Single Responsibility Principle (SRP)

Produced by:

Eamonn de Leastar (edeleastar@wit.ie)

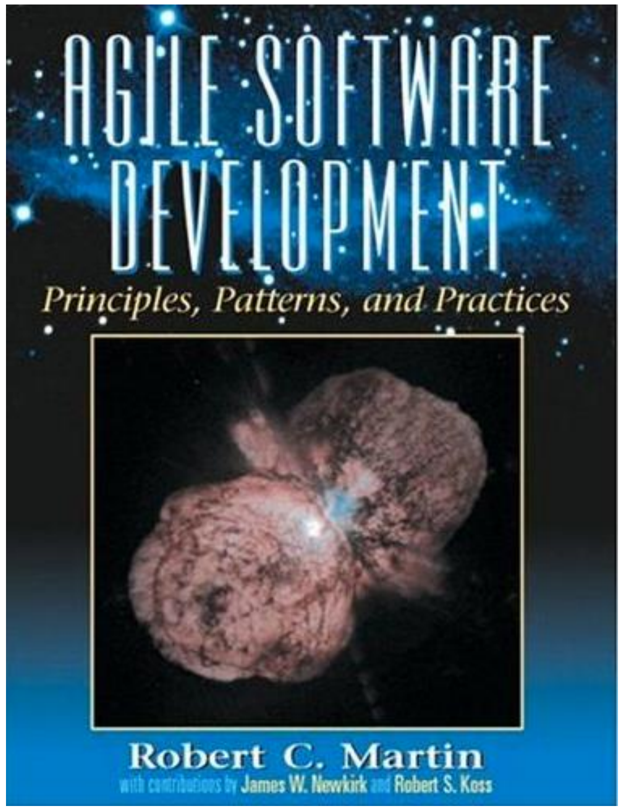
Dr. Siobhán Drohan (sdrohan@wit.ie)

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

SOLID Class Design Principles

In the mid-1990s, Robert C. Martin gathered five principles for object-oriented class design, presenting them as the best guidelines for building a maintainable object oriented system.


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

Michael Feathers attached the acronym SOLID to these principles in the early 2000s.

SOLID Class Design Principles

S	Single Responsibility Principle (SRP). Classes should have one, and only one, reason to change. Keep your classes small and single-purposed.
O	Open-Closed Principle (OCP). Design classes to be open for extension but closed for modification; you should be able to extend a class without modifying it. Minimize the need to make changes to existing classes.
L	Liskov Substitution Principle (LSP). Subtypes should be substitutable for their base types. From a client's perspective, override methods shouldn't break functionality.
I	Interface Segregation Principle (ISP). Clients should not be forced to depend on methods they don't use. Split a larger interface into a number of smaller interfaces.
D	Dependency Inversion Principle (DIP). High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

SOLID Class Design Principles – Module Scope

S Single Responsibility Principle (SRP). Classes should have one, and only one, reason to change. Keep your classes small and single-purposed.

O Open-Closed Principle (OCP). Design classes to be open for extension but closed for modification; you should be able to extend a class without modifying it. Minimize the need to make changes to existing classes.

L Liskov Substitution Principle (LSP). Subtypes should be substitutable for their base types. From a client's perspective, override methods shouldn't break functionality.

I *Interface Segregation Principle (ISP). Clients should not be forced to depend on methods they don't use. Split a larger interface into a number of smaller interfaces.*

D *Dependency Inversion Principle (DIP). High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.*

SOLID Principles in Poster form…

http://blogs.msdn.com/b/cdndevs/archive/2009/07/15/the-solid-principles-explained-with-motivational-posters.aspx

SOLID

Software development is not a Jenga game.

Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

“S” in SOLID - Single Responsibility Principle

Every object should have a single responsibility and all of its services should be aligned with that responsibility.

“Responsibility” is defined as “a reason to change”, and ***Wikipedia does a pretty good job of explaining it***:

- As an example, consider a module that compiles and prints a report. Such a module can be changed for two reasons. First, the content of the report can change. Second, the format of the report can change. These two things change for very different causes; one substantive, and one cosmetic. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

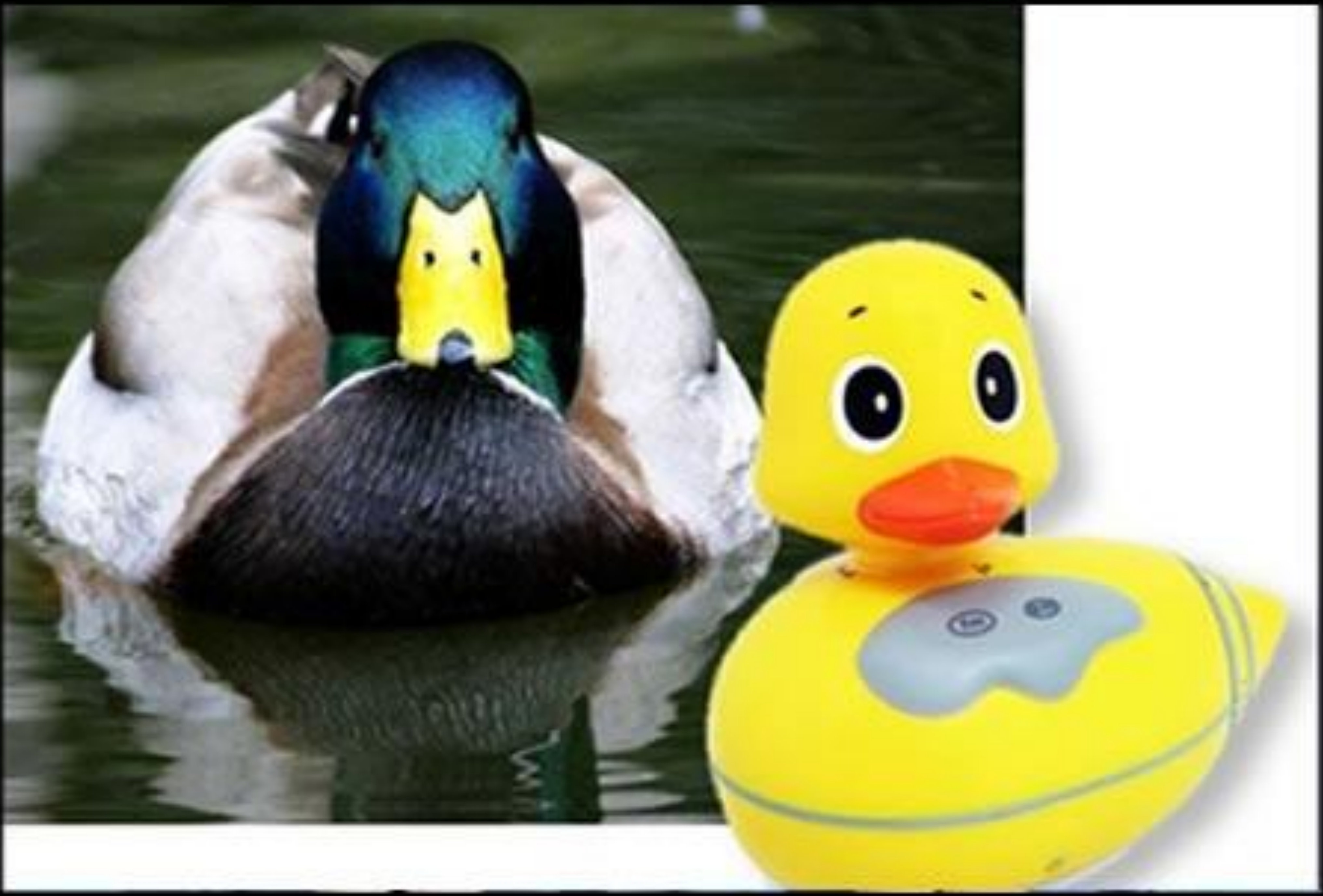
Open-Closed Principle

Open-chest surgery isn't needed when putting on a coat.

“O” in SOLID - Open-Closed Principle

- Software entities – such as classes, modules, functions and so on – should be open for extension but closed for modification.

- The idea is that it’s often better to make changes to things like classes by adding to or building on top of them (using mechanisms like subclassing or polymorphism) rather than modifying their code.



Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

“L” in SOLID - Liskov Substitution Principle

- Subclases should be substitutable for the classes from which they were derived.
- For example, if MySubclass is a subclass of MyClass, you should be able to replace MyClass with MySubclass without bunging up the program.

Interface Segregation Principle

You want me to plug this in *where?*

“I” in SOLID - Interface Segregation Principle

Many client specific interfaces are better than one general purpose interface.

Make fine grained interfaces that are client specific.

outside the scope of this module

DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

“D” in SOLID - Dependency Inversion Principle

- High-level modules shouldn’t depend on low-level modules, but both should depend on shared abstractions.

- In addition, abstractions should not depend on details – instead, details should depend on abstractions.

Depend on abstractions, not on concretions.

outside the scope of this module

- In your own time, see the Manager example here.

