「マルチパラダイムデザイン」

Multi-Paradigm Design and Implementation in C++

James O. Coplien

Distinguished Member of Technical Staff
Silicon Prairie Research Department, Bell Laboratories
Naperville, Illinois, USA

Visiting Professor, University of Manchester Institute of Science and Technology
Manchester, United Kingdom

cope@research.bell-labs.com

What Is a Paradigm?

- A way of organizing things
- We organize by abstracting
- Abstracting focuses on what is common
- We treat variations separately

C++ != OO

- Stroustrup has never called C++ an object-oriented programming language
- Most C++ practitioners use OO methods
- C++ is not just an OO language
- C++ begs for a method suited to its design
- Multi-paradigm design: a “universal” paradigm (in the universe of a given programming language)
- Portable to other expressive languages

Multi-paradigm Design

- Domain analysis
- Commonality analysis and variability analysis
- Commonalities and variabilities: in the application domain (“the problem”) and the solution (programming language)
- Mapping from application analysis to solution analysis
  - Distinguishable from patterns and idioms because it’s regularized
  - Distinguishable from OOA because it’s generalized

Part I: Domain Analysis

- Commonly equated with the breadth of analysis
- Multi-paradigm design: Richness of domains
  - Analysis of application sub-domains
  - Solution domain analysis
- Organize systems into software families
- Families form a domain

Properties of Domains

- They follow intuition and business experience
- Analyze to the boundaries of the business, not the application at hand
  - Leads to more maintainable systems
  - Might support reuse
- Domains and modular partitioning
  - The best domains are modularly disjoint
  - Domains often overlap

Example Domains for a Text Editor

- Commands (may be multiple editing languages)
- Buffers (with different memory management schemes)
- Files (with different character sets and formats)
- Keyboard (with/without function/arrow keys)
- Screen (different technologies)

Part II: Commonality Analysis

- The essence of abstraction
- Commonalities define families from family members
- We allow partitioning criteria to arise from the abstractions, not vice versa!
- Many axes of commonality:
  - Behavior
  - Data structure
  - Name
  - Code structure
- We loosely refer to these as *commonality categories*

Commonality Example: Text Editing Buffers

- **Structure:**
  - **number of lines**
  - **current line**
  - **name**
- **Behavior:**
  - **fill from file or stream**
  - **write to file or stream**
  - **yield a given line**
  - **internal memory management**

Part III: Variability Analysis

- Variability is the absence of commonality
- Variabilities distinguish family members
- Variabilities can be overwhelming
- Variability != detail!
- Parameterize variability
- We use the same axes for variability as for commonality (structure, behavior, type, name): commonality categories
- Variability analysis proceeds in parallel with commonality analysis

Variability Example: Text Buffers

- Working Set Algorithm
  - LFU, LRU, page, file
  - Note: *Behavior* is still common!
- Character set (ASCII, EBCDIC, FIELDATA, char, wchar_t)
- Versioned or unversioned

Parameters of Variation

- We want to parameterize the variations
- Establish *parameters of variation*
  - Working Set Algorithm
  - Character Set
  - Versioning
- Parameters of variation are the “genetic code” for the family
- By substituting “values” for each parameter, we can generate family members

Positive and Negative Variability

- Positive variability preserves underlying commonality
  - Public Inheritance preserves base class behavior
  - Template inheritance preserves common structure
- Negative variability violates commonality
- Examples in the solution domain:
  - #ifndef
  - Overriding argument defaulting
  - Template specialization
  - Inheritance with cancellation (private inheritance)

Application Analysis and Solution Analysis

- The problem domain exhibits commonalities and variabilities
- The solution domain also exhibits commonalities and variabilities, at all levels
- Programming languages express pairings of commonalities and variabilities
  - Inheritance: common behavior, different data and algorithm structure
  - Templates: common structure, type-sensitive behavior
  - Overloading: common name and semantics; different algorithms and parameters
  - . . . .
- Solution domain analysis is a peculiar activity

Solution Domain Analysis

- A given application commonality category/variability category pair constitute a *paradigm*
  - We organize software by abstracting
  - Abstracting focuses on what is common
- Object paradigm: common structure and behavior; variable structure and algorithm
- Overloaded procedures form a family: common name; variable algorithm
- Template instantiations form a family: common structure, different types, etc.
- + Note: this is a different sense of *paradigm* than used by Tim Budd.

