Improve Your SystemVerilog OOP Skills

By Learning Principles and Patterns

Jason Sprott – Verilab

Email: jason.sprott@verilab.com

Experts in SV Learn Design Patterns

Bob "Big Dog" Coder

Smart Engineer

Agenda

- The OO Basics Are Not Enough
- OO Principles
- Abstract Factory Pattern example
- Strategy Pattern example
- Composite Pattern example

The OO Basics Are Not Enough

OO Basics

Abstraction

Encapsulation

Polymorphism

Inheritance

- Essential to learning, but …
- Doesn't tell us <u>how</u> to build good OO programs
- We need to know more

Guiding Principles of OOP

OO Principles

Classes should be open for extension but closed for modification

Subclasses should be substitutable for their base classes

Depend on abstractions. Do not depend on concrete classes

Encapsulate what varies

Favour composition over inheritance

Loosely coupled designs between interacting objects

…

- Guide us in how to use our OO basics
- Aim to be principles of <u>good</u> programming
- Apply to SystemVerilog testbench design

Guiding Principles of OOP

OO Principles

Classes should be open for extension but closed for modification

Subclasses should be substitutable for their base classes

Depend on abstractions. Do not depend on concrete classes

Encapsulate what varies

Favour composition over inheritance

Loosely coupled designs between interacting objects

…

- AKA: Open-closed Principle (OCP)
- Maybe the most important of all
- Change what modules do, without modifying them

Guiding Principles of OOP

OO Principles

Classes should be open for extension but closed for modification

Subclasses should be substitutable for their base classes

Depend on abstractions. Do not depend on concrete classes

Encapsulate what varies

Favour composition over inheritance

Loosely coupled designs between interacting objects

…

- AKA: Liskov's Substitution Principle (LSP)
- Contracts must be honoured
- Surprisingly often violated
- Violations can lie dormant for some time

Guiding Principles of OOP

OO Principles

Classes should be open for extension but closed for modification

Subclasses should be substitutable for their base classes

Depend on abstractions. Do not depend

Encaps

Favour inherit

Loosely coupled designs between interacting objects

…

By not passing ‘b’ to the super class, existing code might not work the same anymore.

```
class BaseClass;
   Channel a, b;
   function new(Channel a, Channel b);
      this.a = a;
      this.b = b;
   end function
   virtual function Channel getChA()...
   virtual function Channel getChB()...
endclass
```

```
class SubClass extends BaseClass;
   function new(Channel a, Channel b);
      super.new(a, null);
   end function
endclass
```

```
if(s.getChB == ChY) ... // broken now
```

Guiding Principles of OOP

OO Principles

Classes should be open for extension but closed for modification

Subclasses should be substitutable for their base classes

Depend on abstractions. Do not depend on concrete classes

Encapsulate what varies

Favour composition over inheritance

Loosely coupled designs between interacting objects

…

- AKA: Dependency Inversion Principle (DIP)
- Dependencies should target an abstract interface
- Concrete things change a lot, interfaces less so
- When you see `new` we are talking concrete

Guiding Principles of OOP

OO Principles

Classes should be open for extension but closed for modification

Subclasses should be substitutable for their base classes

Depend on abstractions. Do not depend on concrete classes

Encapsulate what varies

Favour composition over inheritance

Loosely coupled designs between interacting objects

…

- Looks for behaviours that (may) change, e.g. different algorithms
- Encapsulate what changes in a class
- Aims to allow changes to be made without affecting dependent code

Design Patterns – What Are They?

> “Each pattern describes a problems which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the solution again a million times over, without ever doing it the same twice”

Christopher Alexander (Architect),
A Pattern Language: Towns, Buildings, Construction, 1977

Design Patterns

- Patterns tell us how to structure classes and objects to solve certain problems.
- We need to fit that to our application and programming language
- Embody OO Principles
- Show you how to write code with good OO design qualities
- Most patterns address change

Design Patterns And Types

Creational

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Structural

- Facade
- Adapter
- Bridge
- Decorator
- Composite
- Flyweight
- Proxy

Behavioural

- Iterator
- Command
- Strategy
- Chain of responsibility
- Mediator
- and more …

(11 patterns)

Has someone already got a solution to your OO problem?