SRP – Single Responsibility Principle

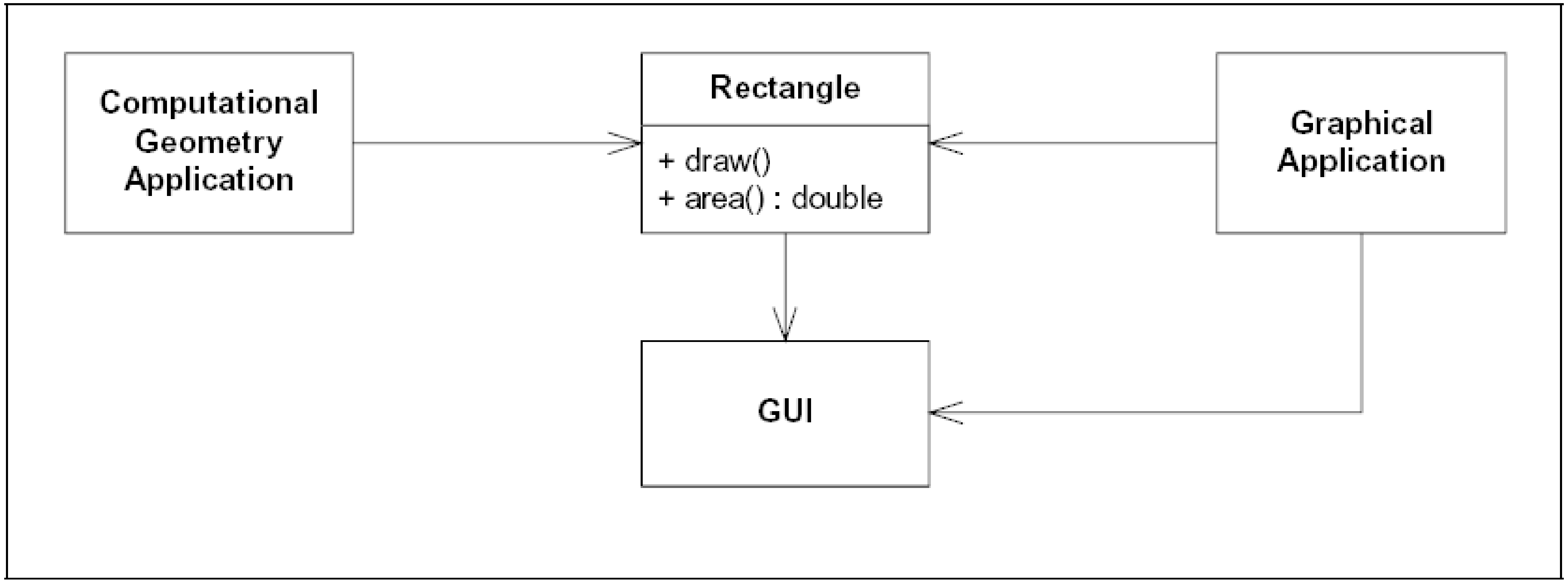
SRP: The Single Responsibility Principle

THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE.

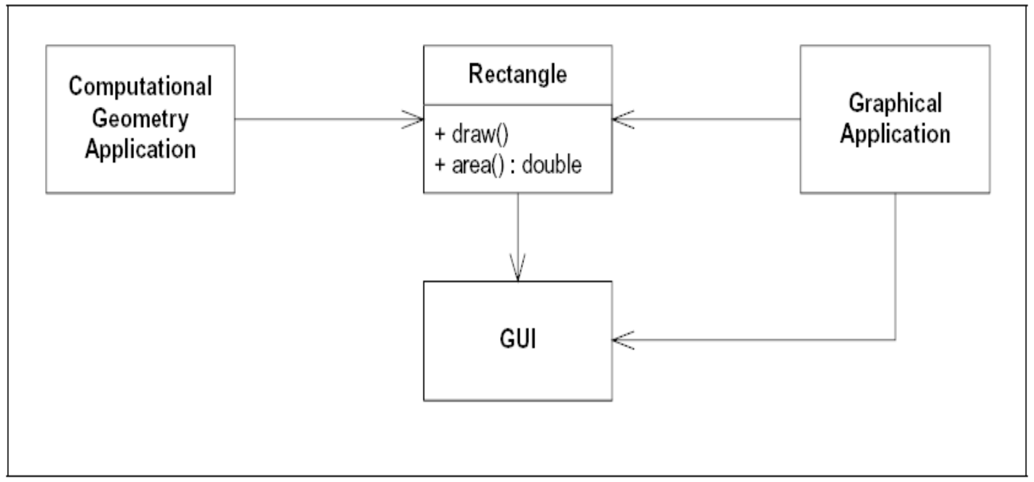
- Each responsibility is an axis of change.
- When requirements change
  → a change in responsibility amongst the classes.
- If a class assumes more than one responsibility
  → more than one reason for it to change.
  → changes to one responsibility may impair or inhibit the class’ ability to meet the others.

Rectangle example

- The Rectangle class has two methods:
  - one draws the rectangle on the screen
  - the other computes the area of the rectangle.
- Two applications use this class:
  - one application uses Rectangle to help it with the mathematics of geometric shapes.
  - the other uses the class to render a Rectangle on a window.



SRP Violation

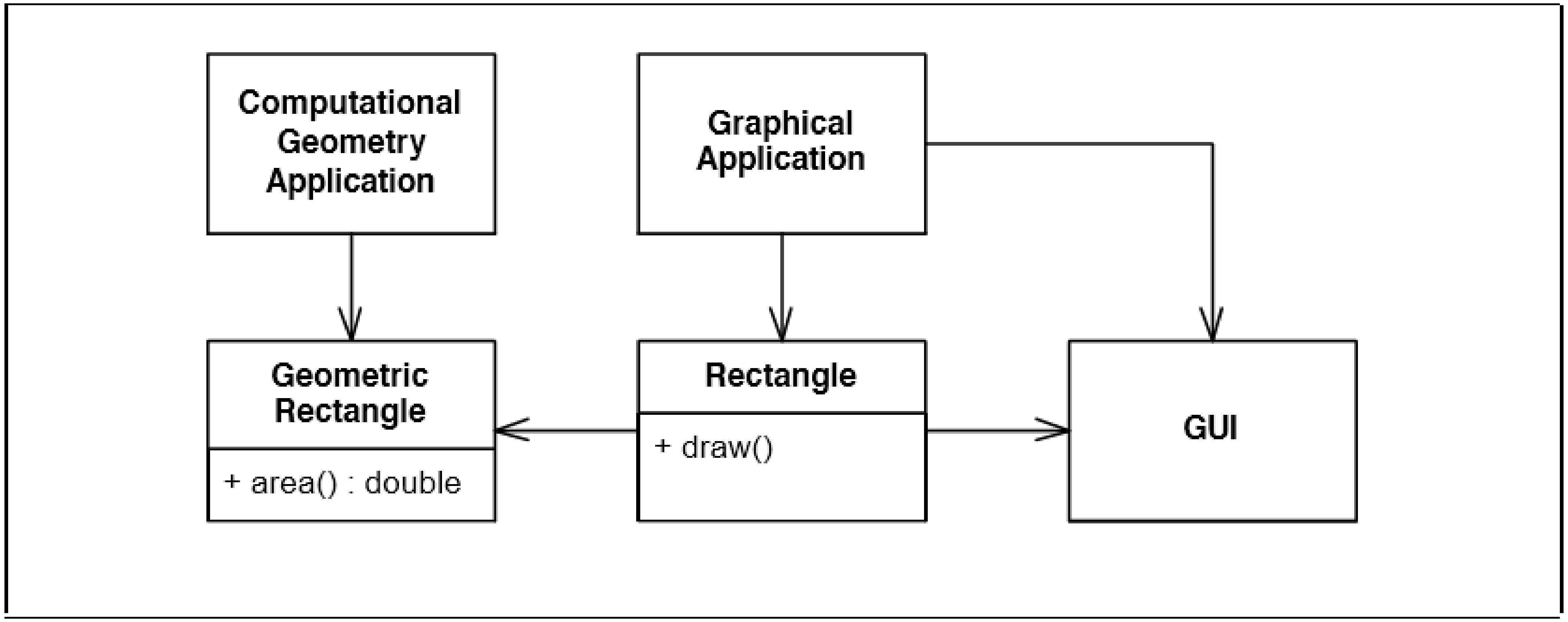


- Rectangle has two responsibilities:
  - provide a mathematical model of the geometry of a rectangle.
  - render the rectangle on a graphical user interface.

- Violation of SRP:
  - the GUI must be included in the in the computational geometry application.
    - the class files for the GUI have to be deployed to the target platform.
  - if a change to the Graphical Application causes the Rectangle to change for some reason, that change may force us to rebuild, retest, and redeploy the Computational Geometry Application.

SRP Fix

- Separate the two responsibilities into two separate classes
  - Moves the computational portions of Rectangle into the GeometricRectangle class.
- Now changes made to the way rectangles are rendered cannot affect the ComputationalGeometry Application.



What is a Responsibility?

- “A reason for change.”
- If you can think of more than one motive for changing a class, then that class has more than one responsibility.

```
interface Modem
{
  void dial(String pno);
  void hangup();
  void send(char c);
  char recv();
}
```

Modem Responsibilities

```
interface Modem
{
  void dial(String pno);
  void hangup();
  void send(char c);
  char recv();
}
```

- Two responsibilities:
  - connection management (dial and hangup functions)
  - data communication (send and recv functions)
- They have little in common
  - may change for different reason
  - will be called from different parts of the applications
- They will change for different reasons.

