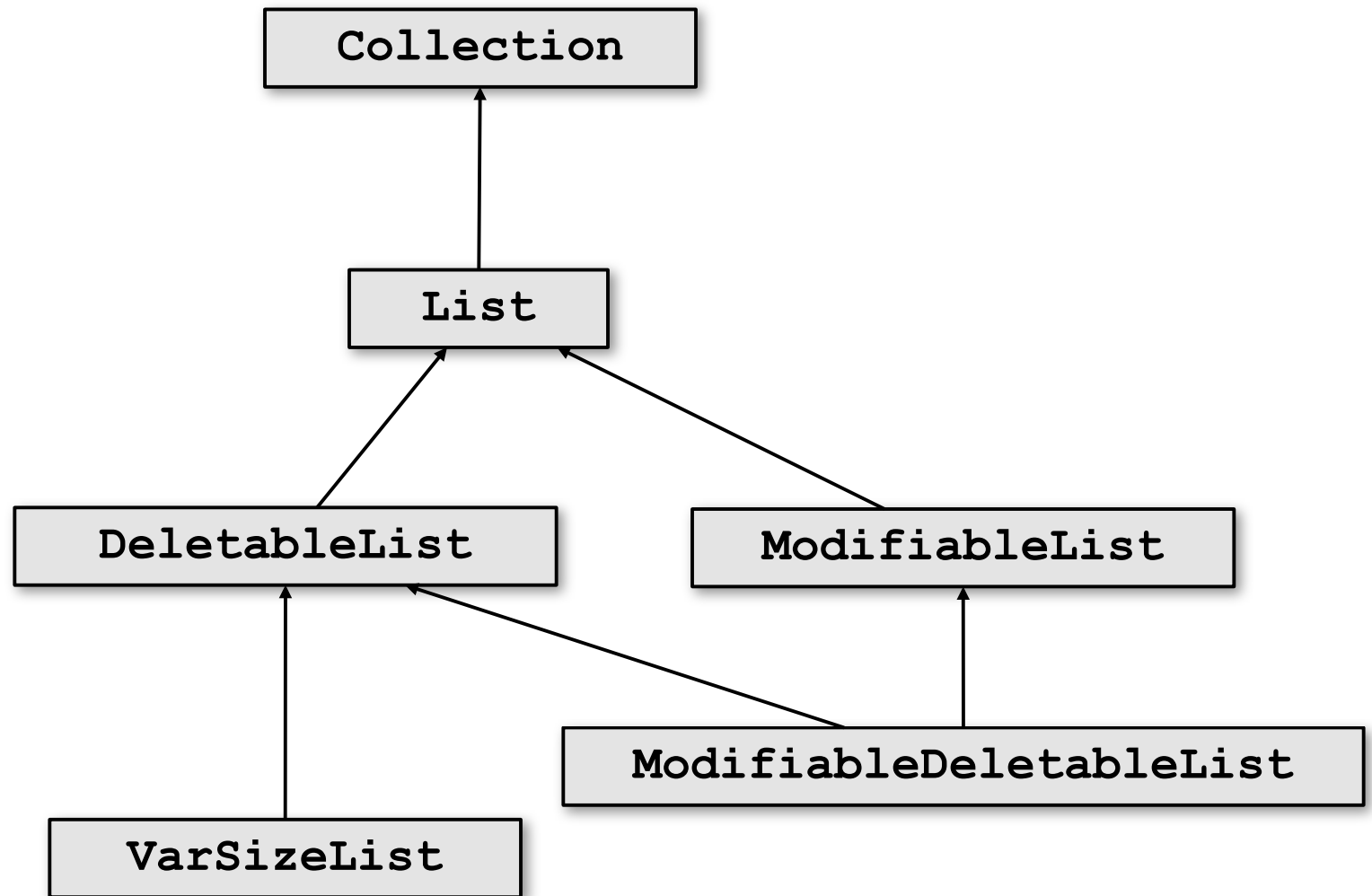
A Type-safe Implementation?



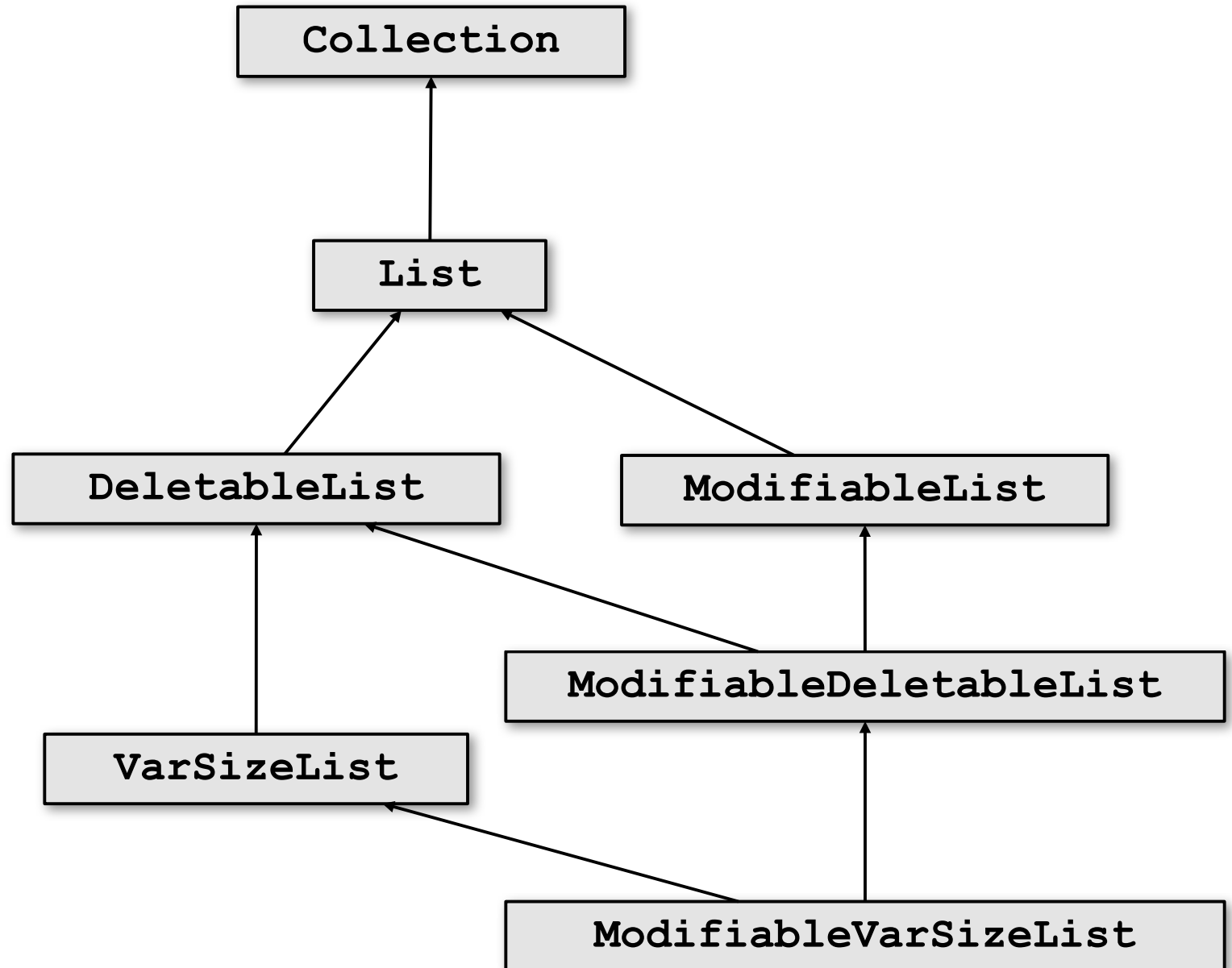
A Type-safe Implementation?



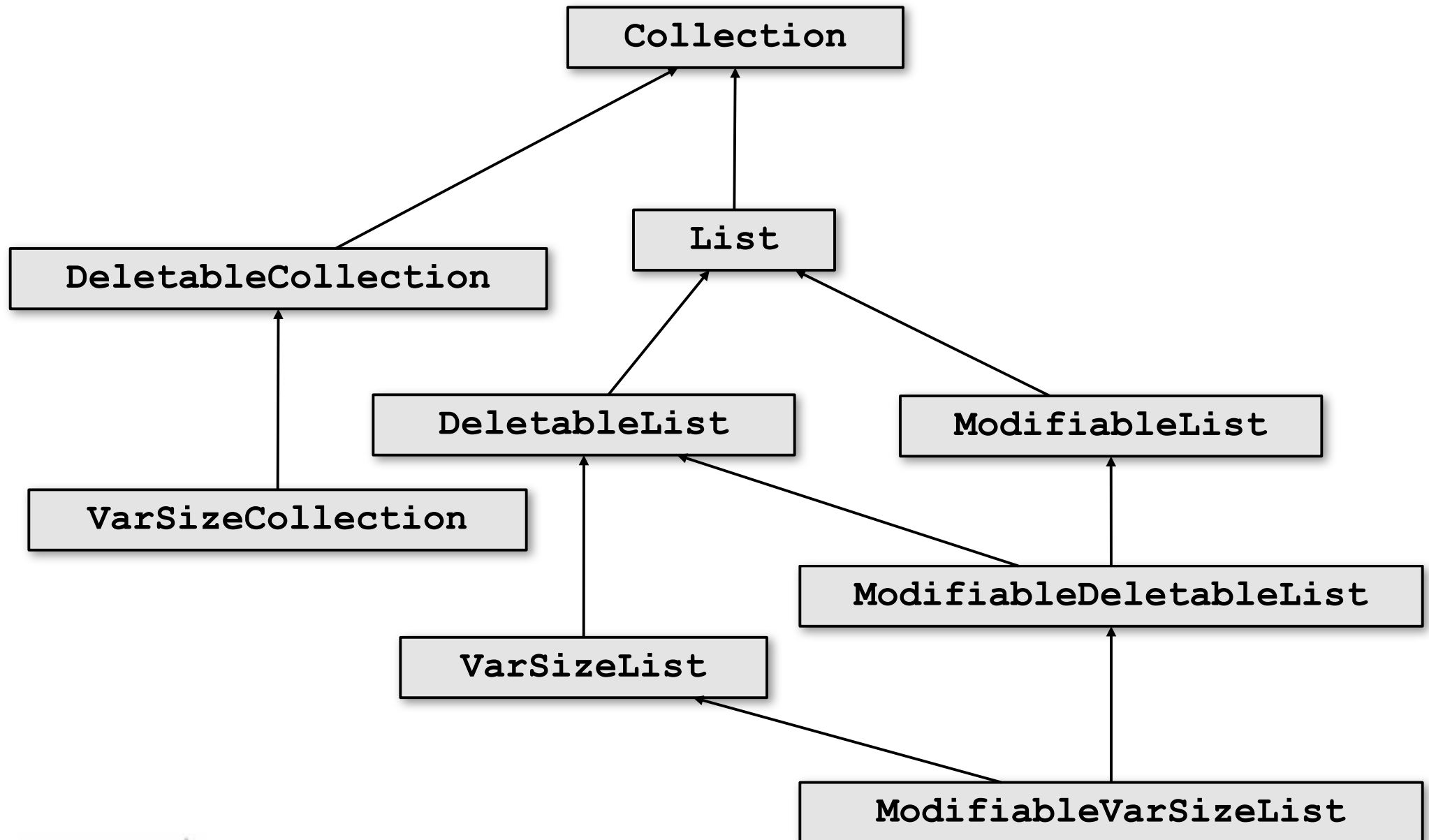
A Type-safe Implementation?



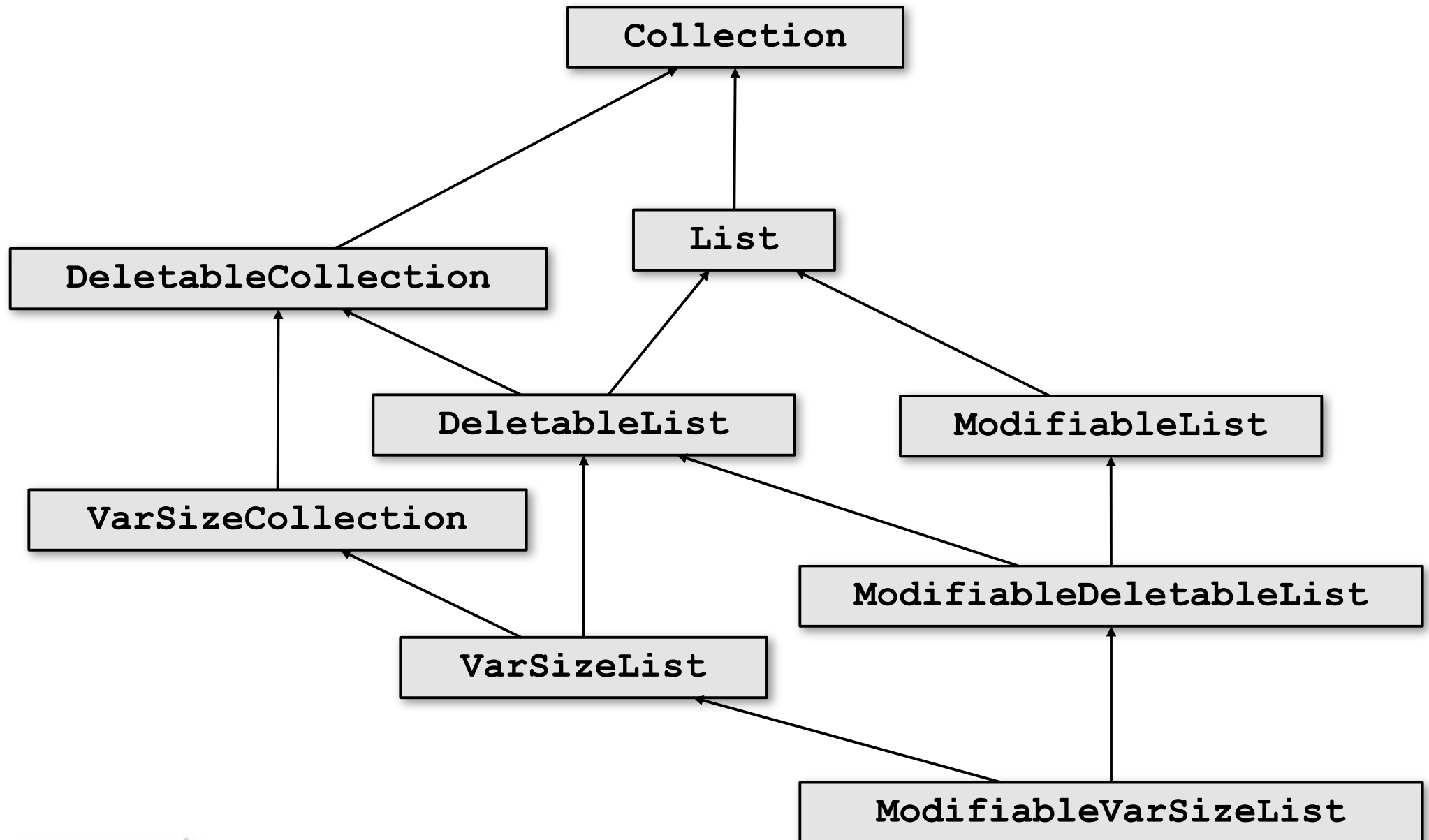
A Type-safe Implementation?