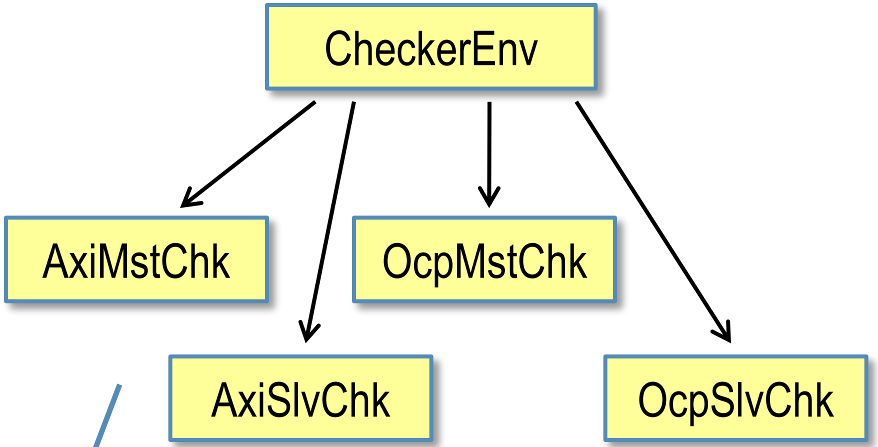
Abstract Factory Pattern (Creational)

Problem: I want to instantiate different types of checkers, depending on the type of bus being used. I don't want my environment change when new bus types are added.

Abstract Factory

Provides and interface for creating families of related or dependent objects without specifying their concrete classes

