Object-Oriented Programming

Lessons Learned

Johnny Bigert, Ph.D., Skype/Microsoft
johnny.bigert@skype.net
October 26, 2011

Software Development

Why do we do the things we do?

Software Development

- Object-orientation, design principles, timeboxing, teams, geographic colocation, Scrum, lean, continuous improvement, refactoring, design patterns, prototyping, continuous integration, unit tests, test driven development, modularity, readability, testability, layering, reuse, interfaces, APIs, decoupling, cohesion, revision control…
- Sure, but why?

Problem Description

- ”Develop a software system according to requirements (functional and non-functional)”

- Sounds simple!

Constraints

- Cost
  - of development, testing, on-boarding, …
- Time to market
  - changing/unclear requirements, new features, …
- Quality
  - bugs, regressions, user experience, …

Or ”good/cheap/fast – choose two”

Tools At Our Disposal

- Focus on internal quality (good code)
- Quality assurance (testing)
- Software development methodology (Scrum etc, not covered today)
- Etc

Constraints

- Cost
  - of development, testing, on-boarding, …
- Time to market
  - changing/unclear requirements, new features, …
- Quality
  - bugs, regressions, user experience, …

Internal Quality

What is it? Why do we need it? How do we get it?

Internal Quality

- = Good code!
- = Clean, consistent, well-structured, readable, extensible, adaptable, decoupled, testable, well-tested etc…

Constraints (revisited)

- Cost
  - of development, testing, on-boarding, …
- Time to market
  - changing/unclear requirements, new features, …
- Quality
  - bugs, regressions, user experience, …
- Poor internal quality affects them all!

Obtaining Internal Quality

Examples:

- Object-Oriented Design Principles
  - ”Head First Design Patterns” by Freeman and Freeman
- Design Patterns
  - ”Design Patterns” by Gamma et. al.
- Code should be easy to read
  - (you write it once, but read it hundreds of times!)
  - ”Code Complete” by McConnell
- Code reviews, Pair programming, Code metrics, Test-driven development, …

Example

A networking application

Example

- Networking application

- ```
  class Application
  {
      void buttonPushed()
      {
          network->sendData(data);
      }
      Network *network;
  };
  ```
  ```

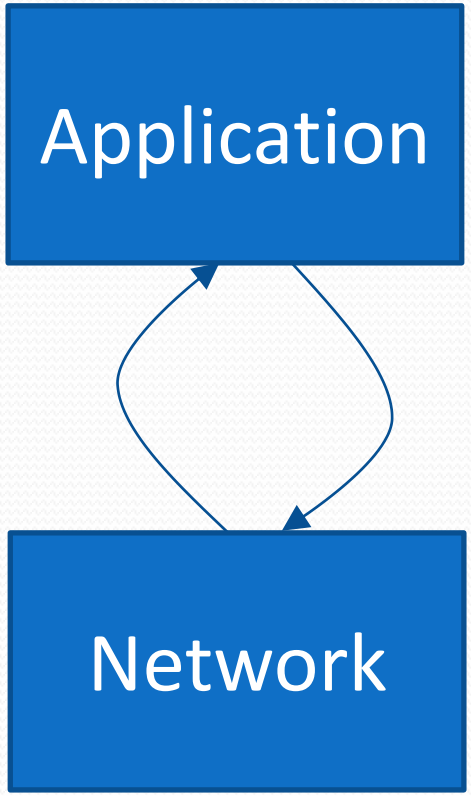
Example (continued)

