CMPT 373
Software Development Methods

Design

Nick Sumner
wsumner@sfu.ca

What is design?

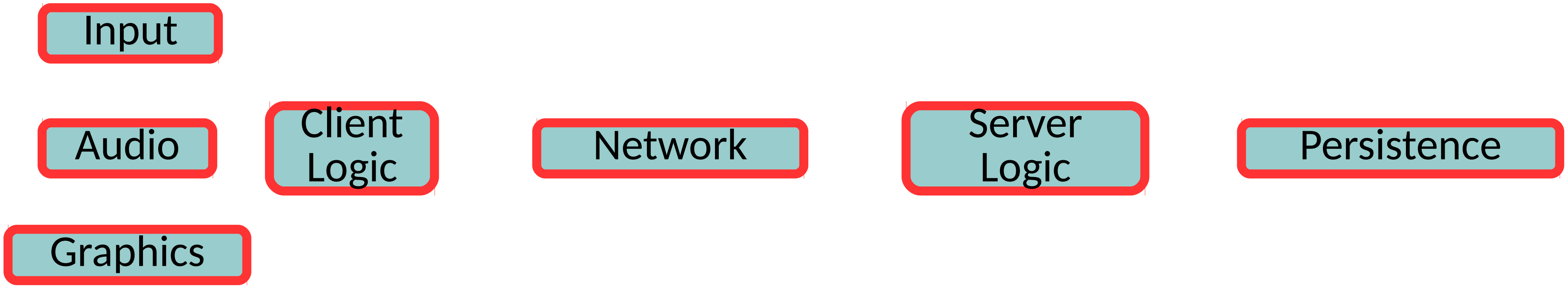
- Not referring to UX (even though it’s important)

What is design?

- Not referring to UX (even though it’s important)
- Includes many things:

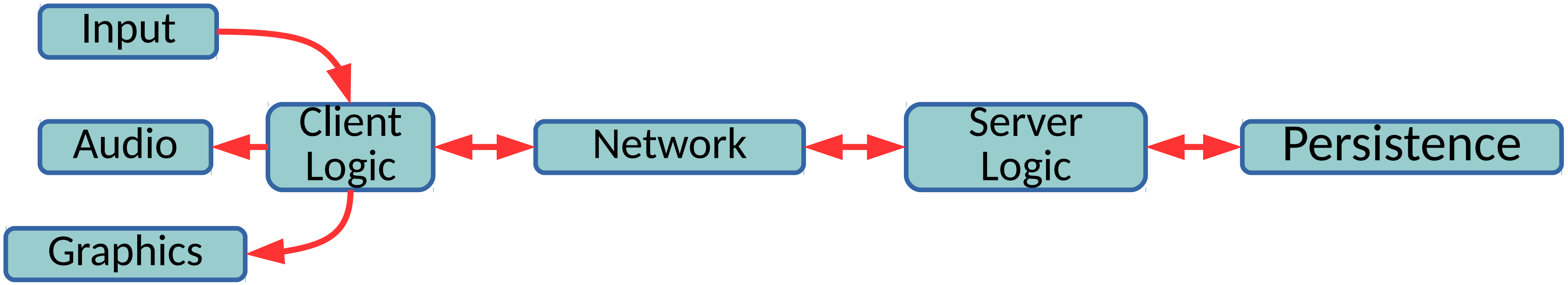
What is design?

- Not referring to UX (even though it's important)
- Includes many things:
  - The components of the system



What is design?

- Not referring to UX (even though it's important)
- Includes many things:
  - The components of the system
  - How they interact



What is design?

- Not referring to UX (even though it's important)
- Includes many things:
  - The components of the system
  - How they interact

} ?

What is design?

- Not referring to UX (even though it's important)
- Includes many things:
  - The components of the system
  - How they interact

} *architecture*

What is design?

- Not referring to UX (even though it's important)
- Includes many things:
  - The components of the system
  - How they interact
  - The interfaces & abstractions they expose (or hide!)

What is design?

- Not referring to UX (even though it's important)
- Includes many things:
  - The components of the system
  - How they interact
  - The interfaces & abstractions they expose (or hide!)

What is an abstraction?

What is design?

- Not referring to UX (even though it’s important)
- Includes many things:
  - The components of the system
  - How they interact
  - The interfaces & abstractions they expose (or hide!)

```
Server server{port};
while (true) {
  auto incoming = server.receive();
  ...
  server.send(outgoing);
}
```

What does the networking library that I gave to you expose/hide?

What is design?

- Not referring to UX (even though it’s important)
- Includes many things:
  - The components of the system
  - How they interact
  - The interfaces & abstractions they expose (or hide!)

Is design UML?

What is design?

- Not referring to UX (even though it’s important)
- Includes many things:
  - The components of the system
  - How they interact
  - The interfaces & abstractions they expose (or hide!)

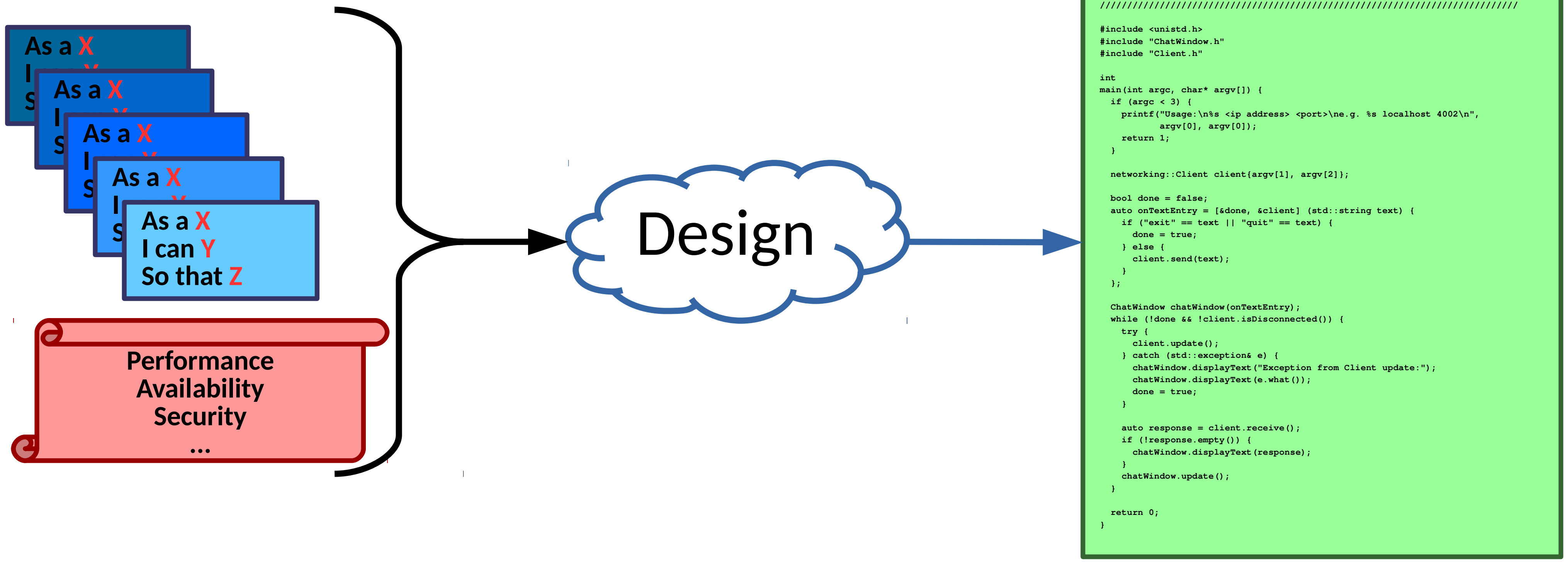
Is design UML?

Is UML design?

Why does design matter?

Why does design matter?

- Translating requirements and stories to code



```
////////////////////////////////////////////////////////////////////////////////
//                          Single Threaded Networking
//
// This file is distributed under the MIT License. See the LICENSE file
// for details.
////////////////////////////////////////////////////////////////////////////////

#include <unistd.h>
#include "ChatWindow.h"
#include "Client.h"

int
main(int argc, char* argv[]) {
  if (argc < 3) {
    printf("Usage:\n%s <ip address> <port>\ne.g. %s localhost 4002\n",
           argv[0], argv[0]);
    return 1;
  }

  networking::Client client{argv[1], argv[2]};

  bool done = false;
  auto onTextEntry = [&done, &client] (std::string text) {
    if ("exit" == text || "quit" == text) {
      done = true;
    } else {
      client.send(text);
    }
  };

  ChatWindow chatWindow(onTextEntry);
  while (!done && !client.isDisconnected()) {
    try {
      client.update();
    } catch (std::exception& e) {
      chatWindow.displayText("Exception from Client update:");
      chatWindow.displayText(e.what());
      done = true;
    }

    auto response = client.receive();
    if (!response.empty()) {
      chatWindow.displayText(response);
    }
    chatWindow.update();
  }

  return 0;
}
```

Why does design matter?

- Translating requirements and stories to code
- Understandability

Why does design matter?

- Translating requirements and stories to code
- Understandability

How much time do professional programmers spend reading code?

Why does design matter?

- Translating requirements and stories to code
- Understandability
- Performance & reliability

Why does design matter?

- Translating requirements and stories to code
- Understandability
- Performance & reliability
- Reusability

Why does design matter?

- Translating requirements and stories to code
- Understandability
- Performance & reliability
- Reusability
- Determines ease & risk for ***change***.

Why does design matter?

- Translating requirements and stories to code
- Understandability
- Performance & reliability
- Reusability
- Determines ease & risk for ***change***.
  - ***Understanding*** of requirements will change

Why does design matter?

- Translating requirements and stories to code
- Understandability
- Performance & reliability
- Reusability
- Determines ease & risk for ***change***.
  - Understanding of requirements will change
  - ***Requirements*** will change

Why does design matter?