Abstract Factory Pattern

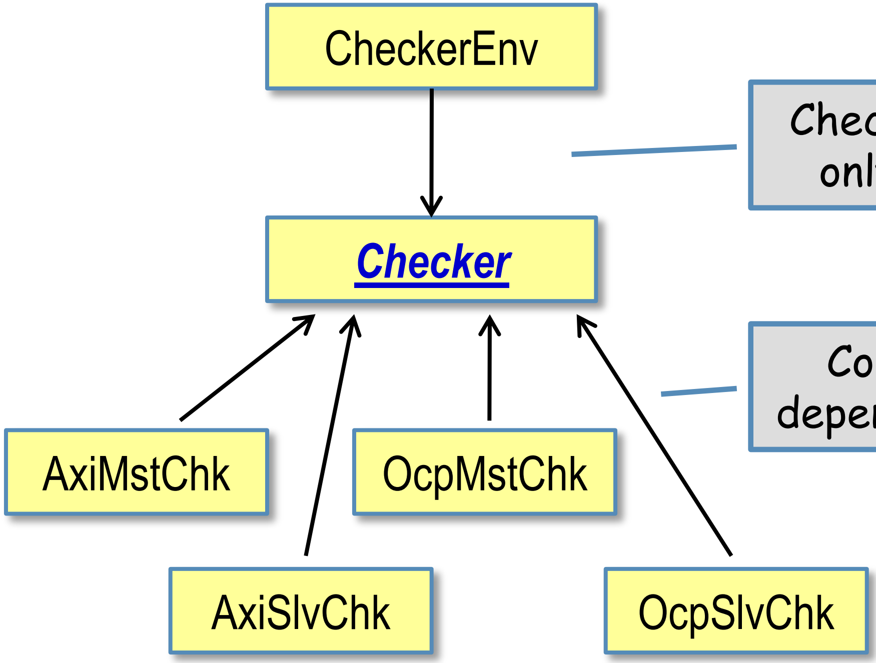


Depends on all protocol checker objects, because it's creating them directly

Any changes to the concrete implementation of these could affect CheckerEnv

Every new kind of checker added creates another dependency for CheckerEnv

Abstract Factory Pattern – DIP

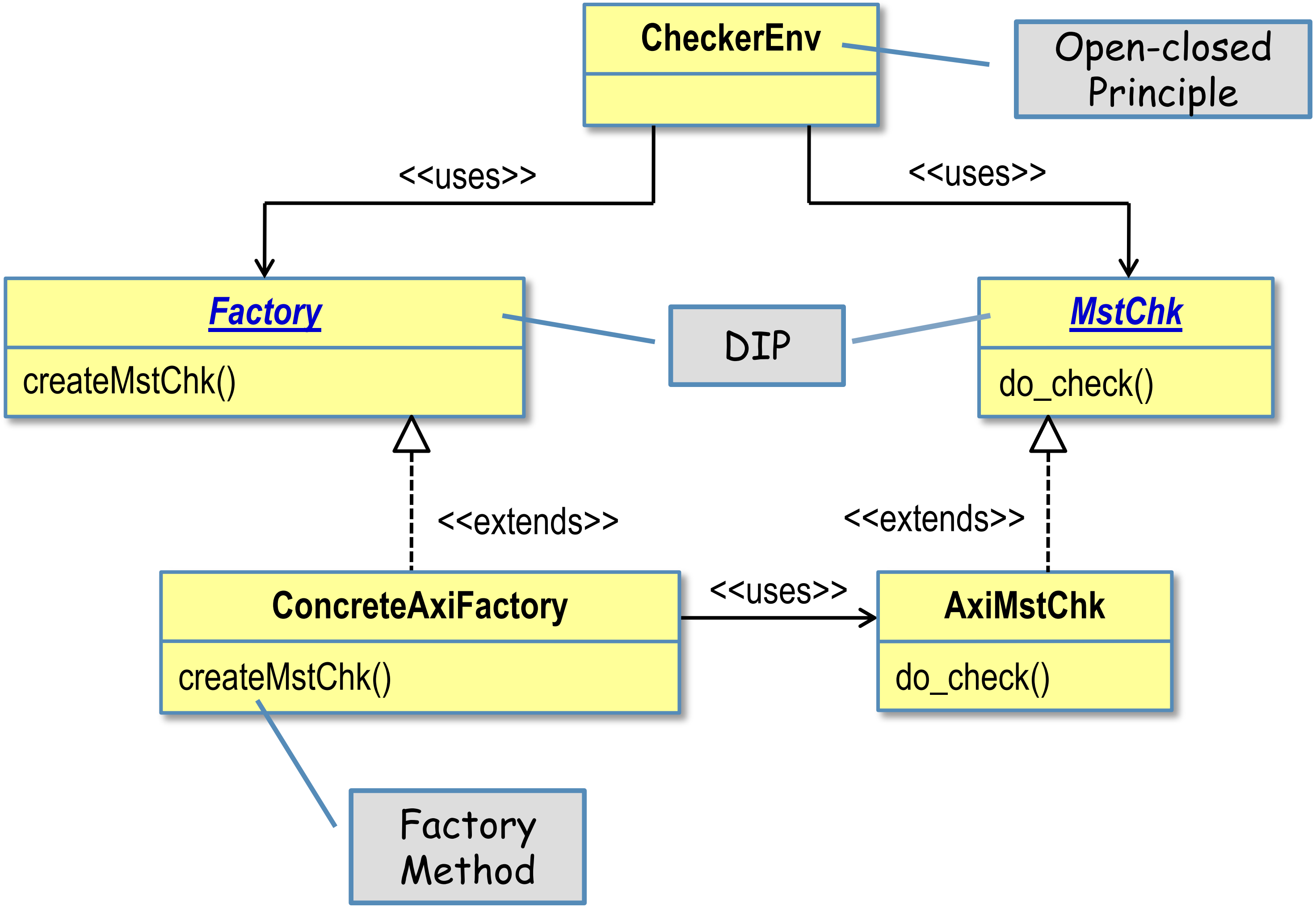


CheckerEnv now depends only on an abstraction

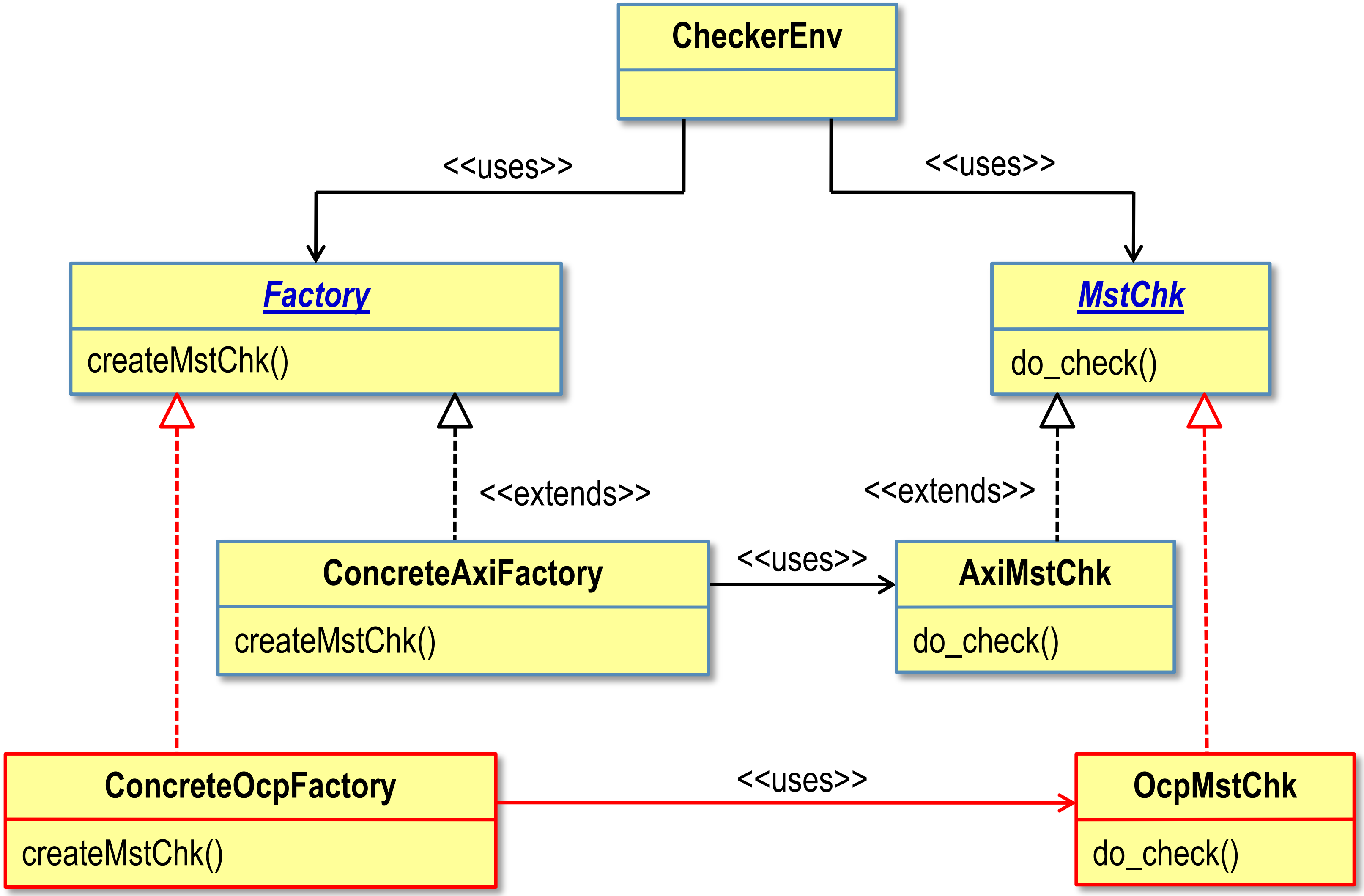
Concrete classes now depend on the abstraction

Dependency Inversion Principle
Depend on abstractions not concrete classes

Abstract Factory Pattern



Abstract Factory Pattern



Abstract Factory Pattern

```
class ConcreteAxiFactory extends Factory;
   ...
   function MstChk createMstChk();
      AxiMstChk p;
      p = new(<AXI specific args>);
      return(p);
   endfunction
   ...
endclass
```

```
class ConcreteOcpFactory extends Factory;
   ...
   function MstChk createMstChk();
      OcpMstChk p;
      p = new(<OCP specific args>);
      return(p);
   endfunction
   ...
endclass
```

The factory knows how to create its own concrete products. If the way to do this changes, the client is protected from that

Abstract Factory Pattern

```
class CheckerEnv;
   Factory checker_factory;

   function new(Factory f);
      checker_factory = f;
   endfunction

   function start_checking_stuff();
      MstChk my_chk;
      my_chk = checker_factory.createMstChk();
      my_chk.do_check(); // do something
   endfunction

endclass
```

CheckerEnv uses abstract types – it is generic code.

Remember DIP

Whoever uses the CheckerEnv class ...

```
   CheckerEnv my_env;
   ConcreteOcpFactory ocp_factory = new;
   my_env = new(ocp_factory);
   my_env.start_checking_stuff();
```