```
class Network
{
    void packetReceived(...)
    {
        app->onDataReceived(data);
    }
    Application *app;
};
```

Downsides

Tightly coupled:

- Can't easily replace network
- Can't easily test application logic without real network
- Can't reuse network
- Hard to understand which parts are used by the other



Dilemma

- The Application needs to communicate with the Network (and vice versa)
- The Application must not know about the Network (and vice versa)

Object-Oriented Design Principles

Object-Oriented Design Principles

- High-level guidelines for object-oriented development
- Fundamental to good software architecture

SOLID Design Principles

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

- http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29

Single Responsibility Principle

- ”A class should have a single responsibility”
- = A class (or interface, or module) should have a **single reason to change**

- http://en.wikipedia.org/wiki/Single_responsibility_principle

Single Responsibility Principle Applied

- Example: A network class that both
  1. formats the data to send and
  2. sends the data over the wire
- The class will change if
  1. the data format changes or
  2. the sending of the data changes
- Changes to one might break the other
- Split into two classes!

Open/Closed Principle

- ”A class (or module) should be open for extension but closed for modification”
- = Make sure that there are ways to **modify the behavior** of your software **without changing** it

- http://en.wikipedia.org/wiki/Open/closed_principle

Open/Closed Principle Applied

- We want to avoid code changes:
  - if(condition) listener = new SomeListener; else if(condition2) listener = …;
- Instead, **move the decision out of the class**
  - Constructor/setter injection (setListener)
  - Add/remove (addListener)

Open/Closed Principle Applied (cont)

- We want to create an application that can send messages over the network
- Application doesn't care how things are sent
  - UDP/TCP, HTTP, binary, email, smoke signals
- Application::Application(INetworkSender)
  - Network strategy - the strategy design pattern