- Translating requirements and stories to code
- Understandability
- Performance & reliability
- Reusability
- Determines ease & risk for ***change***.
  - Understanding of requirements will change
  - Requirements will change
  - Your code may outlast your time at a company

Why does design matter?

- Translating requirements and stories to code
- Understandability
- Performance & reliability
- Reusability
- Determines ease & risk for change.
  - Understanding of requirements will change
  - Requirements will change
  - Your code may outlast your time at a company
- Once software is too complex to reason about, it is too late

What makes a design bad?

- Too many possible ways to design poorly to list

What makes a design bad?

- Too many possible ways to design poorly to list
- Common attributes of a bad design: [Ousterhout 2018]

What makes a design bad?

- Too many possible ways to design poorly to list
- Common attributes of a bad design: [Ousterhout 2018]
  - ***Change Amplification***
    An apparently simple change requires modifying many locations

What makes a design bad?

- Too many possible ways to design poorly to list
- Common attributes of a bad design: [Ousterhout 2018]
  - *Change Amplification*
    An apparently simple change requires modifying many locations
  - *Cognitive Load*
    The developer needs to know a great deal in order to complete a task

What makes a design bad?

- Too many possible ways to design poorly to list
- Common attributes of a bad design: [Ousterhout 2018]
  - *Change Amplification*
    An apparently simple change requires modifying many locations
  - *Cognitive Load*
    The developer needs to know a great deal in order to complete a task
  - *Unknown unknowns*
    Potions of code to modify for a task may be hard to identify

What makes a design bad?

- Too many possible ways to design poorly to list
- Common attributes of a bad design: [Ousterhout 2018]
  - *Change Amplification*
    An apparently simple change requires modifying many locations
  - *Cognitive Load*
    The developer needs to know a great deal in order to complete a task
  - *Unknown unknowns*
    Potions of code to modify for a task may be hard to identify

These are symptoms of complexity.

What makes a design good?

- It identifies & manages complexity
  - ***Inherent*** (essential) complexity

What makes a design good?

- It identifies & manages complexity
  - ***Inherent*** (essential) complexity
  - ***Incidental*** (accidental) complexity

What makes a design good?

- It identifies & manages complexity
  - ***Inherent*** (essential) complexity
  - ***Incidental*** (accidental) complexity

$\xrightarrow{}$ hide

What makes a design good?

- It identifies & manages complexity
  - ***Inherent*** (essential) complexity $\rightarrow$ $\boxed{\text{hide}}$
  - ***Incidental*** (accidental) complexity $\rightarrow$ $\boxed{\text{minimize}}$

What makes a design good?

- It identifies & manages complexity
  - Inherent (essential) complexity
  - Incidental (accidental) complexity
- What is complexity?

What makes a design good?

- It identifies & manages complexity
  - Inherent (essential) complexity
  - Incidental (accidental) complexity
- What is complexity?
  - No agreed upon universal definition; many variants

What makes a design good?

- It identifies & manages complexity
  - Inherent (essential) complexity
  - Incidental (accidental) complexity
- What is complexity?
  - No agreed upon universal definition; many variants
  - Grows as entities/concepts in project are connected/woven together

What makes a design good?

- It identifies & manages complexity
  - Inherent (essential) complexity
  - Incidental (accidental) complexity
- What is complexity?
  - No agreed upon universal definition; many variants
  - Grows as entities/concepts in project are connected/woven together

    [Watch "Simple Made Easy" for one interesting perspective]

What makes a design good?

- It identifies & manages complexity
  - Inherent (essential) complexity
  - Incidental (accidental) complexity
- What is complexity?
  - No agreed upon universal definition; many variants
  - Grows as entities/concepts in project are connected/woven together
    [Watch “Simple Made Easy” for one interesting perspective]
  - One other heuristic is risk of *change*

What makes a design good?

Broadly

- Divides the system into ***independent*** components

What makes a design good?

Broadly

- Divides the system into independent components
- Makes it easy for developers to get their jobs done

What makes a design good?

- Not clever

What makes a design good?

- Not clever!

```
int x = foo(bar(baz(bam(a), b), c), d);
```

What makes a design good?

- Not clever!!

```
int x = foo(bar(baz(bam(a), b), c), d);
```

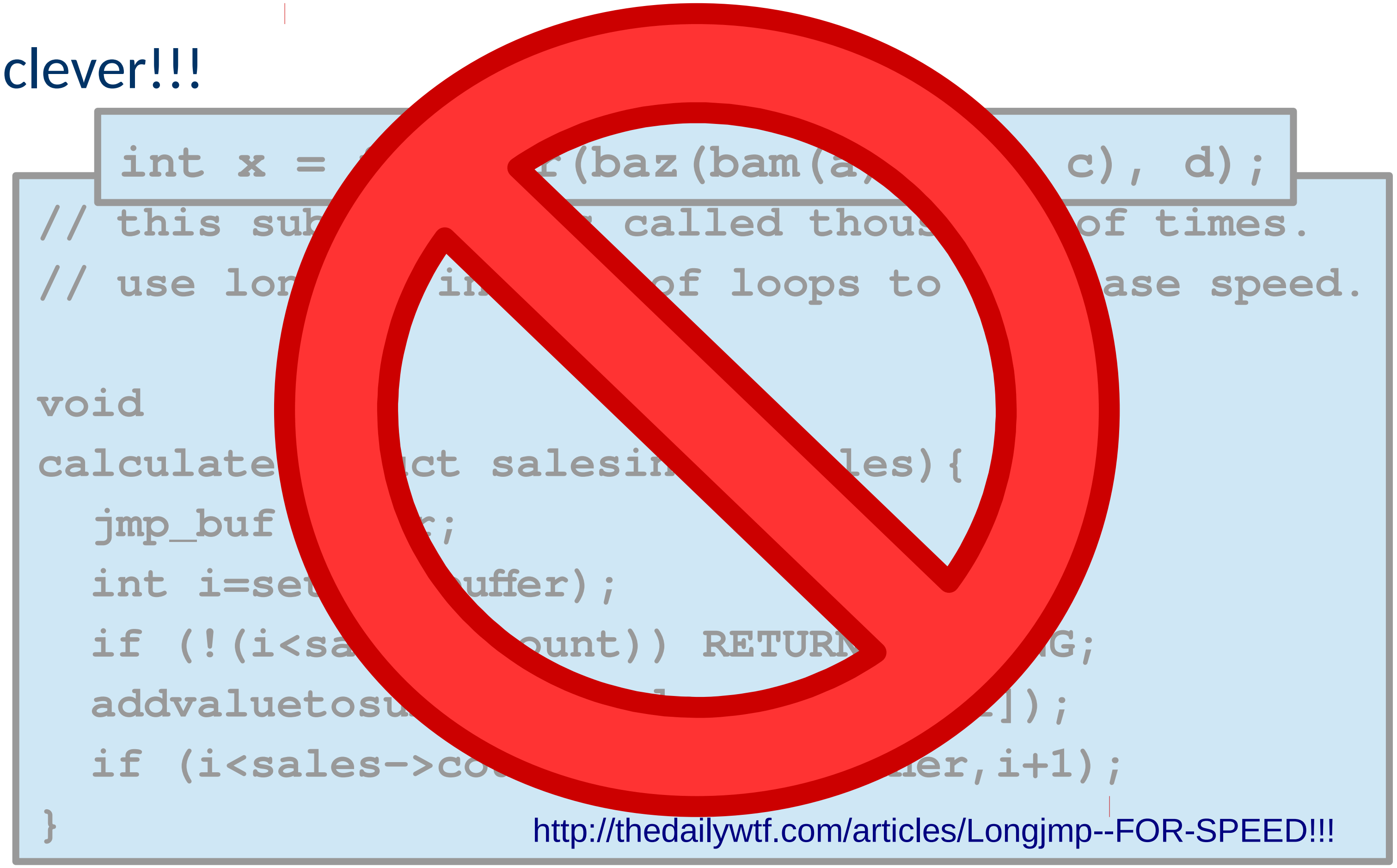
```
// this subroutine is called thousands of times.
// use longjmp instead of loops to increase speed.

void
calculate(struct salesinfo* sales){
  jmp_buf buffer;
  int i=setjmp(buffer);
  if (!(i<sales->count)) RETURN_NOTHING;
  addvaluetosubtotal(sales->values[i]);
  if (i<sales->count) longjmp(buffer,i+1);
}
```

http://thedailywtf.com/articles/Longjmp--FOR-SPEED!!!

What makes a design good?