Should the responsibilities be separated?

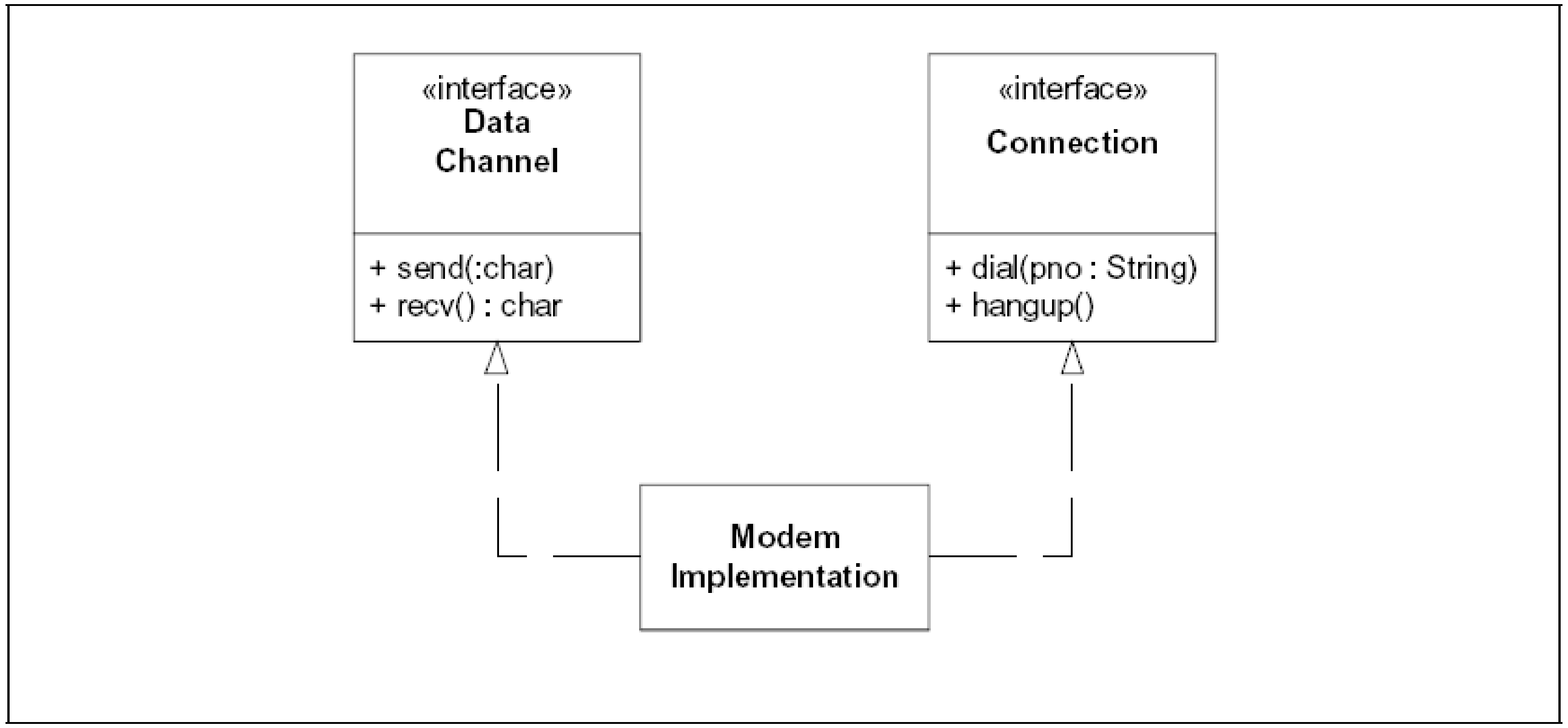
- It depends!

- How do you forsee the application changing?
  - e.g. could the signature of the connection methods potentially change, without any change to the send/receive mechanism?

```
interface Modem
{
  void dial(String pno);
  void hangup();
  void send(char c);
  char recv();
}
```

Should the responsibilities be separated?

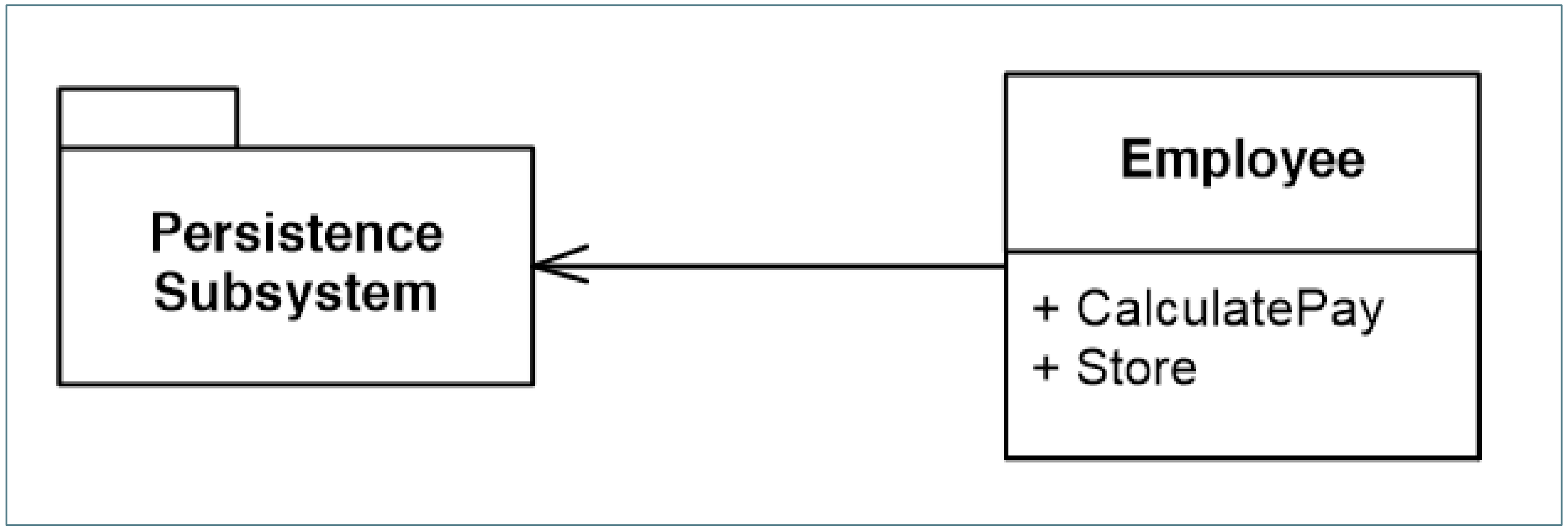
- If the application can change in ways that cause the two responsibilities to change at different times → separate the responsibilities.
- Separation here is at interface level and not class level.



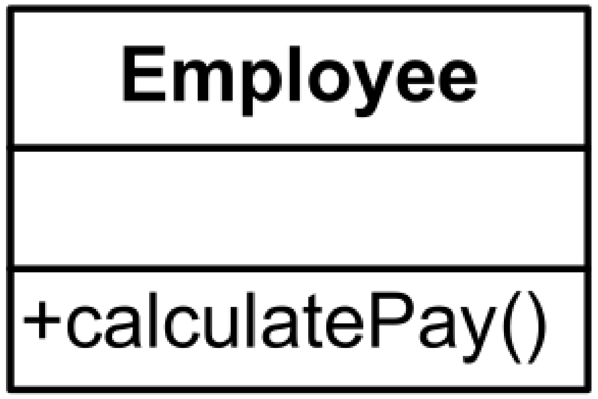
- *CAUTION*: ***Needless complexity*** can occur when there is no ***foreseeable*** need to separate the responsibilities.

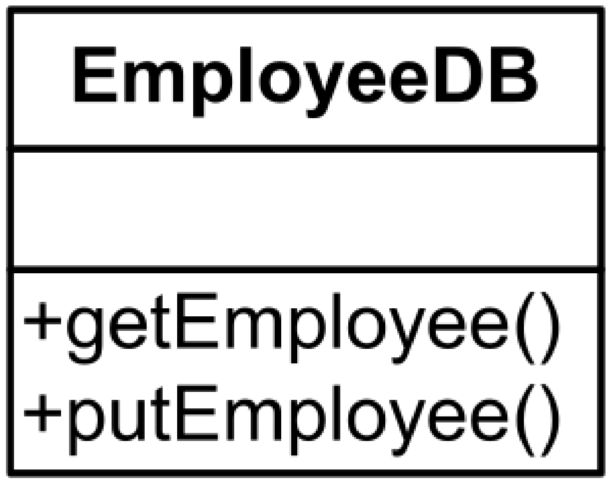
Another SRP Violation

- Coupling persistence services (store) with business rules (calculatePay) violates SRP.



