Software Engineering 301:
Software Analysis and Design

Evolvability

Agenda

- Basic concepts
- Evolvability in standard object-orientation
- Evolvability in APIs
- Frameworks
- Qualitative evolvability analysis

What does evolvability mean?

- "The ability to be evolved"
  - "That's correct in a way that demonstrates complete lack of understanding of the question"
- Potential to respond to the pressures to change with minimal modifications
  - bug fixes
  - enhancements
  - refactoring
  - porting

Agenda

- ~~Basic concepts~~
- Evolvability in standard object-orientation
- Evolvability in APIs
- Frameworks
- Qualitative evolvability analysis

Evolvability in standard OO

- Addition of subclasses accommodated
  - Effects of overriding methods?
  - Comprehensibility of system?
    - Lots of “look-alike” classes, like the subtypes of java.util.Queue (often worse than that): usability problems
  - Some part needs to know about new subclasses
- Changes to method implementations possible
  - Little need to worry about effects on subclasses, other methods

Evolvability in standard OO

- Addition of superclasses usually OK
- Movement of methods up the class hierarchy usually OK
- These might break some special cases where reflection is used
  - i.e., if you check what the superclass of a class is, or which class declares a method

Evolvability in standard OO

- Addition of concrete methods is usually possible
  - Inherited by subclasses without need to change them
  - Attempts by subclass to add an abstract method of same signature would cause trouble
  - Depends on independent compilation properties of language in use
  - What if subclasses already implement the method?
- Addition of abstract methods & deletion of methods are problematic
  - Any dependent might break, including subclasses and clients
  - Widespread changes, including of customer code

Agenda

- ~~Basic concepts~~
- ~~Evolvability in standard object-orientation~~
- Evolvability in APIs
- Frameworks
- Qualitative evolvability analysis

APIs

- API = application programming interface
- An API is provided by a piece of software
  - The API abstracts away the implementation of the software
- An API is used by other pieces of software
  - The two pieces of software interact via the API
  - The API acts as a contract between them

API content

- What is in an API?
  - Types (classes, interfaces, etc.)
  - Methods and fields
- These will generally need to be public or protected to be considered API
- Sometimes:
  - Documentation (i.e., JavaDoc) will specify more details
  - Preconditions, postconditions, invariants
  - Any other constraints API consumers can expect

Language support for APIs

- Java has visibility modifiers, method signatures
- Until recently, no compiler support for contracts
  - Now, annotations can be used
- Visibility modifiers are not always enough
  - Eclipse has additional rules about what is or is not “API” within its own source code
  - E.g., anything in a package “internal” is NOT API

API evolution

- In general, APIs are intended to never change in a non-backwards compatible fashion
- E.g., Eclipse does not permit APIs to change in a non-backwards compatible way, except across major release boundaries
  - Even then, they try to avoid it
- Problem: APIs sometimes need to change
  - New features require new designs that require different APIs
  - Problems with old APIs (bugs, performance issues)
- @deprecated

Agenda

- ~~Basic concepts~~
- ~~Evolvability in standard object-orientation~~
- ~~Evolvability in APIs~~
- Frameworks
- Qualitative evolvability analysis