- Not clever!!!



```
int x = f      r(baz(bam(a,       c), d);
```

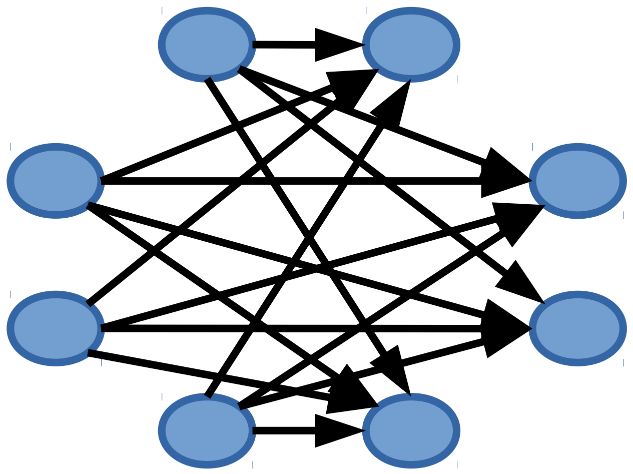
```
// this sub          called thous         of times.
// use lon       in      of loops to          ase speed.

void
calculate      uct salesin         les){
  jmp_buf      r;
  int i=se      uffer);
  if (!(i<sa         ount)) RETUR        G;
  addvaluetosu                      ]);
  if (i<sales->co                  ler,i+1);
}
```

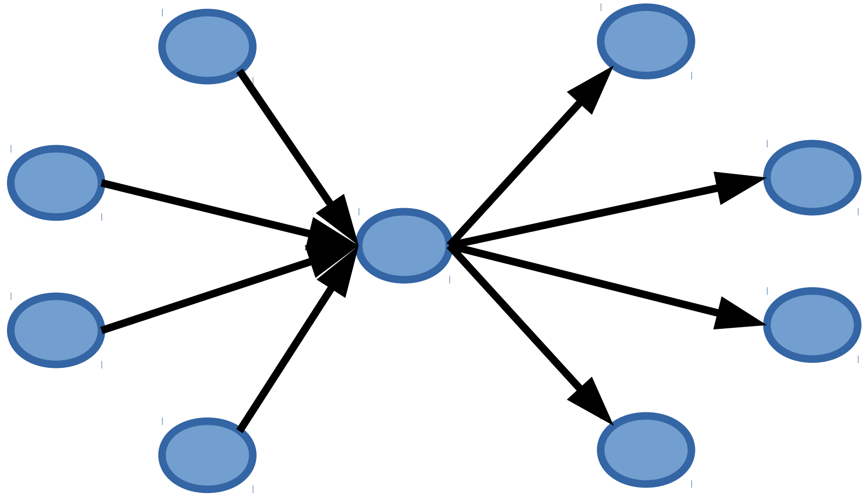
http://thedailywtf.com/articles/Longjmp--FOR-SPEED!!!

What makes a design good?

- Not clever
- Loose coupling

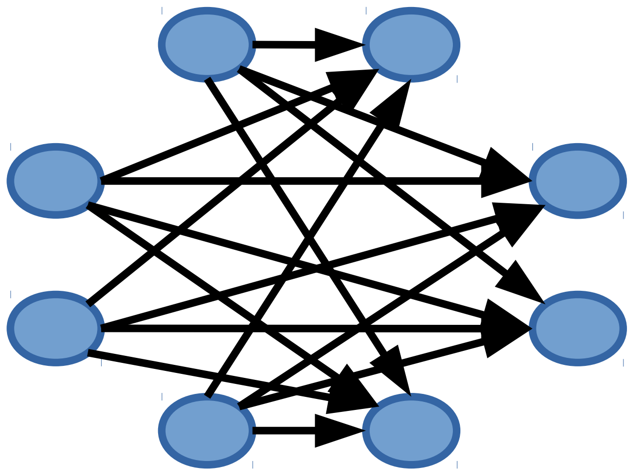


vs

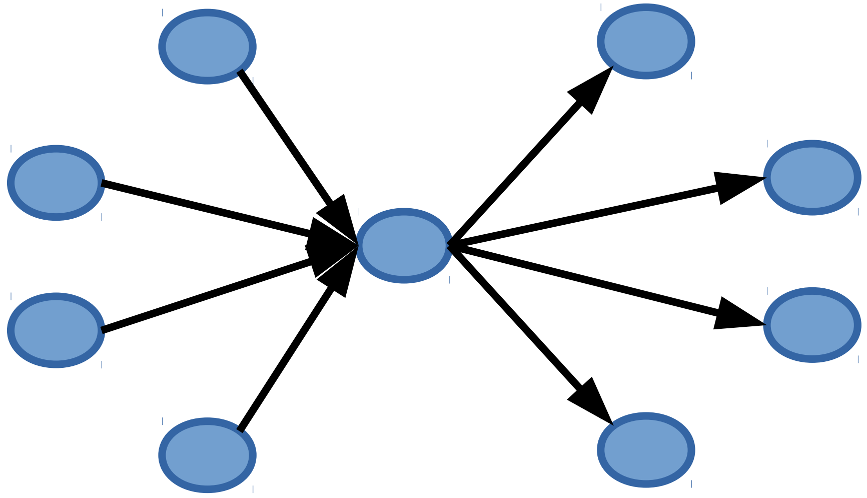


What makes a design good?

- Not clever
- Loose coupling



vs



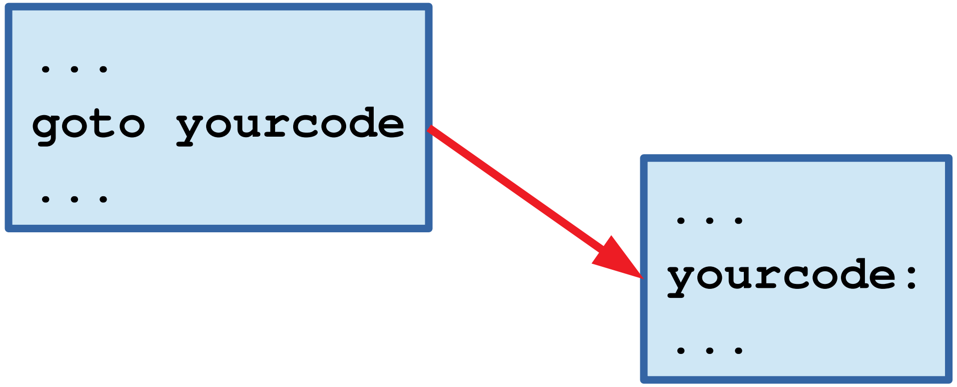
Why?

What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)

What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)



```
...
goto yourcode
...
```

```
...
yourcode:
...
```

What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data

What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data

```
... = global
```

```
int global = ...
```

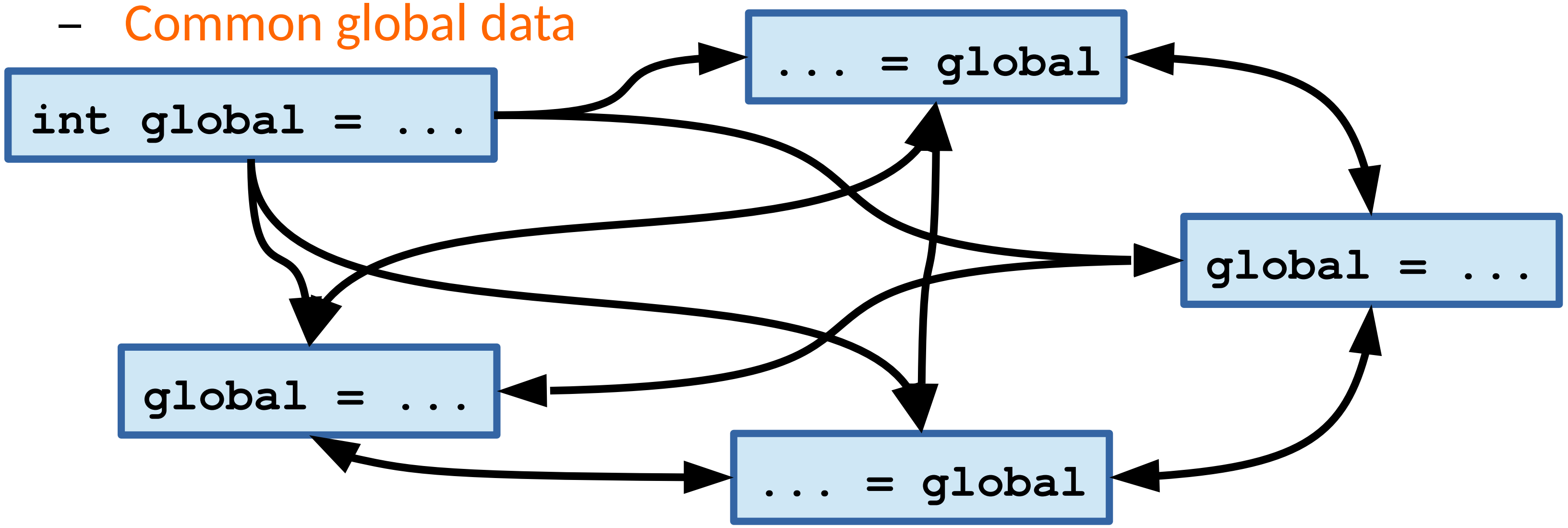
```
global = ...
```

```
global = ...
```

```
... = global
```

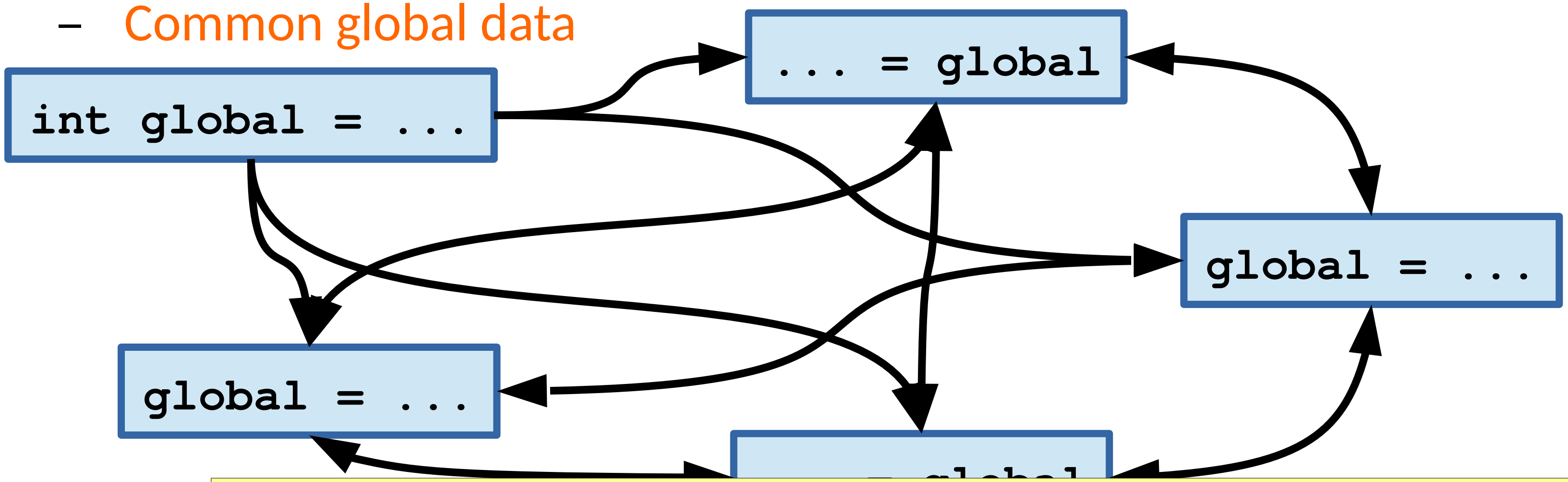
What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data



What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data



Singletons have these constraints and worse.