- We can **modify** the application behavior **without changing** its code

Liskov Substitution Principle

- ”You should be able to substitute a class with any of its subclasses without altering the properties of the program (correctness etc.)”
- = Subclasses should **obey the semantics** of super-classes they inherit (or interfaces they implement)
- Design by contract

- http://en.wikipedia.org/wiki/Liskov_substitution_principle

Liskov Substitution Principle Applied

- A classic: Is a circle an ellipse? (= Is Circle a subclass of Ellipse)?
- Contract on Ellipse: calling setHeight should not affect getWidth
- Circle: setHeight will change width
- If Circle is a subclass of Ellipse, it violates LSP

Interface Segregation Principle

- “Many client specific interfaces are better than one general purpose interface”
- = A class should **only depend on functions it uses**

- http://en.wikipedia.org/wiki/Interface_segregation_principle

Interface Segregation Principle Applied

- Example: A Network class delivers messages to the application
- If the Network class knows about the Application class…
- …then Network will know Application's full public interface = bad
- Create a client specific interface INetworkReceiver, let Application inherit

Dependency Inversion Principle

- ”Depend on abstractions, do not depend on concretions”
- The ”inversion” is the core of object-oriented programming

- http://en.wikipedia.org/wiki/Dependency_inversion_principle

Decoupling Through Interfaces

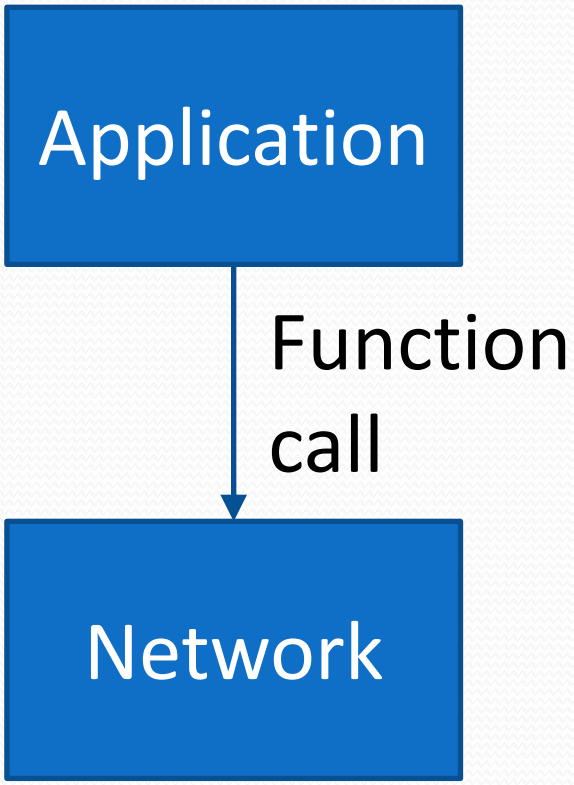
Breaking dependencies

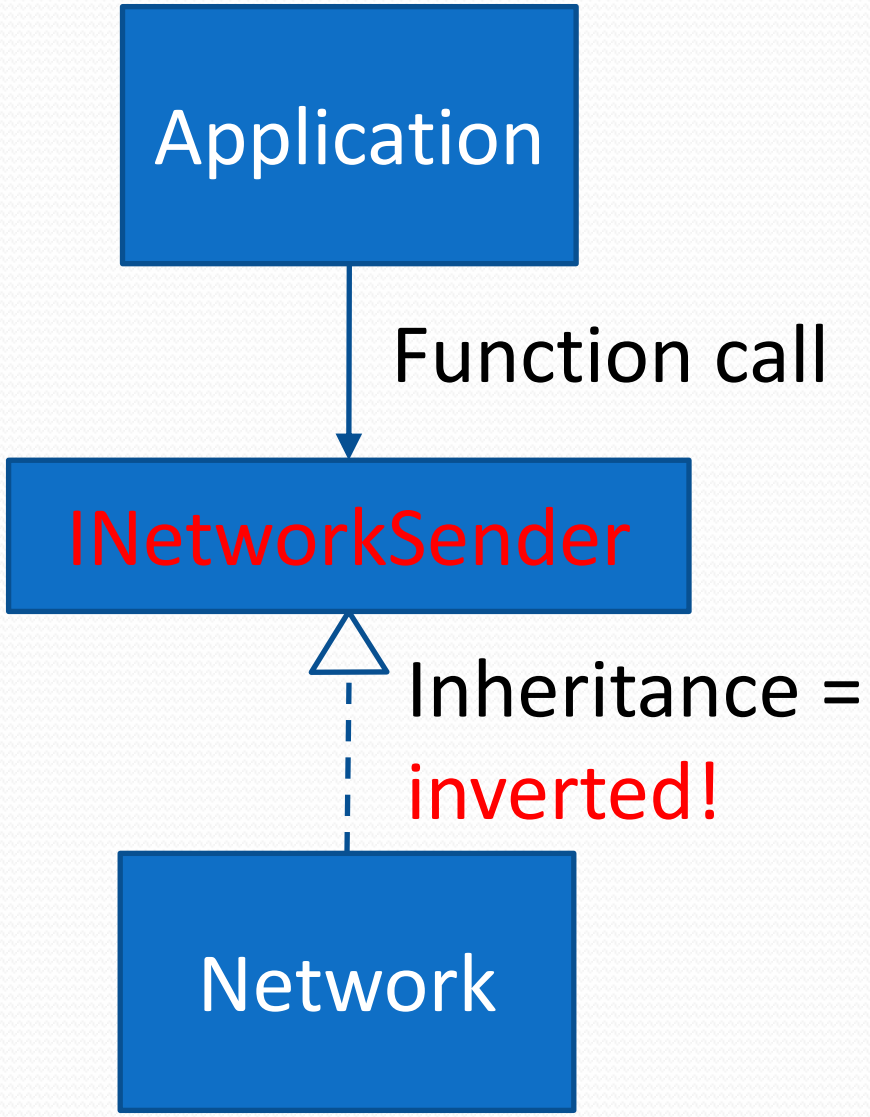
The Dependency Inversion Principle

- “Depend on abstractions, not on concretions.”
- = Application should not talk directly to the Network implementation (and vice versa)
- = Application should talk to a Network abstraction
- We will use an interface!

The Dependency Inversion Principle

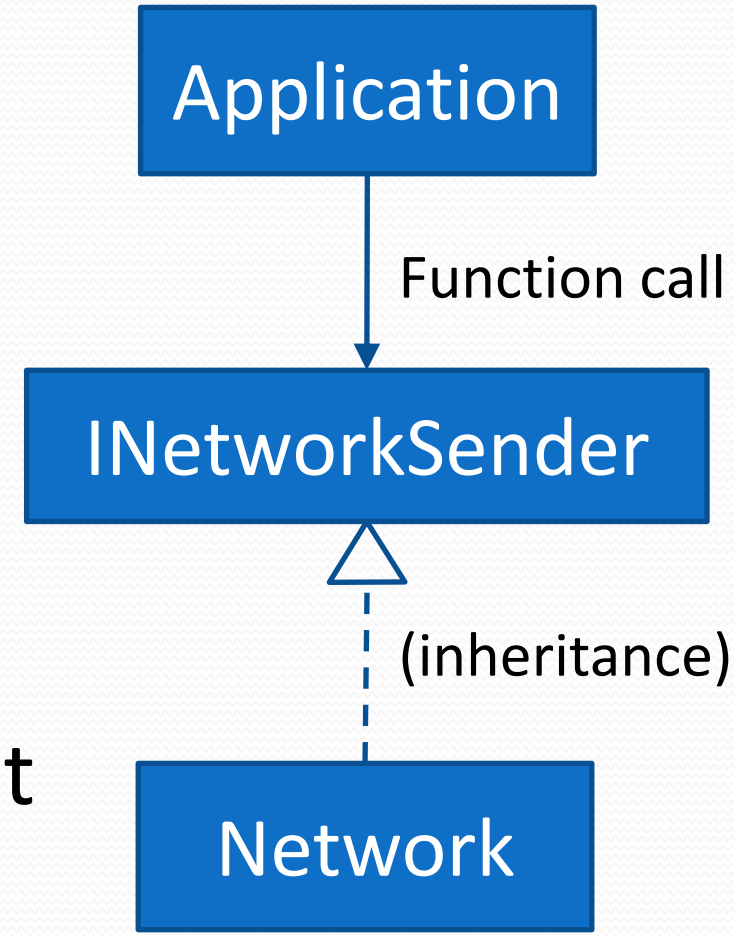
Procedural programming





Observation

- Three principles:
  - Open/closed
  - Interface segregation
  - Dependency inversion
- All give us the same result
- But for different reasons!



Interfaces

- Abstract base classes in C++