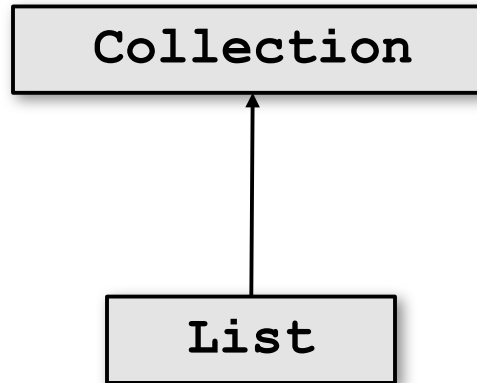
A Type-safe Implementation?



A Type-safe Implementation?



A Type-safe Implementation?



The Full Implications of a Type-safe Implementation

- JCF has 16 main interfaces
- More variations may arise in practice
 - e.g. Persistent vs. not.

The Full Implications of a Type-safe Implementation

- JCF has 16 main interfaces
- More variations may arise in practice
 - e.g. Persistent vs. not.

“Much as it pains me to say it, strong static typing does not work for collection interfaces in Java.”

- Doug Lea

The Full Implications of a Type-safe Implementation

YES WE CAN

Collections with Static Type Conditions

Collections with Static Type Conditions

Size Variability

```
interface Deletable {}  
interface VarSize extends Deletable {}
```

Collections with Static Type Conditions

Size Variability

```
interface Deletable {}  
interface VarSize extends Deletable {}
```

Modifiability

```
interface Modifiable {}
```

Conditional Methods

```
public interface List extends Collection {
    public int size();
    public E get(int index);
    ...

    // Size modifying operations: OPTIONAL METHODS
    public E add(int index, E elmt);

    // Content modifying operations: OPTIONAL METHODS
    public E set(int index, E elmt);
}
```

Conditional Methods

```
public interface List<M> extends Collection<M> {  
    public int size();  
    public E get(int index);  
    ...  
  
    // Size modifying operations: OPTIONAL METHODS  
    public E add(int index, E elmt);  
  
    // Content modifying operations: OPTIONAL METHODS  
    public E set(int index, E elmt);  
}
```

Conditional Methods

```
public interface List<M> extends Collection<M> {
    public int size();
    public E get(int index);
    ...

    <M extends VarSize>?
    public E add(int index, E elmt);

    // Content modifying operations: OPTIONAL METHODS
    public E set(int index, E elmt);
}
```

Conditional Methods

```
public interface List<M> extends Collection<M> {  
    public int size();  
    public E get(int index);  
    ...  
  
    <M extends VarSize>?  
    public E add(int index, E elmt);  
  
    <M extends Modifiable>?  
    public E set(int index, E elmt);  
}
```

Conditional Methods

```
public interface List<M> extends Collection<M> {  
    public int size();  
    public E get(int index);  
    ...  
  
    <M extends VarSize>?  
    public E add(int index, E elmt);  
  
    <M extends Modifiable>?  
    public E set(int index, E elmt);  
}
```

```
List<VarSize> l = ...;  
l.add(0, "foo"); // OK  
l.set(0, "bar"); // COMPILE-TIME ERROR!
```

Conditional Methods

```
public interface List<M> extends Collection<M> {
    public int size();
    public E get(int index);
    ...

    <M extends VarSize>?
    public E add(int index, E elmt);

    <M extends Modifiable>?
    public E set(int index, E elmt);
}
```

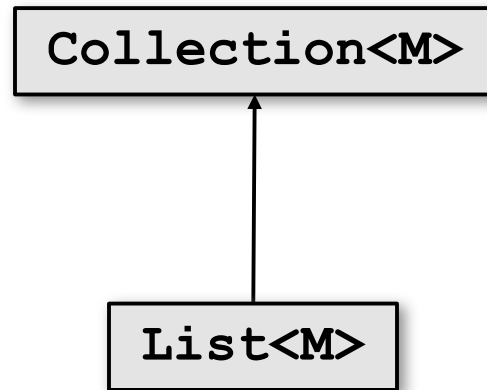
```
List<VarSize> l = ...;
▶ l.add(0, "foo"); // OK
  l.set(0, "bar"); // COMPILE-TIME ERROR!
```

Conditional Methods

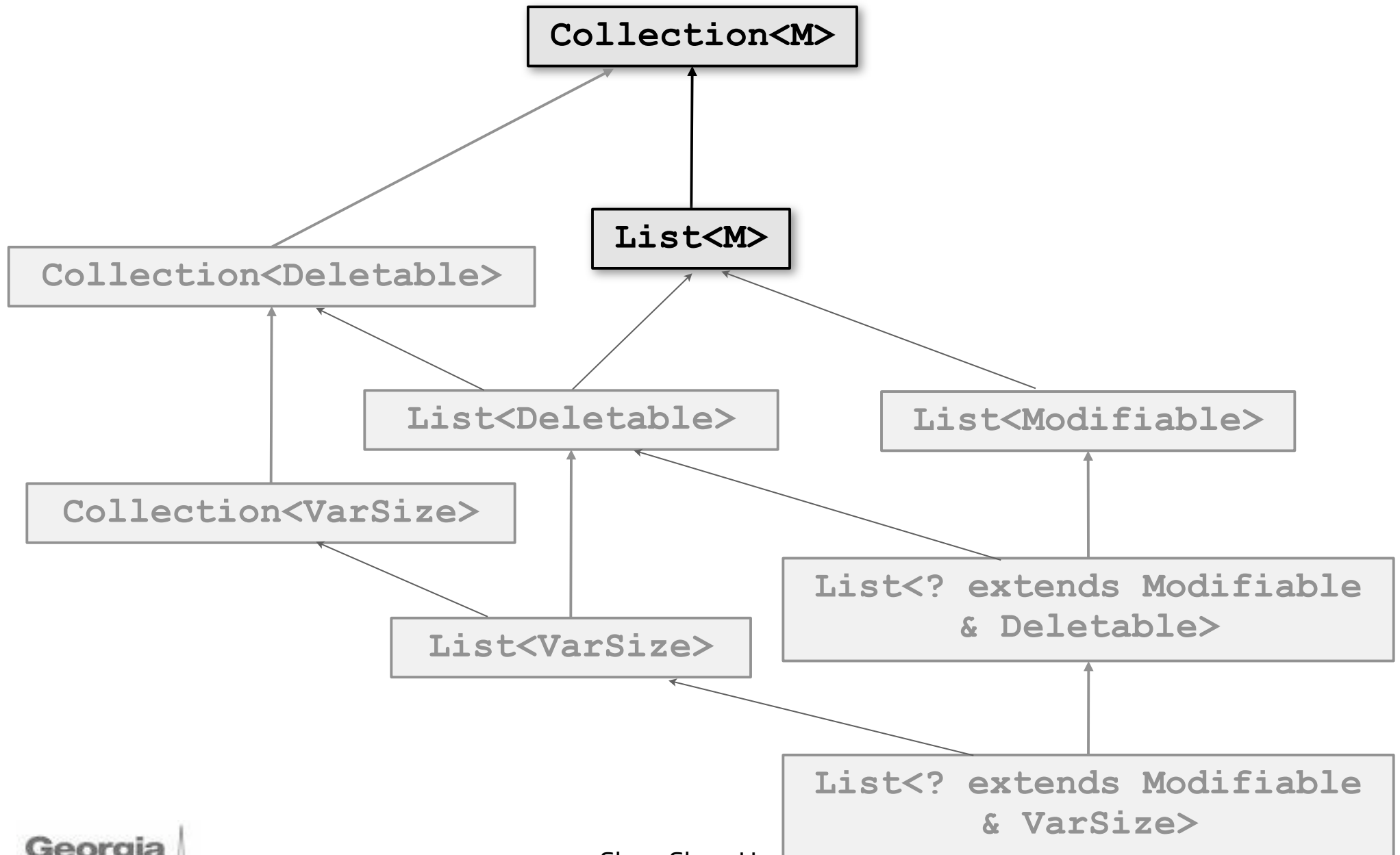
```
public interface List<M> extends Collection<M> {  
    public int size();  
    public E get(int index);  
    ...  
  
    <M extends VarSize>?  
    public E add(int index, E elmt);  
  
    <M extends Modifiable>?  
    public E set(int index, E elmt);  
}
```

```
List<VarSize> l = ...;  
l.add(0, "foo"); // OK  
▶ l.set(0, "bar"); // COMPILE-TIME ERROR!
```

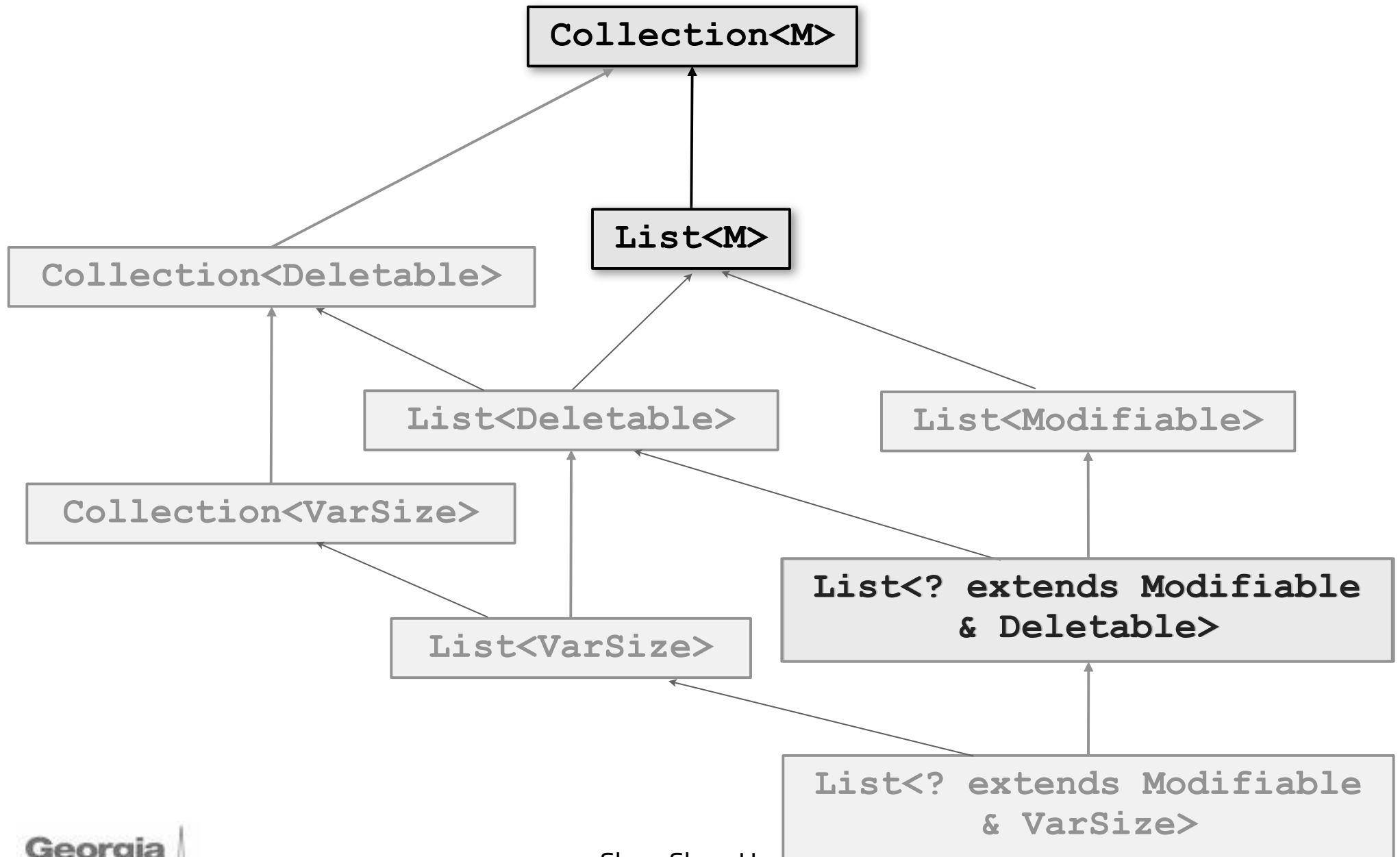
Type Hierarchy using Static Type Conditions



Type Hierarchy using Static Type Conditions



Type Hierarchy using Static Type Conditions



Static Type Conditions: Summary

- Separately type checked conditional code
 - “Correctness” is statically guaranteed!
 - No exhaustive testing

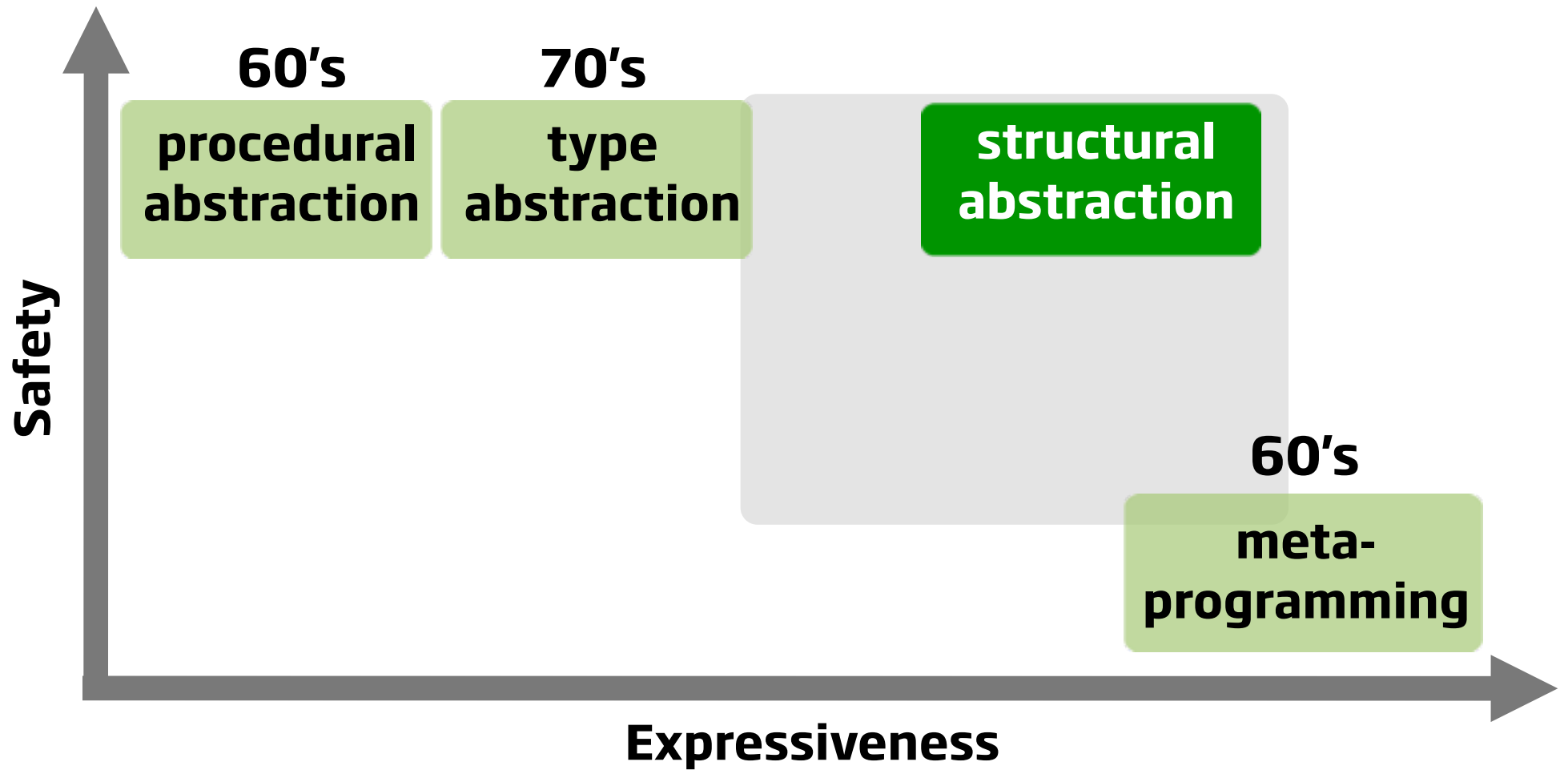
Static Type Conditions: Summary

- Separately type checked conditional code
 - “Correctness” is statically guaranteed!
 - No exhaustive testing
- **#ifdef** done right!

Static Type Conditions: Summary

- Separately type checked conditional code
 - “Correctness” is statically guaranteed!
 - No exhaustive testing
- `#ifdef` done right!
- Practical impact: Concise and safe solution for JCF's well-known “optional methods” problem.

Abstraction Mechanism Design Space



Citations

- ***“Morphing: Safely Shaping a Class in the Image of Others”***.
Shan Shan Huang, David Zook, and Yannis Smaragdakis. [ECOOP 2007]
- ***“Expressive and Safe Static Reflection”***. Shan Shan Huang and
Yannis Smaragdakis. [PLDI 2008]
- ***“cj: Enhancing Java with Safe Type Conditions”***. Shan Shan
Huang and Yannis Smaragdakis. [AOSD 2007]

Citations

- ***“Generating AspectJ Programs with Meta-AspectJ”***. David Zook, Shan Shan Huang, and Yannis Smaragdakis. [GPCE 2004]
 - **Best Paper Award**
- ***“Domain-Specific Languages and Program Generation with Meta-AspectJ”***. Shan Shan Huang, David Zook, and Yannis Smaragdakis. [TOSEM 2008]
- ***“Morphing: Safely Shaping a Class in the Image of Others”***.
Shan Shan Huang, David Zook, and Yannis Smaragdakis. [ECOOP 2007]
- ***“Expressive and Safe Static Reflection”***. Shan Shan Huang and Yannis Smaragdakis. [PLDI 2008]
- ***“cj: Enhancing Java with Safe Type Conditions”***. Shan Shan Huang and Yannis Smaragdakis. [AOSD 2007]

Citations

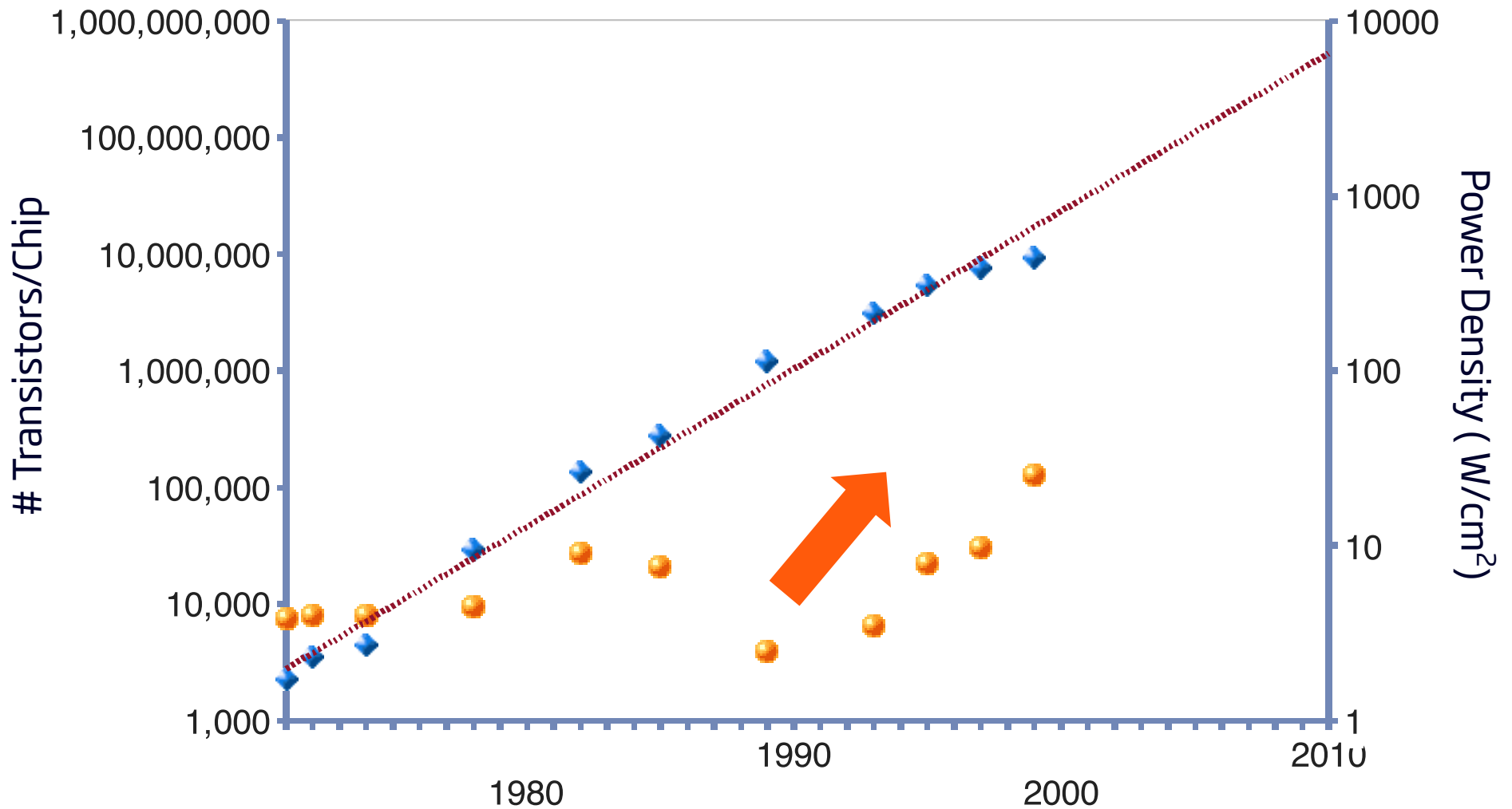
- **“Statically Safe Program Generation with SafeGen”**. Shan Shan Huang, David Zook, and Yannis Smaragdakis. [GPCE 2005, SCP 2008]
 - **“Generating AspectJ Programs with Meta-AspectJ”**. David Zook, Shan Shan Huang, and Yannis Smaragdakis. [GPCE 2004]
 - **Best Paper Award**
 - **“Domain-Specific Languages and Program Generation with Meta-AspectJ”**. Shan Shan Huang, David Zook, and Yannis Smaragdakis. [TOSEM 2008]
 - **“Morphing: Safely Shaping a Class in the Image of Others”**. Shan Shan Huang, David Zook, and Yannis Smaragdakis. [ECOOP 2007]
 - **“Expressive and Safe Static Reflection”**. Shan Shan Huang and Yannis Smaragdakis. [PLDI 2008]
 - **“cj: Enhancing Java with Safe Type Conditions”**. Shan Shan Huang and Yannis Smaragdakis. [AOSD 2007]

Citations

- **“Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary”**. Shan Shan Huang, Amir Hormati, Rodric Rabbah, and David Bacon. [ECOOP 2008]
 - **“Statically Safe Program Generation with SafeGen”**. Shan Shan Huang, David Zook, and Yannis Smaragdakis. [GPCE 2005, SCP 2008]
 - **“Generating AspectJ Programs with Meta-AspectJ”**. David Zook, Shan Shan Huang, and Yannis Smaragdakis. [GPCE 2004]
 - **Best Paper Award**
 - **“Domain-Specific Languages and Program Generation with Meta-AspectJ”**. Shan Shan Huang, David Zook, and Yannis Smaragdakis. [TOSEM 2008]
 - **“Morphing: Safely Shaping a Class in the Image of Others”**. Shan Shan Huang, David Zook, and Yannis Smaragdakis. [ECOOP 2007]
 - **“Expressive and Safe Static Reflection”**. Shan Shan Huang and Yannis Smaragdakis. [PLDI 2008]
 - **“cj: Enhancing Java with Safe Type Conditions”**. Shan Shan Huang and Yannis Smaragdakis. [AOSD 2007]

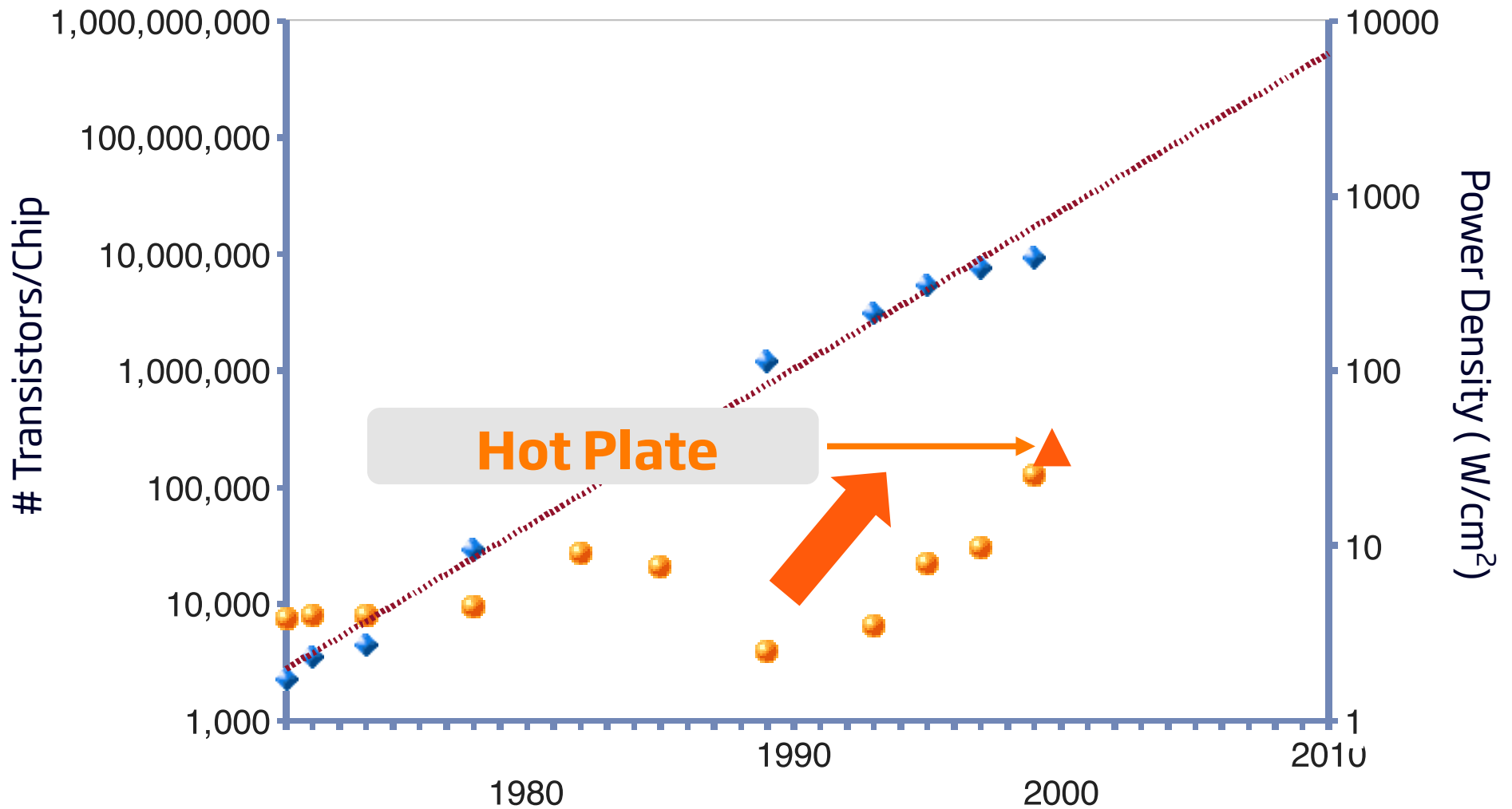
Obligatory Moore's Law Slide

- Number of Transistors double every two years.



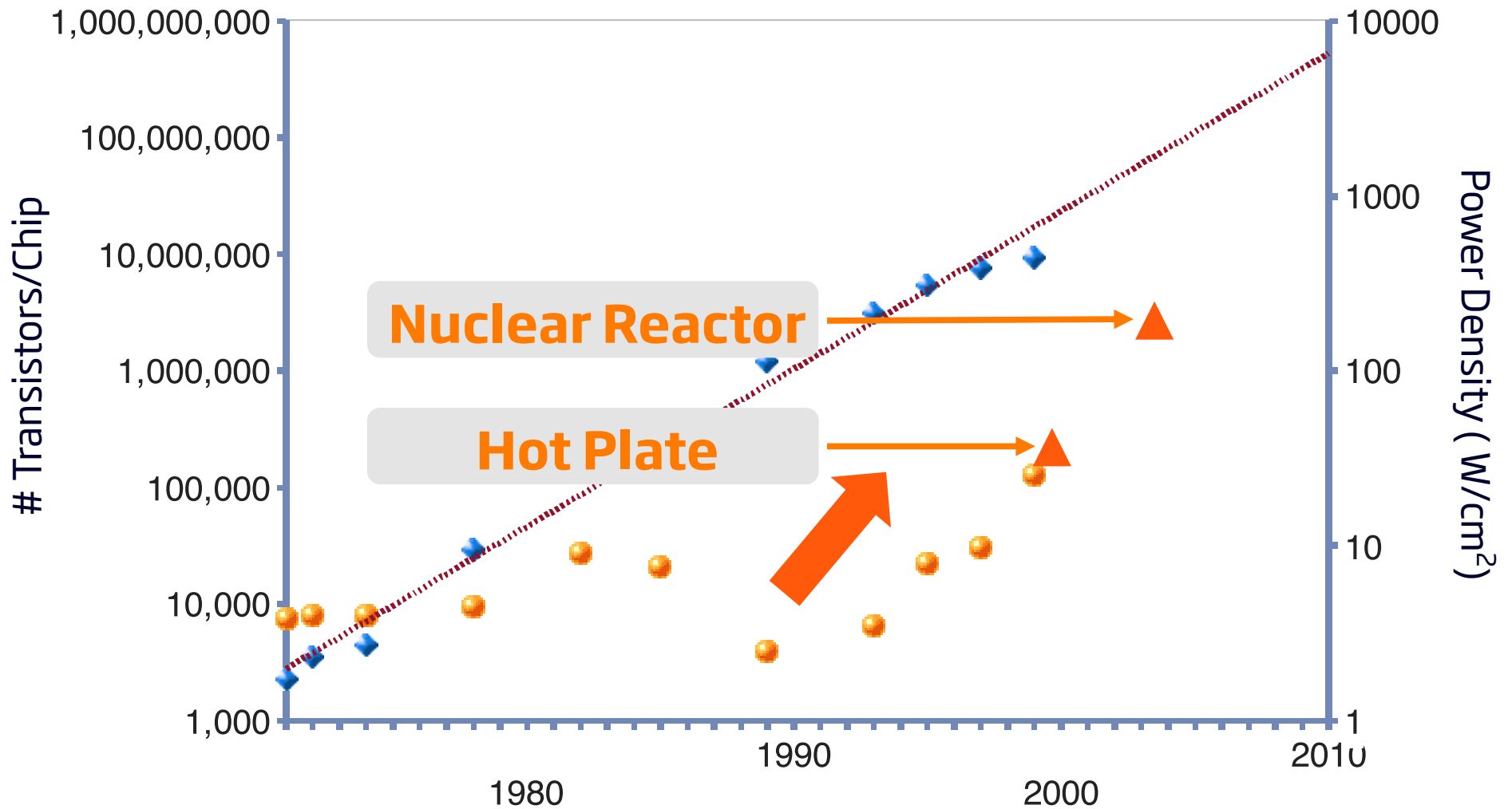
Obligatory Moore's Law Slide

- Number of Transistors double every two years.



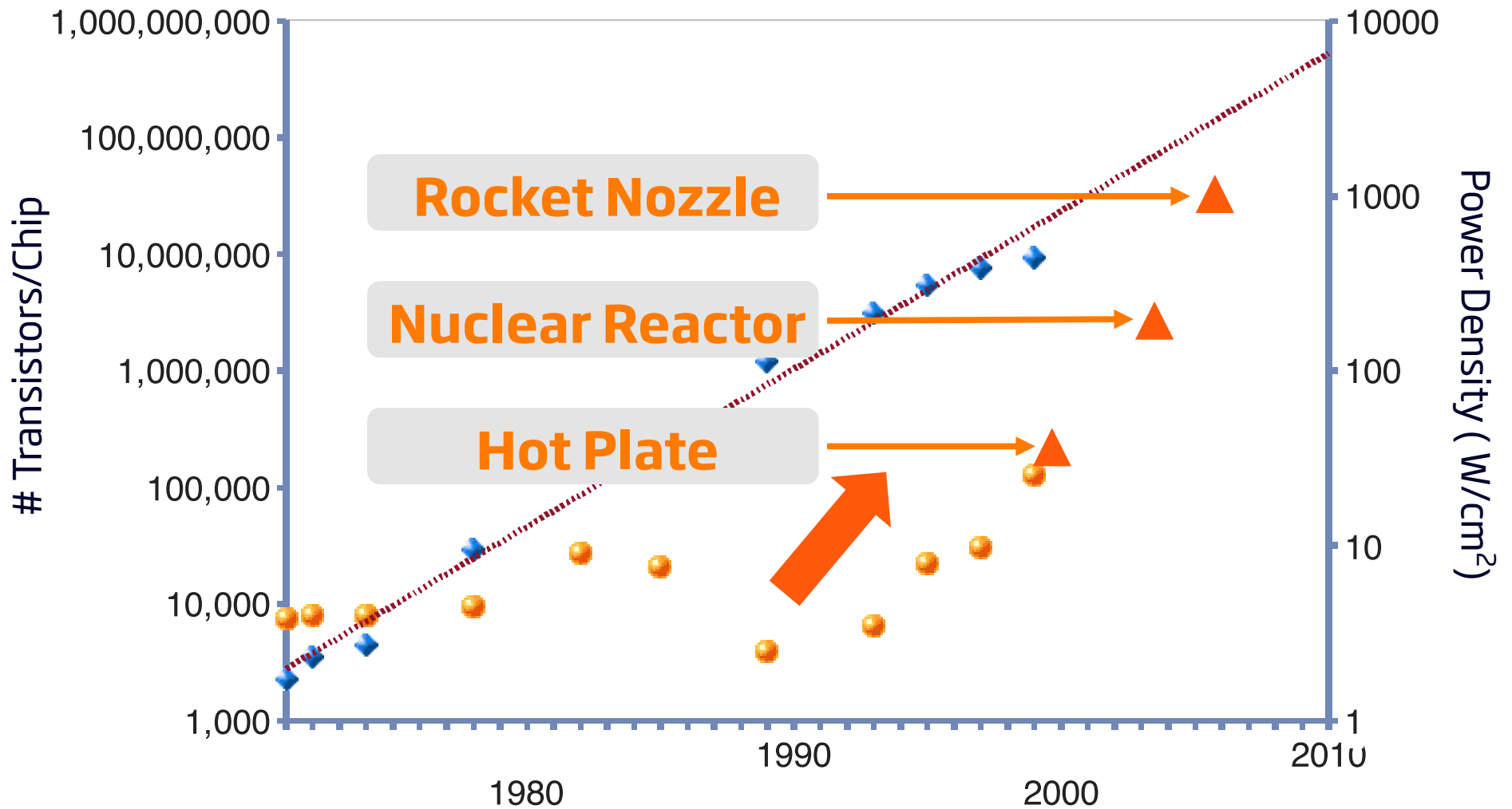
Obligatory Moore's Law Slide

- Number of Transistors double every two years.



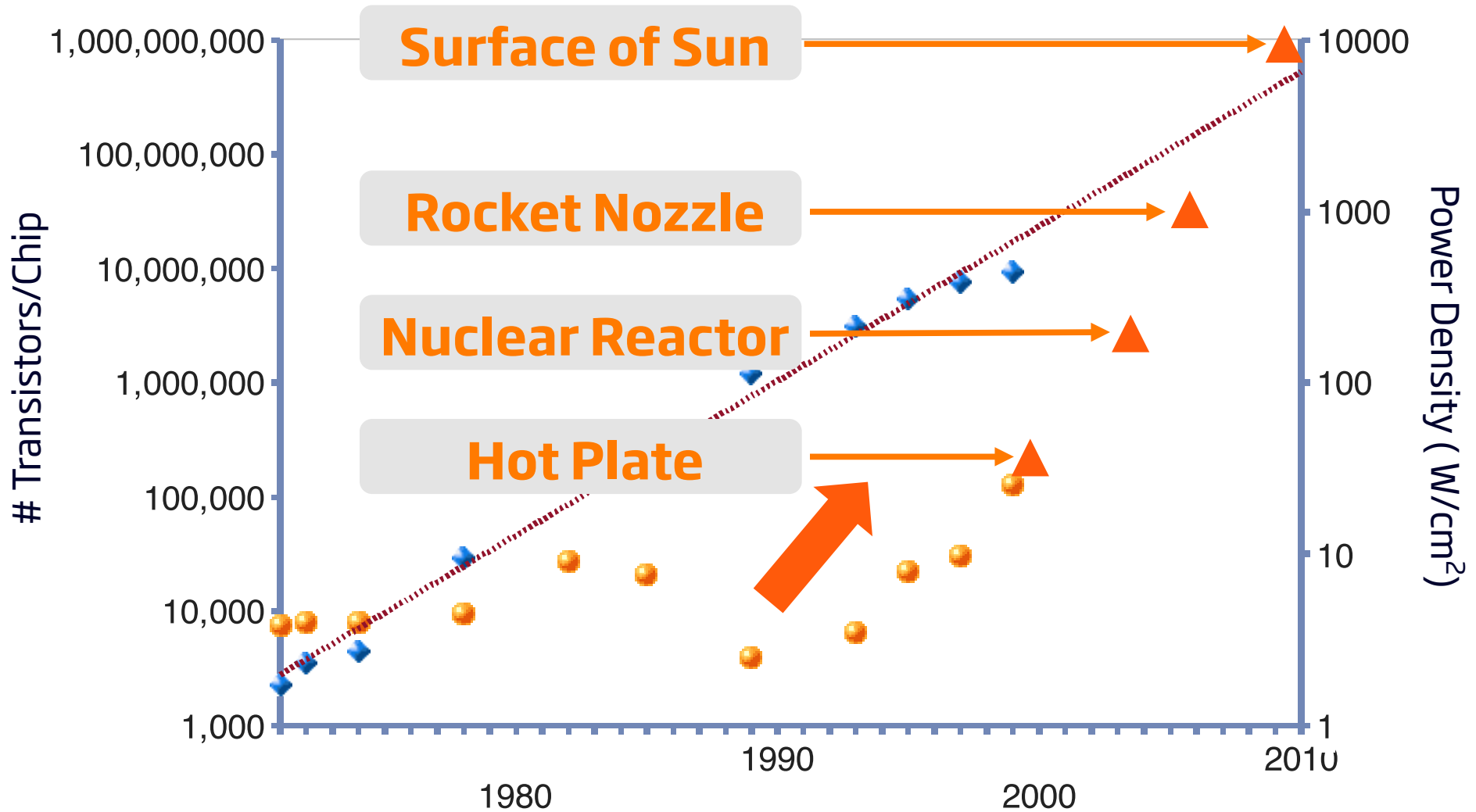
Obligatory Moore's Law Slide

- Number of Transistors double every two years.



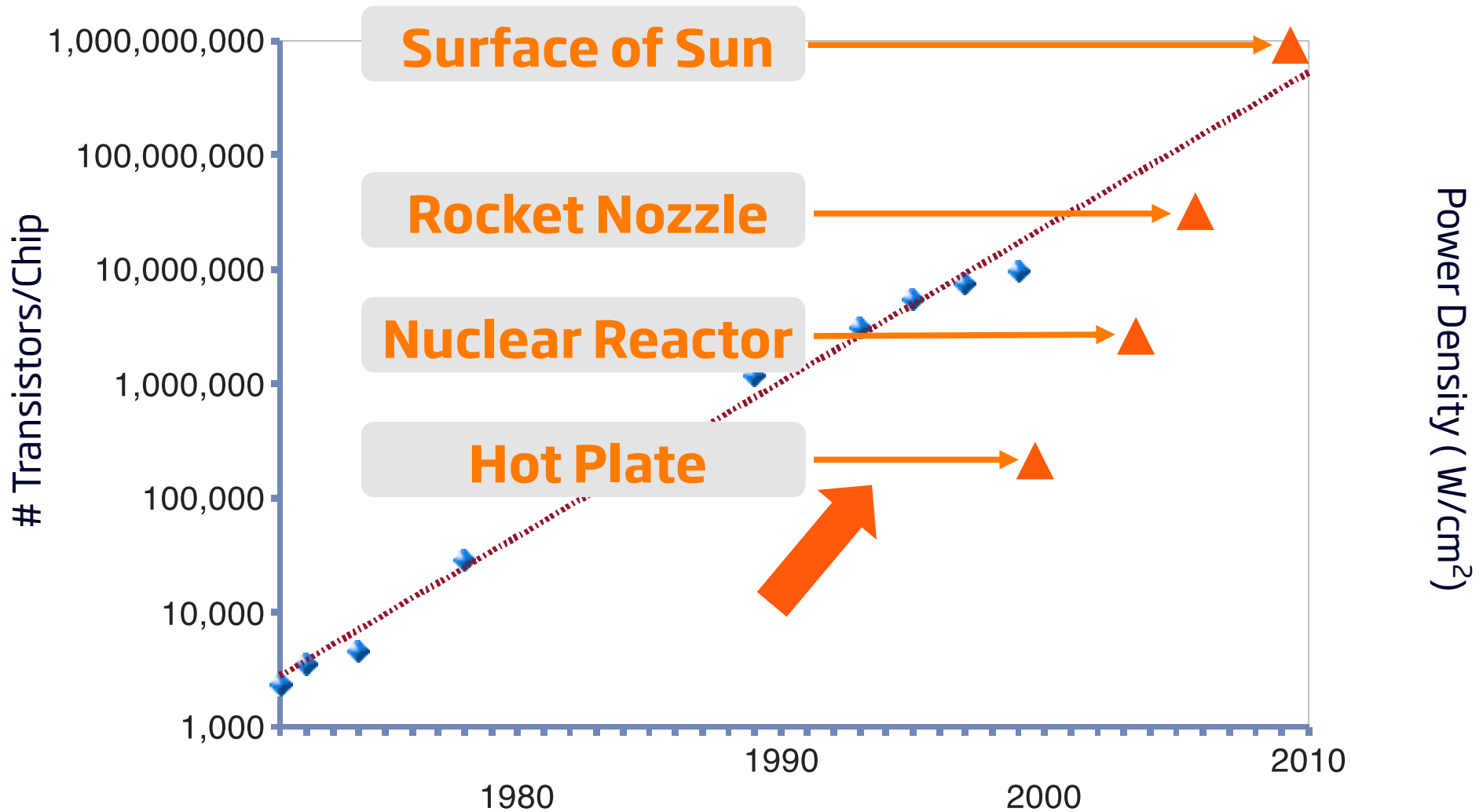
Obligatory Moore's Law Slide

- Number of Transistors double every two years.



Obligatory Moore's Law Slide

- Number of Transistors double every two years.



Alternatives?

- Yes, we've heard of multicores

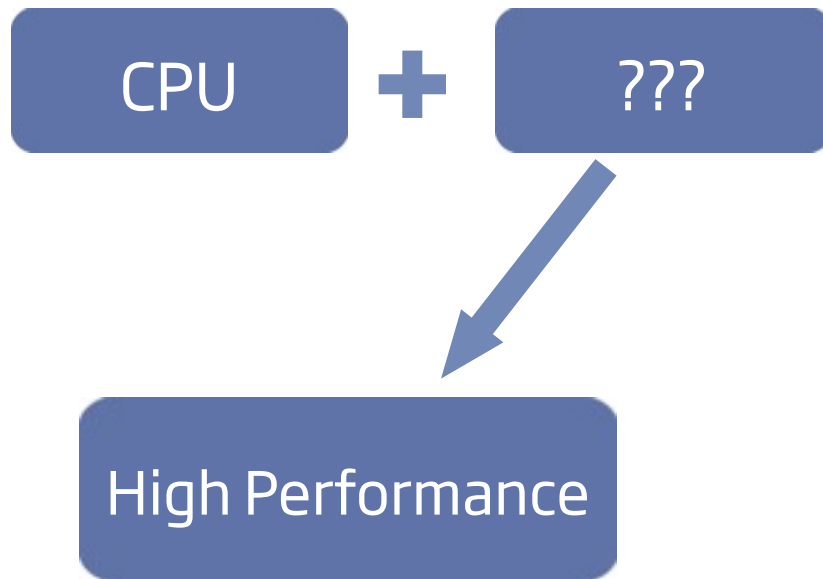
Alternatives?

- Yes, we've heard of multicores
- Heterogenous architecture:



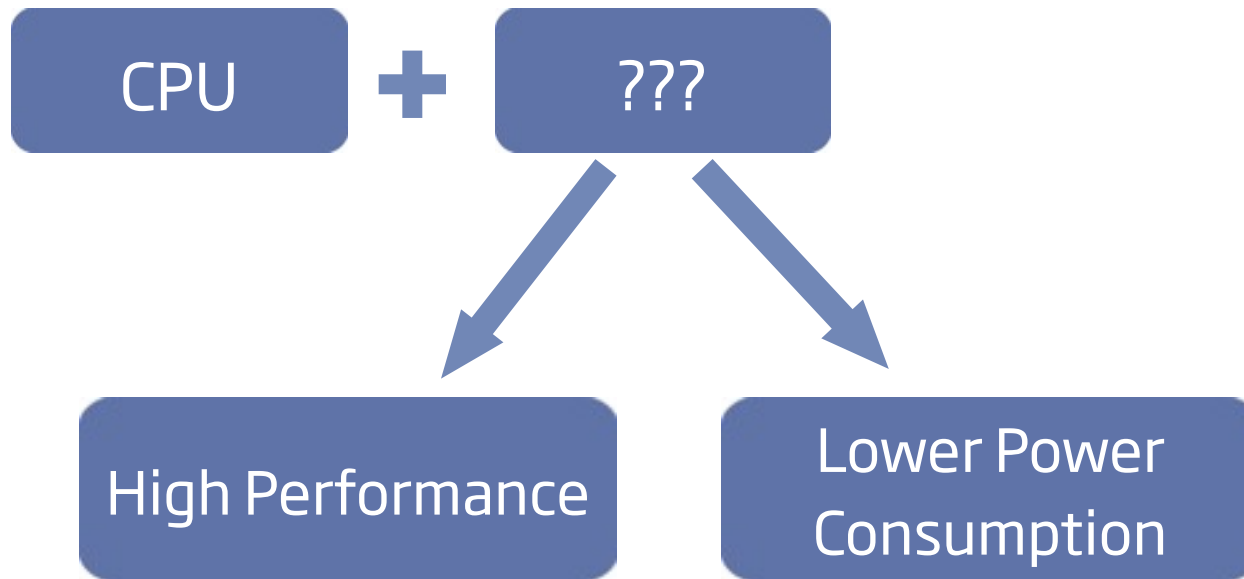
Alternatives?

- Yes, we've heard of multicores
- Heterogenous architecture:



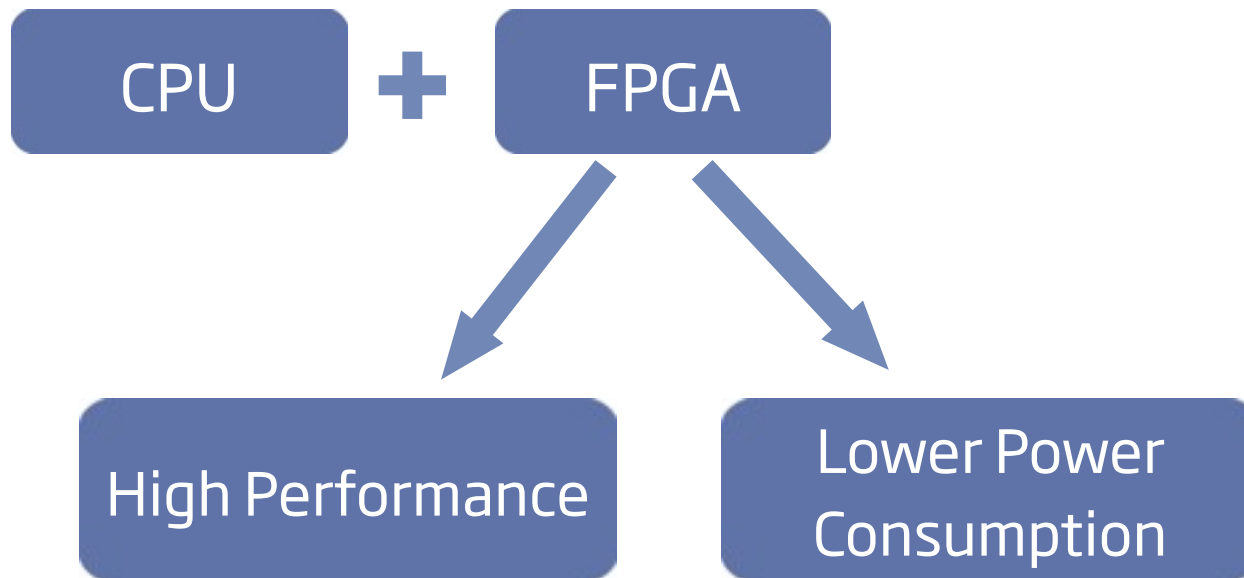
Alternatives?

- Yes, we've heard of multicores
- Heterogenous architecture:



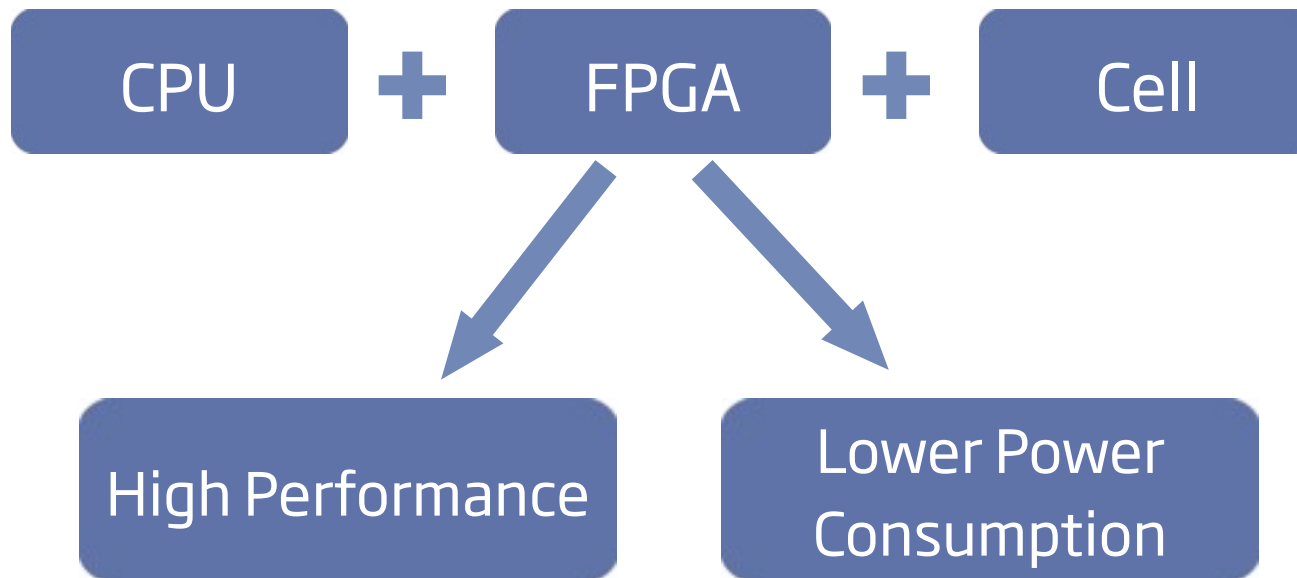
Alternatives?

- Yes, we've heard of multicores
- Heterogenous architecture:



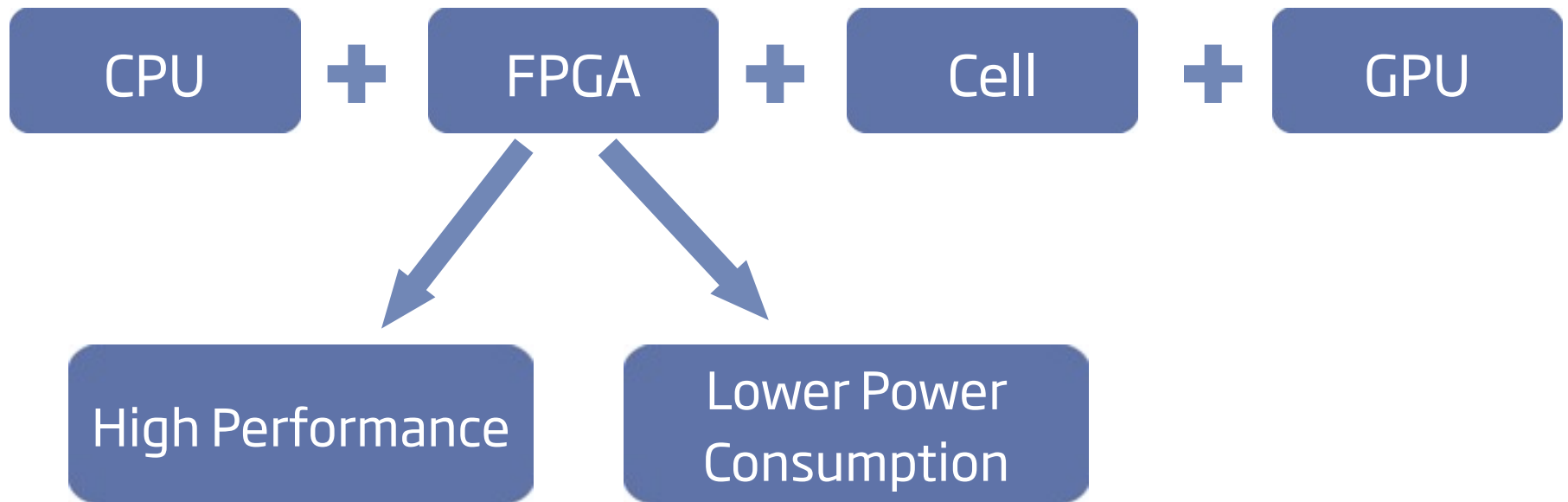
Alternatives?

- Yes, we've heard of multicores
- Heterogenous architecture:



Alternatives?

- Yes, we've heard of multicores
- Heterogenous architecture:



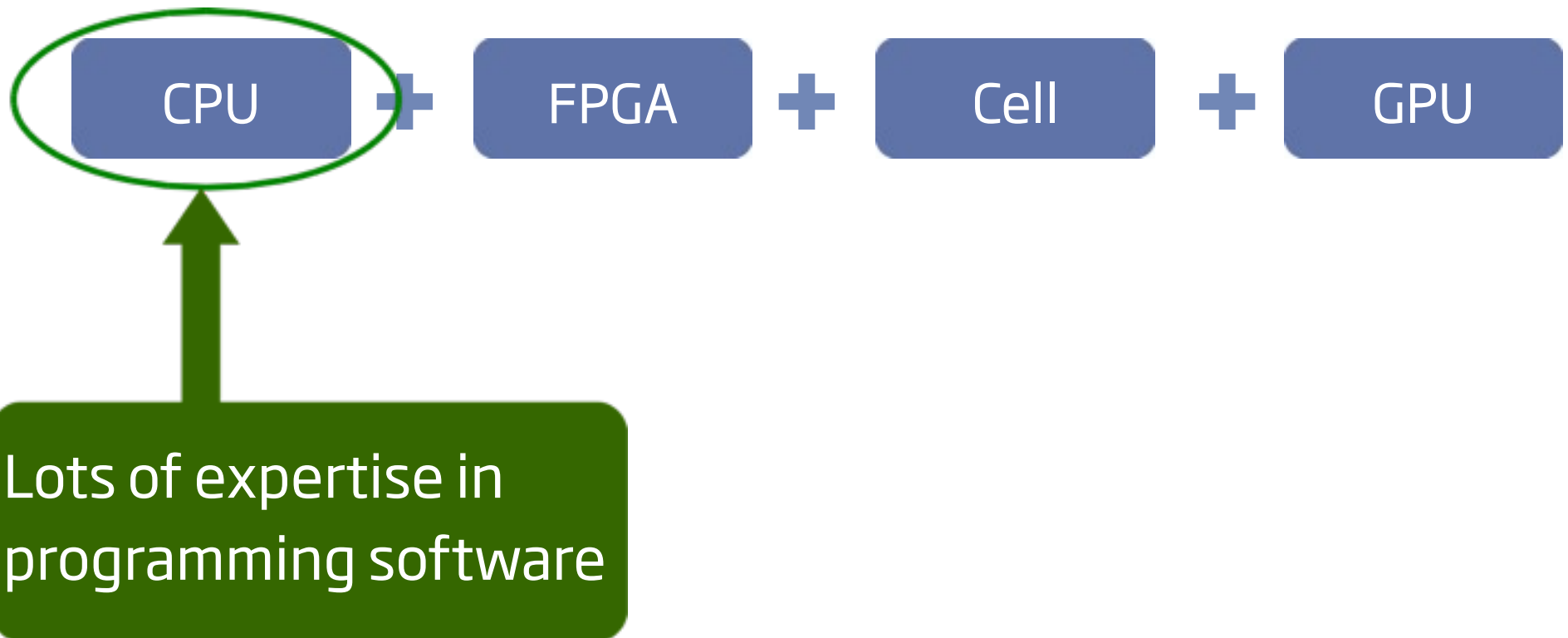
Alternatives?

- Yes, we've heard of multicores
- Heterogenous architecture:



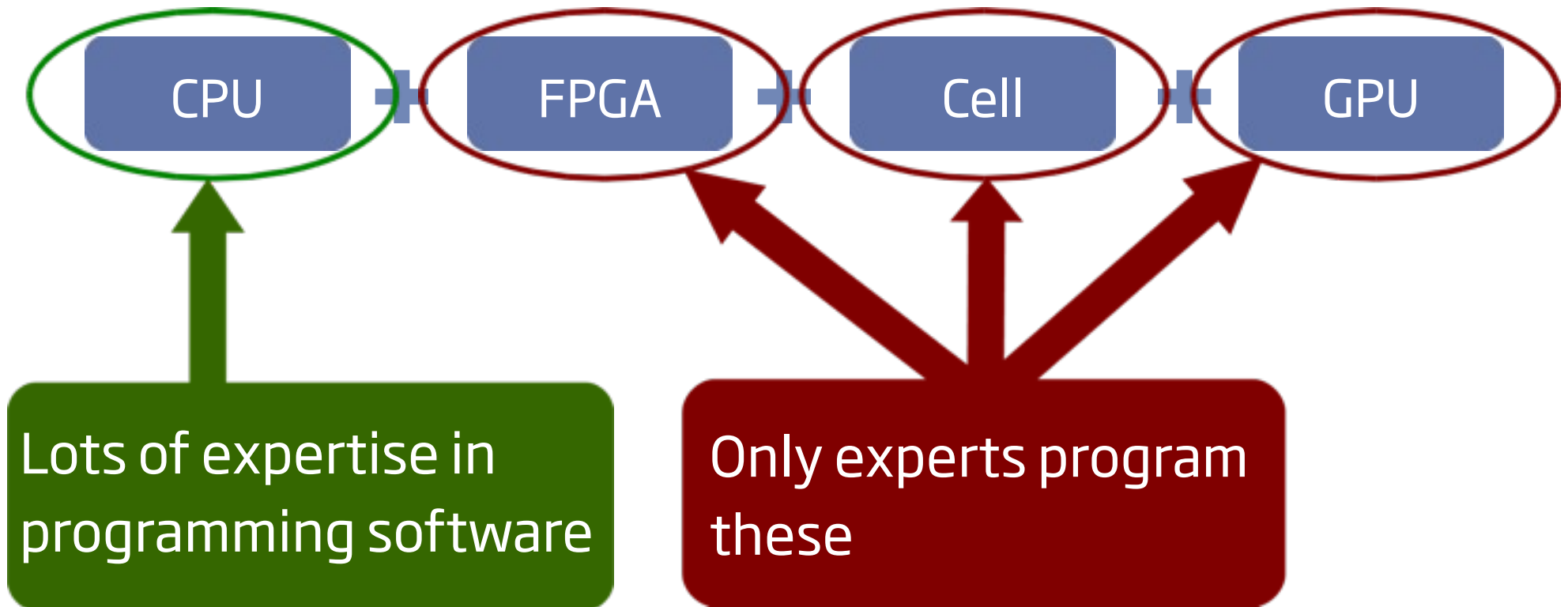
Alternatives?

- Yes, we've heard of multicores
- Heterogenous architecture:



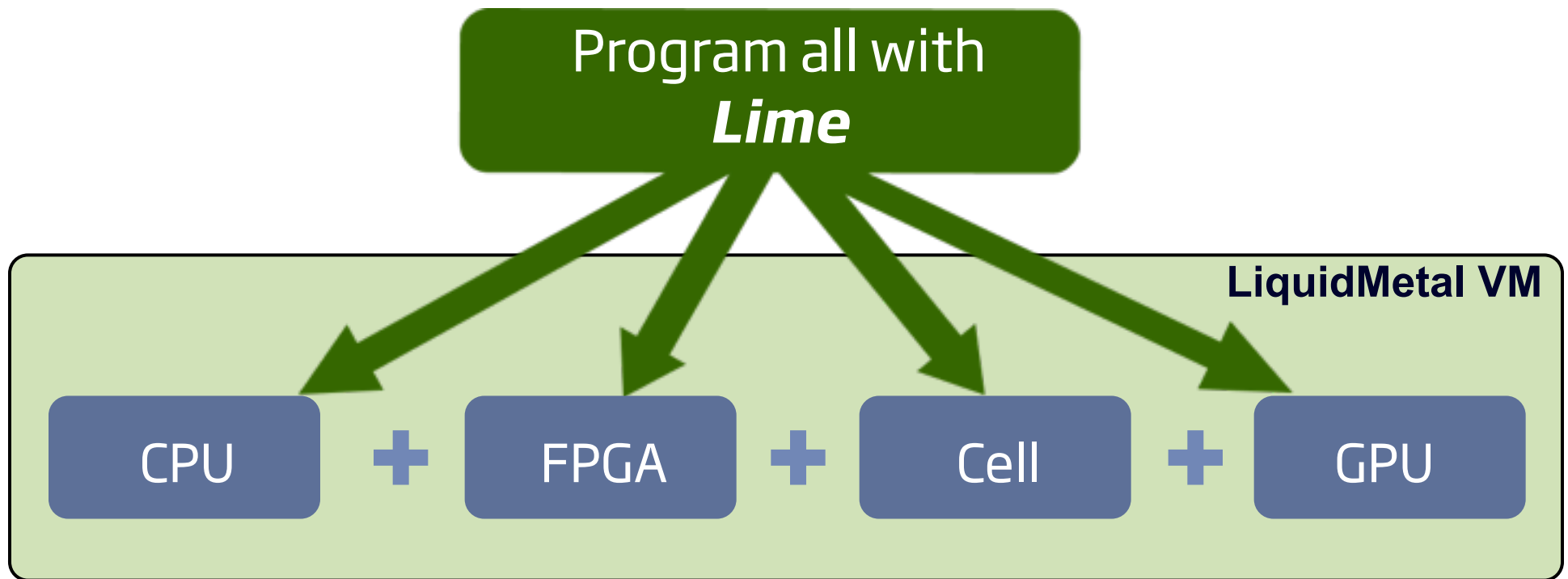
Alternatives?

- Yes, we've heard of multicores
- Heterogenous architecture:



Liquid Metal: Grand Vision

- One Unified language for programming heterogeneous architectures
 - Lime : Extension of Java



A Taste of Lime

```
Unsigned<T> permute( T permTable[T] ) {  
    bit newBits[T];  
    for ( T i )  
        newBits[i] = data[permTable[i]];  
    return new Unsigned<T>(newBits);  
}
```

A Taste of Lime

Lime

```
Unsigned<T> permute( T permTable[T] ) {
    bit newBits[T];
    for ( T i )
        newBits[i] = data[permTable[i]];
    return new Unsigned<T>(newBits);
}
```

C

```
work = ((left >> 4) ^ right) & 0x0f0f0f0f;
right ^= work;
left ^= work << 4;
work = ((left >> 16) ^ right) & 0xffff;
right ^= work;
left ^= work << 16;
work = ((right >> 2) ^ left) & 0x33333333;
left ^= work;
right ^= (work << 2);
work = ((right >> 8) ^ left) & 0xff00ff;
left ^= work;
right ^= (work << 8);
right = (right << 1) | (right >> 31);
work = (left ^ right) & 0xaaaaaaaa;
left ^= work;
right ^= work;
left = (left << 1) | (left >> 31);
```

A Taste of Lime

Lime

```
Unsigned<T> permute( T permTable[T] ) {  
    bit newBits[T];  
    for ( T i )  
        newBits[i] = data[permTable[i]];  
    return new Unsigned<T>(newBits);  
}
```

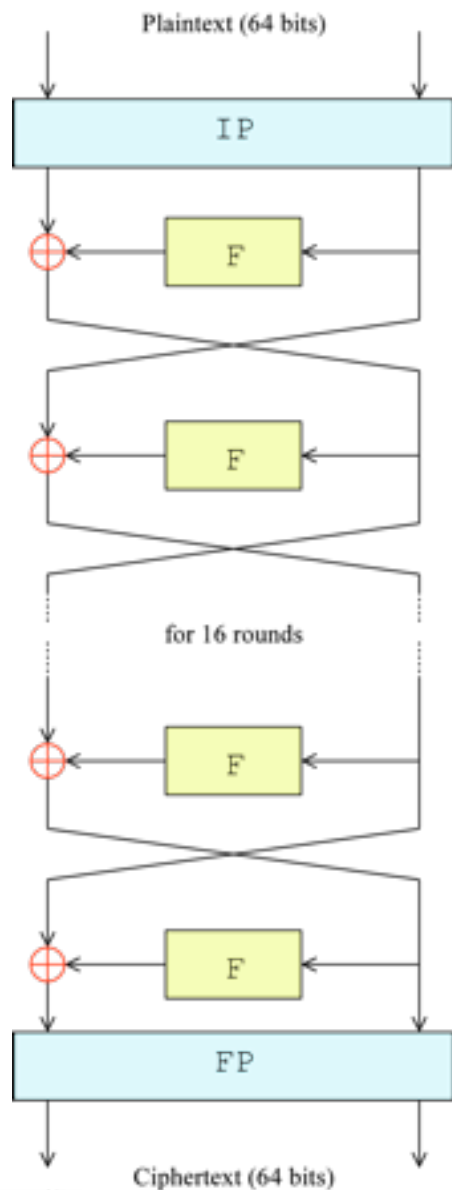
C

```
work = ((left >> 4) ^ right) & 0x0f0f0f0f;  
right ^= work;  
left ^= work << 4;  
work = ((left >> 16) ^ right) & 0xffff;  
right ^= work;  
left ^= work << 16;  
work = ((right >> 2) ^ left) & 0x33333333;  
left ^= work;
```

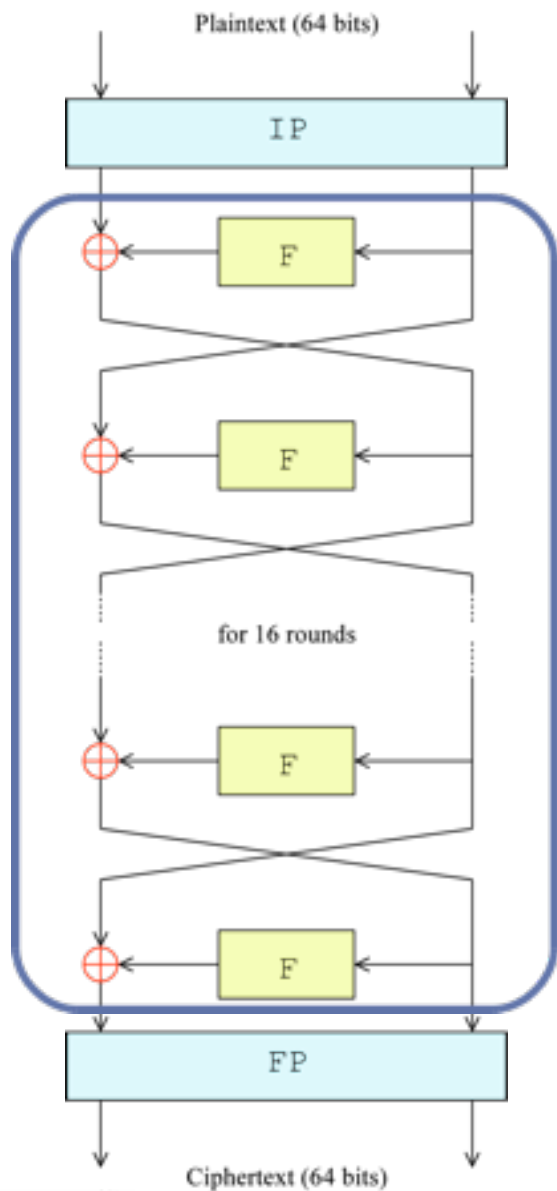
```
((left >> 4) ^ right) & 0x0f0f0f0f;
```

```
right ^= (work << 8);  
right = (right << 1) | (right >> 31);  
work = (left ^ right) & 0xaaaaaaaa;  
left ^= work;  
right ^= work;  
left = (left << 1) | (left >> 31);
```

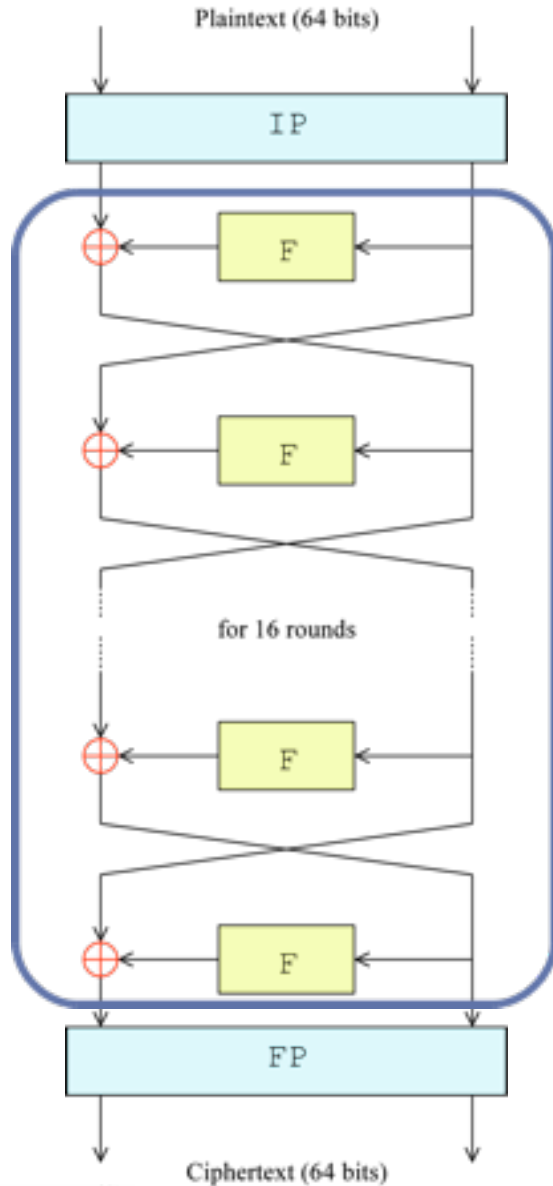
Example: Data Encryption Standard



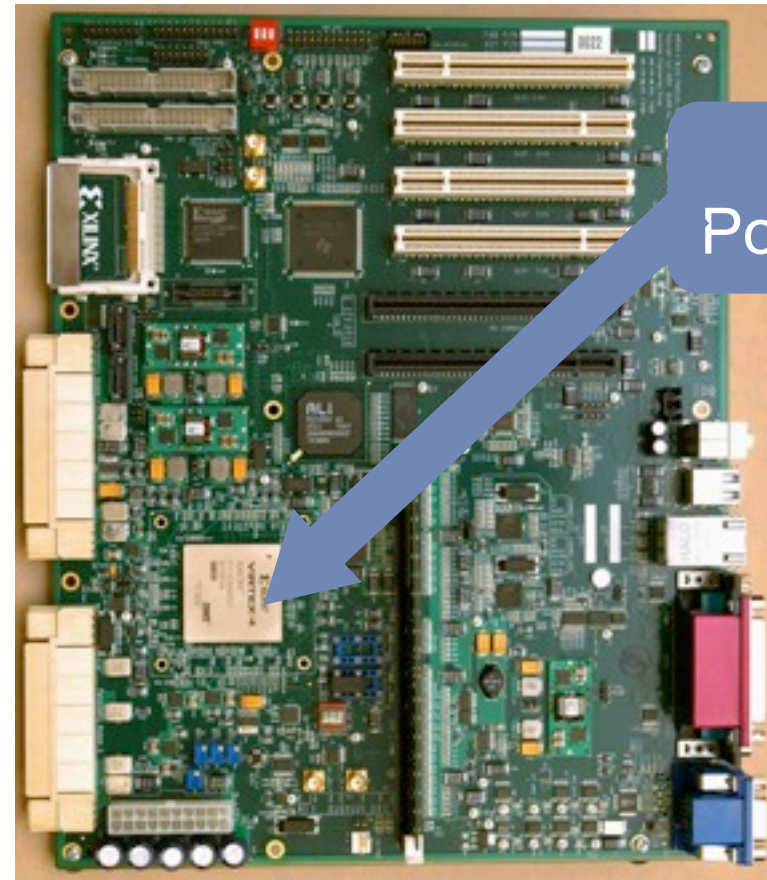
Example: Data Encryption Standard



Example: Data Encryption Standard

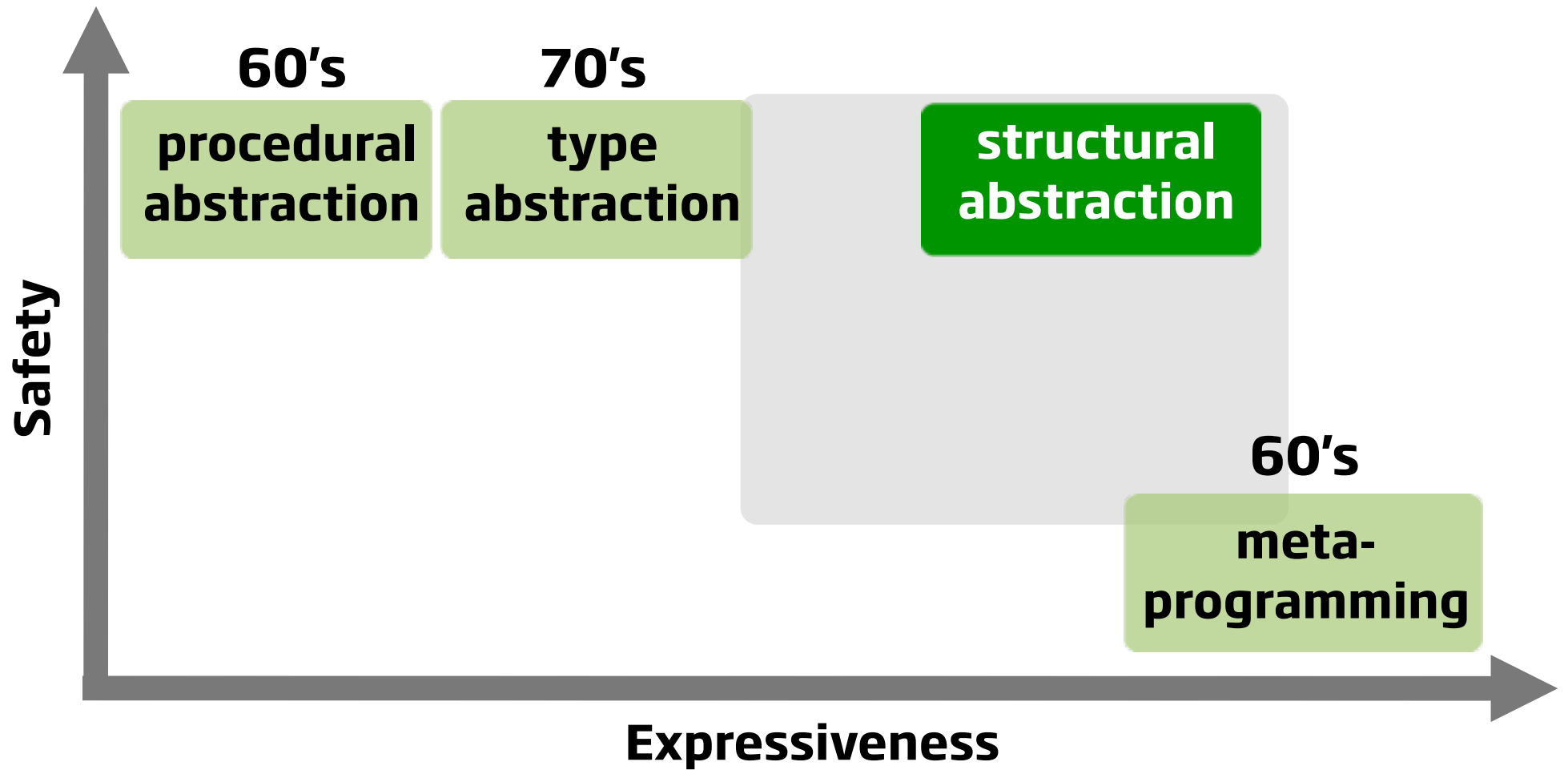


Xilinx Virtex-4

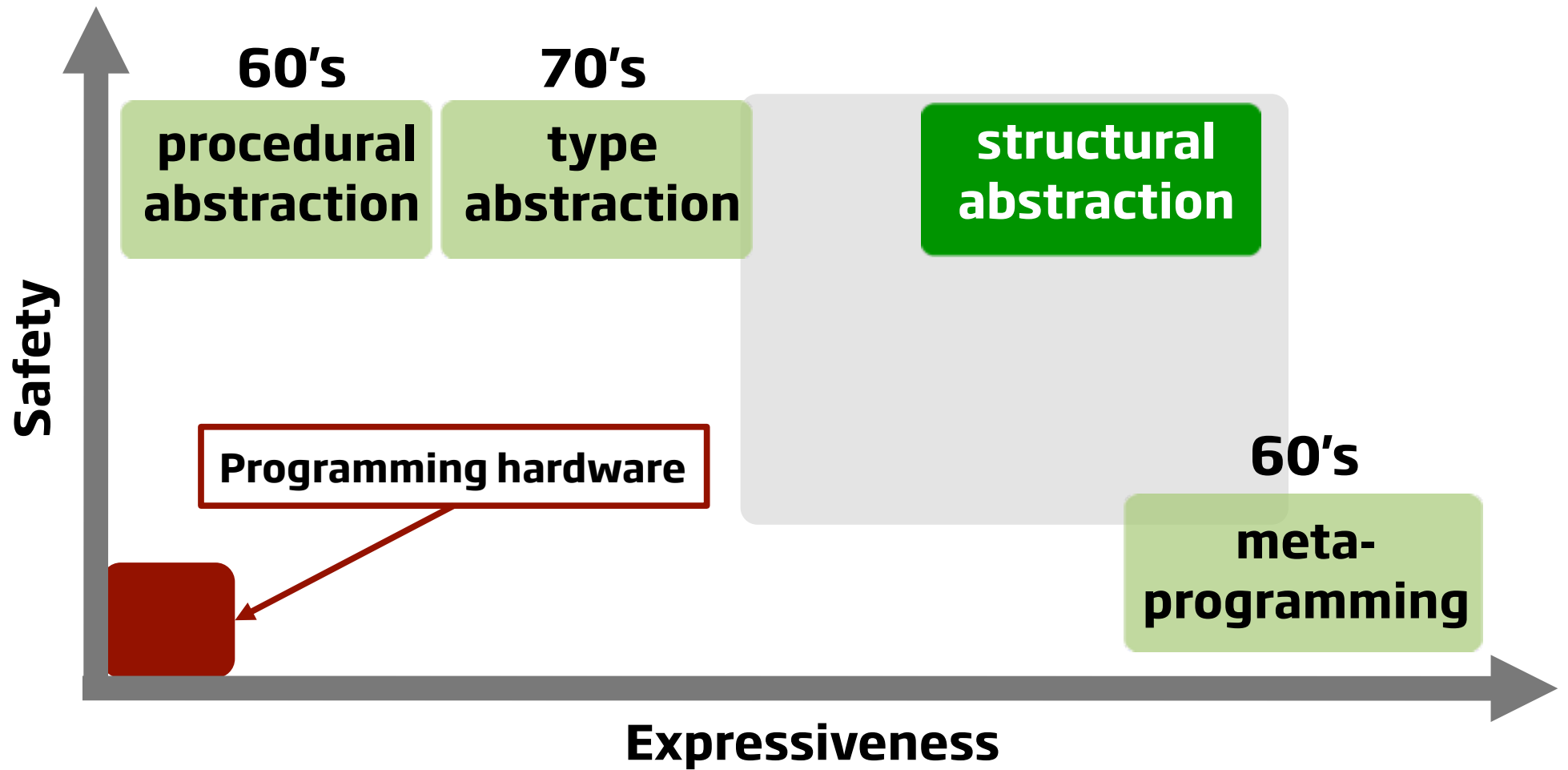


FPGA +
PowerPC405

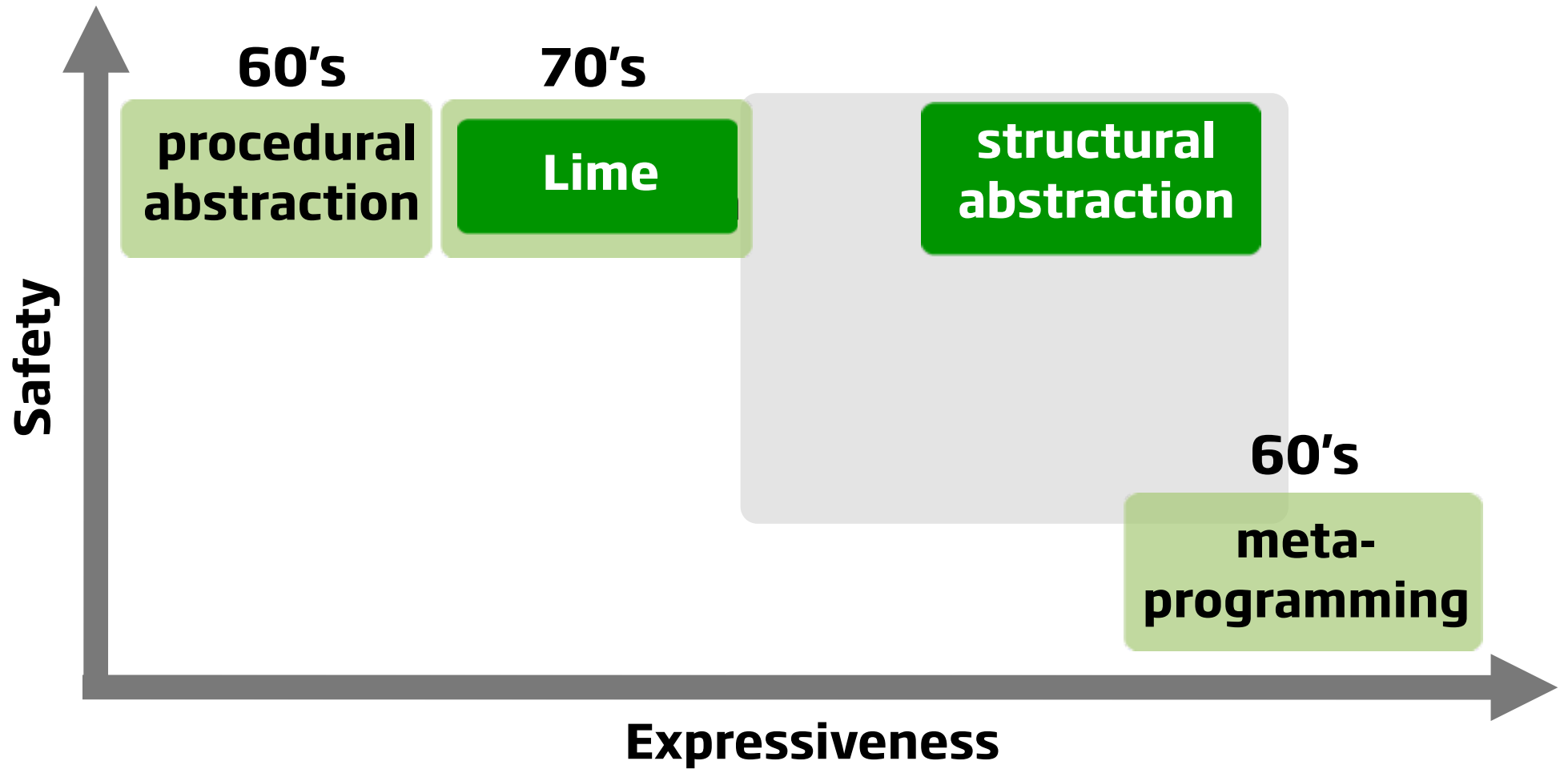
Abstraction Mechanism Design Space



Abstraction Mechanism Design Space

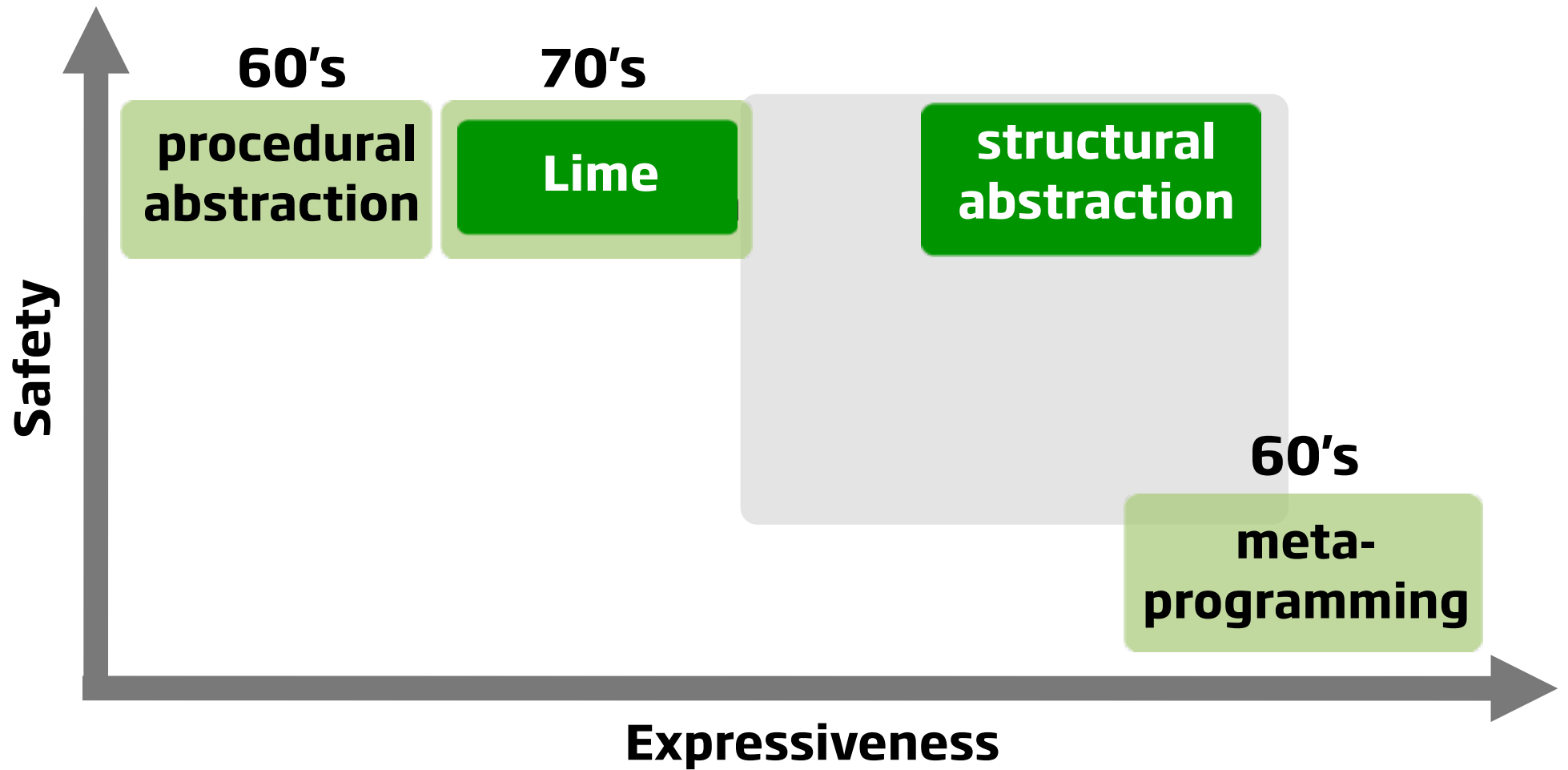


Abstraction Mechanism Design Space



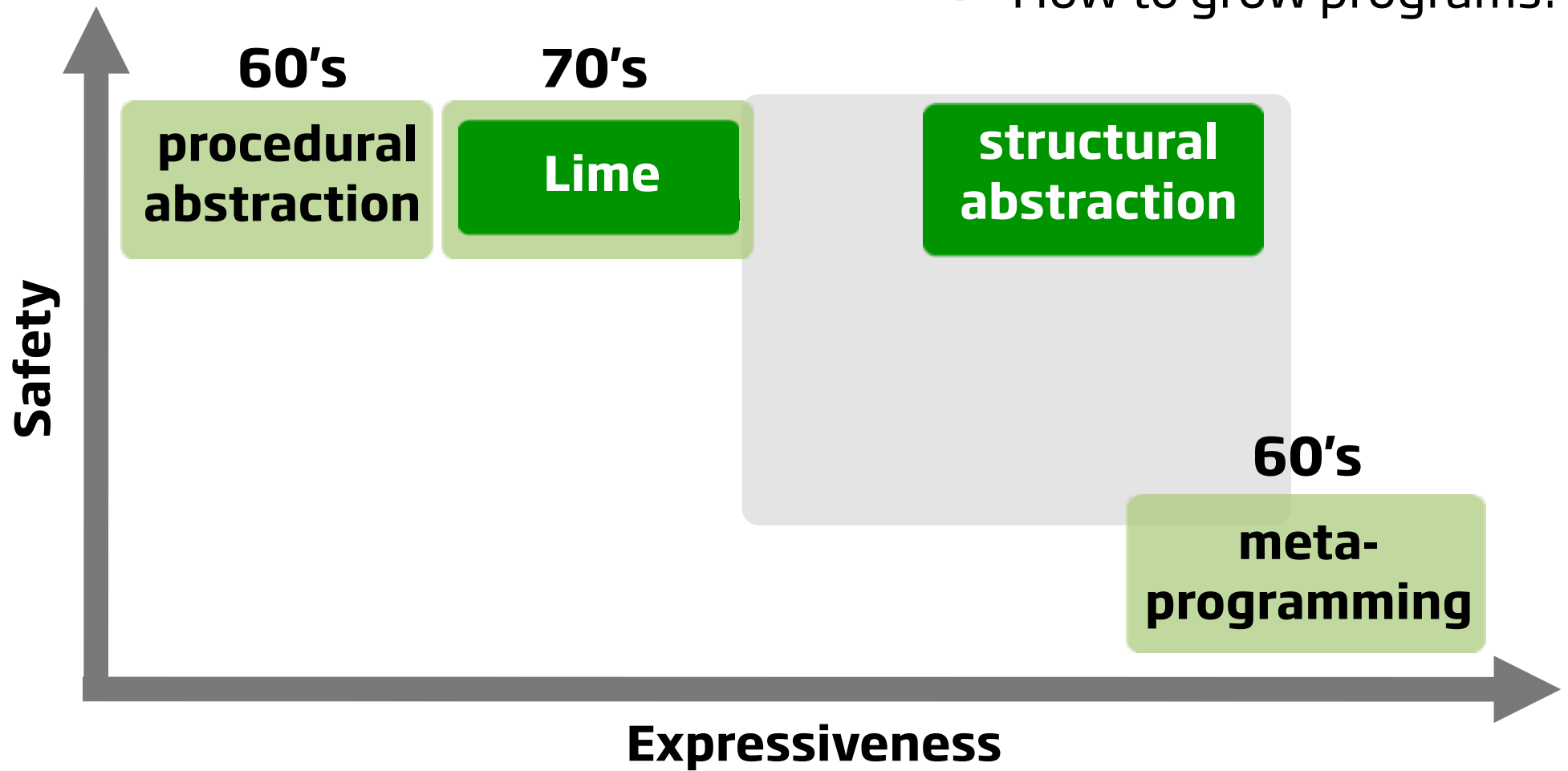
Abstraction Mechanism Design Space

- Structure: More than just methods and fields



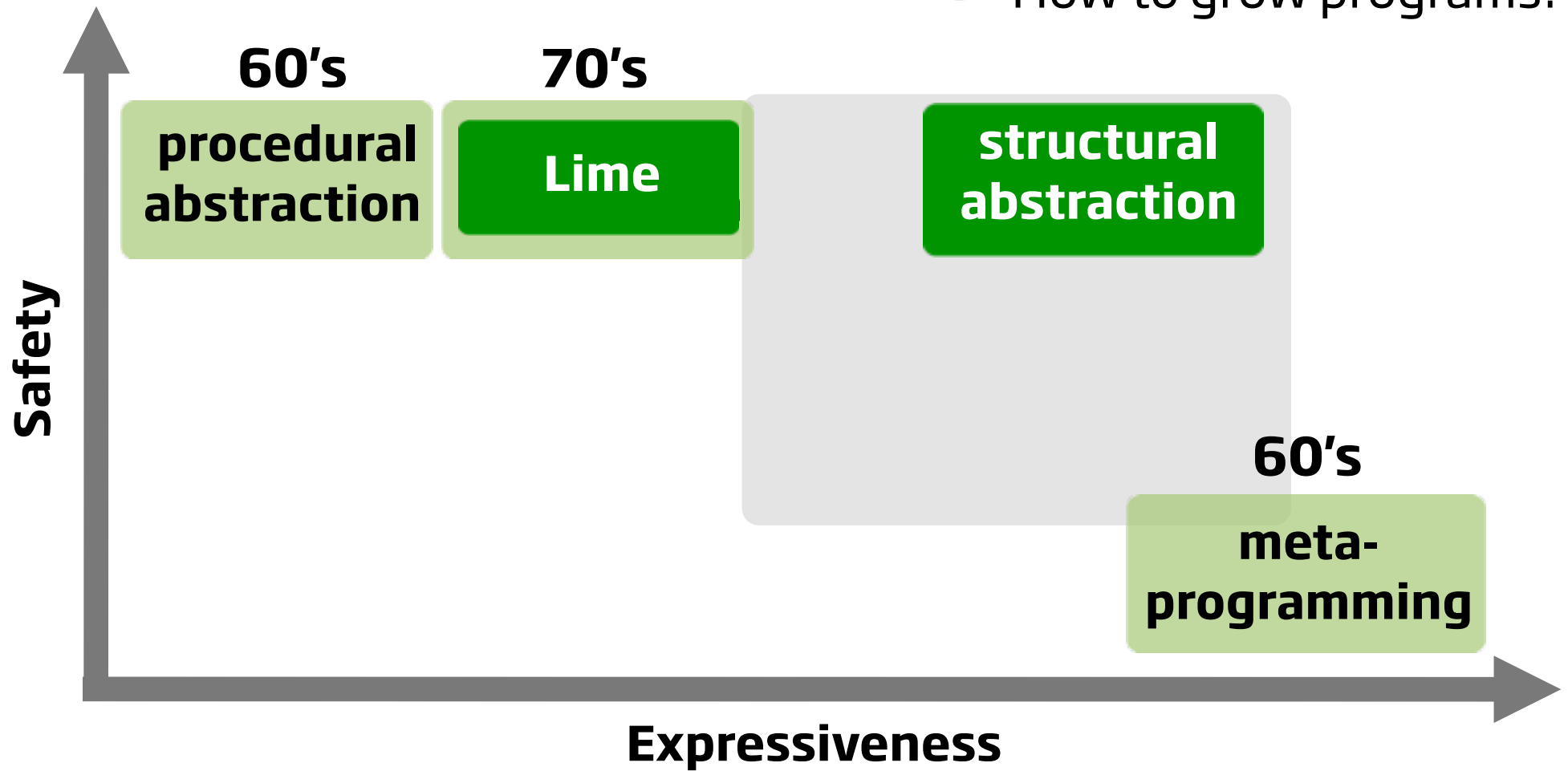
Abstraction Mechanism Design Space

- Structure: More than just methods and fields
- How to grow programs?



Abstraction Mechanism Design Space

- Collaboration with other domain experts
- Structure: More than just methods and fields
- How to grow programs?



Thank You!

Questions?