What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data
  - Subclassing

We will spend a day in the future on this.

What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data
  - Subclassing
  - Temporal

What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data
  - Subclassing
  - Temporal

```
Cat cat = new Cat;
...
delete cat;
```

What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data
  - Subclassing
  - Temporal

```
Process p;
p.doStep1();
p.doStep2();
p.doStep3();
```

```
Cat cat = new Cat;
...
delete cat;
```

What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data
  - Subclassing
  - Temporal

```
Process p;
p.doStep1();
p.doStep2();
p.doStep3();
```

```
Process p;
p.foo();
p.bar();
p.baz();
```

```
Cat cat = new Cat;
...
delete cat;
```

This is more insidious!

What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data
  - Subclassing
  - Temporal
  - Passing data to/from each other

What makes a design good?

- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data
  - Subclassing
  - Temporal
  - Passing data to/from each other

```
x = foo(1,2)
```

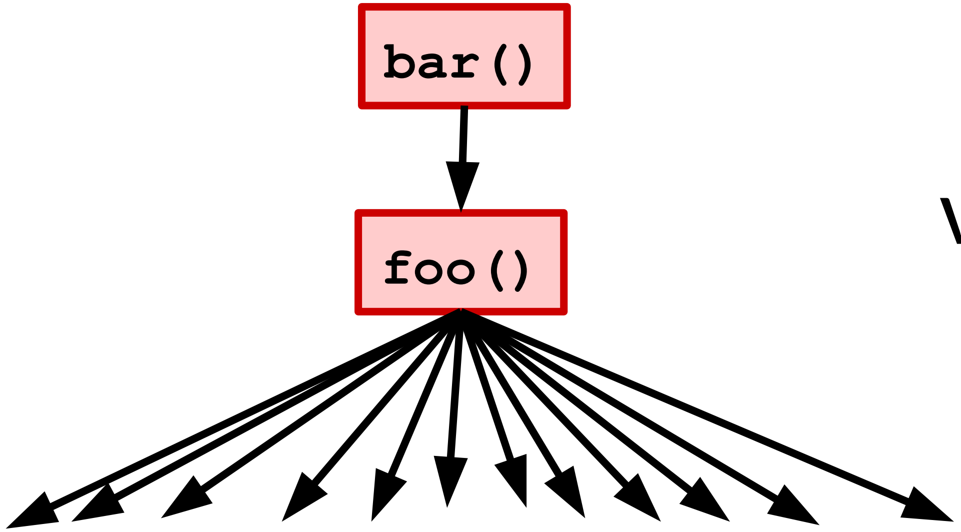
```
def foo(a, b):
    ...
```

What makes a design good?

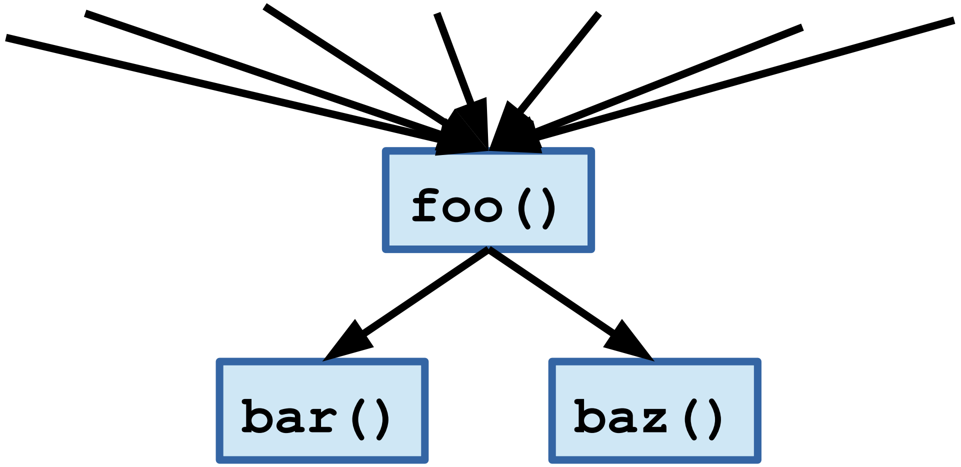
- Not clever
- Loose coupling
  - Content (accessing implementation of another component)
  - Common global data
  - Subclassing
  - Temporal
  - Passing data to/from each other
  - Independence

What makes a design good?

- Not clever
- Loose coupling
- High fan in / low fan out

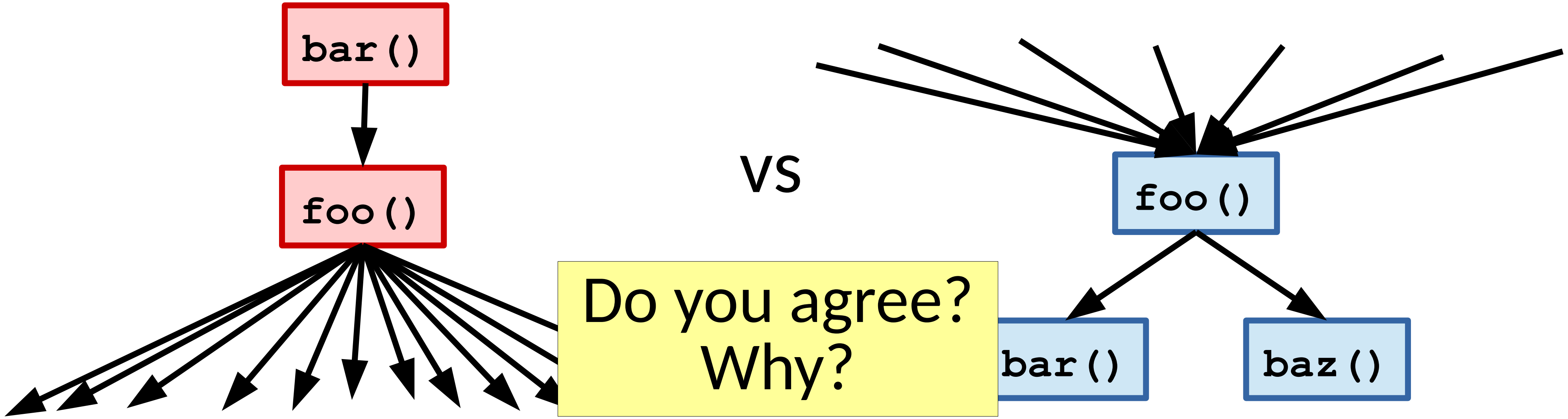


VS



What makes a design good?

- Not clever
- Loose coupling
- High fan in / low fan out



Do you agree?
Why?

What makes a design good?

- Not clever
- Loose coupling
- High fan in / low fan out
- Layers / Stratification

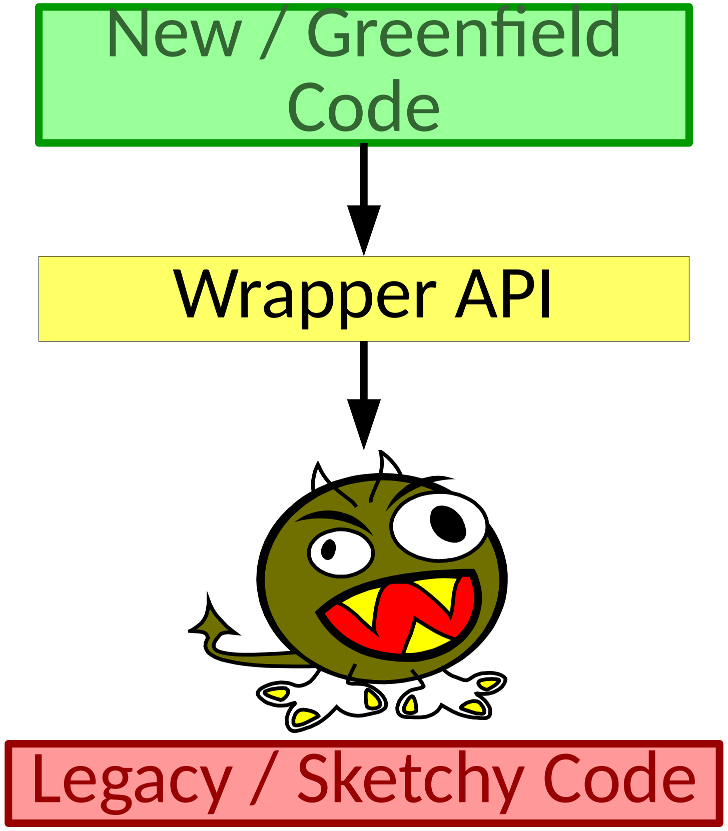
What makes a design good?