```
class INetworkSender
{
    virtual ~INetworkSender() {}
    virtual void sendData(
        const std::string &data) = 0;
};
```

Interfaces (continued)

- ```
  class INetworkReceiver
  {
      virtual ~INetworkReceiver() {}
      virtual void onDataReceived(
          const std::string &data) = 0;
  };
  ```
  ```

Example (revisited)

```
class Application :
      public INetworkReceiver
{
      void buttonPushed()
      {
            network->sendData(data);
      }
      INetworkSender *network;
};
```

Example (continued)

```
class Network : public INetworkSender
{
    void packetReceived(...)
    {
        app->onDataReceived(data);
    }
    INetworkReceiver *app;
};
```

Example (continued)

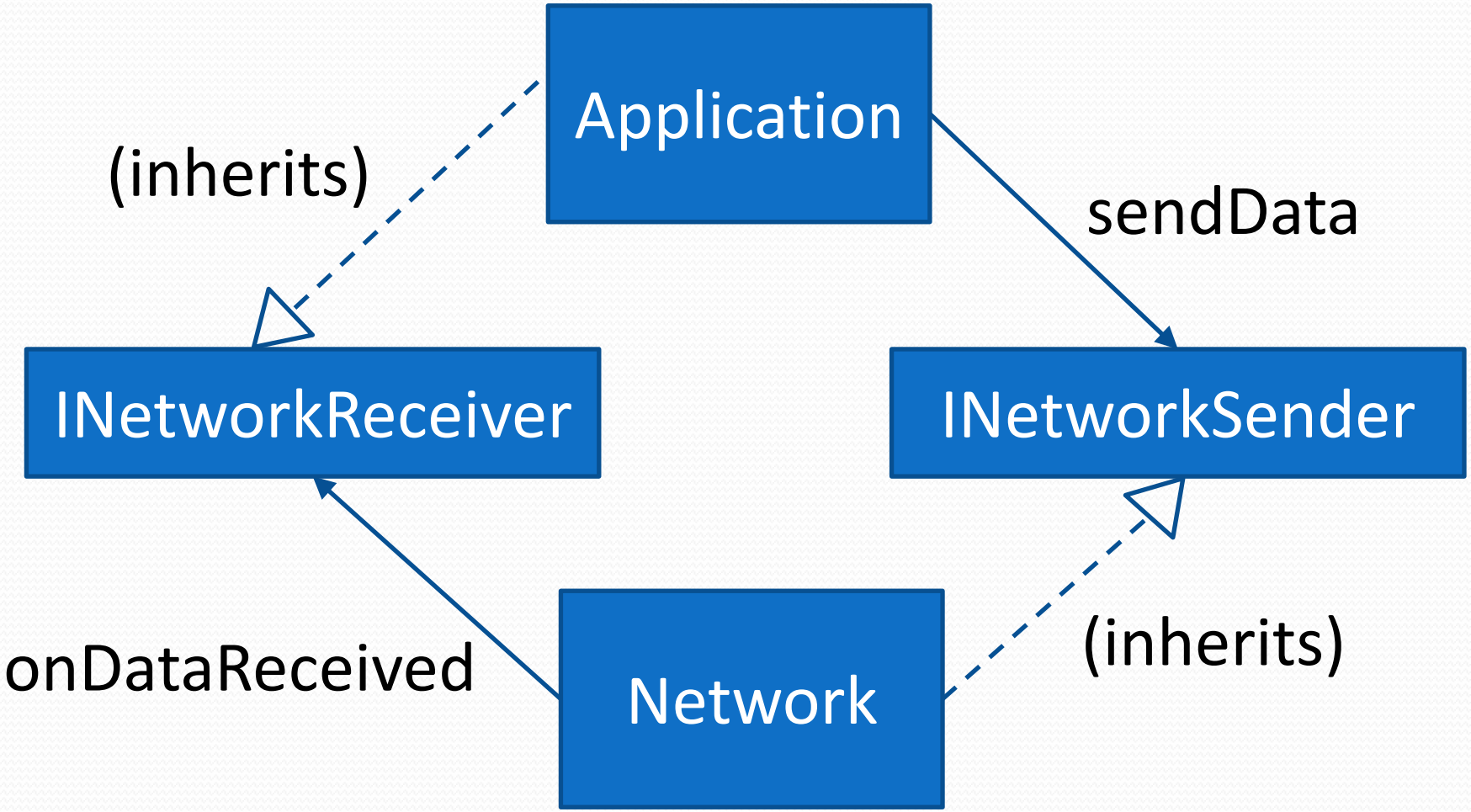
- Constructor and setter injection:

```
Network network;
// ctor takes INetworkSender!
Application app(network);
// setter takes INetworkReceiver!
network.setReceiver(app);
```

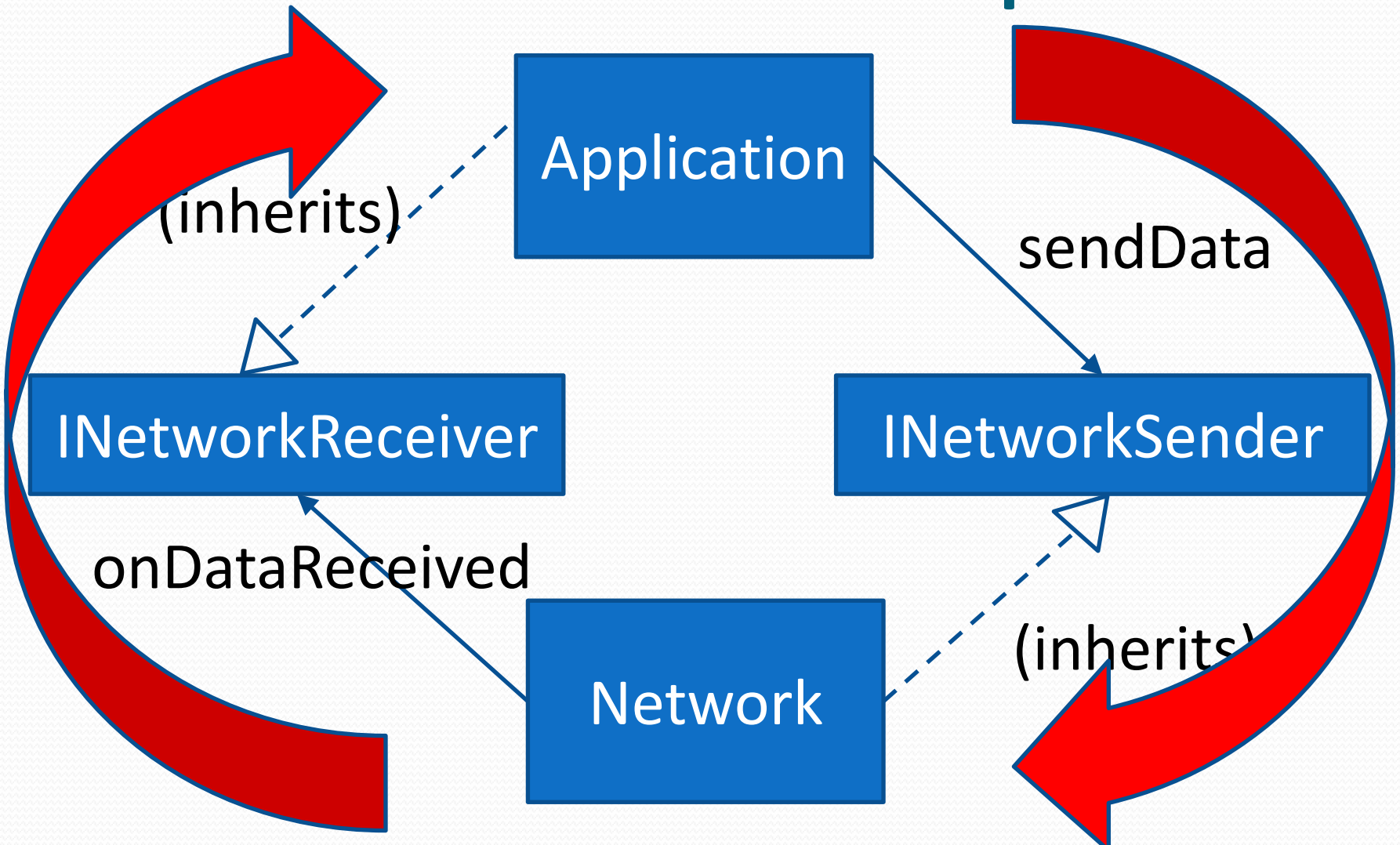
The ”Diamond”

