Object Oriented Programming

Introduction to Computer Science II

Christopher M. Bourke
cbourke@cse.unl.edu

Object Oriented Programming

Introduction

Definition

Object-Oriented Programming is a programming paradigm that uses objects (data structures which have data fields and methods) which interact together within a computer program.

Object Oriented Programming

Key Ideas

▶ Modularity of design
▶ Encapsulation of data into Objects
▶ Abstraction of an object’s behavior and representation
▶ Hierarchical organization of objects with subtypes inheriting behavior and state
▶ Polymorphism - ability of objects to have multiple forms in different contexts at different times

Object Oriented Programming

Keep in mind

When talking about OOP, keep in mind:

▶ Definitions are often blurred, authors/contexts may differ slightly
▶ Languages can have some, all, or none OOP features
▶ Programs can ignore or utilize some subset of features
▶ Non-OOP languages can support some (or all!) OOP features without being “OOP”
▶ Debate still going as to “best” practices, how to do OOP, etc.
▶ Understanding some concepts requires understanding other concepts

Object Oriented Programming

Advantages

▶ Facilitates modeling of real-world agents & problems
▶ Supports modularity & decomposition
▶ Clear demarcation of responsibilities and functionality
▶ Separates interface (the what) from implementation (the how)
▶ Promotes code reuse
▶ Bugs, side-effects, etc. can be localized, less and more predictable testing

Motivation I

Consider the evolution of computer languages

▶ Machine Code → Assembly → Imperative Language
▶ Each iteration still forces you to think in terms of a computer’s structure
▶ Other Programming language paradigms force models/problems into their paradigm
▶ Example: LISP assumes that all problems involve lists
▶ Example: C assumes all problems are procedural (algorithmic)
Motivation II

- In general, problems are forced into a language’s paradigm
- Involves some “translation” or pigeonholing
- Good for domain-specific problems or just number-crunching, bad for generality
- Opposite of what it should be: computers should be serving our needs
- Languages should model our world and problems

Versus Imperative

**Imperative Paradigm**

- Specifies a specific process (algorithm) that changes a program’s state
- Procedural & Structured paradigms (state changes are local or parameterized), not encapsulated
- Examples: FORTRAN, C

Contrasting with Other Paradigms

- Most modern programming languages support some OOP features
- Features back-added due to demand or advantages that OOP provides
- OOP vs. Declarative paradigm
- OOP vs. Imperative paradigm

Versus Declarative

**Declarative Paradigm**

- “Declares” behavior of a program
- Specifies logic of computation, not control flow
- Functional, avoids state (Lisp, Scheme, Haskell)
- Logic programming
- Domain languages: Regular Expressions, HTML, Excel, SQL
- Not necessarily Turing-complete

History of OOP I

- Paradigms have “evolved” to OOP
- MIT 50s - 60s: LISP “atoms”
- 1960: Sketchpad (first CAD, revolutionary GUI) by Ivan Sutherland, introduced initial concepts
- 1967: Simula 67 by Dahl & Nygaard (bank simulation; agents: tellers, customers, accounts, transactions; physical modeling)
- 1972: Smalltalk by Alan Kay
  - Based on Plato’s Theory of Forms
  - Allegory of the Cave
  - “A form is an objective ‘blueprint’ of perfection” (an archetype)
  - No perfect circle, just shadows, representations of it verything has a quintessence, an archetype, an apotheosis
- 1983: Bjarne Stroustrup develops C++
  - Inspired to add OOP features to C from Simula

History of OOP II

- First compilers simply translated C++ syntax to pure C
- 1983: Brad Cox & Tom Love develop Objective-C
  - Inspired to add OOP features to C from Smalltalk
- 1990s: OOP came to dominate programming methodologies (especially due to GUI development)
- Languages updated to include OOP features
- 1995: Java
  - 1995 by James Gosling, Sun Microsystems
  - “simple, object-oriented and familiar”
  - “robust & secure”
  - “architecture-neutral and portable”
  - “high performance”
  - “interpreted, threaded, dynamic”
Continued Refinements I

- Design Patterns - a general reusable solution to a common and reoccurring problem in software design (also called idioms)
- Anti-patterns and "smells"
- Modeling tools (UML – Unified Modeling Language)
- Design-by-contract frameworks
- Integrated Development Environments (IDEs)
- Development of libraries, frameworks
- Development of methodologies, best practices, etc.