- Not clever
- Loose coupling
- High fan in / low fan out
- Layers / Stratification

& a consistent, self contained view per level

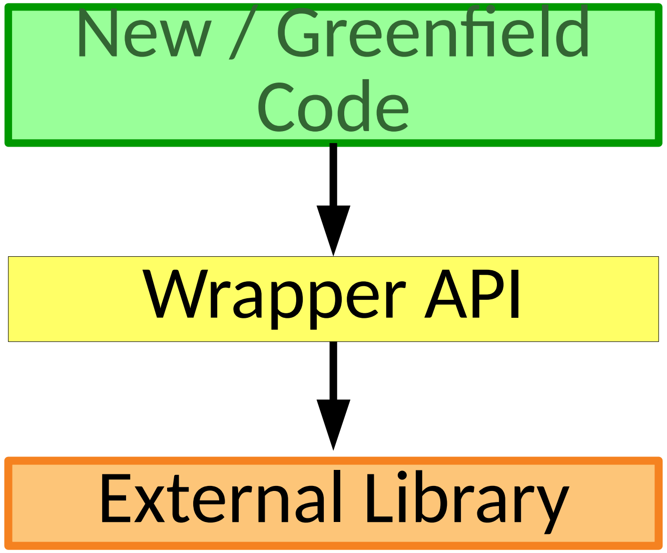
What makes a design good?

- Not clever
- Loose coupling
- High fan in / low fan out
- Layers / Stratification



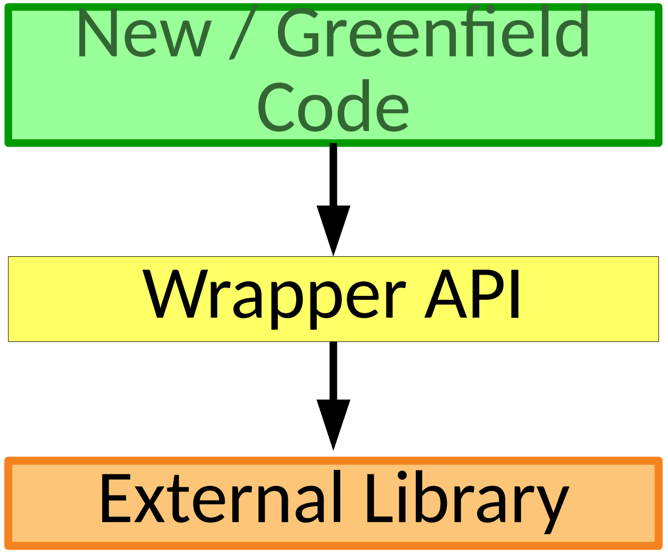
What makes a design good?

- Not clever
- Loose coupling
- High fan in / low fan out
- Layers / Stratification



What makes a design good?

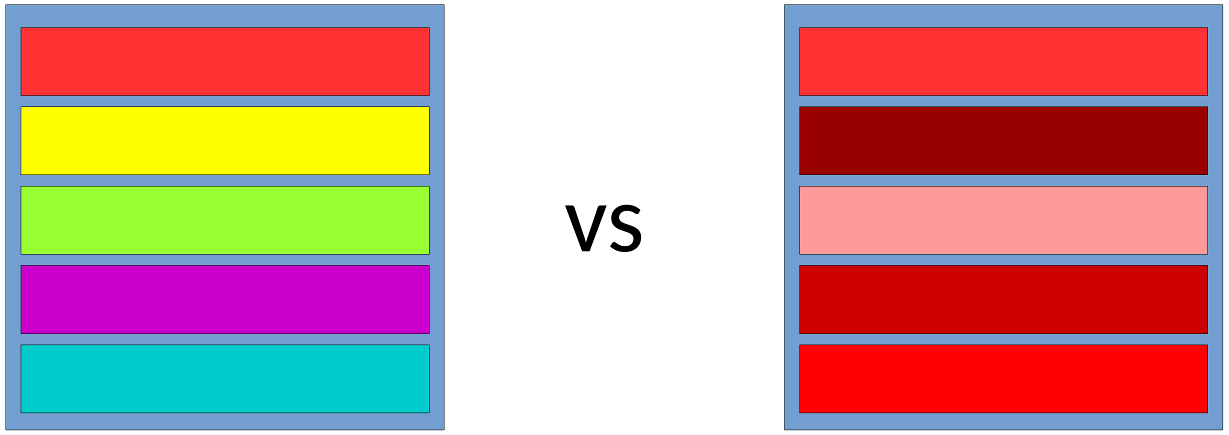
- Not clever
- Loose coupling
- High fan in / low fan out
- Layers / Stratification



What impact does this have on invariants & types?

What makes a design good?

- Not clever
- Loose coupling
- High fan in / low fan out
- Layers / Stratification
- Cohesion
- …



What makes a design good?

- Not clever
- Loose coupling
- High fan in / low fan out
- Layers / Stratification
- Cohesion
- ...

But these are the ends, not the means

Revisiting Complexity

- We can characterize ***causes*** of complex designs [Ousterhout 2018]

Revisiting Complexity

- We can characterize ***causes*** of complex designs [Ousterhout 2018]
  - ***Dependencies***
    Code cannot be understood in isolation because of relationships to other code.

Revisiting Complexity

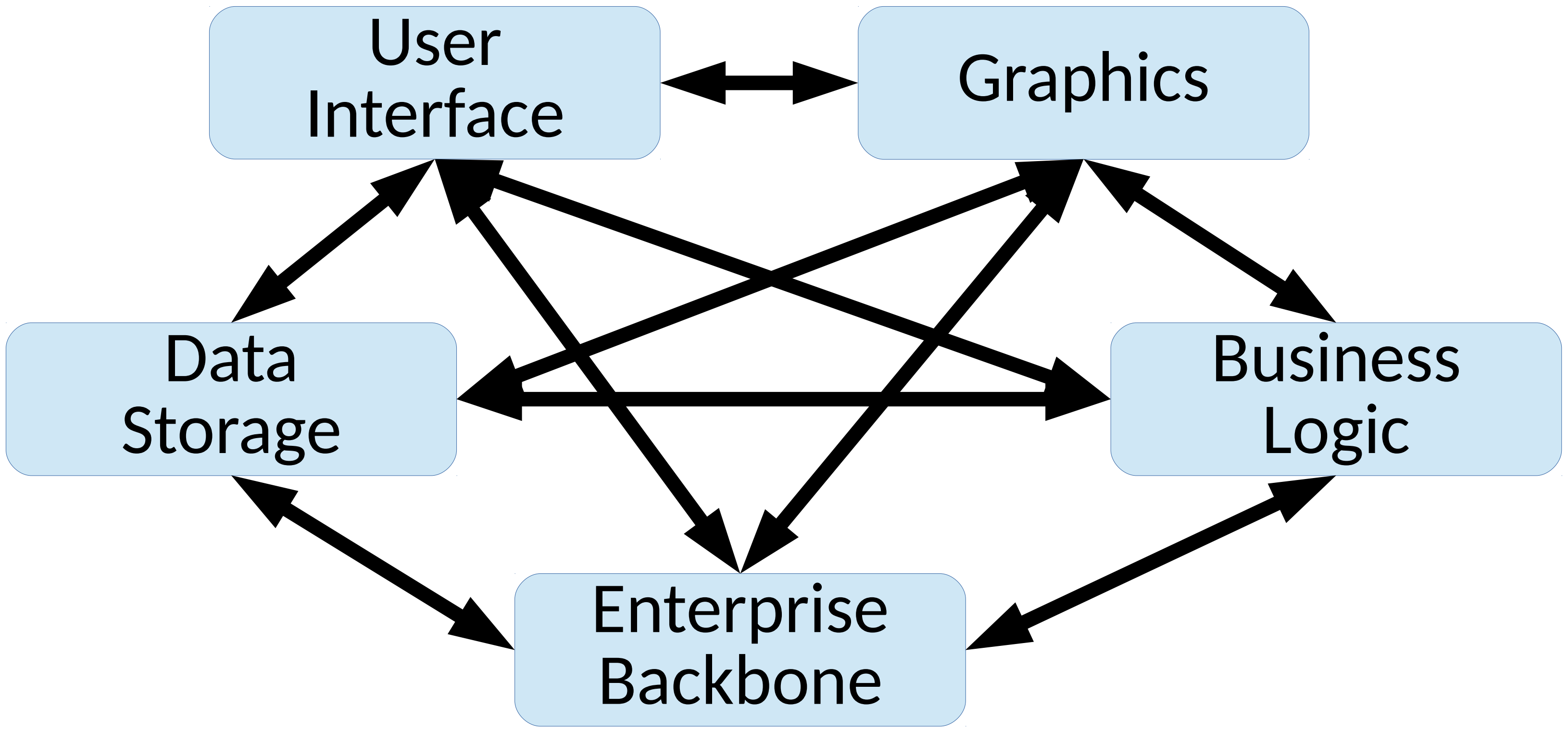
- We can characterize ***causes*** of complex designs [Ousterhout 2018]
  - ***Dependencies***
    Code cannot be understood in isolation because of relationships to other code.
  - ***Obscurity***
    Important information about code is not obvious.