When using CheckerEnv we just pass it the type of factory we want it to use

We can create new factory types whenever we like

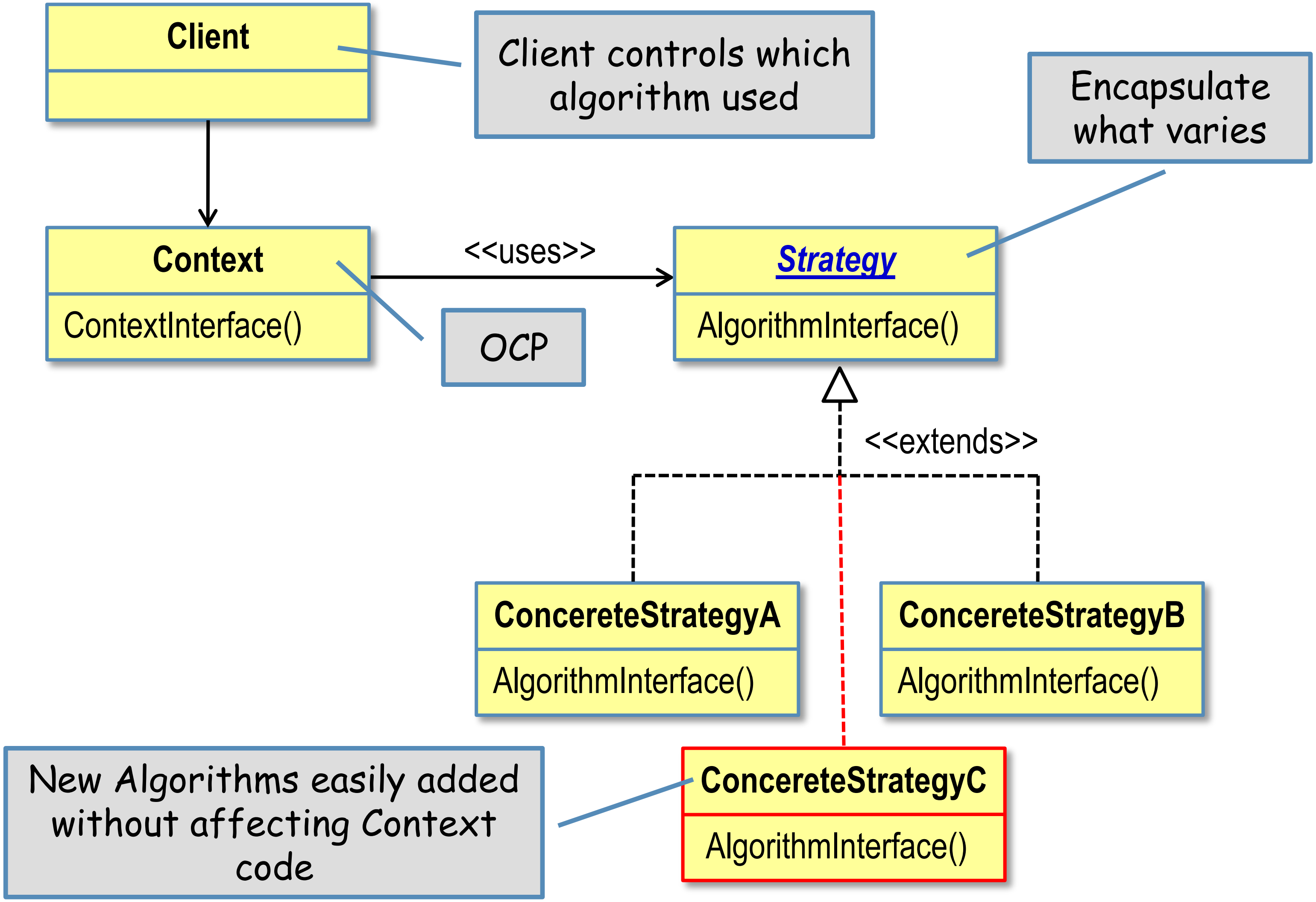
Strategy Pattern (Behavioural)

Problem: I have code that uses an algorithm that can change or new ones can be added. I want to allow the algorithm to change without breaking my code.

Strategy

Defines a family of algorithms, encapsulates each one and makes them interchangeable. Strategy lets the algorithm vary independently from the client using it.

Strategy Pattern



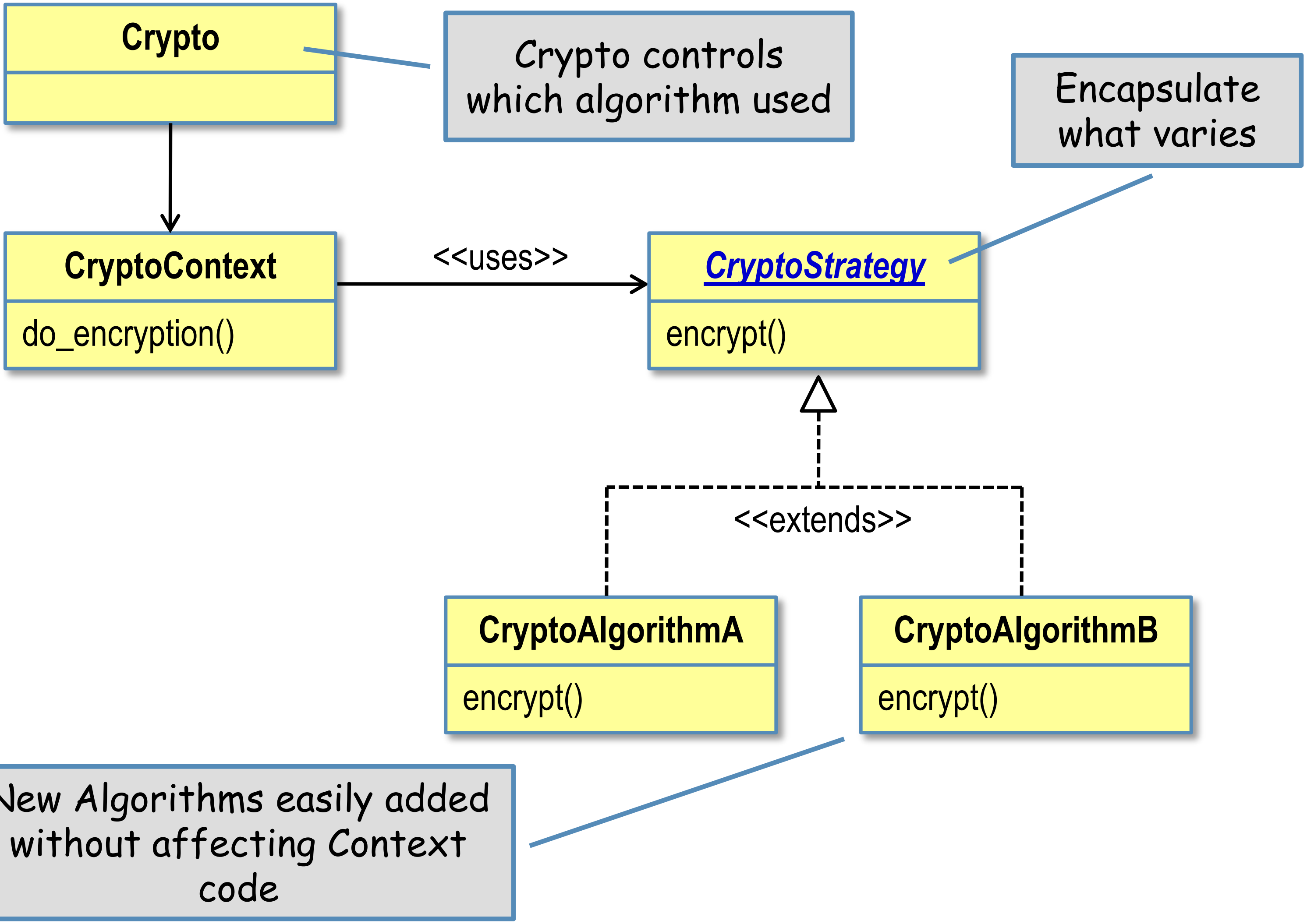
Client controls which algorithm used

Encapsulate what varies

OCP

New Algorithms easily added without affecting Context code

Strategy Pattern



Crypto controls which algorithm used

Encapsulate what varies

New Algorithms easily added without affecting Context code

Strategy Pattern

```
virtual class CryptoStrategy;
   pure virtual function MyData encrypt(MyData d);
endclass
```

Abstract interface capturing behaviour that varies

```
class CryptoAlgorithmA extends CryptoStrategy;
   virtual function MyData encrypt(MyData d);
      // some algorithm
   endfunction
endclass
```

Concrete implementation of the algorithm.

```
class CryptoContext;
   CryptoStrategy my_algorithm;
   function new(CryptoStrategy s);
      my_algorithm = s;
   endfunction
   function MyData do_encryption(MyData d);
      return (my_algorithm.encrypt(d));
   endfunction
endclass
```

Generic code that can handle any algorithm. Only depends on the abstract interface

In this case algorithm to use is passed on constructor

Executes the algorithm

Strategy Pattern

```
virtual class CryptoStrategy;
```

```
class CryptoAlgorithmA;
```

```
class CryptoContext;
```

```
// *** Client code ***

// instance of crypto container
CryptoContext    my_crypto;

// create desired algorithm
CryptoAlgorithmA cryptoA = new;

// pass in the algorithm to be
// used as an argument (or similar)
my_crypto = new(cryptoA);

// Do the encryption
foo = my_crypto.do_encryption(data);
```