Transformational Analysis Table

Commonality	Variability	Binding	Instantation	C++ Feature
Function Name and Semantics	Anything other than algorithm structure	Source	N/a	Template
	Fine algorithm	Compile	N/a	#ifdef
	Fine or gross algorithm	Compile	N/a	Overloading
Data Structure	Value of State	Run Time	Yes	Struct, simple types
	A small set of values	Run time	Yes	Enum
	Types, values and state	Source	Yes	Template
Related Operations and Some Structure	Value of State	Source	No	Module
	Value of State	Source	Yes	struct, class
	Data Structure and State	Compile	Optional	Inheritance
	Algorithm, Data Structure and State	Compile	Optional	Inheritance
		Run	Optional	Virtual Functions

Multi-Paradigm Design: Transformational Analysis

- Divide the application into intuitive sub-domains
- Tabulate the commonalities and variabilities of the application domain
- Align those with the catalogued commonalities and variabilities of the programming language
- Implement the analysis structure in terms of the transformational analysis

An Example: Text Editing Buffers

- Commonality: Behavior, some data structure
- Variabilities:
  - Character Set (Type)
  - Working Set Management (Gross Algorithm)
- Solution:
  - Templates (Common data structure, different type)
  - Inheritance with Virtual Functions (Common behavior, different functions)

TextBuffer Transformational Analysis

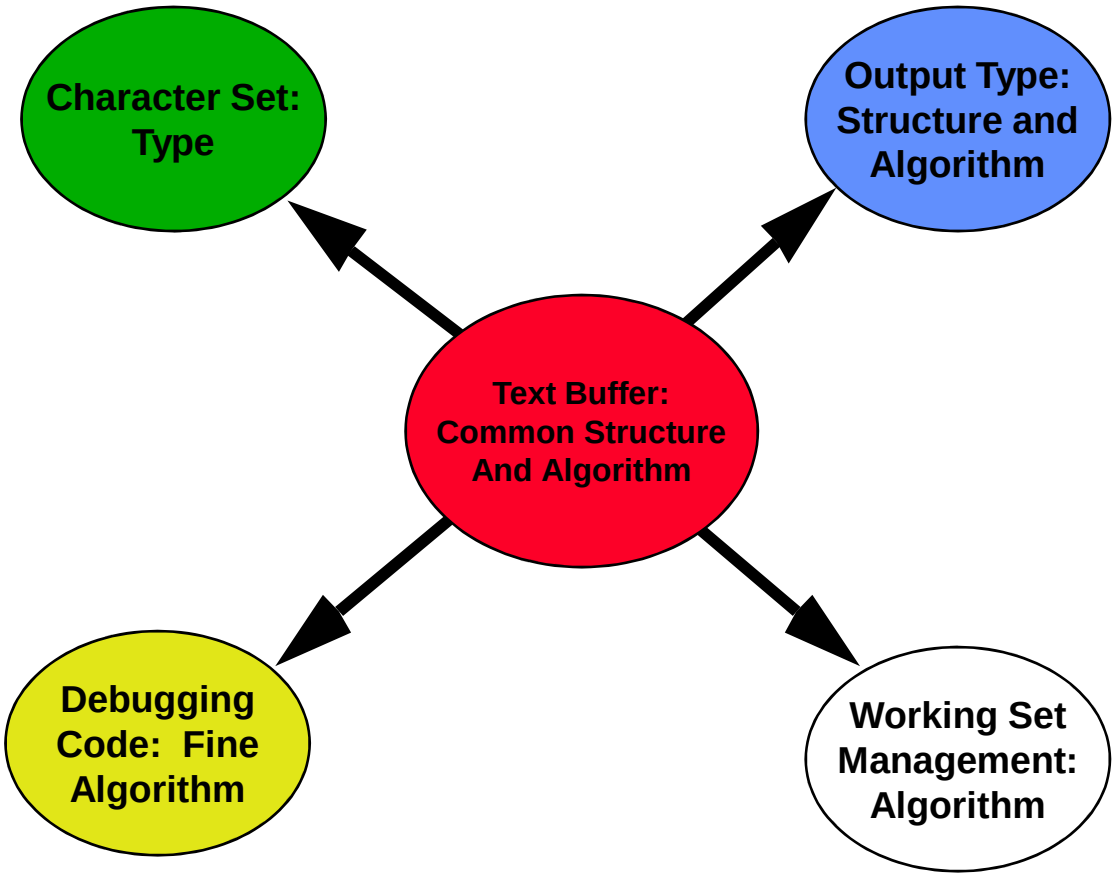
TextBuffer: Common Structure and Algorithm

Parameters of Variability	Meaning	Domain	Binding	Default Technique
Output Type Structure, *Algorithm*	The formatting of text lines is sensitive to the output medium	Database, RCS, TTY, UNIX file	Run	UNIX File *Virtual Functions*
Character Set *Non-structural*	Different buffer types support different character sets	ASCII, EBCDIC, FIELDATA	Compile	ASCII *Templates*
Working Set Management *Algorithm*	Different applications need to cache different amounts of memory	Whole file, whole page, LRU fixed	Compile	Whole file *Inheritance*
Debugging Code *Code Fragments*	Debug in-house only, but keep tests in source code	Debug, production	Compile	None *#ifdef*

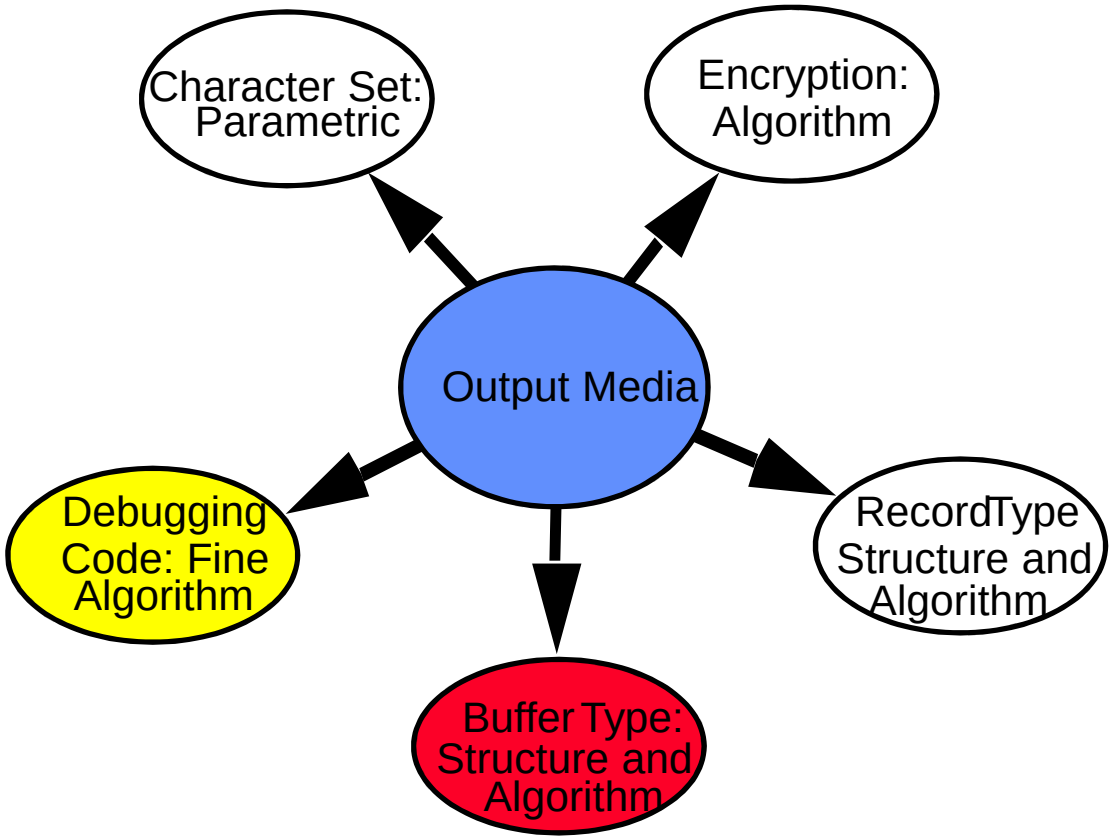
The Solution for Text Buffers

```
template <class CharSet> struct TextBuffer {
    virtual Line line(const LineNumber&) const;
    virtual void write(File&);
    . . . .
private:
    virtual void pageManagement(const LineNumber&);
};
class EmacsBuffer1: public TextBuffer<wchar_t> {
    void pageManagement(const LineNumber &);
    . . . .
};
class EmacsBufferJap: public TextBuffer<Katakana> {
    void pageManagement(const LineNumber &);
};
```