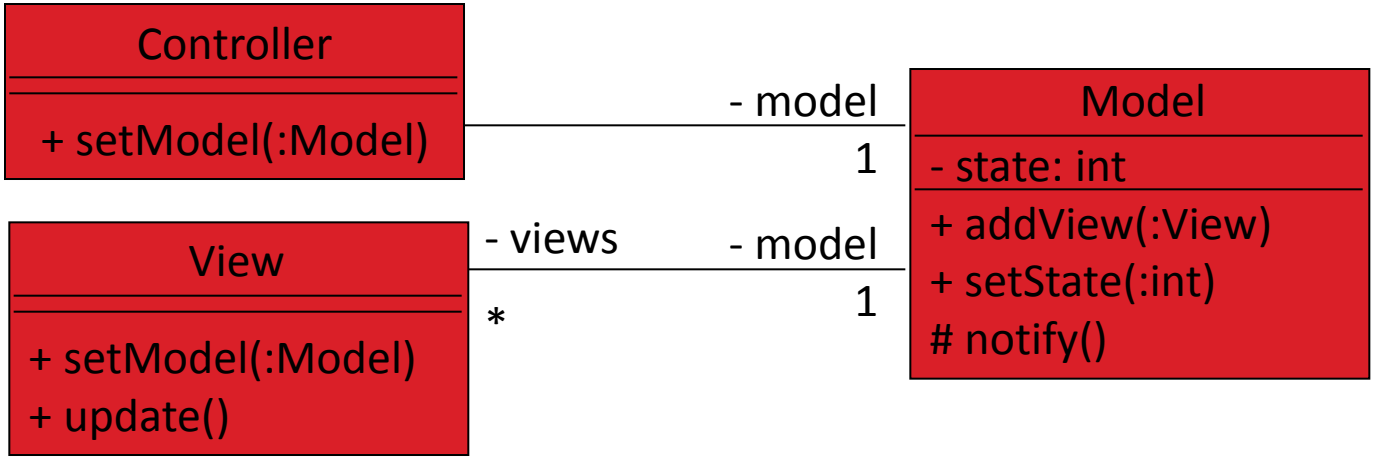
Planned extensibility

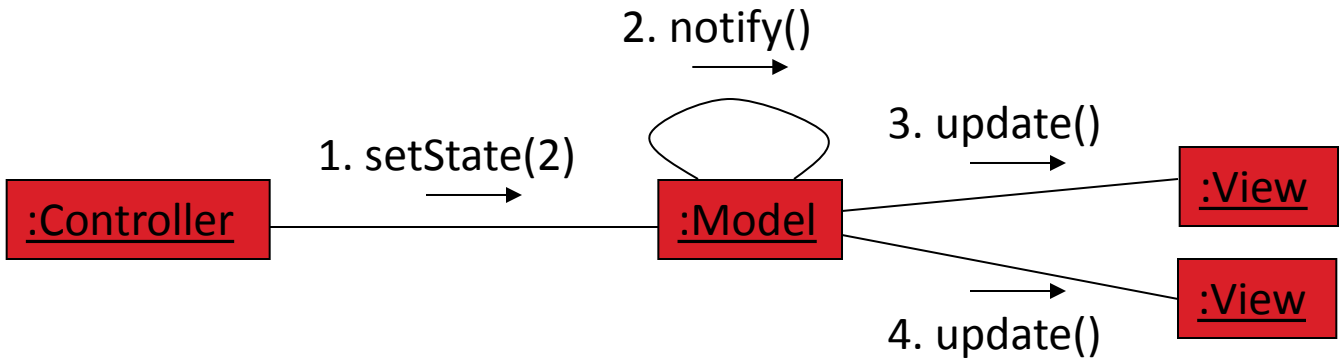
- Consider some domain
  - e.g., drawing apps., web servers, Eclipse plug-ins
  - Every program in this domain has a common subset of functionality
  - Some details very different between them
- We want to implement a solution so that:
  - common functionality can be shared
  - variations can be added easily