Separate the Responsibilities



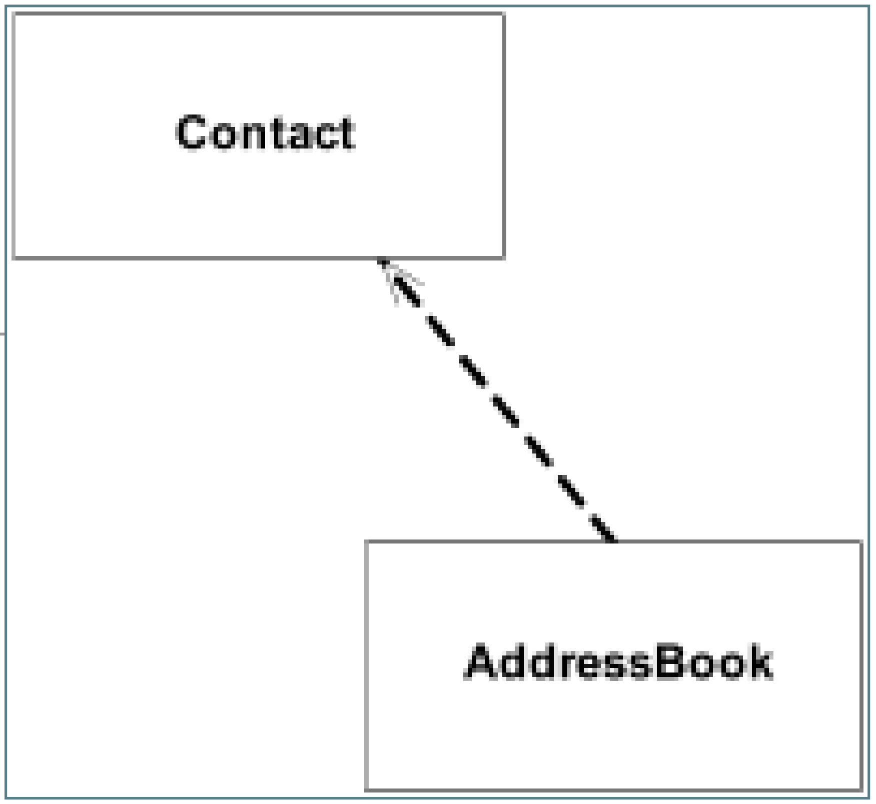


Example - Personal Information Manager

- Design an Application to manage a contact list.

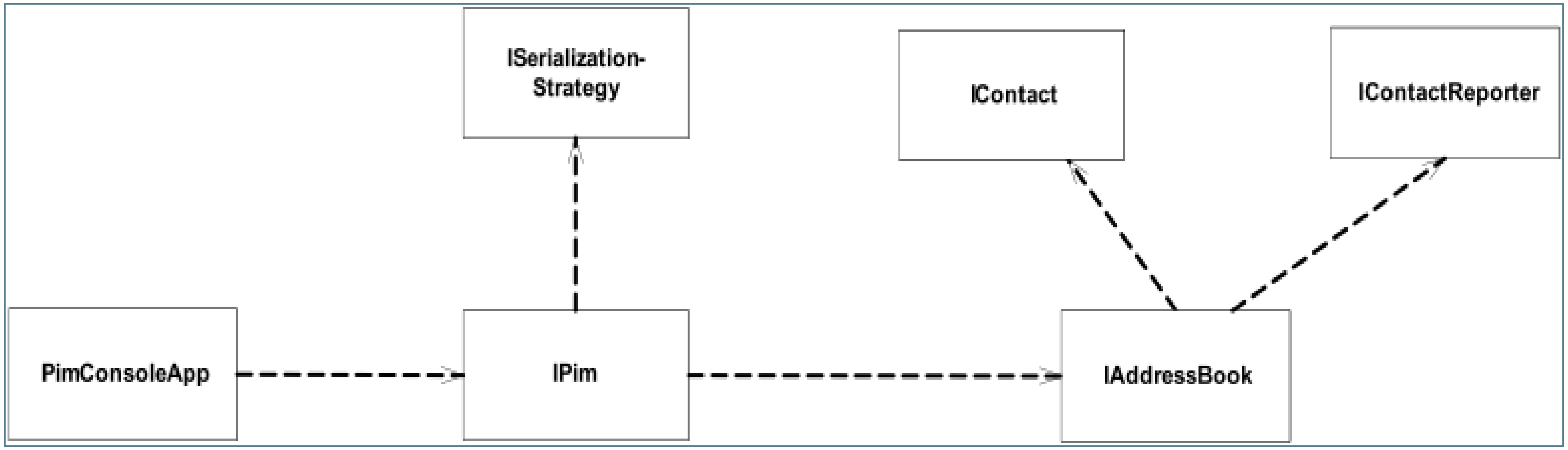
- It should support:
    - Console based UI
    - Load/save to/from a file on disk
    - Simple reports and search functions

AddressBook



- Propose two classes:
  - Contact - to represent each contact
  - AddressBook - to incorporate
    - serialization
    - reporting
    - UI
    - etc…
- Violates SRP as AddressBook has multiple reasons to change
  - Data structure change (e.g. HashMap to TreeMap)
  - Serialization mechanism (e.g. binary to XML)
  - Alternative reports (e.g. different formats and content)
  - Command line syntax changes

Refactor Addressbook

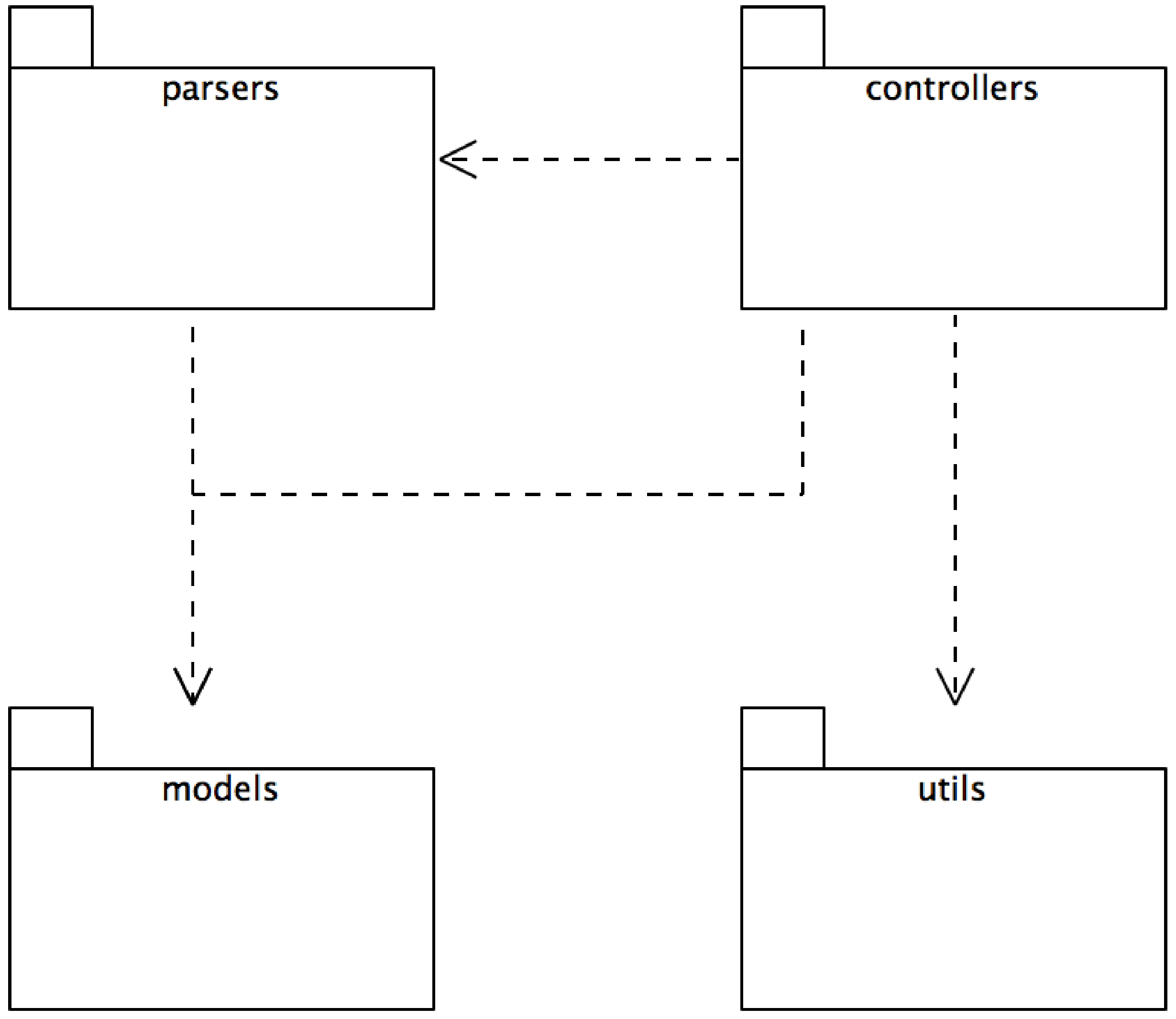


IAddressBook	responsible for contact data structure
IContactReporter	responsible for format and content of reports
ISerializationStrategy	responsible for persistence
IPim	responsible for binding address book to serialization mechanism – and for exposing coherent top level functionality
PimConsoleApp	responsible binding an running application to an IPim.