Revisiting Complexity

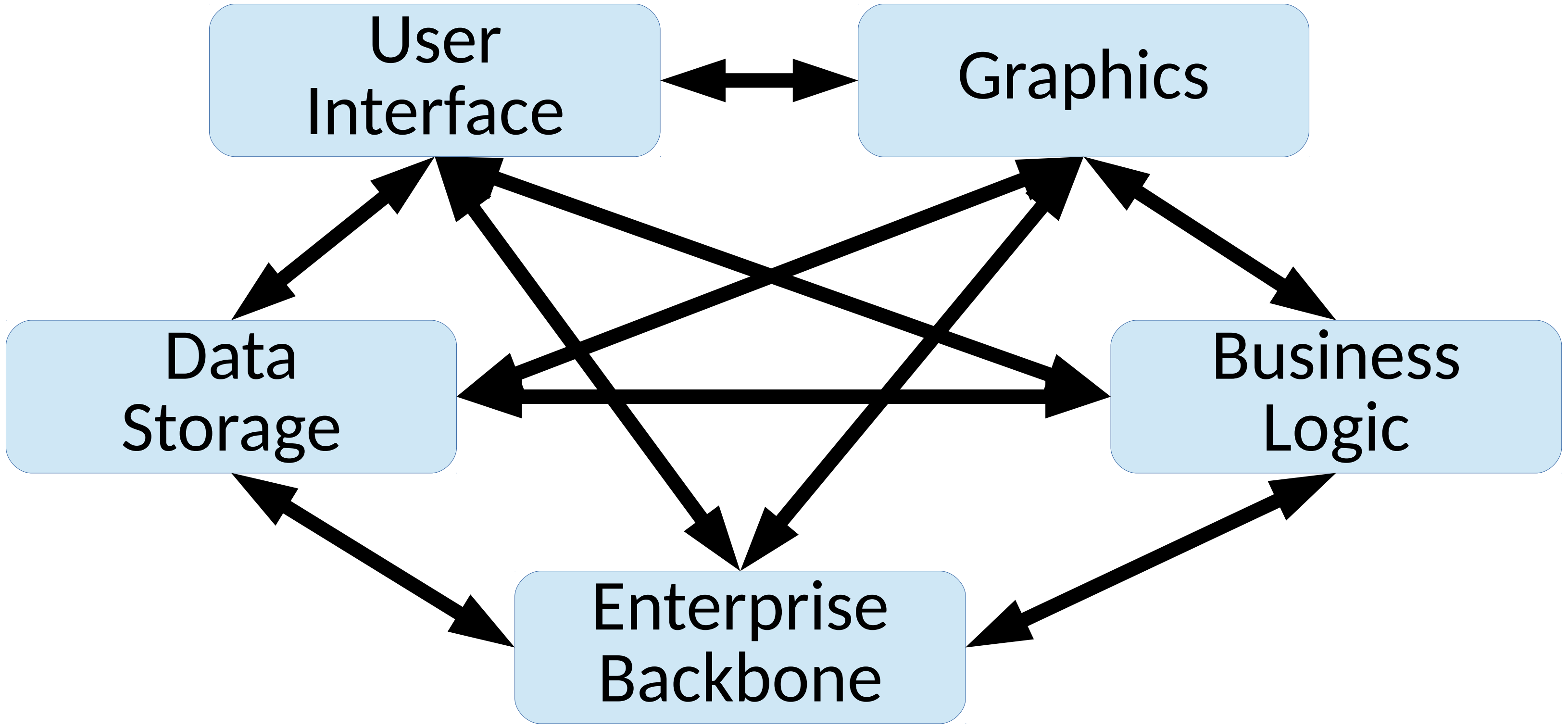
- We can characterize ***causes*** of complex designs [Ousterhout 2018]
  - ***Dependencies***
    Code cannot be understood in isolation because of relationships to other code.
  - ***Obscurity***
    Important information about code is not obvious.

These directly relate to the qualities of good code we just saw.

Consider a design



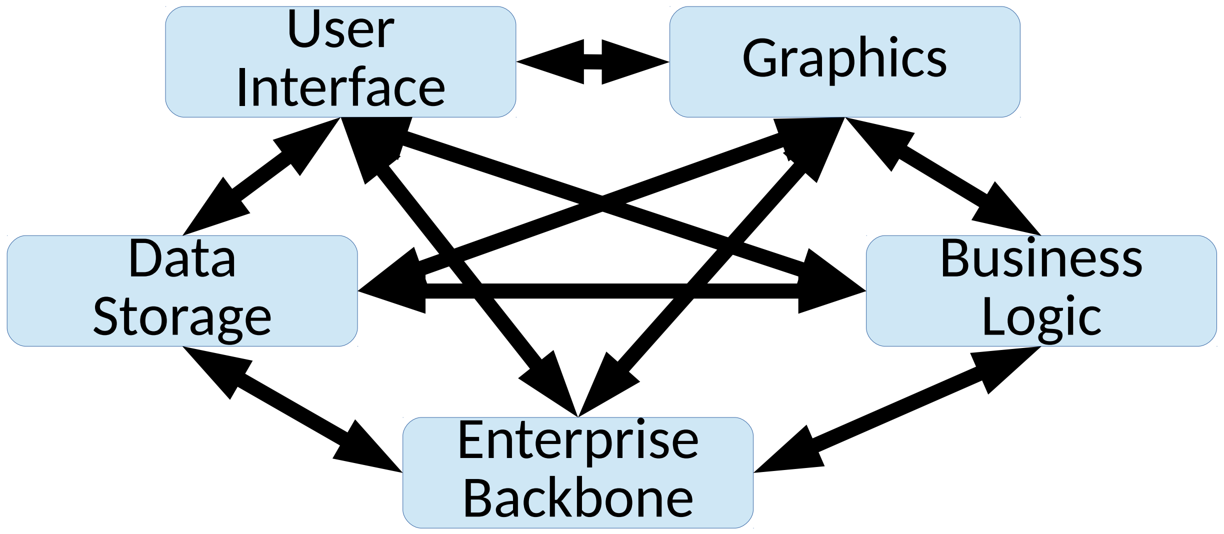
Consider a design



Is this simple? Why?

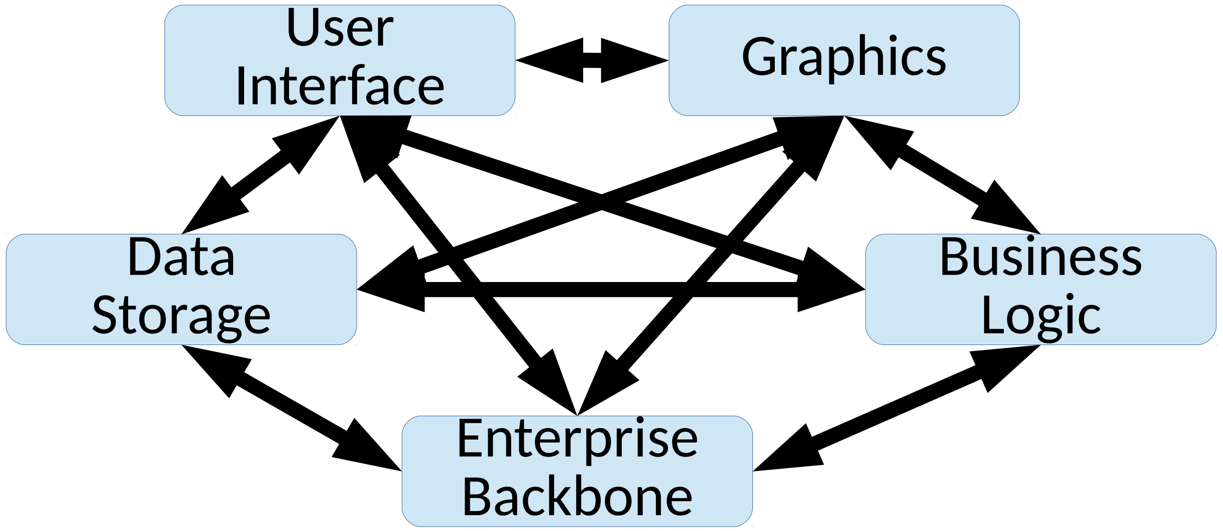
Consider a design

- What if you want to ***modify*** the business logic?



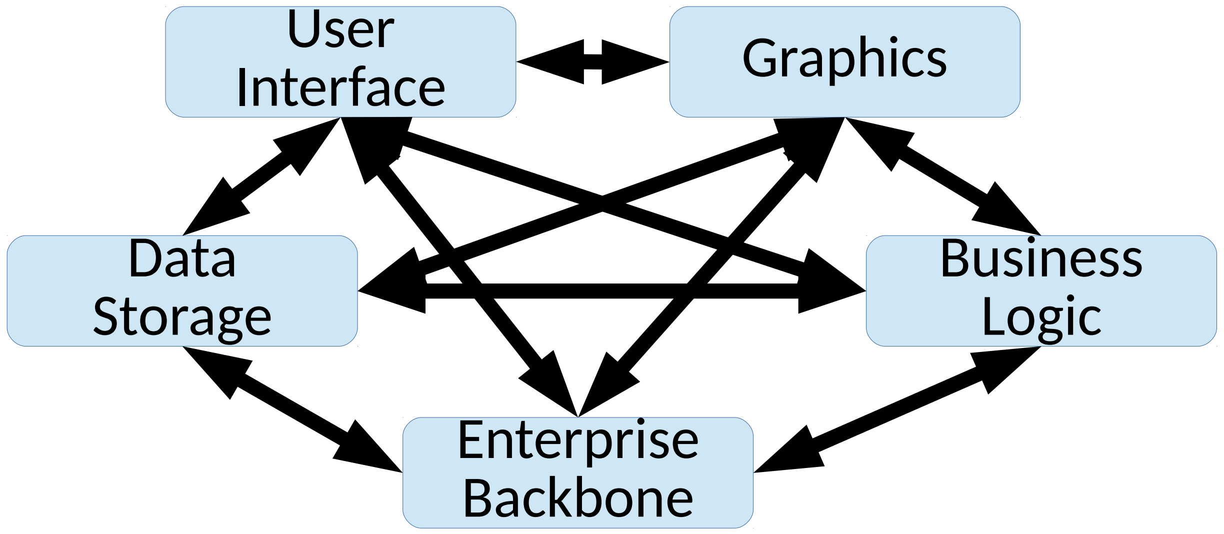
Consider a design

- What if you want to ***modify*** the business logic?
- What if you want to ***reuse*** the business logic?