A useful building block

The ”Diamond” shape

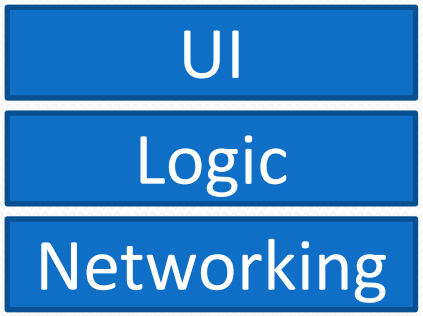


The ”Diamond” shape



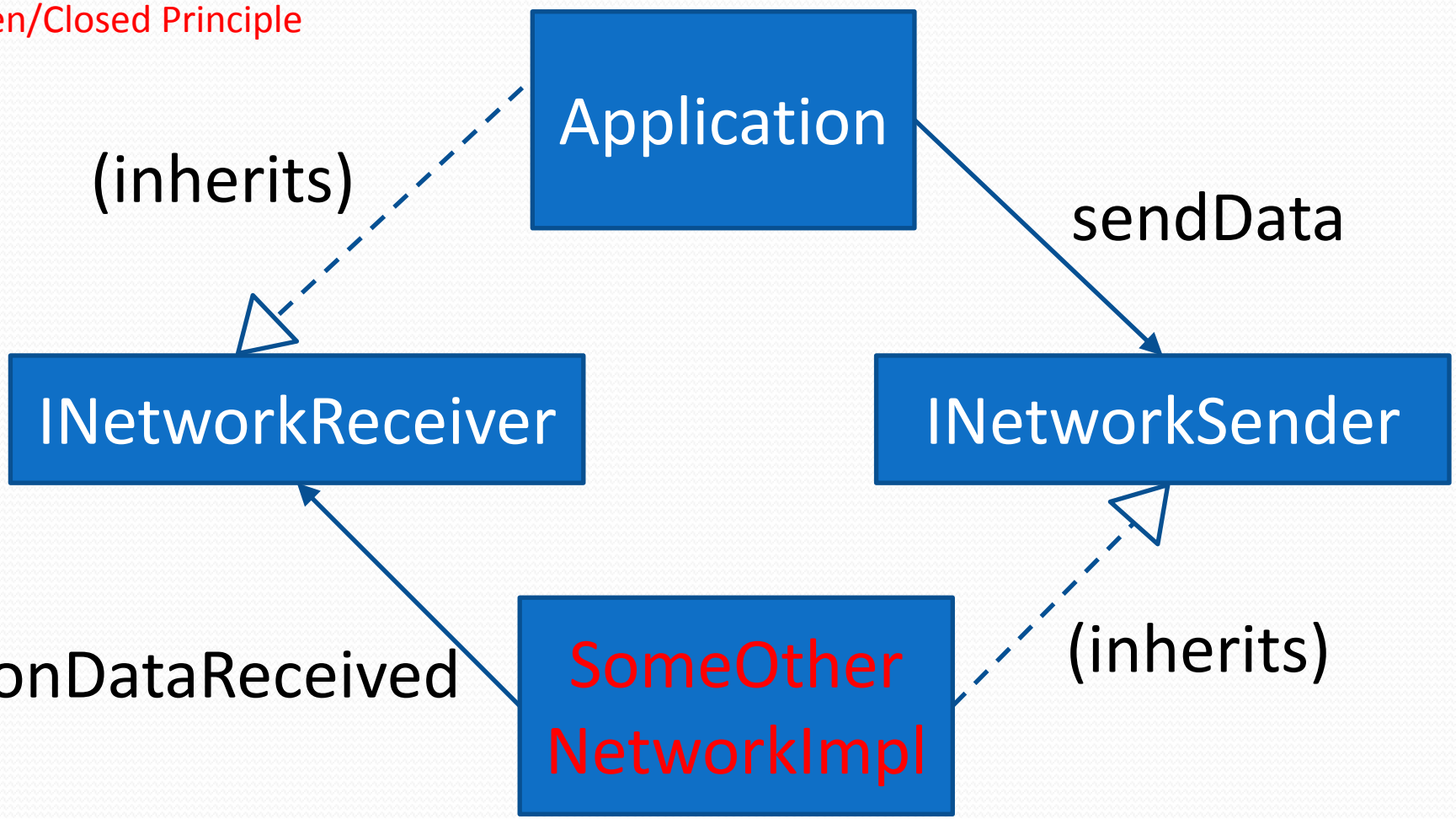
The ”Diamond”

- Fundamental building block
- **Decoupling**
  - despite bi-directional communication
- Frequently used to **layer** your application
  - Lower layers do not know of upper layers
  - Lower layers can be reused



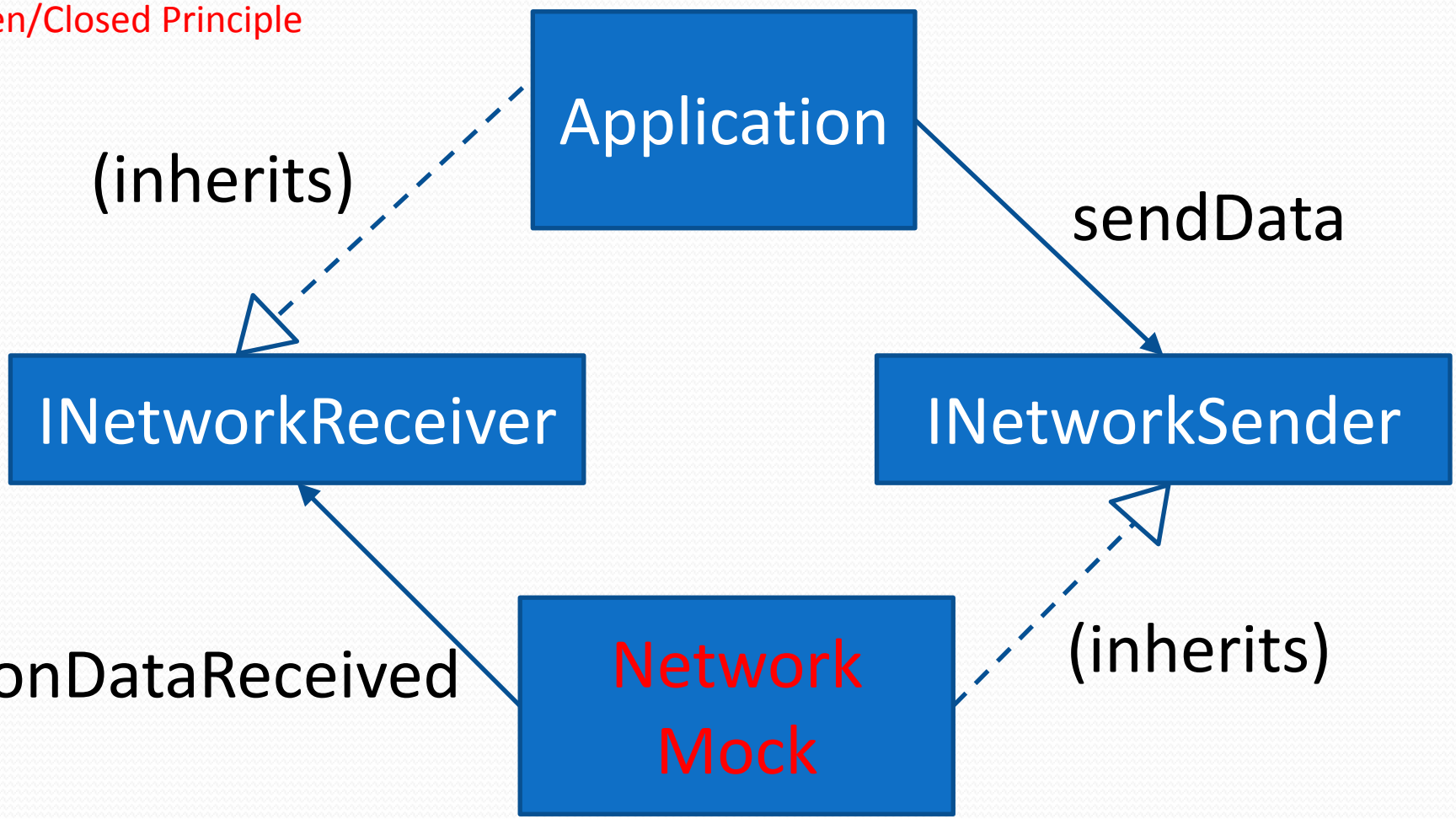
Example Revisited: "Can't easily replace network"

Open/Closed Principle

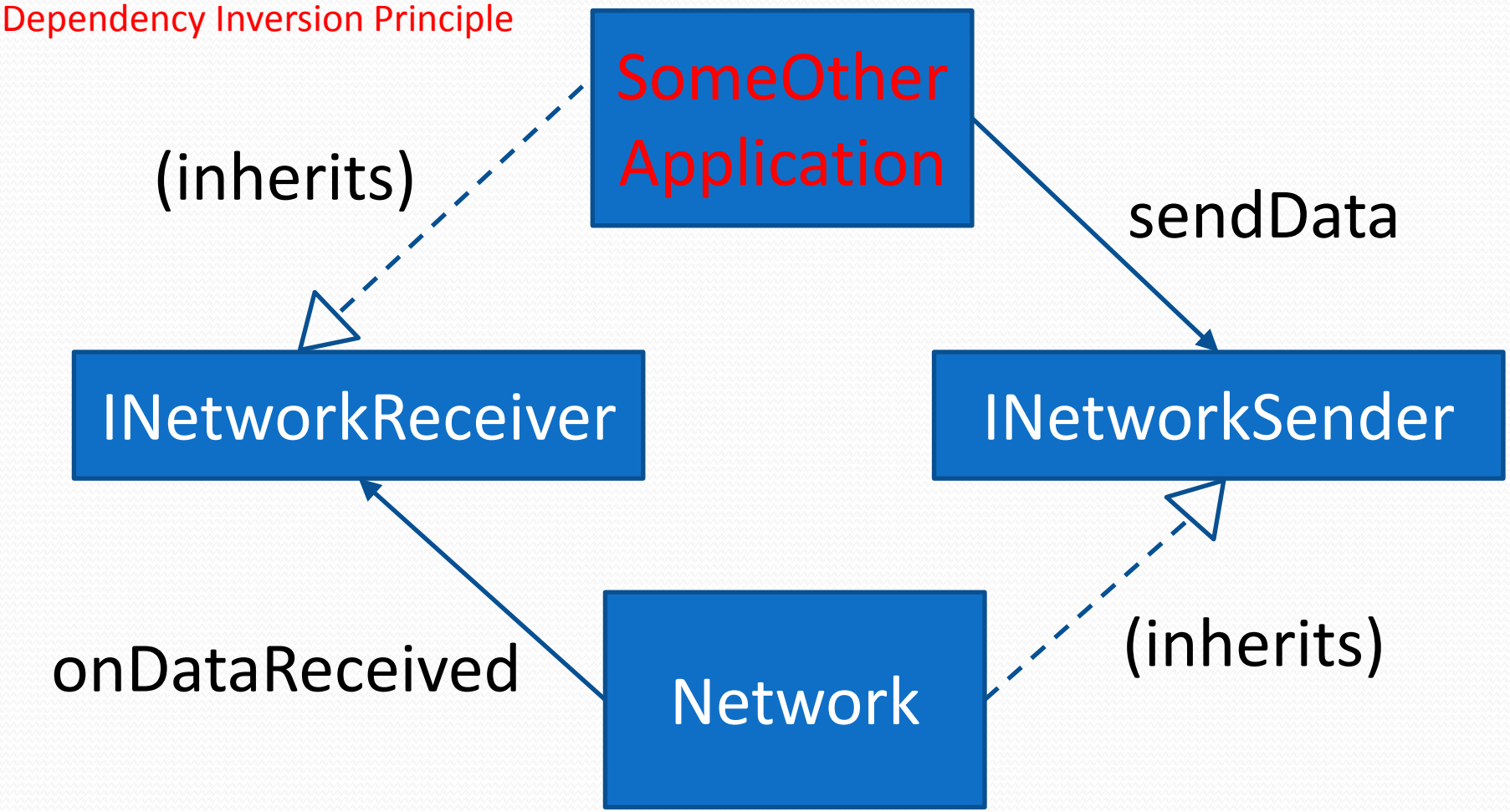


"Can't easily test application logic without network"