Pacemaker - package responsibilities

transform the model into various formats

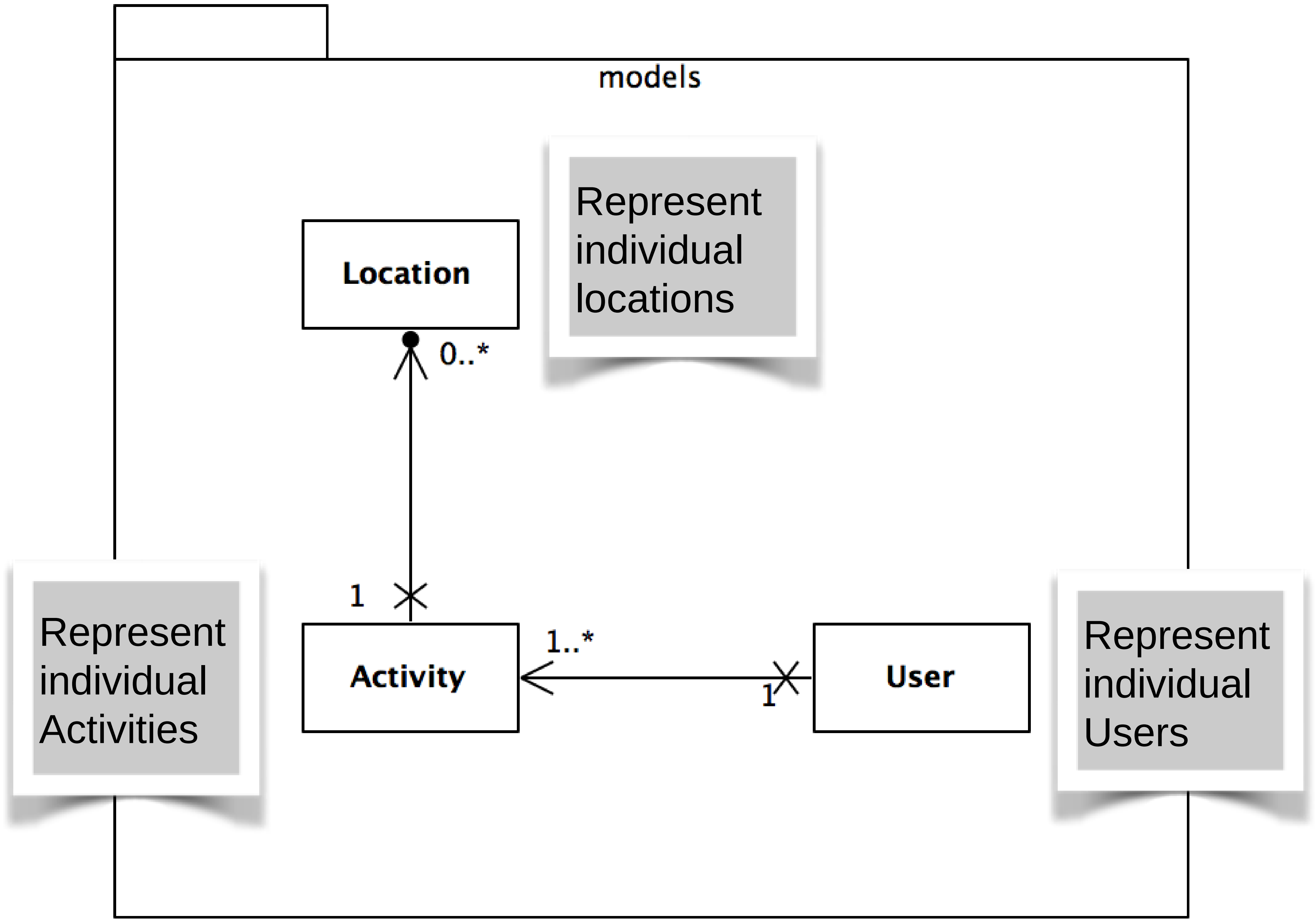
Application services + user interface



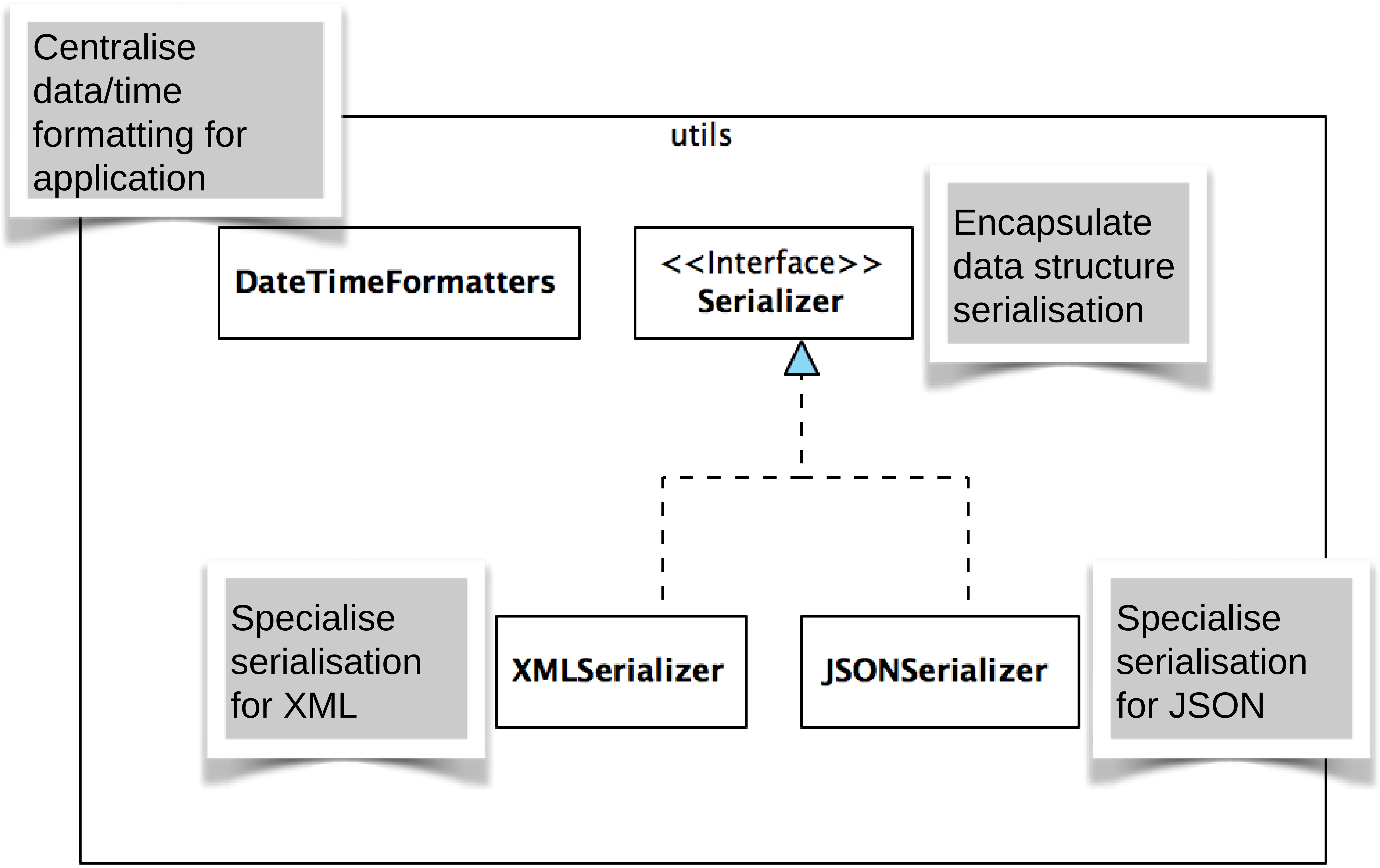
information model for the app

general purpose application independent utilities

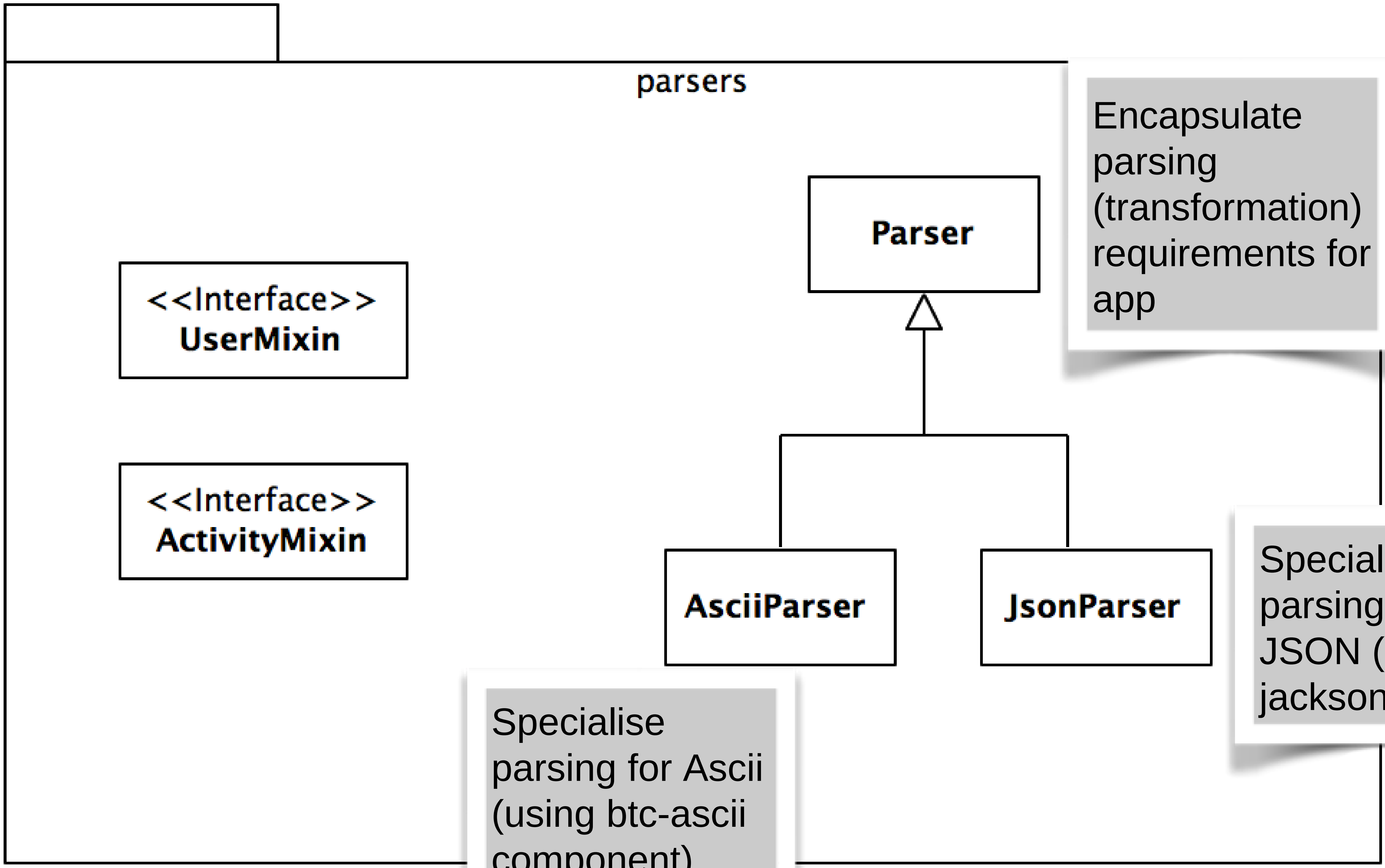
Pacemaker – model responsibilities


models
Location
0..*
1
Activity
1..*
1
User
Represent individual locations
Represent individual Activities