Continued Refinements II

Figure: UML Diagram Example

Objects

Abstract Data Types

- Basis for Objects: Abstract Data Types (ADTs)
- Mathematical model for classes of data structures with similar behavior or semantics
- Purely theoretical entities
- Generalization applicable to areas other than OOP
- Algebraic structures, AI, algorithms, etc.
- Examples: stacks, queues, maps, strings, and their operations

Definition

An object is a general entity characterized by the following.

- **Identity** – An aspect that distinguishes it from other objects
- **State** – properties (attributes, elements, data, fields)
- **Behavior** – describes an object’s *interface* (methods, routines, functions)

Objects I

Key Concepts

Message sending

- Objects provide services
- Client code (user) can access these services by sending messages
- Example: a signal to execute one of its methods
- A signal to turn a Light object on() or off()
- Facilitated by an externally available *interface*

Objects II

Key Concepts

Complex Data Types

- Collection of multiple pieces of information
- Mixed types
- Varying visibility
Objects III

Key Concepts

**Construction**
- Objects are constructed rather than assigned
- Construction may be specified through *constructors*
- Assignment through *references*
- Contrast with primitive types

Objects IV

Key Concepts

**Object Design**
- General approach: decompose an entity to the point where it is simple enough to implement or a suitable object already exists
- Keep it simple
- Semantics may dictate design
- Avoid the God-Object anti-pattern

Objects V

Key Concepts

**Examples**
- An animal is a general type
- Dogs, Cats, Snakes, Lizards are all animals
- How can we further develop a taxonomy?
- Should the following be incorporated?
  - Pets?
  - Gender?
- Strict is-a relationship
- Examples: vehicle, automobile, truck, car, boat plane

4 Main Principles

Object Oriented Programming involves 4 main principles
1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Abstraction I

**Definition**
Abstraction is the mechanism by which objects are given their representation and implementation details are hidden.

Abstraction is apparent in many areas:
- Application layers (database, middle-tier, front-end)
- System Layers
- Protocol layers (OSI networking model)

Abstraction II

Already familiar with *Procedural Abstraction*: the implementation of a process is codified into a reusable piece of code inside a function.

Example: a square root function may be implemented using various algorithms.
- Taylor series
- Babylonian method
- Some other interpolation method
Abstraction III

Example: A sorting function
- Selection sort? Quick sort? Other?
- Temporary space?
- How does it swap?

In the end: who cares?
Procedural abstraction relieves us of the need to worry about implementation details

Abstraction IV

- Abstraction in OOP takes this further: methods implementations are abstracted, but so are the internal representations of the objects
- Objects are defined with a representation similar to is meaning (semantics) while hiding implementation details
- Defines the conceptual boundaries of an object, distinguishing between objects
- Allows the client (programmer, user) to deal with objects based on their public interface without worrying about details
- It just works!

Abstraction in languages

- Abstraction of objects is achieved through its publicly available interface
- Interface specifies how you can use the object
- Specifies which methods may be signaled by client code
- May specify a contract
- In conjunction with inheritance: behavior of subtypes may change, be enhanced, etc.

Abstraction Example 1

Date/Time

A date/time stamp object may be defined to represent a particular point in time (with functionality for formatting, comparing and operating with durations, intervals, other date/time instances).

How could such an object be represented?
- A collection of individual numbers (year, month, day, etc.)
- A single numerical value (time from a particular epoch)
- ISO 8601-formatted string (ex: 2012-05-04T12:20:34.000343+01:00)

Abstraction relieves us of the need to worry about these details
- Representation is internal to the object
- Consistency and changes are handled by the object
- We only interact with its interface

Abstraction Example 2

Strings

Most languages support strings, but may use different representations
- A null-terminated character array
- A length-prefixed character array
- Dynamic character array
- ASCII or Unicode?

Representing a string as an object relieves us of the need to worry about these details
- Contrast with procedural language (C): memory management, consistency, expectations must be handled manually
- Instead of worrying about details: just use it!

Example 3

A Graph has vertices (nodes) and edges; may be represented as:
- matrix
- adjacency list
- maps
**Encapsulation I**

**Definition**

*Encapsulation* is a mechanism for restricting access to (some of) an object’s components and for bundling data and methods operating on that data.