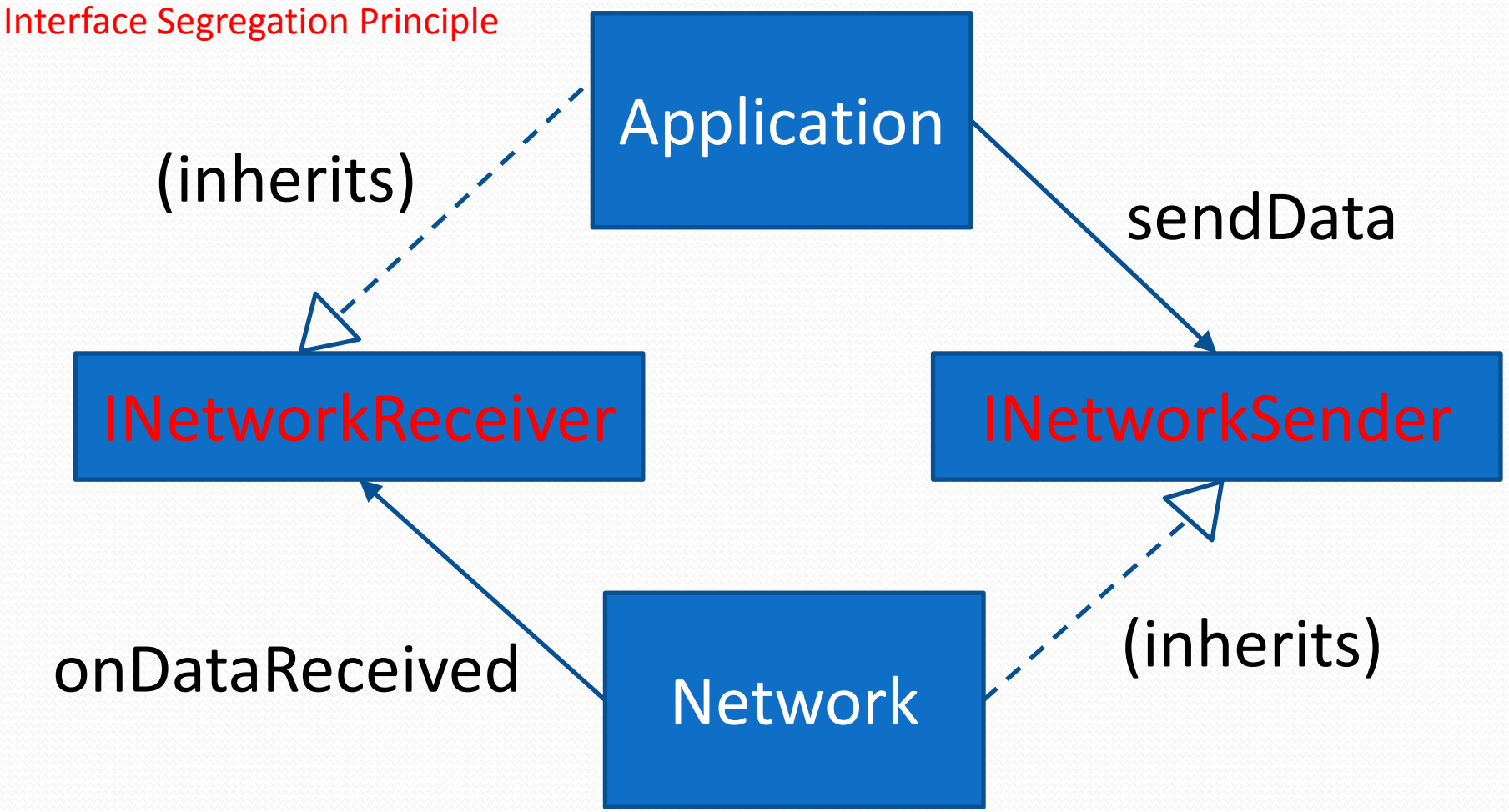
Open/Closed Principle



”Can’t reuse network”



”Hard to understand which parts are used by the other”



Testing

Testability, Test-Driven Development, Unit Testing

Testing

- What is testing?
  - Verify that our system conforms to requirements
  - Make sure the system contains no apparent bugs
- Automated testing
  - Since we need to do the above repeatedly

Testing (continued)

- Testing on what level?
  - E.g. system, module, single function
- Testing what characteristics?
  - E.g. behavior, performance, memory footprint
- Testing how?
  - E.g. from UI, over network, function calls
- We will focus on ”unit testing”

”Unit Testing”

- No clear definition/many names
  - E.g. unit testing, component testing, basic testing, …
- Definition here:
  - Testing through **function calls**:
    - Test executable by compiling test code with application code
  - Testing **behavior** (with some exceptions)
  - Testing on **function to module level**

What is Testability?

- The ability to test code at a reasonable cost
  - E.g. some network failure or timing scenarios are very hard to reliably reproduce
- **Very** expensive to introduce testability late
  - E.g. networking code everywhere
  - Legacy code = code not covered by tests

Test-Driven Development

- How do we make sure our code is testable?
- Write the test first!

- ”Growing Object-Oriented Software Guided by Tests” by Freeman, Pryce

Test-Driven Development (cont)

1. Write a failing unit test.
2. Run tests and watch it fail.
3. Write code to pass the test.
4. Run tests and watch it pass.
5. Refactor. (By definition, the test should still pass.)
6. Start over from (1).

Test-Driven Development (cont)

- Upsides:
  - All code will be testable, and covered by tests!
  - Decoupling comes natural
  - Good interfaces (your test code is a user)
- Downsides:
  - Refactoring not sufficient for large-scale architecture
  - Tedious (I know how to implement it, now let me!)

TDD Example

- ”When the user clicks the button, data ’abc’ should be sent over the network.”
- First, write a failing unit test

TDD Example (continued)

- First iteration:
  - Created and instantiated class-under-test Application
  - Added (empty) function buttonClicked()
  - Compiles but doesn't fail

```
void whenUserClicksButtonAbcIsSentOverNetwork()
{
    Application app;
    app.buttonClicked();
}
```

TDD Example (continued)

- Second iteration:
  - Added verification code
  - Success!, test fails

```
void whenUserClicksButtonAbcIsSentOverNetwork()
{
    TestNetworkSender testSender;
    Application app;
    app.buttonClicked();
    testSender.verifyCallToSend('abc');
}
```

TDD Example (continued)

- No real Network class!
  - Networking is slow and hard to control (may introduce errors, hard to reproduce errors/induce state)
- We introduced a TestNetworkSender class
  - Implements interface INetworkSender
  - Remembers when sendData is called
  - Alternatively: use mock library (e.g. googlemock, http://code.google.com/p/googlemock/)

TDD Example (continued)

- Next, write code to pass unit test!
- Hmm, we cannot get test to pass without changing the test code!

TDD Example (continued)

- Third iteration:
  - Added new Application(INetworkSender) ctor
  - Good, test still fails

```
void whenUserClicksButtonAbcIsSentOverNetwork()
{
    TestNetworkSender testSender;
    Application app(testSender);
    app.buttonClicked();
    testSender.verifyCallToSend('abc');
}
```

TDD Example (continued)

- Now we can write code to pass unit test!
  - See code from earlier
- Watch test pass
- Refactor

”Unit” Testing Modules

- A ”unit” is what you want it to be
- I like **unit tests on module/application level**
- Why?
  - Stable interfaces, internal **restructuring will not change tests**
  - Brings **business value**: clear connection between use-cases and unit tests
  - **Less waste** from testing things that will never be used etc.

Q&A

Q&A

- Skype? Microsoft? Cost/quality/time-to-market? Internal quality? Development methodology? Scrum? Lean? Software architecture? Design principles? SOLID? Diamond? Layers? Interfaces? Decoupling? Abstraction/compression? Testability? Unit testing? Test-driven development? Books?

- www.johnnybigert.se/blog