Represent individual Users

Pacemaker – utils responsibilities


Centralise data/time formatting for application
utils
DateTimeFormatters
<<Interface>> Serializer
Encapsulate data structure serialisation
Specialise serialisation for XML
XMLSerializer
JSONSerializer
Specialise serialisation for JSON

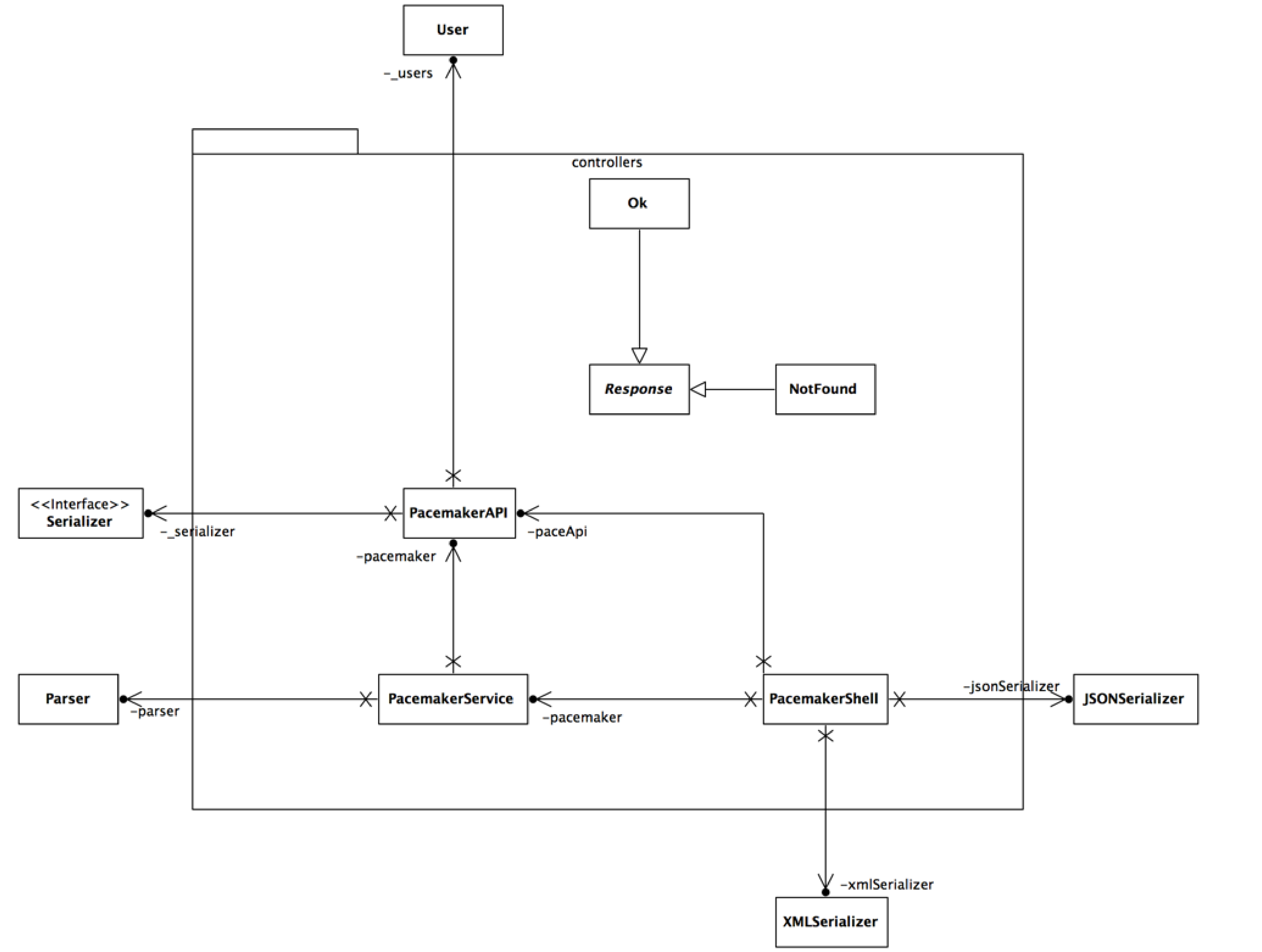
Pacemaker – parsers responsibilities



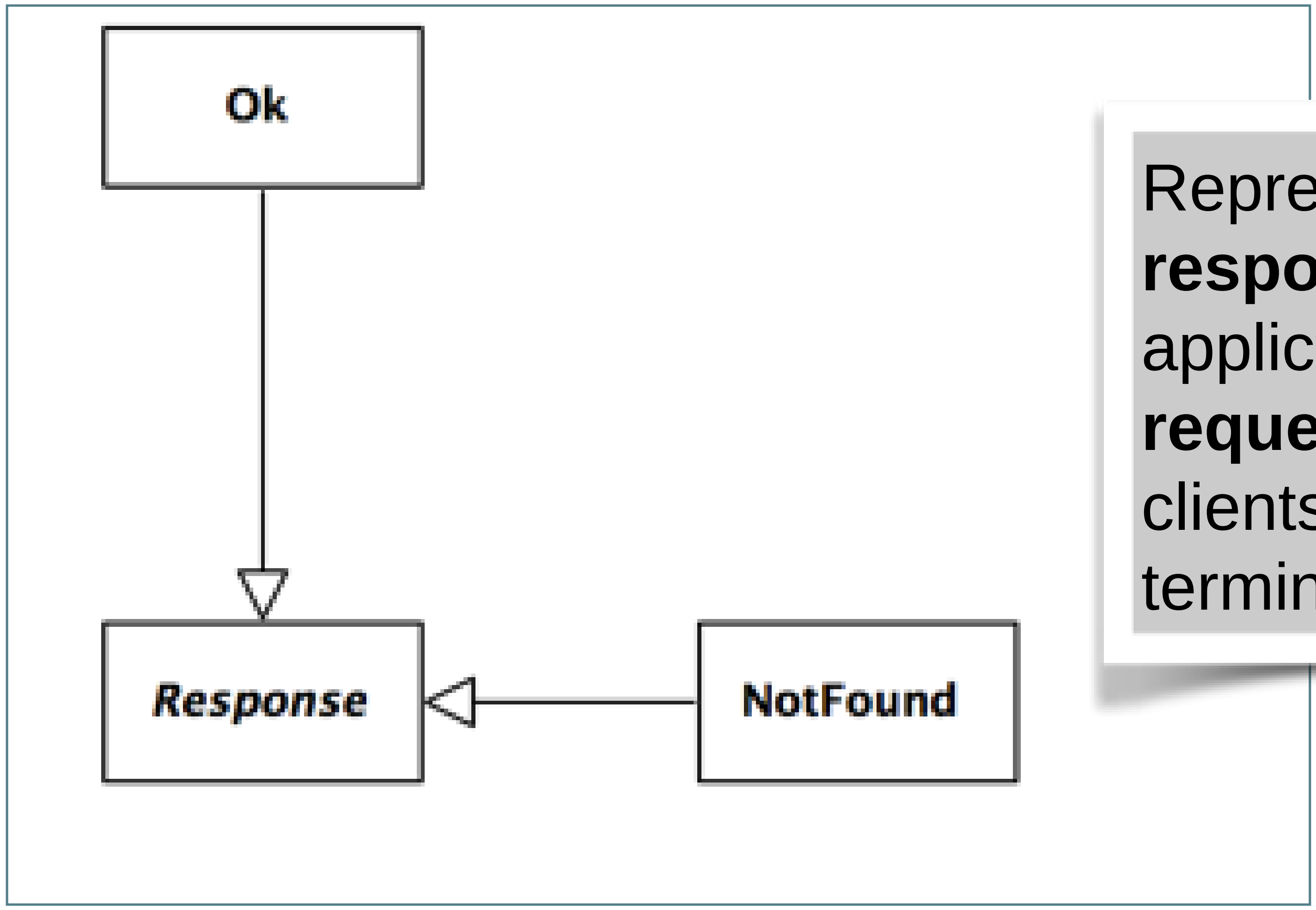
Encapsulate parsing (transformation) requirements for app