Includes:
1. Grouping of data
2. Protection of data
3. Grouping of methods that act on an object’s data

**Encapsulation II**

▶ A form of information hiding
▶ In contrast to abstraction which is implementation hiding
▶ Includes the aspect of grouping data – modularity
▶ Protection of data
▶ Allows restriction to access and control of data
▶ Allows internal behavior to change, evolve, be fixed without affecting client code
▶ Usually breakable (through reflection, exposing in subclasses)

**Encapsulation III**

▶ Enables mutator and accessor methods
  - Changes are made to be intentional
  - *Side-effects* can be controlled and localized
  - Allows validation
▶ Enables objects to be immutable (predictable, thread-safe)
▶ Key idea: abstraction allows a user to access essential information, encapsulation hides the data, they compliment each other

**Encapsulation**

**Best Practice**

▶ Best practice: make everything private, control access through getters/setters
▶ Objects should not just be data containers
▶ Any method that acts on an object’s data should be a member method
▶ Do not break encapsulation by placing related functionality elsewhere

**Encapsulation I**

```java
public class BankAccount {
    private double balance;
    private double apr;
    ...
    // constructors, getters/setters
    ...
    public double getMonthlyEarnings() {
        return this.balance * (this.apr / 12.0);
    }
}
```

1. Grouping of data
2. Protection of data
3. Grouping of functionality that acts on that data

**Encapsulation II**

```java
public class BankUtils {
    public static double getMonthlyEarnings(BankAccount acct) {
        return acct.getBalance() * (acct.getAPR() / 12.0);
    }
}
```

Breaking encapsulation:
Java Example

```java
public class Bird {
    public void move () {
        System.out.println("Moving somehow ...");
        System.out.println("Walking ...");
    }
    @Override
    public void move () {
        this.walk();
    }
}
```

Inheritance

**Definition**

Inheritance is the ability to extend the definition of objects; creating objects from previously created or defined objects.

Most common form: classical inheritance (subclassing)

- subclass/superclass
- derived/base classes
- child/parent classes

Examples:

- A sort method that reorders the original list
- A setter method that modifies an object’s state that can be observed through its public interface
- A subroutine used in a logical expression that makes changes to its parameters
- Any changes to global, static variables, parameters, data I/O, etc.

Side-effects

**Definition**

A function or expression has a side-effect if it modifies some state or has an observable interaction with calling functions or other parts of the program (either globally, to other threads, the outside world, etc.).

Examples:

- A sort method that reorders the original list
- A setter method that modifies an object’s state that can be observed through its public interface
- A subroutine used in a logical expression that makes changes to its parameters
- Any changes to global, static variables, parameters, data I/O, etc.

Aspects of Inheritance

- Allows us to define a hierarchy of related objects
- Subclasses inherit behavior and/or fields from superclasses
- Subclasses can augment, modify, or change the behavior of methods
- Facilitates code reuse
- Generalized common behavior and/or interface can be defined in superclasses
- Subclasses provide specialization (more specificity)
- General properties or behavior is specialized into more concrete implementations
- Super/subclass defines an is-a relationship

Why Encapsulation

Beginners question: why can’t everything just be public?

Without Encapsulation:

- Internal state could be modified with impunity
- Everything essentially becomes global
- Difficult to test/control
- Anti-modular
- No way to enforce rules

Save on typing: utilize IDE macros (Eclipse: Source → generate getters/setters)

Java Example

```java
public class Ostrich extends Bird {
    @Override
    public void fly () {
        System.out.println("Flying ...");
    }
}
```

```java
public class Robin extends Bird {
    @Override
    public void fly () {
        System.out.println("Flying ...");
    }
}
```
**Collections Hierarchy I**

*Figure: Java Collections Library Inheritance structure*

**Collections Hierarchy II**

- A general collection is a data structure that holds stuff
- A set is a collection that holds stuff in an unordered manner; a list is a collection that holds stuff in an ordered manner
- A HashSet is a set that holds stuff in an unordered manner using a hash table
- An ArrayList is a list that holds stuff in an ordered manner using an array, etc.

**Virtual Functions**

A *virtual function* is a function that can be *overridden* in a subclass.