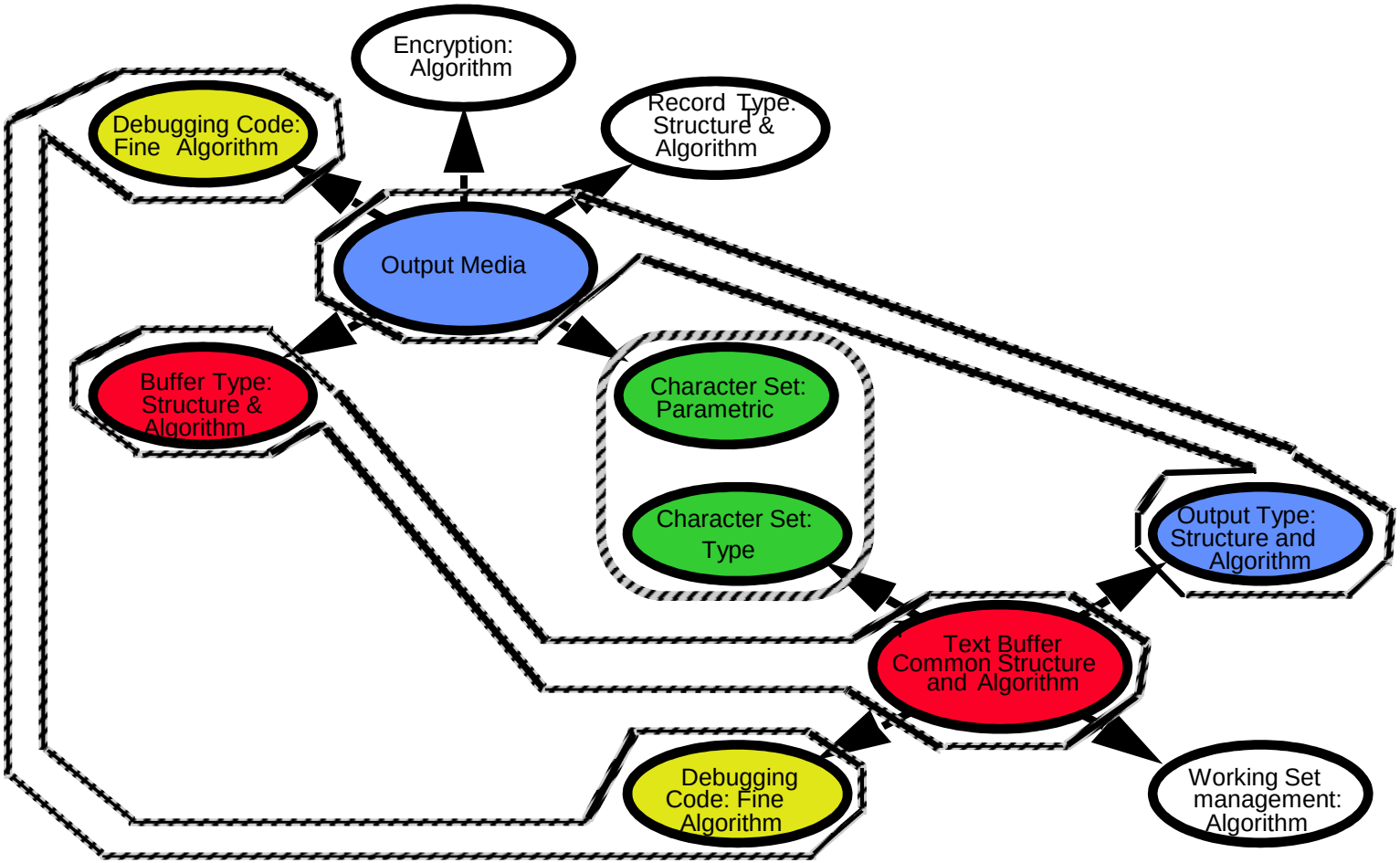
Text Buffer Dependency Graph



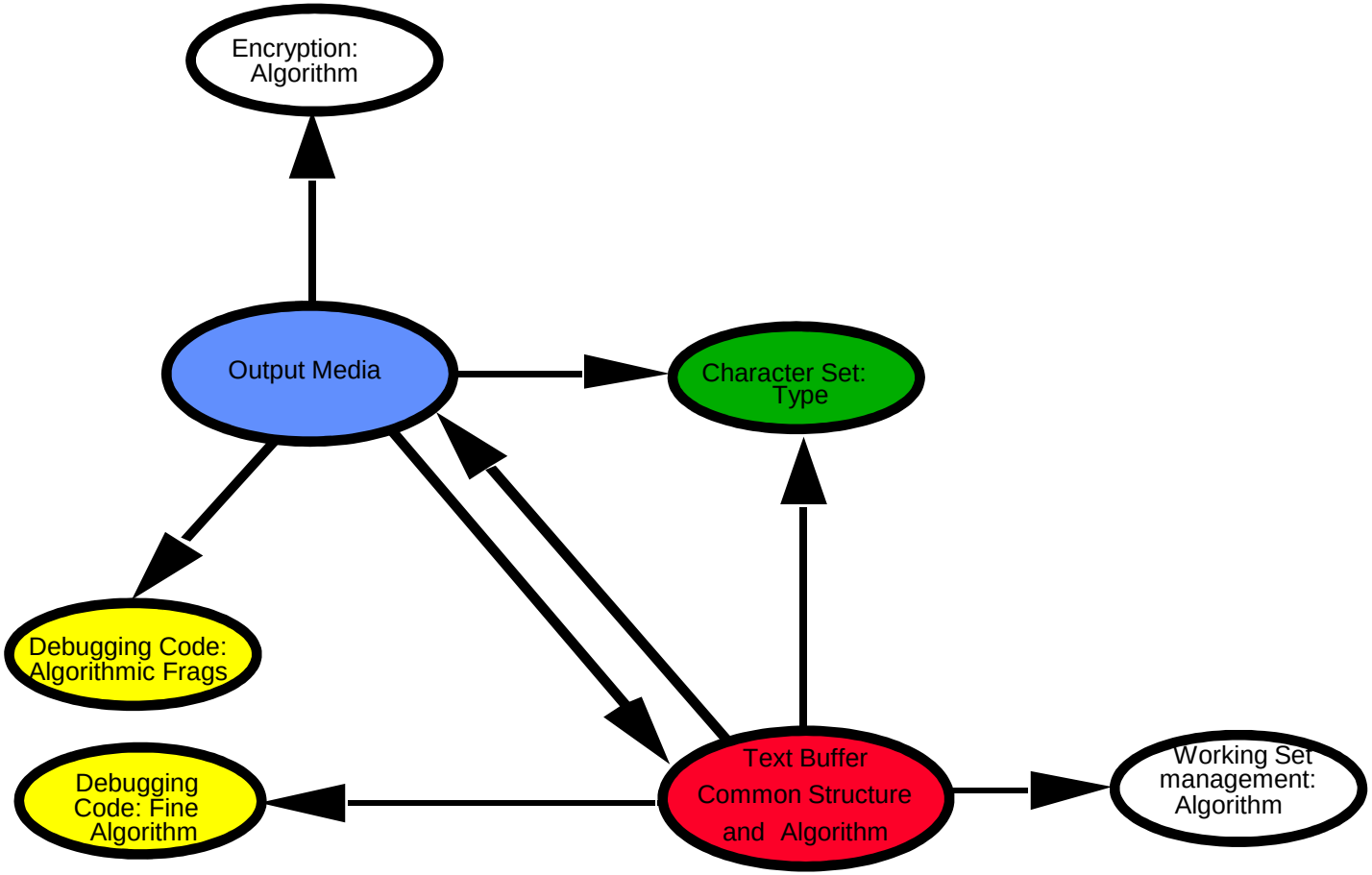
The File Domain



Multi-Domain Dependency Cycles



The unified design



One potential solution

```
template <class TextBuffer, class CharSet>
class OutputMedium {
public:
  void write() {
       . . . .
       subClass->getBuffer(writeBuf);
  }
  OutputMedium(TextBuffer *sc): subClass(sc) { }
protected:
  TextBuffer *subClass;
  CharSet writeBuf[128];
};

template <class TextBuffer, class Crypt, class CharSet>
class UnixFile: public OutputMedium<TextBuffer, CharSet>,
  protected Crypt {
public:
  UnixFile(TextBuffer *sc):
       OutputMedium<TextBuffer, CharSet>(sc) { }
  void read() {
       . . . .
       Crypt::decrypt(buffer);
  }
};
```

