# Lecture 4

CSE 58x
Winter 2000
Mark Morrissey

## Polymorphism

A concept first identified by Christopher Strachey (1967) and developed by Hindley and Milner, allowing types such as list of anything. E.g. in Haskell:

```
length :: [a] -> Int
```

is a function which operates on a list of objects of any type, a (a is a type variable). This is known as parametric polymorphism. Polymorphic typing allows strong type checking as well as generic functions. ML in 1976 was the first language with polymorphic typing.

Ad-hoc polymorphism (*overloading*) is the ability to use the same syntax for objects of different types

e.g. "+" for addition of reals and integers or "-" for unary negation or diadic subtraction.

Parametric polymorphism allows the same object code for a function to handle arguments of many types

Overloading only reuses syntax and requires different code to handle different types.

In OOP, the term is used to describe variables which may refer at run-time to objects of different classes.

Source: *The Free On-line Dictionary of Computing, © 1993-2000 Denis Howe*



# Polymorphism Defined

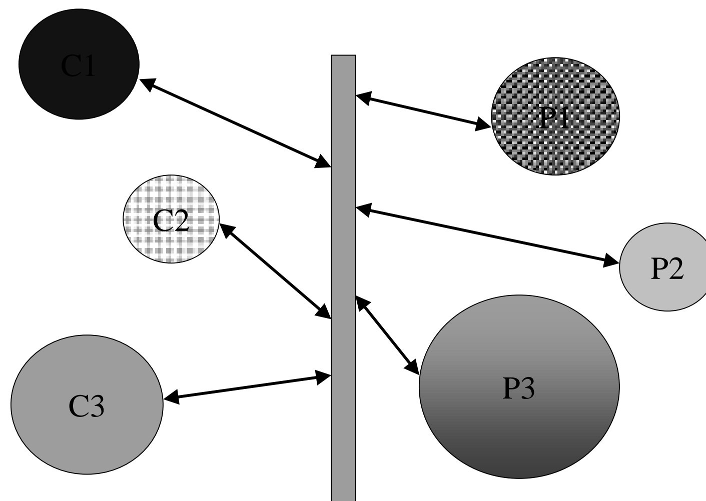
For our purposes:

- » The ability of something to appear in multiple forms, depending on context
- » The ability of different things to appear to be the same in a certain context
- » Refinement: “something” is not restricted to be only what it appears to be

# Substitutability

- The ability of one component to be substituted for another component under certain conditions
- Possible because contractually specified interfaces decouple clients from providers (and vice-versa)
- Difficult to obtain, in general
- Requires a carefully crafted interface contract






How can one interface relate all these different clients and providers

# Interface Crafting

- Require too little or too much, interface loses all value
  - "give me data and I will return data" interface
    - Is this a useful interface?
    - Could you implement a component that is what the client expects?
  - Too much and usage becomes either highly specializes or ignored
- The "trick" is to require no more that is essential and guarantee not more than is reasonable
- Allows both sides to overfill the contract
  - Client can establish more than is required
  - Provider can provide more service than is guaranteed



# Relaxing the Contract

- Consider the TextModel interface from the text
- Provider relaxation of pre-conditions
  - The general contract requires text to be inserted within the current range of text
  - A provider could relax this condition and allow for prepending or appending text to the current range
- Client relaxation of post-conditions:
  - Assume the above provider
  - If the client only ever uses the provider to append text, it is requiring less of the provider

## Invariants

- Pre-condition: text.length() < text.max()
- Post-condition: text.length() < text.max()
- What does this say about the relationship while the provider is executing?
- What if it is necessary for the relationship to hold continuously?
- Invariants are hard for contracts based on pre- and post-conditions alone
- Histories, with their idea of legal transition states, are more suited



## Implications v. Equivalences

- Implications are key to contract relaxation
- If the interface makes guarantees, it is implied that any provider implementing that interface will meet those guarantees
- This implies that as long as the client's expectations do not exceed the guarantees, the client may safely use the provider
- Does not require that the client expect all that is guaranteed
- Implies that two providers that both meet the guarantees do not have to only provide equivalent services

# Substitutability

- If a provider satisfies the same contract as another, the former is said to be *substitutable* for the latter (p. 76)
- For example, if the interface expects an array of Integer and returns a sorted array of Integers, an object that accepts an array and sorts the array depending on type is substitutable provided it meets the guarantees for the interface contract



# Types, Subtypes and Type Checking

- Specifying all contract conditions can be difficult
- Checking them formally can be even harder
- However, some can be checked and whenever we can, they should be checked as early as possible to avoid faults
- Checking mantra:
  - compile-time checking is better than load-time checking is better than runtime checking is better than no checking at all
- One of the largest sources of errors are memory errors. Type checking is a strong ally here

- Type checking ensures that all access to memory is type compatible
- Combined with hardware protection mechanism helps to eliminate nearly all memory errors
- Java and Smalltalk are languages that support type checking
- C and C++ do not



## What is a Type?

- A type names all operations of an interface and the number and types of parameters and the type of returned values of each of the operations
- The types of input and in-out parameters form a part of the pre-conditions
- The types of output and in-out parameters form a part of the post-conditions
- Types typically do not specify all pre- and post-conditions
- Types are therefore a simplified contract
- This implies that type correct programs can still violate the contract

## Type Substitution

- Under what conditions can an object implementing one type be substituted in a context expecting another type?
- Answer follows naturally from implications and relaxation
- Clients do not expect a certain type, but rather expect contract fulfillment
- A subtype based on interface inheritance, where the base interface is extended, can promise more but still meet the base contract
- Thus subtypes are substitutable for base types
- Implementation is still able to violate the contract