OO for planned flexibility

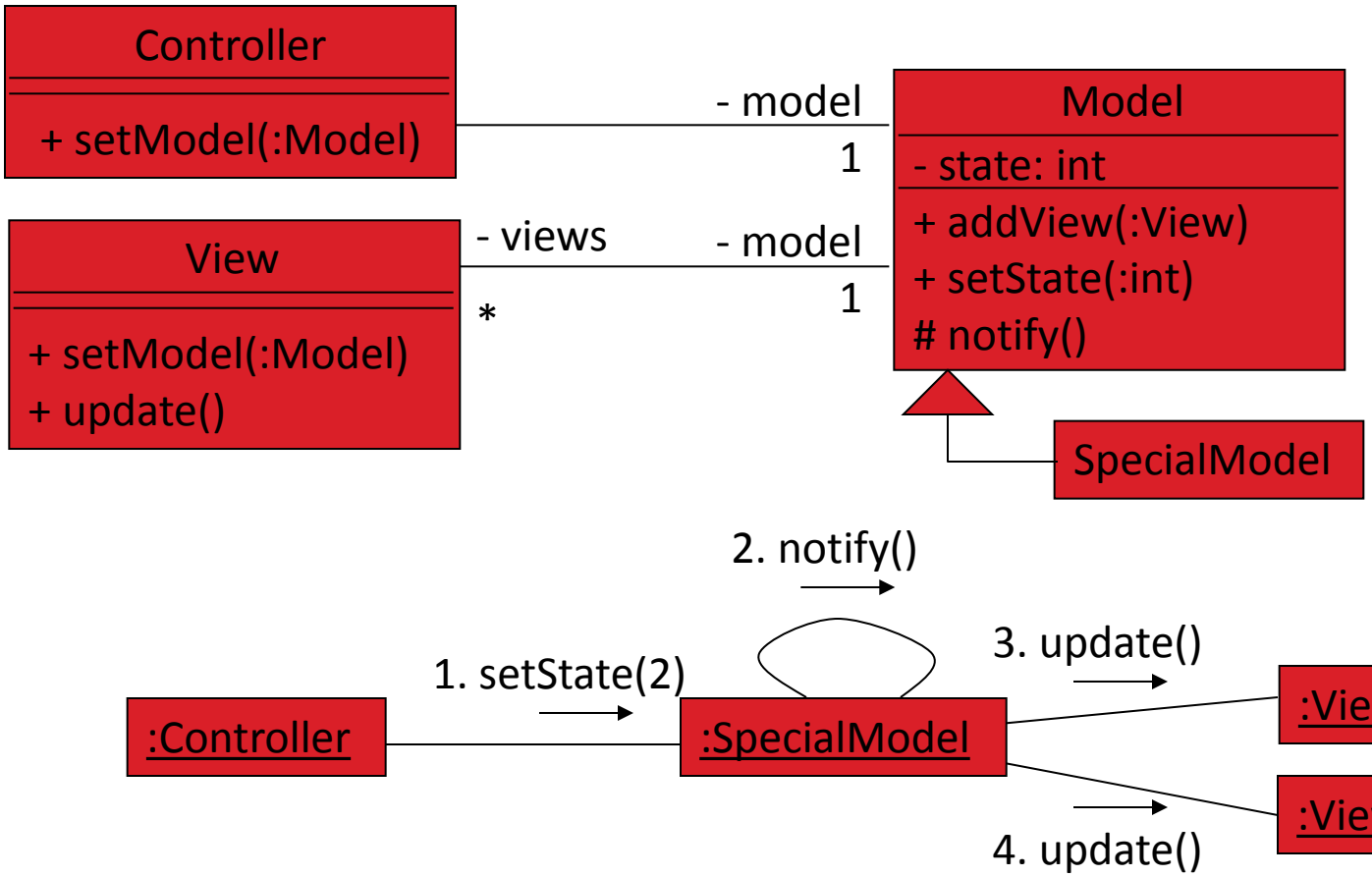
- To provide flexible abstractions, is it sufficient to be able to subclass an existing class?
- Non-trivial behaviour usually requires multiple objects to implement it
- What if some of those objects vary between applications in the domain?
  - How does subclassing help?

OO for planned flexibility





OO for planned flexibility



OO for planned flexibility

- Controller and View classes only need to know about Model, not SpecialModel
- To create the Model class, we need to think of concrete class that:
  - can be generally subclassed
  - does something useful, since instances can be created
- This is hard, since there might not be a general purpose "Model" class that makes sense

Object-oriented frameworks

- A framework is [Johnson, 1997]:
  - the skeleton of an application that can be customized by an application developer
  - a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact
- Reused via subclassing

Frameworks

- Describe the architecture of part of an OO system
  - Kinds of objects and how they interact
- Invert control
  - To reuse library, need to write parts that call into the library
    - Need to know correct sequencing
  - Framework does the calling, instead
    - Less need to worry about sequencing

Examples

- Frameworks are very common:
  - GUIs, drawing apps. (e.g., JHotDraw, Swing)
  - VLSI algorithms
  - operating systems
  - network protocol software
  - manufacturing control
- Java standard class “library” contains many frameworks
- If you see an abstract class, there’s a framework

MVC: A stereotypical framework

- Model-View-Controller (MVC)
  - Model is some set of data
  - Multiple views of data can simultaneously display model in different ways
  - Controller might receive user events => updates model, which in turn, causes views to alter
    - Different interaction patterns possible

Domains

- Important to distinguish between:
  - target domain
    - the space of possible target programs with functional similarities
  - framework domain
    - the space of possible designs that a given framework address
- These overlap, but consider that different designs can accomplish the same task sometimes