Instance of the algorithm we want to use

Program the generic context code to use it

Call to do encryption just does the right thing

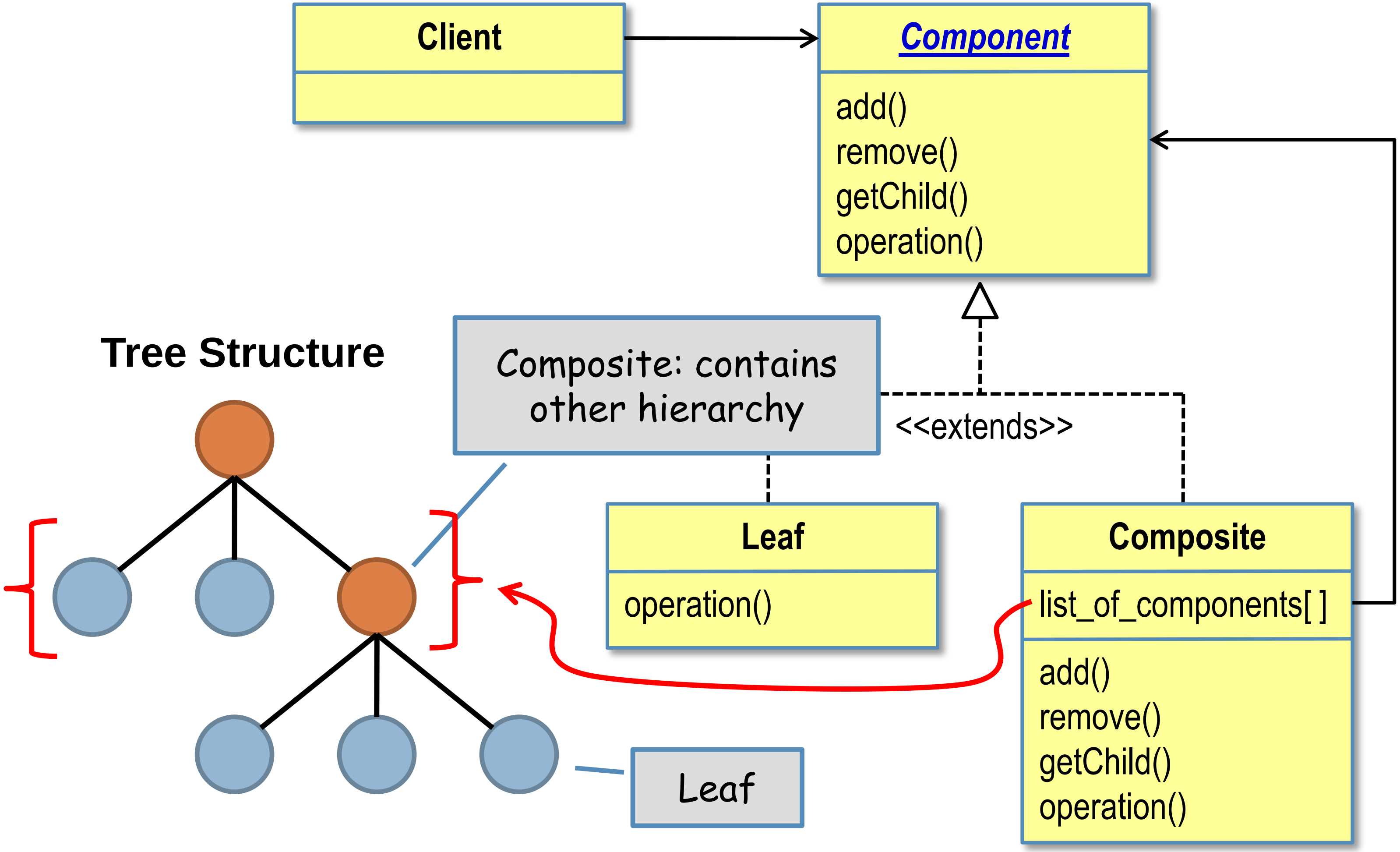
Composite Pattern (Structural)

Problem: I want to build a tree structure, where objects can be leafs or other nodes to further hierarchy.

Composite

Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects or compositions of objects uniformly

Composite Pattern



Composite Pattern

```
virtual class BusComponent;
   virtual function void add(BusComponent c);
      // default behaviour(error?)
   endfunction
   virtual function void remove(BusComponent c);
      // default behaviour (error?)
   endfunction

   pure virtual function void enable(BusComponent c);
   pure virtual function void disable(BusComponent c);

endclass
```

Interface for node or leaf. Some methods not applicable in each case. We could add default behaviour.