## Covariance

- A provider may establish more than is required by a contract, that is, subtype
- $\Rightarrow$ a subtype interface can replace the types of output parameters and return values with something more specific, that is a subtype
- Covariance: varying the types of output parameters and return values <u>in the same direction</u> as the types of the containing interface
- That is, postcondition types and interface types varying in the same direction is called covariance

# Contravariance

- A provider may expect less than is guaranteed by a contract, that is, supertype
- $\Rightarrow$ a subtype interface could replace types of input parameters by a more general type, that is a supertype
- Contravariance: varying the types of input parameters <u>in the opposite direction</u> of the types of the containing interface
- That is, precondition types and interface types varying in the opposite direction is called contravariance



# Invariance

- in-out parameters are simultaneously in the pre- and post-conditions and therefor cannot be covariant or contravariant
- Instead, they must remain the same type
- This is called the invariance of the in-out parameters

## Types, interfaces and components

- Client-Provider independence relies on a self-standing interface that is fully and explicitly typed
- Otherwise, type checking really does no good
- Enables independent development of clients and providers
- Avoids the component programming version of the “chicken and the egg” problem



## Subtype Inference

- Some languages do not have types or do not require explicit type specification
- Instead, if an interface of one type contains the operations of another interface of another type - and all operations are appropriately typed, then the former may be *inferred* to be a subtype of the latter
- May seem far fetched, but can happen, especially for base types, e.g. Event and Property

## Subtype Inference

- The problem here is that the type is a simplified, not complete contract
- There is no way of knowing if this inferred, or structural, subtype respects the base contract
- The way to avoid this is to require the programmer to specify all legal subtype relationships and disallow inferred subtypes



## Independent Extensibility

- A key property of component systems
- Definition: a system is independently extensible if it is extensible and if independently developed extensions can be combined
- Components can be independently developed, acquired, and deployed
- As such, the system is open and not subject to global analysis
- Components are analyzed and tested against individual specifications

## Independent Extensibility

- Browsers and microkernel architectures are examples of independently extensible systems
- Browsers are extendible via plug-ins
- Performance becomes critical in extensible architectures
- One approach to performance is to carefully pick the granularity of the component so that most interactions stay within the component boundaries
  - Certain boundary crossings, such as those involving context switches, can greatly affect performance



## Module Safety

- Module safety is required to that only those operations exposed by interfaces are accessible to clients
- Can’t rely on the type system since any properly formed invocation would be type safe
- Should just any well-formed request be able to shut down the system?
- A system may have a service to check the validity of an invocation for a particular operation or interface
  - e.g. Java has the security manager
- Moves the responsibility for checking to the provider

## Safety, Security and Trust

- Type safety, module safety and the absence of memory errors do not imply trustworthiness
  - In fact, trust is variable
- The idea of proving something to be trustworthy at all stage, while a laudable goal, is practically impossible
- So to obtain trust, we mostly rely on reducing the unknown to the known and trusted
- But how does something become known and trusted?



## Dimensions of Extensibility

- Each feature that is open for separate extension is called a dimension
- Overlapping extensions (non-orthogonal) are non-optimal
  - Choices can lead to non-substitutability
  - So are gaps in the extension space
- Bottleneck Interfaces
- Singleton Configurations
- Parallel Extensions
- Orthogonal Extensions
- Recursive Extensions

# Bottleneck Interfaces

- A self-standing contract that couples families of independent extensions
- Cannot itself be extended, obviously
- Example:

» Controls and Containers are independently extensible

  » That is, new controls and containers, as well as new capabilities, can be independently developed and deployed

» Interfaces that allow Controls and Containers to collaborate are considered Bottleneck Interfaces



# Singleton Configurations

- Some component frameworks allow for extensibility, but require that in some dimensions a singleton configuration be used
- Classic example is a security manager
- Obviously, having multiple independent security managers is not desirable
- Solution is to require that there be at most one in any configuration

# Extensions

- Parallel - components that extend along the same dimension
  - Resource contention
  - Arbitration - e.g. separate controls competing for keyboard focus
- Orthogonal - components that extend orthogonally
- Recursive - create hierarchical extensions
  - framework within a framework
  - e.g. A resource management framework that contains a compound document framework



# Evolution v. Immutability

- A contract - interface + specification - is a point of mediation between clients and providers
- All clients and providers are independent of each other
- The only rendezvous point for service is the contract
- This requires that the contract not be subject to modifications once it has been published
  - Unless, of course, you have control over all clients and providers
  - Possible in a component-based corporate solution
  - Not possible in the grand world of commodity components
- But contracts must evolve, so what do you do?

## Syntactic v. Semantic Change

- Syntactic - interface is changed
- Semantic - specification is changed
- When contract change occurs, problems can result
  - Typically called the Fragile Base Class (FBC) problem
- Change is possible if all parties agree
- What if you can't locate all parties
  - If you publish the specification, ala COM, Beans, and CORBA, you have this problem
- Versioning and immutability are possibilities here as is contract expiry



## Contract Expiry

- Basically, you say "this contract only good until this time"
  - Sort of like the expiration date on milk cartons
- Many possible problems with this
  - Suddenly, clients and/or providers may stop working
  - Who's to say that any really pays attention to expiry
  - End users get a nasty surprise
  - What about contract renegotiation
- Some plusses
  - Legacy support goes away cleanly - provided the expiry is well known
  - Evolution can be very clean