- C++: all functions are non-virtual by default; can be made virtual by use of the keyword `virtual`
- Java: all non-private functions are virtual by default; can be made non-virtual by use of the keyword `final`
  - static methods can be overridden
  - visibility of methods can be more open, but not decreased
  - within a class, access to the superclass’s method is accessible via the `super` keyword

1 but when static methods are invoked, they should be done so via the class name anyway

**How this works: Dynamic Dispatch**

- Upon instantiation, a particular constructor is called
- This creates the object in memory as a *virtual table* (vtable)
- Object has data in memory, but also references to the code for its methods
- When a method is called, the method which the vtable refers to is invoked
- This process is a *run-time* mechanism (since methods can be invoked via a reference to the super class)

**How this works: Dynamic Dispatch**

*Java Example*

```java
Bird b = new Bird();
b.move(); // prints "Moving somehow..."
Robin r = new Robin();
r.move(); // prints "Flying..."
r.fly(); // prints "Flying..."
b = r; // subclass referred to as its superclass
b.move(); // prints "Flying..."
//b.fly(); ---compile error
b = new Ostrich();
b.move(); // prints "Walking..."
```

**Multiple Inheritance**

- An object can inherit from multiple parents, called *multiple inheritance*
- Example: Musician (inherits from Person, Performer)
- Supported by some languages
- Problematic to implement properly: the diamond problem
- Example: Animal, Dog, Cat, “DogCat”?
- Necessary to establish a mechanism to disambiguate properties
**Why Inheritance**

- Motivation: Classification & Taxonomy
- Ubiquitous: sheer volume of data requires classification to organize it
- Hierarchical structure is most natural (though not always applicable)
- Fundamental to communication
- Communication requires a common classification/description

---

**Liskov Substitution Principle**

The Liskov Substitution Principle states: if \( S \) is a (subtype of) \( T \), then any instance of \( T \) should be replaceable by an instance of \( S \) without altering any of the desired properties of the program.

- A principle, not a lemma
- I.e. should be the case
- Not guaranteed
- Certainly possible to violate (breakable)

---

**Beware of Pitfalls**

- Good practice: avoid the yo-yo anti-pattern (keep relations shallow)
- When in doubt: use composition instead!
- Avoid the rectangle problem (or the circle problem):
  - Object types should not be mutable to sub/super types
  - The is-a relationship (should be) an invariant
- **Code Check**: take a look at `unl.cse.rectangle_problem`

---

**Polymorphism**

**Definition**

Polymorphism is the ability to create a variable, method, or an object that has more than one form (type).

- Derived from Greek: “having multiple forms”
- Inspired by biology (organisms may have different phenotypes: observable traits; genetic, behavioral, or consequence there of)
- Allows more abstract, reusable code

---

**Subtype Polymorphism**

- The “default” type of polymorphism
- You can refer to subclass instances as their superclass
- Instance’s methods are called dynamically using late-binding
  - Because subtypes can provide their own method implementation (overrides),
  - There is not enough information at compile time to determine which method to call
  - Example: Bird may a `move()` method; while an Ostrich has a `run()` method
  - **Code Check**: let’s take a look at `unl.cse.oop.inheritance`
- Dynamic Dispatch: compiler is able to create a virtual method table (v-table) with references to the correct method to invoke

---

**Behavioral Subtyping**

Behavioral subtyping is a strict type of polymorphism where:

- Liskov Substitution Principle is strictly enforced
- Much stronger form of polymorphism
- Pre- and Post- conditions are defined & enforced (design-by-contract)
- Preconditions cannot be strengthened in subtypes
- Postconditions cannot be weakened in subtypes
- Invariants are guaranteed
- Subtypes must be pure inheritance (no new methods)
- Not generally used in practice
Method Overriding

- Subclass augments, modifies, or provides a completely different implementation of a method
- Method has the same name and signature
- May support covariant return types
- *Not* the same as (operator) overloading (which is done at compile time)

Ad-hoc Polymorphism I

- Ad-hoc Polymorphism covers a wide variety of as-needed polymorphic behavior.
  - Ad-hoc: done as needed for a particular purpose
  - On-the-fly polymorphism

Method Overloading I

Method Overloading: several different “versions” of a method may be defined

- Methods have the same identifier (name)
- In general, have the same return type
- However, must differ in the type or arity (number of) parameters
- Resolution of which method is called is inferred at compile time based on the arguments at invocation
- May be considered *static dispatch*