Specialise parsing for JSON (using jackson)

Specialise parsing for Ascii (using btc-ascii component)

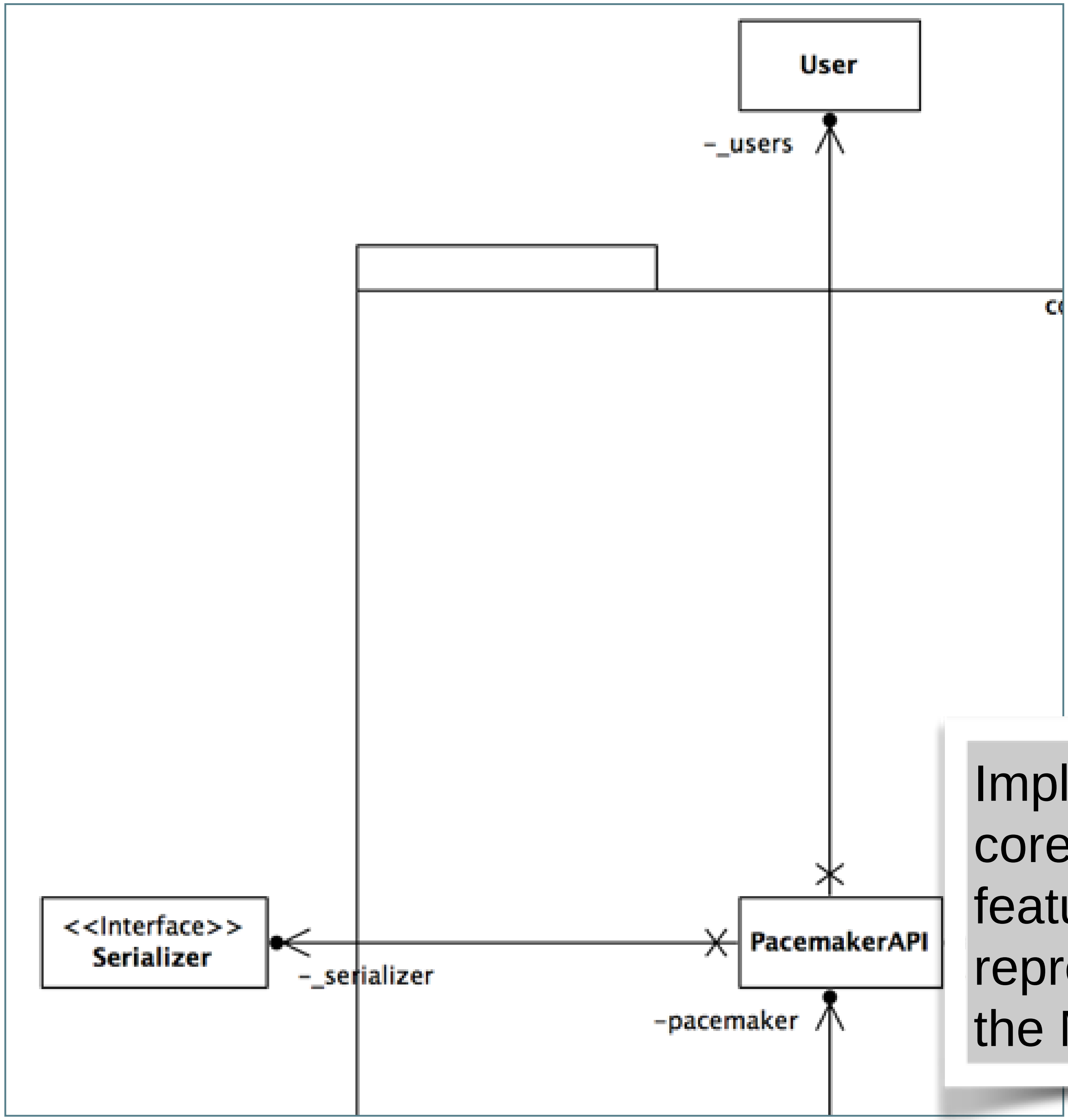

User
-_users
controllers
Ok
Response
NotFound
<<Interface>>
Serializer
-_serializer
PacemakerAPI
-paceApi
-pacemaker
Parser
-parser
PacemakerService
-pacemaker
PacemakerShell
-jsonSerializer
JSONSerializer
-xmlSerializer
XMLSerializer

Pacemaker – Response responsibilities



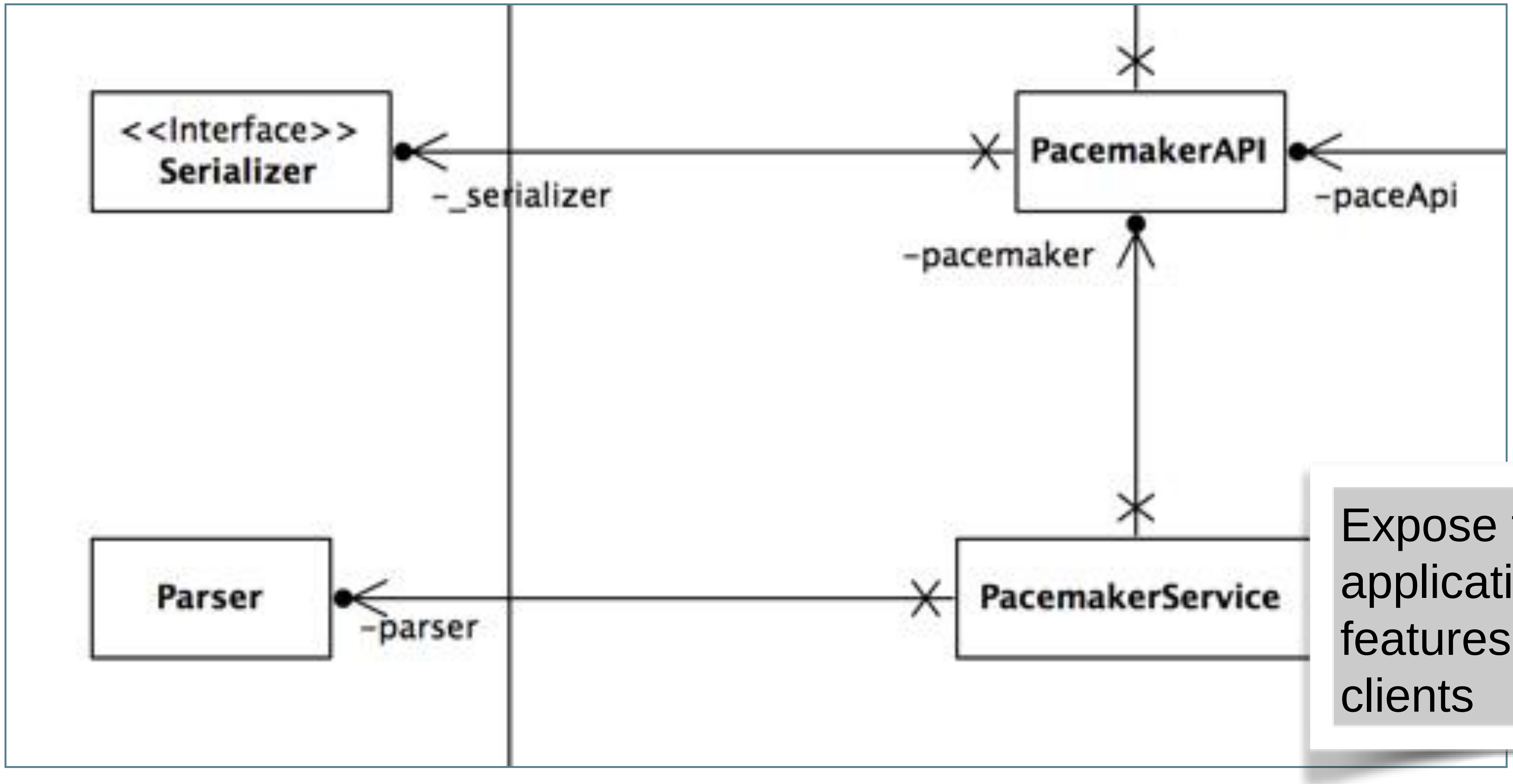
Represent **responses** from an application to **requests** from clients (use HTTP terminology)