TextBuffers

```
template <class CharSet>
class TextBuffer {
public:
   string getLine() {
        string retval;
        . . . .
        return retval;
   }
   void getBuffer(CharSet *) { . . . . }
   TextBuffer() { . . . . }
};

template <class Crypt, class CharSet>
class UnixFilePagedTextBuffer: public TextBuffer<CharSet>,
   protected UnixFile<UnixFilePagedTextBuffer<Crypt,CharSet>,
        Crypt, CharSet> {
public:
   UnixFilePagedTextBuffer(): TextBuffer<CharSet>(),
        UnixFile<UnixFilePagedTextBuffer<Crypt,CharSet>,
                Crypt, CharSet>(this) { . . . . }
   string getLine() { . . . . read(); . . . . }
};
```

Encryption; main

```
class RSA {
protected:
   void encrypt(string &);
   void decrypt(string &);
};


int main() {
   UnixFilePagedTextBuffer<RSA, wchar_t> buffer;
   string buf = buffer.getLine();
   . . . .
}
```

Another example: A Finite-State Machine

- **AbstractFSM domain: the behavioral essence of all Finite State Machines**
- **UserFSM domain: the particular structure of a given FSM with its states, transitions, and actions**
- **GenericFSM domain: the structure common to all FSM implementations (transition tables, etc.)**
- **State, Stimulus: as commonly used in FSM terminology**

AbstractFSM Variability Table

Commonality: Structure and Behavior

Parameters of Variation	Meaning	Domain	Binding	Default
UserFSM *Structure, algorithm*	All generic FSMs understand how to add transitions in any UserFSM	Any class having member functions accepting a Stimulus argument	Compile time	None *Templates*
State *Type*	How to represent the FSM state	Any discrete type	Compile time	None *Templates*
Stimulus *Type*	The type of the message that sequences the machine between states	Any discrete type	Compile time	None *Templates*

ImplementationFSM Variability Table

Commonality: Structure and Behavior

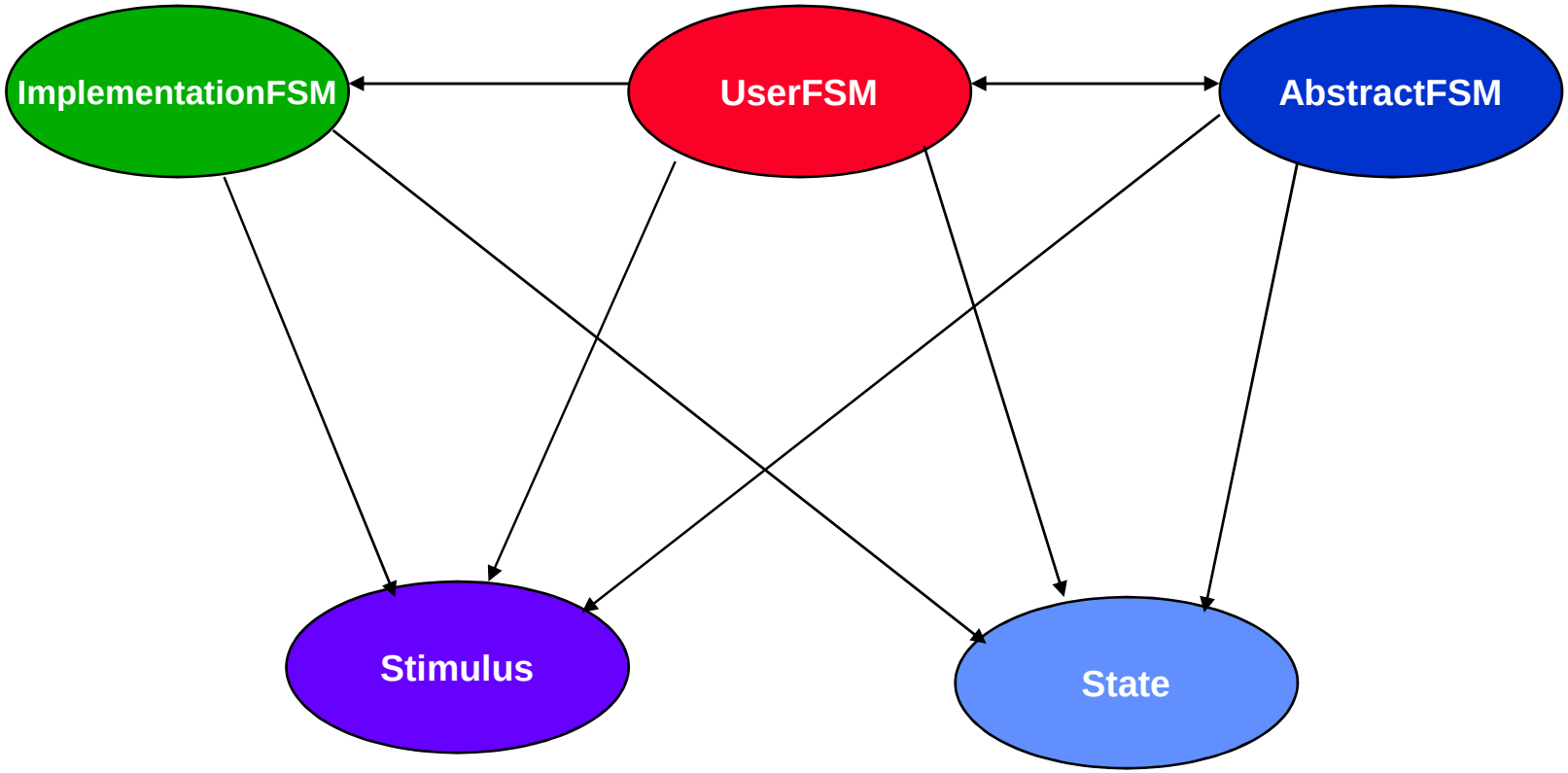
Parameters of Variation	Meaning	Domain	Binding	Default
UserFSM *Structure, algorithm*	To implement the state/action map, the implementation must know the type of user-defined actions and transitions	See previous slide	Compile time	None *Templates*
State *Type*	How to represent the FSM state	Any discrete type	Compile time	None *Templates*
Stimulus *Type*	The type of the message that sequences the machine between states	Any discrete type	Compile time	None *Templates*

UserFSM Variability Table

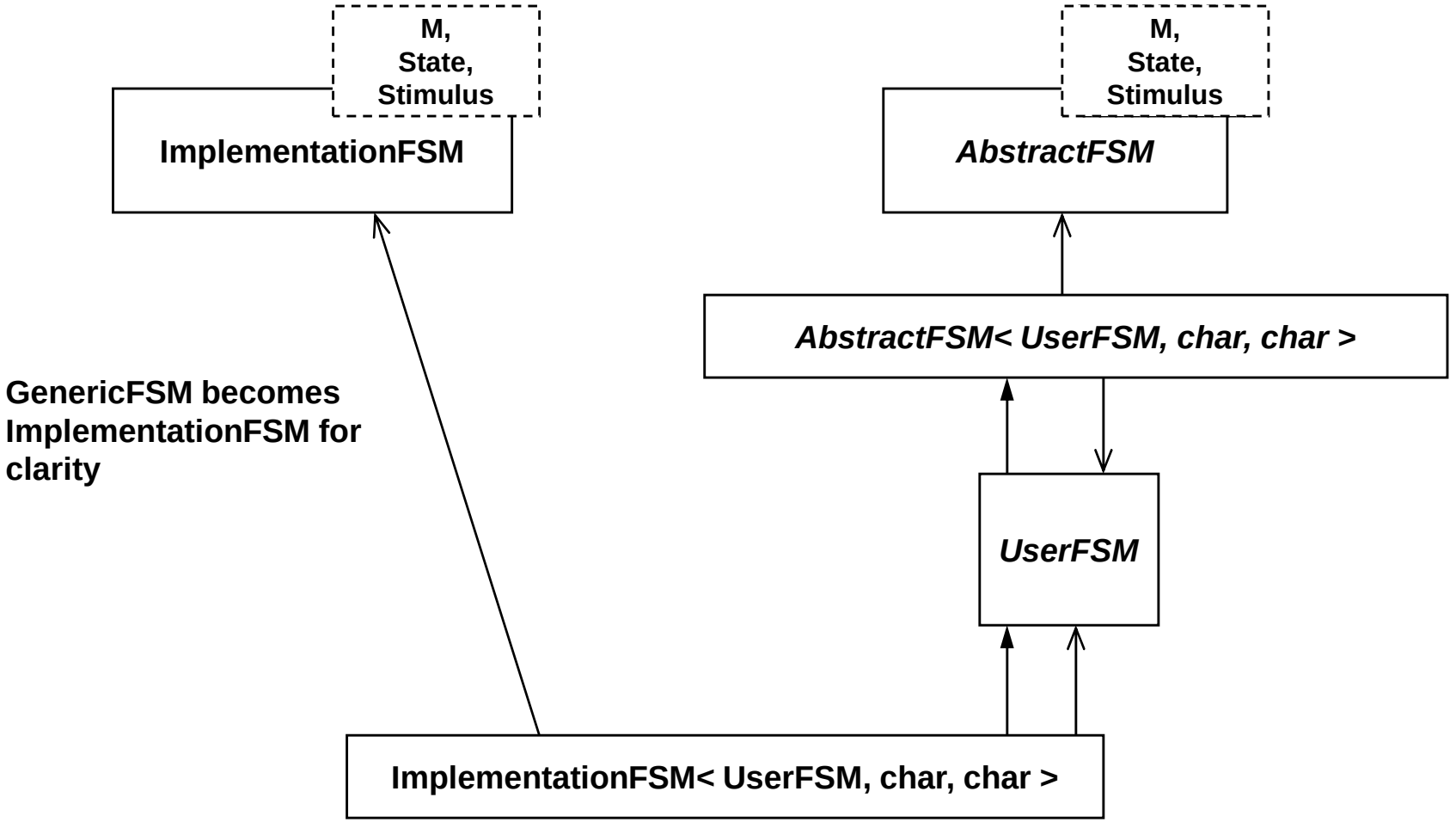
Commonality: Aggregate Behavior

Parameters of Variation	Meaning	Domain	Binding	Default
AbstractFSM *Structure, algorithm*	The UserFSM uses the protocol from the AbstractFSM domain	See previous slide	Compile time	None *Inheritance*
State *Type*	How to represent the FSM state	Any discrete type	Compile time	None *Hand-coded or typedef*
Stimulus *Type*	The type of the message that sequences the machine between states	Any discrete type	Compile time	None *Hand-coded or typedef*
Actions *Algorithm*	Each UserFSM implements its own semantics in transition functions	Any number of functions that map a Stimulus and current state to a new state	Compile time	None *Inheritance*

The FSM Domain Diagram



UML Spec of FSM



Code for the Design

```
#include <Map.h>

template<class M, class State, class Stimulus>
struct AbstractFSM {
    virtual void addState(State) = 0;
    virtual void addTransition(Stimulus, State, State,
        void (M::*)(Stimulus));
    virtual void fire(Stimulus) = 0;
};

struct MyFSM: public AbstractFSM<MyFSM, char, char> {
    void x1(char);
    void x2(char);
    void init() {
        addState(1);  addState(2);
        addTransition(EOF,2,3,&MyFSM::x1);
    }
};
```

More code for the design

```
template <class UserMachine, class State, class Stimulus>
class FSM: public UserMachine {
public:
    FSM() { init(); }
    virtual void addState(State);
    virtual void addTransition(Stimulus, State from,
           State to, void (UserMachine::*)(Stimulus));
    virtual void fire(Stimulus);
private:
    State nstates, *states, currentState,
    Map<Stimulus, void (UserMachine::*)(Stimulus)>
        *transitionMap;
};

FSM<MyFSM, char, char> myMachine;
```

Objects, Multi-Paradigm Design, and Patterns

- **Patterns: A technique to apply a fixed set of solutions to a fixed set of problems**
- **OOA/OOD: A technique to find abstractions**
- **Multi-Paradigm Design: A method to find abstraction techniques and reduce them to practice**