Method Overloading II

Some examples in practice:

- The same name can be used for several versions of an absolute value function (rather than different names for each numeric type)
- C: standard math library has three functions with different names (labs, fabs, abs) for three numeric types; Java: 4 methods with the same name, different arguments
- Java: String class has 9 `valueOf(...)` methods to give the String value of every type
- Java: Integer class has 3 different `toString(...)` methods (built-in, conversion, base-conversion)

Operator Overloading I

Operator Overloading: built-in operators’ (+, =, etc.) behavior can be redefined depending on its operands

- Some overloading is already built into languages:
  - String + String may be concatenation
  - Number + Number may be addition
  - Number + String may be conversion-addition or conversion-concatenation
- Some languages allow you to overload an operator
  - Could define + to mean union when used with arrays or graphs
  - Could define + to mean time-addition when used with date-time/interval types

Operator Overloading II

Should be avoided:

- Of limited use (operator-operand combinations can only have one meaning)
- All combinations of operands that are used must be defined
- Coding requires defining a function anyway, why not just use the function?
- Built-in operators have a clear and understood meaning; when applied to user defined types or completely redefined, meaning becomes ambiguous, unclear, open to interpretation
- Example: what does Array + Array mean?
  - Concatenation?
  - Vector addition?
  - Set union?
Type Coercion I

- When operators are overloaded, rules must be defined for how types and behaviors are resolved.
- Types are coerced into other types with specific rules so that the operands become compatible.
- Type Coercion: one type is mapped to another compatible type.
- Type Promotion: mapping a type to a super-type for compatibility.
- Rules may be defined by the language or incompatible types may be disallowed entirely.
- Such type casting may require an explicit instruction by the user.

Type Coercion II

Examples:
- Casting numbers to strings in order to concatenate.
- Integers are cast to floats in order to add/subtract/multiply.
- Floats may be down-casted to integers when the assignment operator is used.

Type Coercion III

General types of conversions:
- Covariant: converting from a specialized type to a generalized type (subclass to superclass, Ostrich to Bird).
- Contravariant: converting from a generalized type to a specialized type (super to subclass, Bird to Ostrich).
- Invariant: no conversion is possible.

Type Coercion IV

In general, only covariance is “safe” since subtypes define an is-a relationship.

More formally, if \( A \leq B \), then a transformation \( f \) is:

- Covariant if \( f(A) \leq f(B) \)
- Contravariant if \( f(B) \leq f(A) \)
- Invariant otherwise

Type Coercion V

- covariant if \( f(A) \leq f(B) \)
- contravariant if \( f(B) \leq f(A) \)
- invariant otherwise

Parametric Polymorphism I

Parametric polymorphism allow us to define methods or classes that have strong static type-safety, but that can be applied to many different types.

Allows us to write generic code that depends only on some aspect(s) of all types that are applicable to the class/method without depending on the actual type.

A method or class is given a generic type, a parameter (parameterized).

The specific type is specified by the client code or deduced at runtime.

The parameterized code itself is generic enough that it applies to all types (does not use any type-specific functionality).
Parametric Polymorphism II

Examples:
▶ A generic “container” class (list or set) can be parameterized to hold a specific type (defined at instantiation)
▶ A generic print method that prints elements in an array or collection in a tidy format
▶ A generic `getMax()` method can be used for all numeric types
▶ A single, generic sorting method can be defined and used for any type that is “comparable” in some manner
▶ Example: a parameterized summation method would need to bound the types to numerical types

Bounded Parametric Polymorphism

▶ Some information may be needed by the implementation
▶ Code needs to know that such objects are numeric types or are “comparable”
▶ Bounding the parameterized type guarantees a minimum set of information implied by the type
▶ Bounding the types provides a minimal guarantee of the interface they provide (which can then be used!)

Open Recursion

▶ Objects are defined, but instances are constructed
▶ An instance may need to refer to its own methods or fields
▶ No variable reference at compile time
▶ Special keyword(s) defined (this or self)
▶ Bound to a specific object (instance) via late-binding

Open Recursion

Java keywords

▶ this can be used inside an object to refer to itself
▶ super can be used to access methods/variables of a super class
▶ this(...) can be used to call an object’s constructors
▶ super(...) can be used to call an object’s superclass’s constructor

Association, Aggregation, Composition I

Association: “a” relation between two classes
▶ An object knows about another object
▶ Can call its methods, interact with it
▶ Most general object-object relationship

Association, Aggregation, Composition II

Aggregation: “the” relation between two classes
▶ One class “has” another (or usually a collection of another) class
▶ Usually by reference
▶ Relationship does not imply ownership
▶ Other instances may also “have” the object
▶ Destruction issues
▶ Example: University – Departments – Faculty
**Association, Aggregation, Composition III**

**Composition:** One object owns another
- Upon destruction of owner, owned object(s) are also destroyed
- Should be private: No one else should depend on them
- Composition in Java:
  - Less important in Java as garbage collection is automatic
  - If multiple owners, owned objects won’t get destroyed

**Other Issues**

**Mixins**

A *mixin* is an object that provides functionality to be inherited or reused
- Object’s functionality is *mixed in* to another object
- Mixin itself is abstract and cannot be instantiated
- Not a form of specialization
- A means to collect functionality into an object
- Java: not directly supported
  - Interfaces provide specification, but not behavior
  - Abstract classes cannot provide functionality, but must be inherited and cannot be multiply inherited
  - Preferred solution: composition

**Association, Aggregation, Composition IV**

**Composition vs Inheritance**
- Inheritance is good when hierarchy is rigid and well-established; design is mature
- Inheritance is fragile as changes up-stream (super classes) will affect everything downstream
- Composition is more extensible; does not lock you into a hierarchy that would require refactoring
- Composition is more flexible, easier to change and changes have less of an effect
- Use composition when in doubt

**Other Issues**

**Copy Constructors**

- It’s often convenient to have a way to create different copies of objects
- Usual to implement a *copy constructor*: a constructor that takes an instance of its own class
- Pitfall: the copy should be a *deep copy* and not just a copy of references
- If not a deep copy, changes to the “shared” resources will be realized by both copies

**SOLID I**

**SOLID Principles (Features, Martin):**
- **SRP:** Single Responsibility Principle – a class should have only a single responsibility (i.e. only one potential change in the software’s specification should be able to affect the specification of the class)
- **OCP:** Open/Closed Principle – software entities should be open for extension, but closed for modification
- **LSP:** Liskov Substitution Principle – objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program
- **ISP:** Interface Segregation Principle – many client-specific interfaces are better than one general-purpose interface
- **DIP:** Dependency Inversion Principle – one should depend upon Abstractions. Do not depend upon concretions

**Duck Typing**

- OOP languages without classes use dynamic typing (JavaScript, Python)
- Methods and properties define a type rather than semantics
- If it walks like a duck (waddle()), talks like a duck (quack()) then it’s a Duck
Summary Example I

Maps

Maps are key-value data structures
Abstraction: get and put methods define how to use this data structure without worrying about its implementation details

```java
Map<String, Student> m = new HashMap<String, Student>();
...
String nuid = "12345678";
Student s = new Student(...);
m.put(nuid, s);
...
String someNuid = "87654321";
Student joe = m.get(someNuid);
```

Summary Example II

Maps

- Encapsulation: maps could be implemented using a tree or hashtable; could use open or closed addressing; representation details are irrelevant and hidden to us.
- Inheritance: `HashMap` is a map that uses a hash table (while a `TreeMap` is a map that uses a tree).
- Polymorphism: through parametric polymorphism, we can map specific types to other specific types.
- Java note: a `Map` is not `Collection` (see why: http://docs.oracle.com/javase/1.4.2/docs/guide/collections/designfaq.html#14)

Summary Example III

Maps

Map data structures in Java Collections library

```
Map
   
   AbstractMap
   |
   
   HashMap
   |
   
   TreeMap
   |

   LinkedHashMap
```

Software Engineering & Design Books I

Figure: Design Patterns: Elements of Reusable Object-Oriented Software (1994) by Gamma et al.

Software Engineering & Design Books II

Figure: Code Complete (1993) by Steve McConnell

Software Engineering & Design Books III

Figure: The Mythical Man-Month (1975) by Frederick Brooks Jr.
Figure: Refactoring: Improving the Design of Existing Code (1999) by Martin Fowler