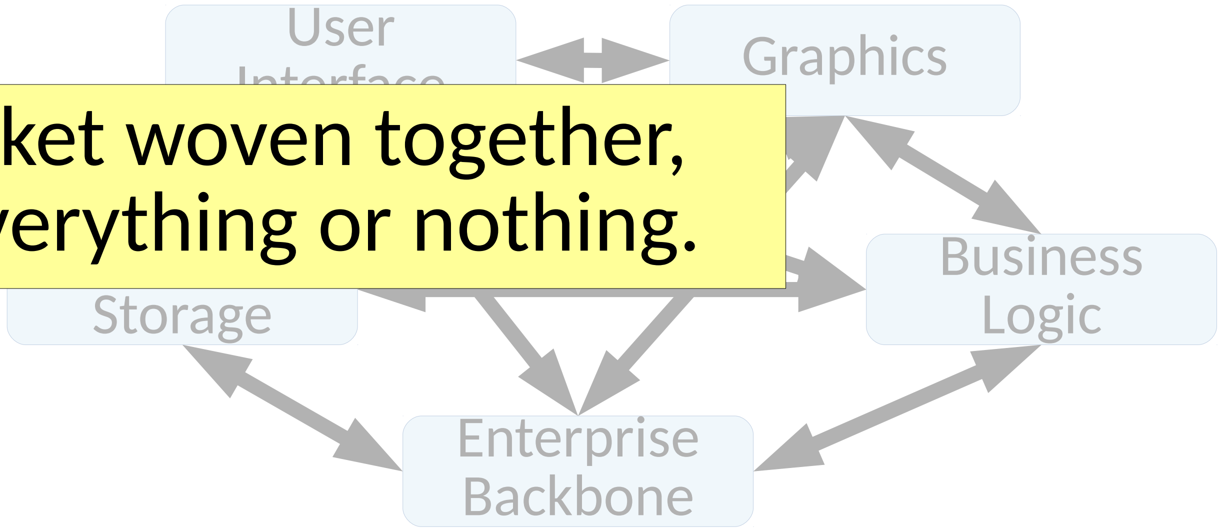
Consider a design

- What if you want to ***modify*** the business logic?
- What if you want to ***reuse*** the business logic?
- What if you want to ***replace*** the display?



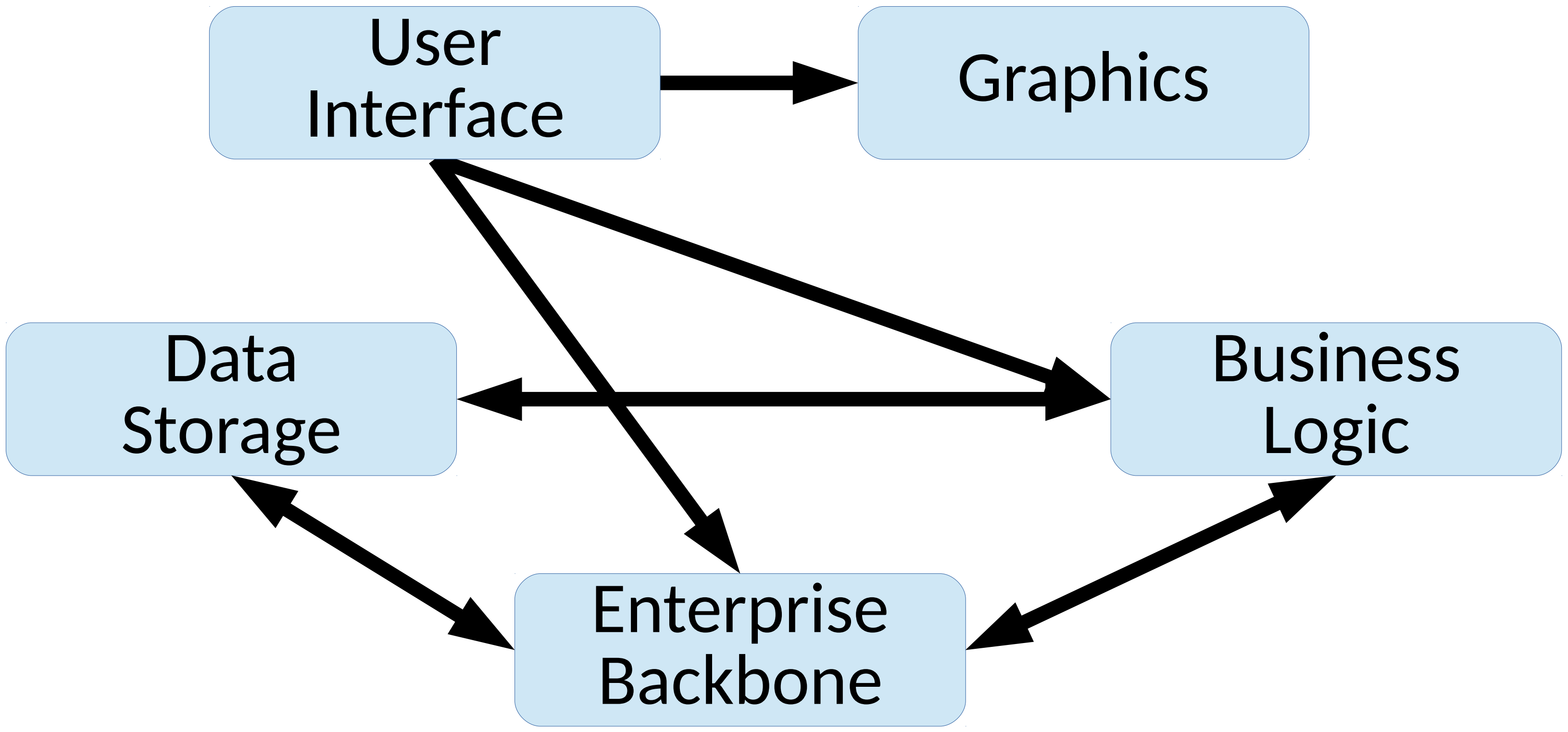
Consider a design

- What if you want to ***modify*** the business logic?
- What if you want to ***reuse*** the business logic?
- What if you want to ***replace*** the display?

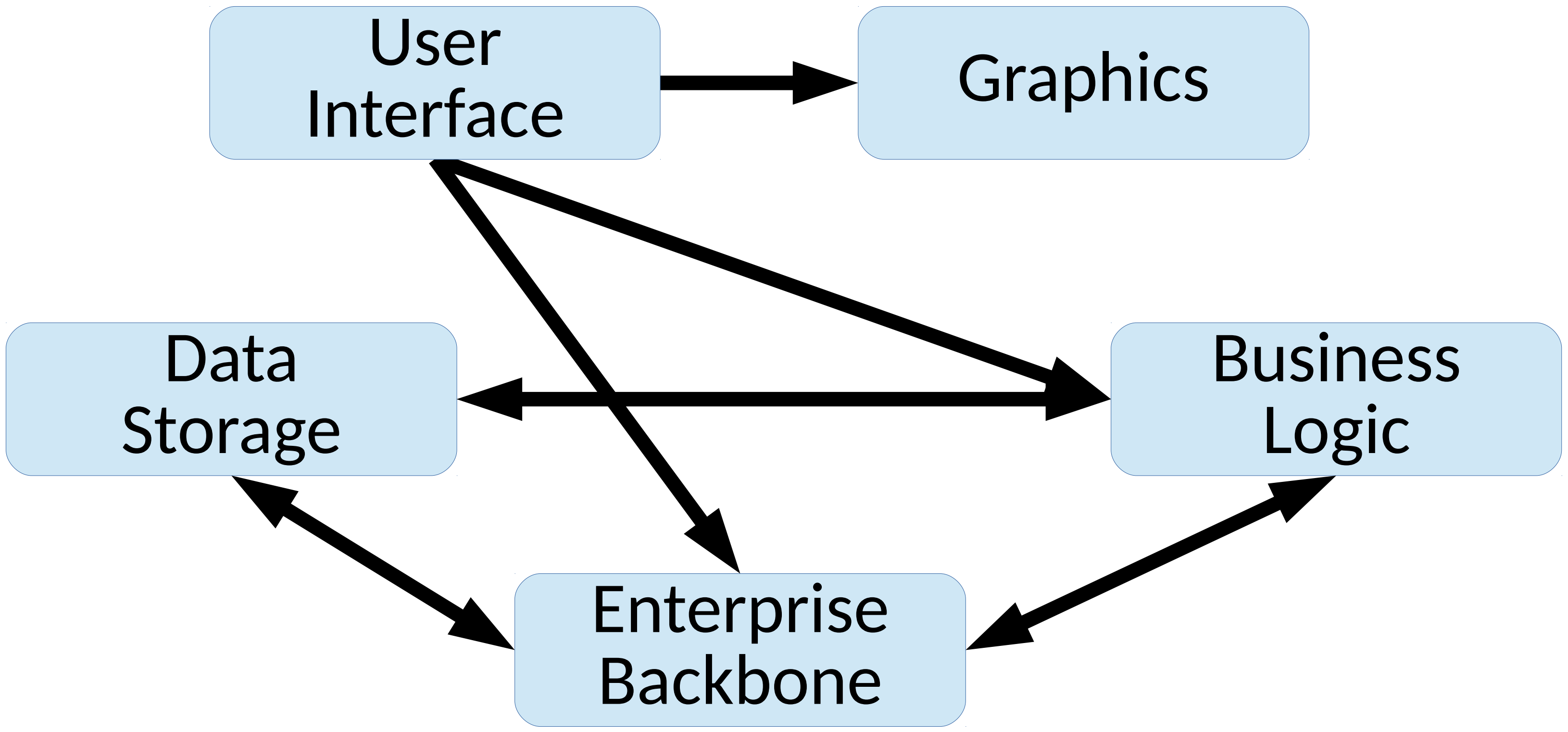


Like a basket woven together, you get everything or nothing.