Pacemaker – PacemakerAPI responsibilities



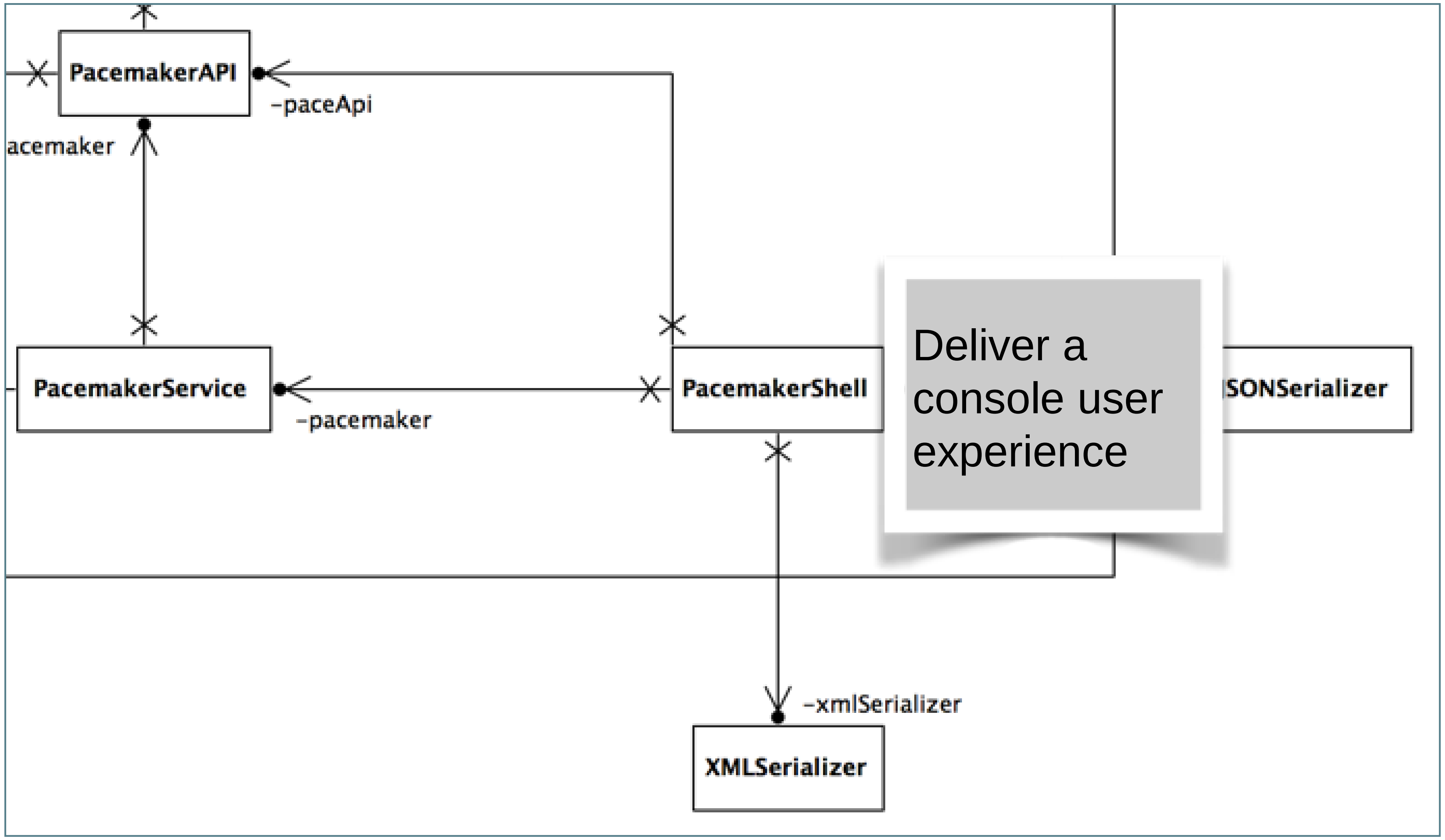
Implement the core application features as represented by the Model.

Pacemaker – PacemakerService responsibilities



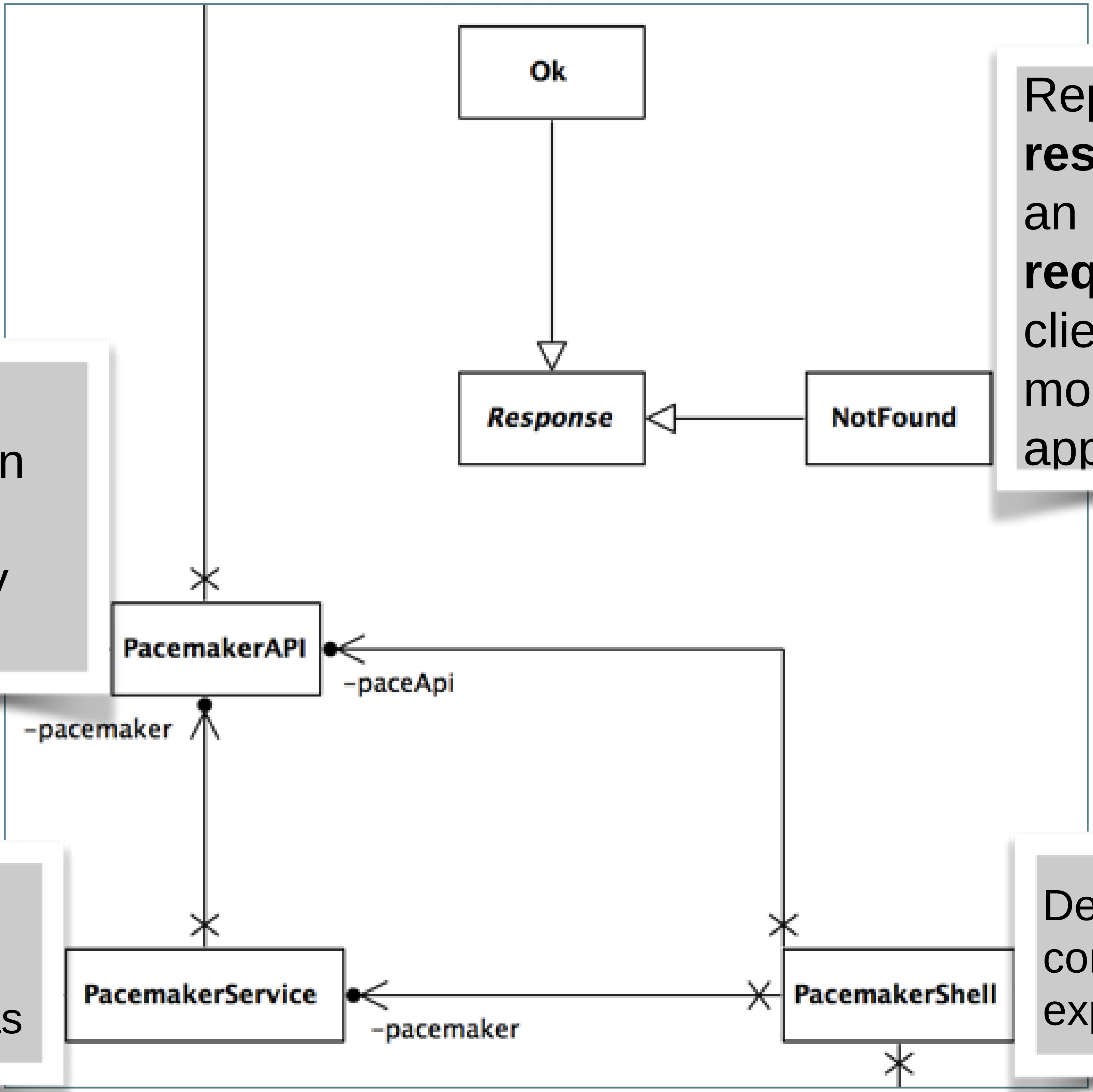
Expose the core application features to clients

Pacemaker – PacemakerShell responsibilities



Deliver a console user experience

Pacemaker – controllers responsibilities



Represent **responses** from an application to **requests** from clients(information modelled for the app on HTTP)

Implement the core application features as represented by the Model.

Expose the core application features to clients

Deliver a console user experience

SRP Summary

- Changes in requirements are manifested as changes in class responsibilities.
- Therefore a ‘cohesive’ responsibility is a single axis of change – requirement changes often are restricted to a few cohesive responsibilities (in a reasonably designed system).
- Thus, to avoid coupling responsibilities that change for different reasons, a class should have only one responsibility, one reason to change.
- Violation of SRP causes spurious dependencies between modules that are hard to anticipate, in other words fragility.

Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit