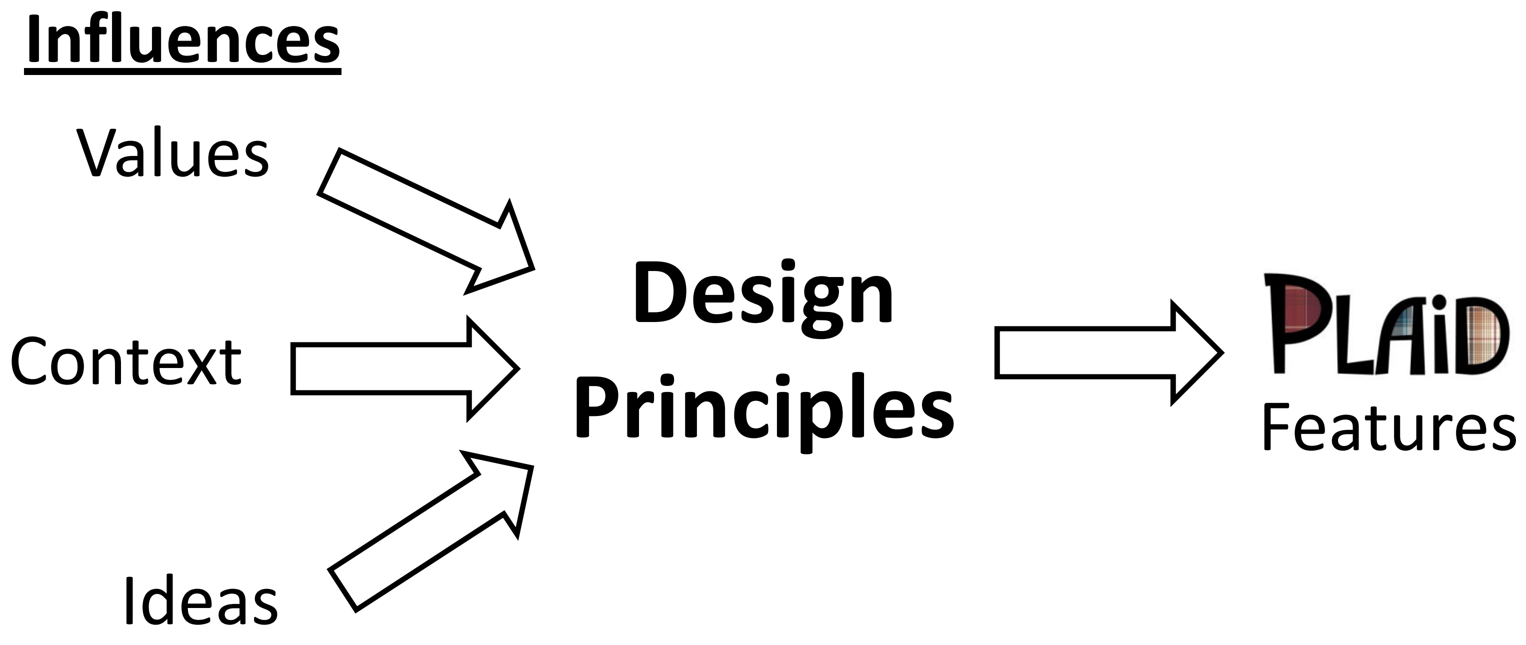
Principles of PLAID: The Design and Rationale for a New Programming Language

Jonathan Aldrich
Carnegie Mellon University
Principles of Programming Seminar
March 5, 2010

The Plaid Language

- Plaid is a new general-purpose language for professional programmers
- This talk is about Plaid's design, focusing on the principles



Influences: Values

- Dave Ungar: Values drove Self principles, which drove design [Dahl-Nygaard Lecture 2010]
  - His values: simplicity, creativity, accessibility, …

- Values driving Plaid
  - **Changeability:** modifying, reusing, and enhancing code
    - Requirements are constantly changing, we must respond effectively
  - **Compositionality:** dividing tasks into parts, working on them separately, synthesizing into a whole
    - Makes it possible to build large systems
  - **Understandability:** reading code, understanding how it works and how to use it
    - Enables achieving the other values

- We value other things as well, but those are the most important

PLAID

Influences: Context

- Increasing use of components (e.g. libraries and frameworks)
  - Programming challenges used to be algorithms and data structures
  - Now major challenge is **effectively leveraging components**
- Rise of multicore processors
  - Achieving speedup requires concurrency
  - **Concurrency is difficult and error-prone** in today's systems
- Ultra-large scale systems [SEI '06]
  - Develop software in a distributed, decentralized, data-driven, heterogeneous **environment**
- There are other major trends (e.g. cyber-physical) but these have less influence on Plaid's design

Influences: CMU Ideas and Strengths

- Principled, sound language design methodology
  - Type theory (CMU a pioneer)

- Deep understanding of software design
  - Designs that support scale and change (Notkin, Garlan & Shaw, …)
- Object models – and synergies with functional programming
  - Self (Ungar), Cecil (Chambers), EML (Millstein), Malayeri thesis, …
- Modular reasoning about state
  - Separation logic (Reynolds, Brookes, Krishnaswami thesis)
  - Typestate and permissions (Boyland, Bierhoff and Beckman theses)

PLAID

Principle 1: Procedural + Type Abstraction

- Core of OO is **procedural abstraction** [Reynolds ’75][Cook ’09]
  - Data abstracted via a set of functions in an existentially typed package
  - Can mix and match multiple implementations of a type (e.g. in a list)
  - Driving values: **changeability** and **compositionality**
    - key to achieving large-scale *reuse* in practice

- Contrast **type abstraction** in languages like ML
  - The name of the abstract type fixes a (hidden) representation
  - Can reason about types that go together (e.g. for binary methods)
  - Driving values: **understandability**
    - effect on *safety* and *performance*

- **Both are important!**
  - Plaid’s goal is to support them equally well
  - Many interesting questions in marrying OO, functional paradigms

PLAID

Features: Procedural + Type Abstraction

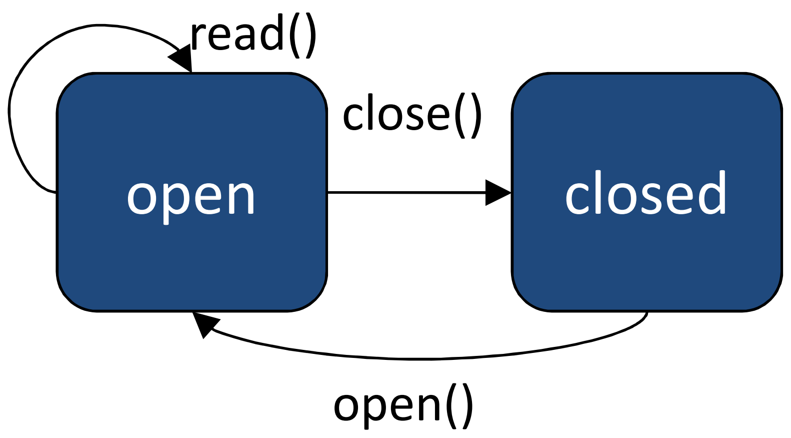
```
type Set = Collection with {
     method Set<E> union(Set<E> other);
}
state SetImpl = Collection with {
    List<E> members;
    method SetImpl<E> union(Set<E> other)
    {
        new SetImpl {
            members =
                members.appendAll(other);
        }
    }
} as Set
```

```
val ADT = new {
    type set = List;
    method set<T> union(
            set<T> s1, set<T> s2) {
        s1.appendList(s2);
    }
} as {
    type set <: { type E; };
    val union: set<T> * set<T> -> set<T>
}
```

Features: OO and Functional

- first-class lambdas = functors = methods
- object = module
- type = signature
- state = class = record = datatype
- abstract type = type member = class type parameter
- signature ascription
- composition (generalizes & cleans up inheritance)
- subtyping
- pattern matching (generalizes casts, external dispatch)
- local type inference (e.g. for local variables, method type parameters)

Principle 2: Support State Abstractions



- Abstract state machine constraints on object usage
  - Ubiquitous: 1/3 of Java classes are clients of an object protocol
  - Complex and undocumented: up to dozens of states per class
  - Motivates **typestate** [Strom & Yemeni ’86][Deline & Fähndrich ’04][Bierhoff & Aldrich ’07]

- Driving values
  - **Understanding** how to use an abstraction
  - **Composing** applications correctly out of components

PLAID

Features: Typestate-Oriented Programming

```
state File { String filename; }
state ClosedFile = File with {
    void open() [ClosedFile>>OpenFile] {
        this <- OpenFile {
            filePtr = fopen(filename);
        }
    }
}
state OpenFile = File with
    private CFile fileResource;
    int read();
    void close() [OpenFile>>ClosedFile];
}
```

State transition

State change primitive

Values specified for each new field

Different representation

New methods

PLAID

Why Typestate in the Language?

- Language influences thought [Boroditsky '09]
  - Language support encourages engineers to **think** about states
    - Better designs, better documentation, more effective reuse
- Improved library specification and verification
  - Typestates define when you can call read()
  - Make constraints that are only implicit today, explicit
- Expressive modeling
  - If a field is not needed, it does not exist
  - Methods can be overridden for each state
- Simpler reasoning
  - Without state: fileResource non-**null** if File is open, **null** if closed
  - With state: fileResource always non-**null**
    - But only exists in the FileOpen state

PLAID

DEMONSTRATION: Plaid Compiler

Principle 3: Describe Sharing of State

- Drivers: values and context
  - **understanding** non-local effects of mutable state
  - correctly **changing** and **composing** stateful components
  - enabling **safe concurrency**

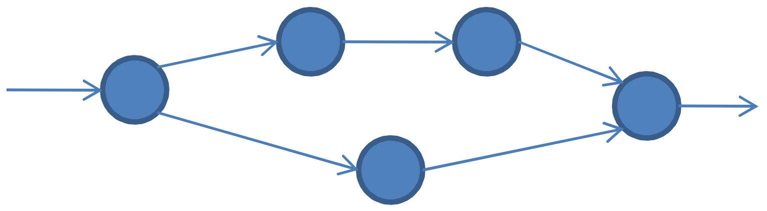
- Design
  - default is **immutable** data, no declarations required
  - for mutable data, default is **uniqueness**

  [Chan et al. '98]

  - shared mutable data annotated with hierarchical **data groups** [Leino '98]
  - lightweight **effect system** summarizes how functions access state
    - research challenge: making this scale [Gifford et al. '87], [Greenhouse & Boyland '99]

PLAID

Principle 4: Describe Dependencies



- Drivers: values and context
  - enabling **safe concurrency**
  - **understanding** computation in a concurrent setting
- Inspiration: functional programming is "naturally concurrent"
  - Up to data dependencies in program
- Goal: make dependencies on state updates explicit as well
  - Easier to track dependencies than all possible concurrent executions
  - Functional programming passes data explicitly to show dependencies
  - For stateful programs, we **pass permissions explicitly** instead
- Consequence: no need to express explicit control flow!

PLAID

Features: Sharing and Dependencies

method unique Data createData();

method void print(**immutable** Data d);

method unique Stats getStats(**immutable** Data d);

method void manipulate(**unique** Data d,
immutable Stats s);

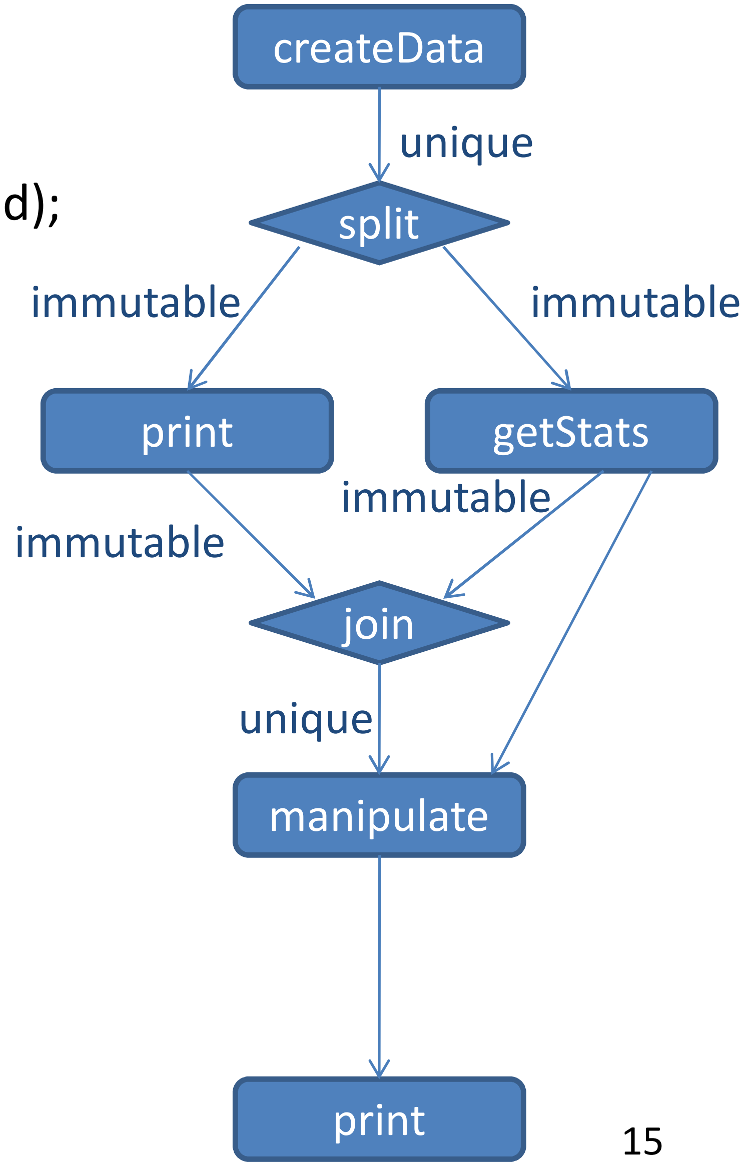
val d = createData();

print(d);

val s = getStats(d);

manipulate(d, s);

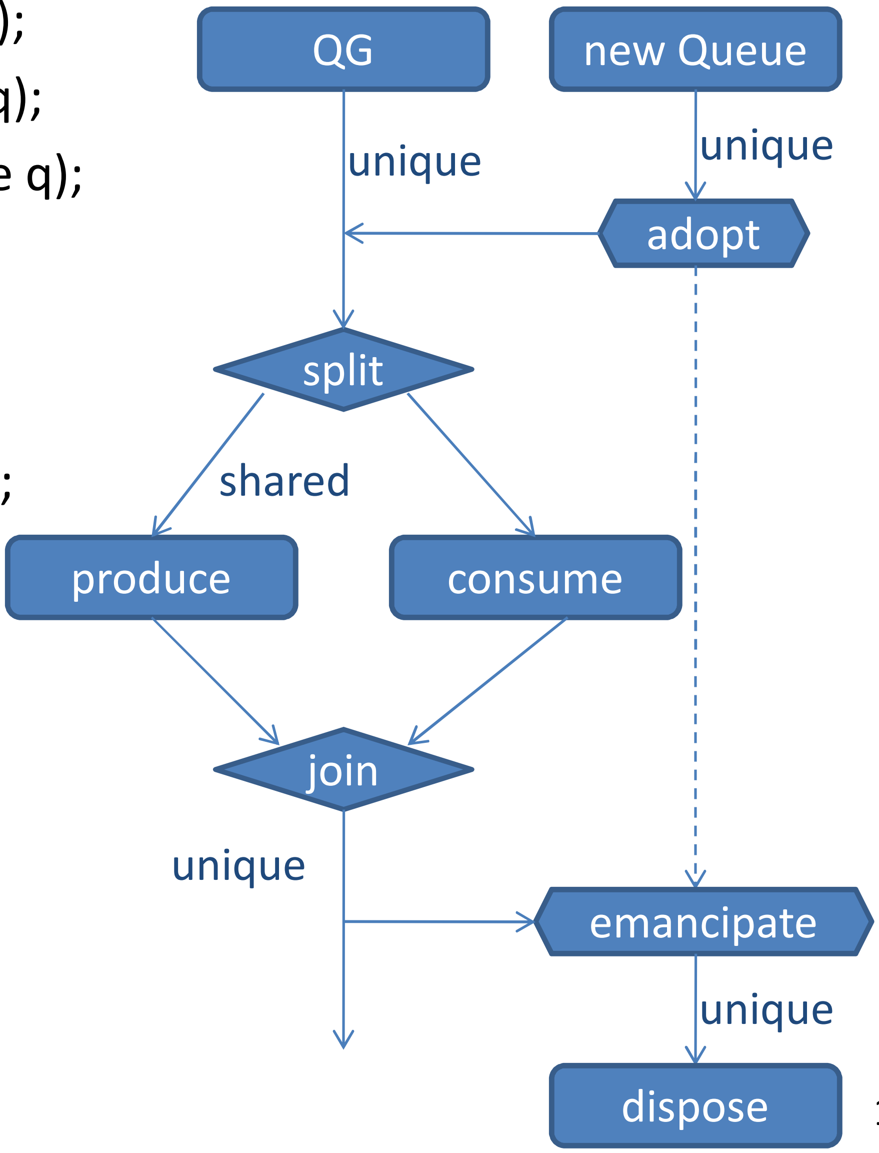
print(d);



Features: Sharing and Dependencies

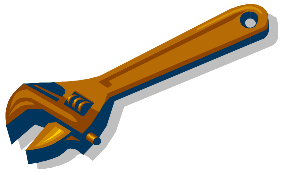
method void produce(‘QG Queue q);
method void consume(‘QG Queue q);
method void dispose(**unique** Queue q);

group QG;
val QG Queue q = new Queue;
split QG: produce(q) || consume(q);
q.dispose();



PLAID

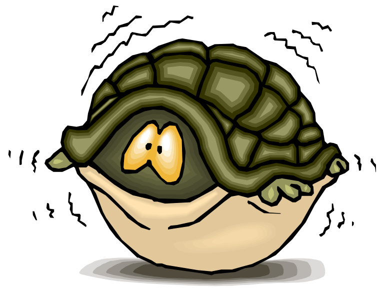
Principle 5: Static & Dynamic Checking



- Driving values
  - want to declare constraints for better **understanding**
  - desirable to check statically – but better to check dynamically than not at all
  - sometimes dynamic checking enables **composition** with dynamically typed code and/or easier **change**
- Principle
  - Every constraint that can be declared can be checked statically or dynamically
- Features
  - Gradual types: can omit some types, statically check as much as possible
  - Casts to types, states, and permissions
    - Research question: how to check a cast to **unique**?

PLAiD

Principle 6: Information Hiding



- Driving values
  - Facilitates **change** by ensuring clients depend only on interfaces [Parnas '72]
  - Enhances **understanding** and **composition** by supporting separate reasoning

- OO gets a bad rap for information hiding (in the PL community)
  - Real issue is industrial languages, not the OO paradigm

- Plaid will be second to none in its support for encapsulation
  - Should be possible to prove results like contextual equivalence

PLAID

Dynamic Types, Tag Tests

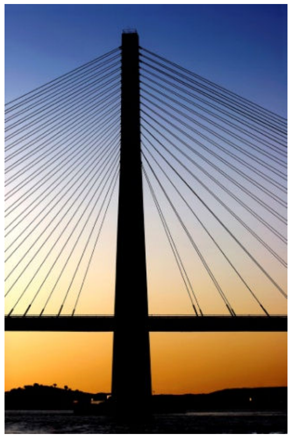
```
set = new Collection with {
    List<E> members;
    method Set<E> union(Set<E> other);
} as Collection with {
    method Set<E> union(Set<E> other);
}

dynamic dset = set;
dset.members.add(e); // FAIL at run time
```

```
type TestMember = {
    boolean isMember(E e); }
state List = { … }
state ArrayList case of List = { … }

List myList = new ArrayList{};
// match OK – ArrayList a case of List
match (myList) {
    case ArrayList al { … }
}

TestMember tm = myList;
// compile-time error: TestMember
// does not support case analysis
match (tm) { … }
```

Principle 7: Pay as You Go

- Should not pay for features of the language not being used
  - compare to Java – many system services built in, some have high overhead
- Research challenges
  - Changing representation
  - Casts that check the state and/or permission

Principle 8: Bridge to Existing Languages



- Drivers: values and context
  - **understandability** for programmers who know other languages
  - **composition** with components written in existing languages
  - in **ultra-large scale systems** we cannot control the language of all components

- Familiarity
  - use Java syntax wherever possible
  - when no clear language design choice, use Java's
    - fix some glaring problems like nulls (what Hoare calls his $1 billion mistake)

- Compatibility
  - compile to platforms, like the JVM, that have good existing libraries

PLAID

Principles Summary

1. Support procedural and type abstraction
2. Support state abstractions
3. Describe sharing of mutable state
4. Describe dependencies, not control flow
5. Combine static and dynamic checking
6. Enforce strict information hiding
7. Pay as you go
8. Provide a bridge to existing languages

Additional Standard Principles

- Abstraction
  - Anything with recursive structure can be abstracted
- Simplicity
  - Ungar: favor simplicity over expressiveness
- Modular checking
  - All typechecking is modular
- Memory safety
  - All behavior is well-defined
- Soundness
  - Well-typed programs do not go wrong
- Design intent
  - Ways of expressing the designer's intent at multiple levels of detail
- Direct manipulation (as in Self)
  - Everything – including modules – is first class; interpretation is supported

Current Plaid Language Research

• Core calculus	Darpan Saini, Joshua Sunshine
• Information hiding	Karl Naden
• Typestate model	Filipe Militão, Luís Caires (FCT)
• Gradual typing	Roger Wolff, Ron Garcia, Eric Tanter (U. Chile)
• Concurrency	Sven Stork, Paulo Marques (U. Coimbra)
• Web programming	Joshua Sunshine
• Permission parameters	Nels Beckman
• Compilation/typechecking	Karl Naden, Joshua Sunshine, Mark Hahnenberg, Sven Stork

PLAID

Future Plaid Research Topics

- Overall type system design
- Module system design
- Efficient compilation – achieving pay as you go
- Practical effect specifications
- Permission-aided modular verification
- Distributed system support
- Module versioning support
- Safer, more useable framework designs

The Plaid Language

- Values: change, composition, understanding
- Context: components, concurrency, ultra-large scale
- Primary principles
  - OO + functional abstraction, tracking state + dependencies, static + dynamic checking, modularity, efficiency, soundness
- Many new research directions
- Compiler implemented (in Java, for now)
- Plaid typechecker (in Plaid) underway

http://www.plaid-lang.org/

PLAID