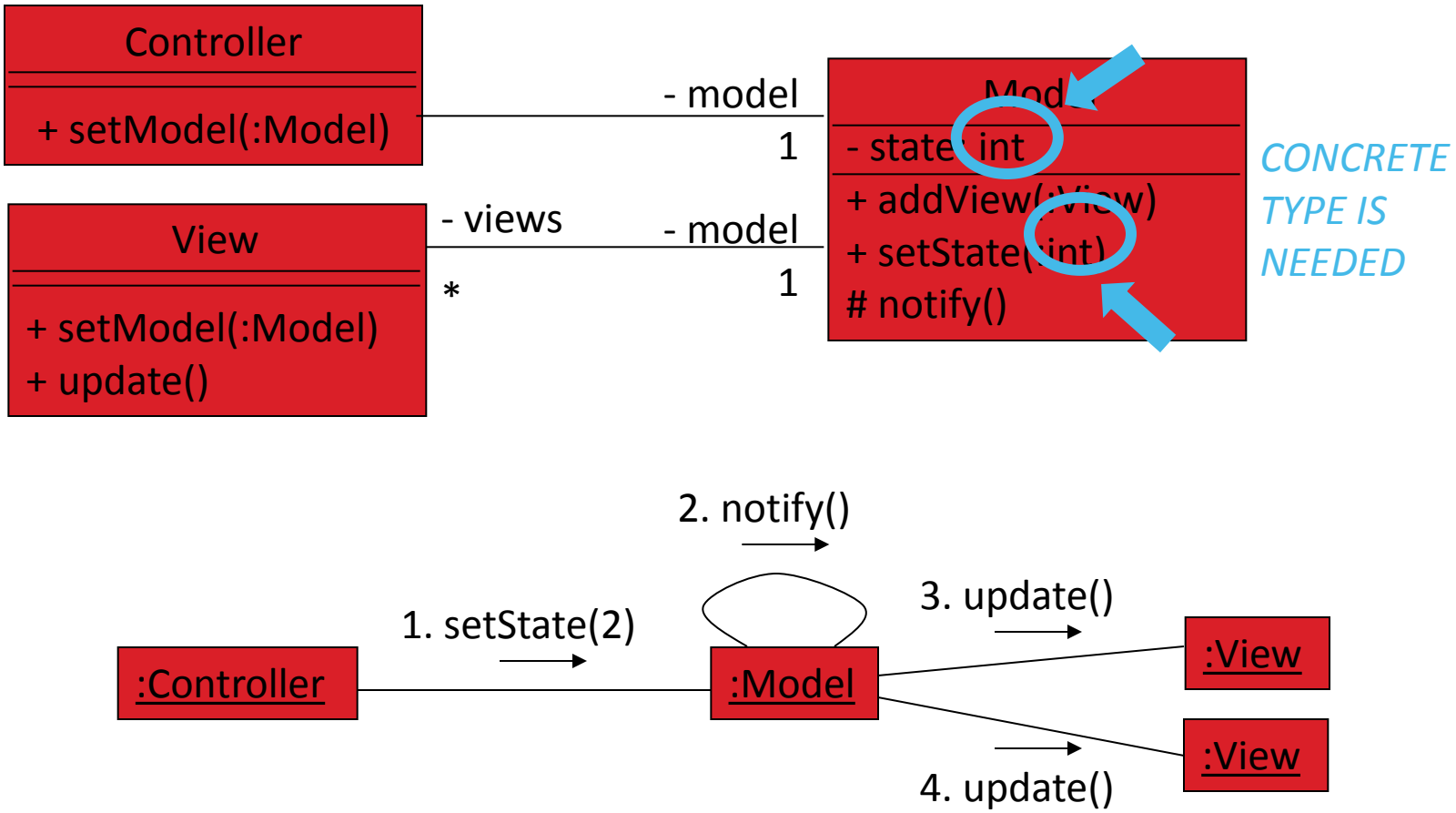
Initial design

- Bottom-up
  - build small bits and pieces, fit them together
  - this works well if the framework domain is well understood
  - results in arbitrarily many pieces needing to be built if the framework domain is not well understood
- Top-down
  - driven by particular, concrete targets
  - may result in too much rigidity: additional targets won't fit
  - may result in too much flexibility: inefficiency and complexity for nothing

Lack of composability

- Since frameworks invert control, what happens if multiple frameworks have to work together?
  - They may break each other's assumptions [van Gurp and Bosch, 2001]
  - Multiple inheritance problems

Inflexibility in MVC



Inflexibility

- Interfaces become set in stone
- Strong typing difficulties:
  - We need the interfaces to be as general as possible
    - But in Model, is there a single instance of Object for state, or multiple ones? Some other type?
  - View and Controller can also be specialized
    - Need to cast state to appropriate type

Framework evolution

- Frameworks are software
  - they will tend to change over time
- Client software has been implemented assuming a particular contract
  - every client will need to change, if the framework is not backwards-compatible
  - alternatively, old interfaces need to be maintained: source of "deprecated" concept in Java
  - changes implemented in complex ways to avoid changes to interface: structural degradation

Agenda

- ~~Basic concepts~~
- ~~Evolvability in standard object-orientation~~
- ~~Evolvability in APIs~~
- ~~Frameworks~~
- Qualitative evolvability analysis

What can vary?