Consider a design

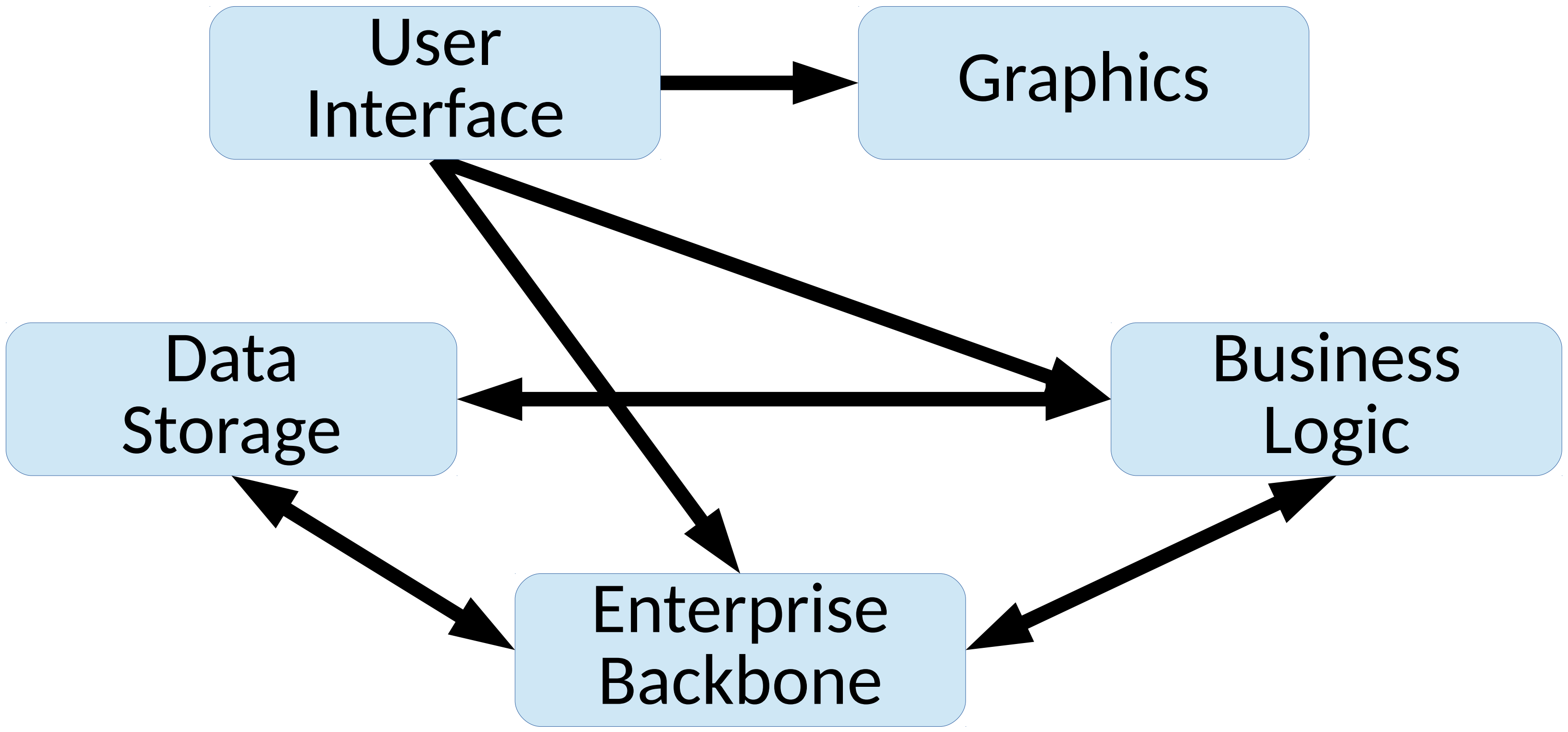


Consider a design



Is this simple*r*? Why?

Consider a design



Is this simpler? Why?

What is still complex? Why?

Consider a design

- The fewer connected or conflated concepts, the better

Consider a function

```
bool
isFasterThanSound(double speed) {
  return speed > MACH1;
}
```

Consider a function

```
bool
isFasterThanSound(double speed) {
   return speed > MACH1;
}
```

Is this simple or complex? Why?

Consider a function

```
bool
isFasterThanSound(double speed) {
  return speed > MACH1;
```

```
[redacted]
[redacted](double speed, double angle) {
  [redacted]

}
```

Is this simple or complex? Why?

Consider a function

```
bool
isFasterThanSound(double speed) {
  return speed > MACH1;
```

```
[redacted]
[redacted](double speed, double angle) {
  [redacted]

}
```

Is this simple or complex? Why?

A good design should be ***hard to misuse***

Consid

```
class Student {
public:
   ...
   ID getID() const;
   Name getName() const;
   Address getAddress() const;

   void storeToDatabase() const;
   static Student readFromDatabase();

   bool canApplyForCoOp();
   bool meetsDegreeRequirements();
};
```

Consic

```
class Student {
public:
   ...
   ID getID() const;
   Name getName() const;
   Address getAddress() const;

   void storeToDatabase() const;
   static Student readFromDatabase();

   bool canApplyForCoOp();
   bool meetsDegreeRequirements();
```

What is *good* about this class?

Consid

```
class Student {
public:
   ...
   ID getID() const;
   Name getName() const;
   Address getAddress() const;

   void storeToDatabase() const;
   static Student readFromDatabase();

   bool canApplyForCoOp();
   bool meetsDegreeRequirements();
```

What is *good* about this class?

What is ***bad*** about this class?

What are our simplifying tools?

- Metaphors – identify “real world” objects & relations

What are our simplifying tools?

- Metaphors – identify “real world” objects & relations

Be careful.
This can be a good place to start,
but a poor place to end.

What are our simplifying tools?

- Metaphors – identify “real world” objects & relations
- Abstraction – use high level concepts

What are our simplifying tools?

- Metaphors – identify “real world” objects & relations
- Abstraction – use high level concepts
- Encapsulation – hide the details

This is the Code Complete definition, not a universal one!

What are our simplifying tools?

- Metaphors – identify “real world” objects & relations
- Abstraction – use high level concepts
- Encapsulation – hide the details

Deeply tied to ***information hiding***

What are our simplifying tools?

- Metaphors – identify "real world" objects & relations
- Abstraction – use high level concepts
- Encapsulation – hide the details
- Consistency

What are our simplifying tools?

- Metaphors – identify “real world” objects & relations
- Abstraction – use high level concepts
- Encapsulation – hide the details
- Consistency
- Inheritance?

What are our simplifying tools?

- Metaphors – identify “real world” objects & relations
- Abstraction – use high level concepts
- Encapsulation – hide the details
- Consistency
- Inheritance?
  - In small, constrained doses
  - Ideally through interfaces

What are our simplifying tools?

- Metaphors – identify “real world” objects & relations
- Abstraction – use high level concepts
- Encapsulation – hide the details
- Consistency
- Inheritance?
  - In small, constrained doses
  - Ideally through interfaces

Use especially for:
1) likely/risky to change code
2) frequently used code

Key Strategy: Mitigate change

- Identify potential areas of change

```
class Student {
public:
   ...
   int getID() const;
   ...
};
```

Key Strategy: Mitigate change

- Identify potential areas of change

```
class Student {
public:
   ...
   int getID() const;
   ...
};
```

Key Strategy: Mitigate change

- Identify potential areas of change
- Separate them structurally

```
class Student {
public:
   ...
   int getID() const;
   ...
};
```

Key Strategy: Mitigate change

- Identify potential areas of change
- Separate them structurally

```
class Student {
public:
   ...
  int getID() const;
   ...
};
```

```
class Student {
public:
   ...
  ID getID() const;
   ...
};
```

Key Strategy: Mitigate change

- Identify potential areas of change
- Separate them structurally
- Isolate their impact through interfaces

Key Strategy: Mitigate change

- Identify potential areas of change
- Separate them structurally
- Isolate their impact through interfaces

```
class IDCreator {
public:
   ...
   virtual ID createID() = 0;
   ...
};
```

Key Strategy: Mitigate change

```
...
ID studentID = student.getID();
...
```

How might this hinder change?

Key Strategy: Mitigate change

```
...
ID studentID = student.getID();
...
```

How might this hinder change?

How can it be resolved?

Key Strategy: Mitigate change

```
...
ID studentID = student.getID();
...
```

How might this hinder change?

How can it be resolved?

What are the trade offs?

Constant Vigilance

- Avoiding complexity requires a planned process

Constant Vigilance

- Avoiding complexity requires a planned process
  - Code review everything

  [metaphors, abstraction, encapsulation, consistency, inheritance]

Constant Vigilance

- Avoiding complexity requires a planned process
  - Code review everything

    [metaphors, abstraction, encapsulation, consistency, inheritance]
  - Write tests (simple code is easier to test)

Constant Vigilance

- Avoiding complexity requires a planned process
  - Code review everything
    [metaphors, abstraction, encapsulation, consistency, inheritance]
  - Write tests (simple code is easier to test)
- Know when & where you make bad decisions
  - *technical debt*

Constant Vigilance

- Avoiding complexity requires a planned process
  - Code review everything
    [metaphors, abstraction, encapsulation, consistency, inheritance]
  - Write tests (simple code is easier to test)
- Know when & where you make bad decisions
  - *technical debt*
    - You end up paying it back!

Design Smells

- A *design smell* is a clue that better design is needed

Design Smells

- A *design smell* is a clue that better design is needed
- Such as: (adapted from John Ousterhout)
  - Thin components
    - Is it really hiding an implementation?
    - Is complexity arising from having to many small classes?

Design Smells

- A *design smell* is a clue that better design is needed
- Such as: (adapted from John Ousterhout)
  - Thin components
    - Is it really hiding an implementation?
    - Is complexity arising from having to many small classes?
  - Information leaks
    - Can I see the implementation details? (unintentional interface)
    - Repeated similar code

Design Smells

- A *design smell* is a clue that better design is needed
- Such as: (adapted from John Ousterhout)
  - Thin components
    - Is it really hiding an implementation?
    - Is complexity arising from having to many small classes?
  - Information leaks
    - Can I see the implementation details? (unintentional interface)
    - Repeated similar code
  - Difficulty making a change

Experience

- Experience hones your sense of design.
  - Hopefully, our discussions this semester will help you be aware of it.