Operations interface

Composite Pattern

```
virtual class BusComponent;
class BusNode extends BusComponent;
   protected BusComponent node_list[BusComponent];
   virtual function void add(BusComponent c);
      node_list[c] = c; // use ref as key
   endfunction
   virtual function void remove(BusComponent c);
      node_list.delete(c);
   endfunction
   virtual function void enable();
      // iterate sub nodes and enable
   endfunction
   virtual function void disable();
      // iterate sub nodes and disable
   endfunction
   ...
endclass
```

Implementation to manage hierarchy for this node

The user doesn't need to care if this is a node or leaf. The interface is the same

Composite Pattern

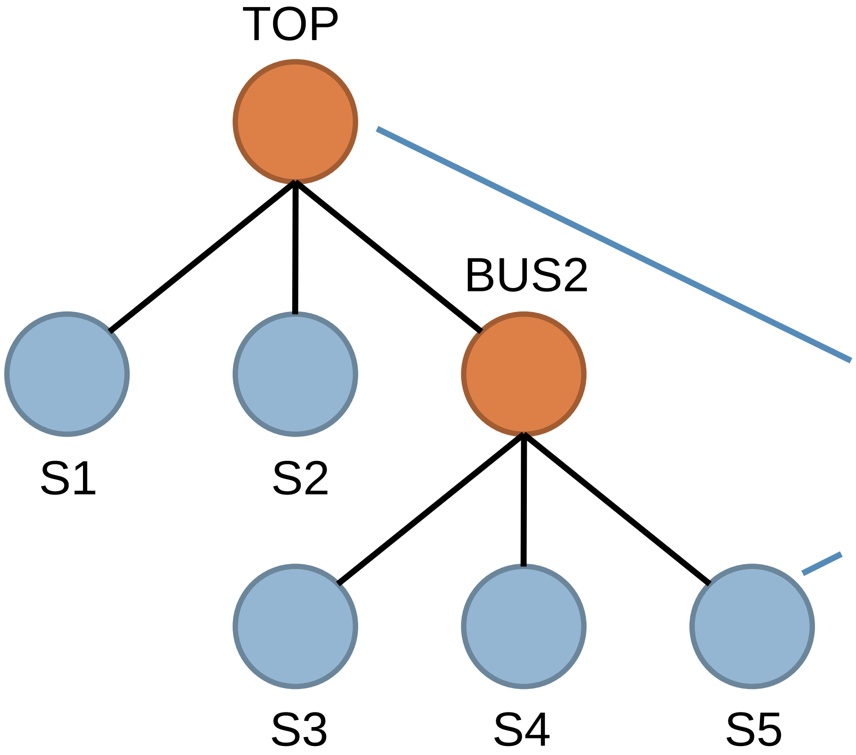
```
virtual class BusComponent;
```

```
class BusNode;
```

```
class BusLeafNode extends BusComponent;
   virtual function void enable();
      // enables a leaf node master or slave
   endfunction
   virtual function void disable();
      // disables a leaf node master or slave
   endfunction
   ...
endclass
```

Functionality for leaf nodes.

Composite Pattern



```
class BusControl;
   function void disable(BusComponent c);
      c.disable();
   endfunction
endclass
...
BusControl bus_ctrl = new;
```

```
// start with instance TOP (BusNode)
bus_ctrl.disable(TOP);
```

```
// start with instance S5 (BusLeafNode)
bus_ctrl.disable(S5);
```

BusControl can treat nodes and leafs the same

Summary

- Learning Principles and Patterns improves your OOP skills

- Patterns give us a useful common language when talking about solutions

- Someone might already have solved the OO problem you are working on elegantly

Bibliography

- Design Patterns: Elements of Reusable Object-Oriented Software By Gamma, Helm, Johnson, Vlissides
- Head First Design Patterns By Eric Freeman & Elizabeth Freeman
- Design Patterns Explained Second Edition By Alan Shalloway; James R. Trott
- Design Patterns For Dummies By Steve Holzner
- Article: Design Principles and Design Patterns, Robert C. Martin
  http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
- The Object-Oriented Thought Process, Third Edition By Matt Weisfeld
- SystemVerilog for Verification Second Edition By Chris Spear
- OVM Class Reference Version 2.0
- Paper: Verification Patterns in Addition to RVM By Cavanagh, Sine Warner (Sun Microsystems), SNUG2008 San Jose

verilab

Improve Your SystemVerilog OOP Skills

By Learning Principles and Patterns

Jason Sprott – Verilab

Email: jason.sprott@verilab.com