- Given a design, consider the points that can be changed easily, and those that cannot
  - classes/interfaces
  - methods and fields
  - relationships
  - added
  - removed
  - modified
- Then, consider likelihood that these would change

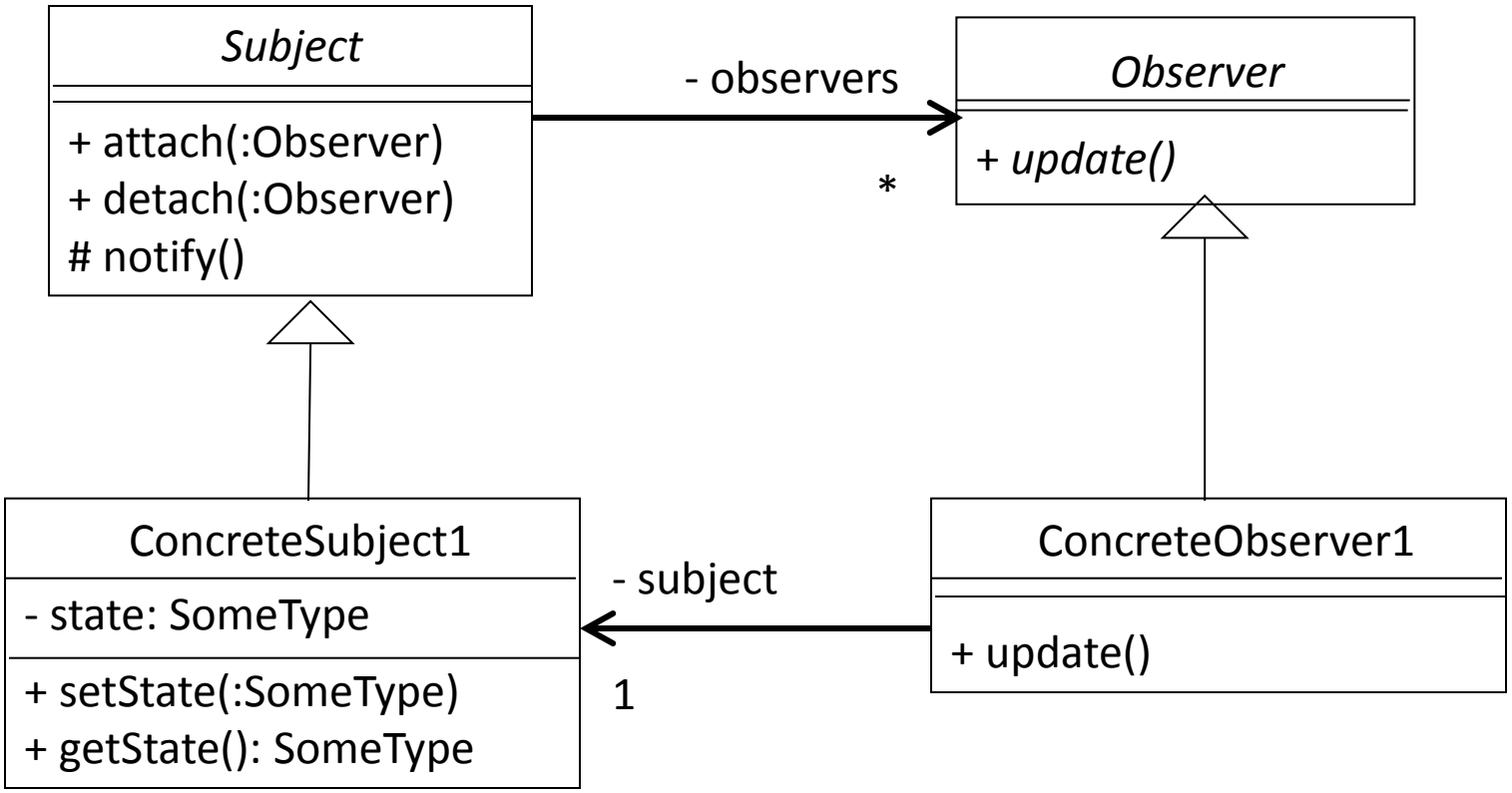
Difficulties

- Abstraction of the design
  - other parts of the system aren't shown, but may matter to the analysis
  - logical roles may be realized by multiple classes
    - complicates the analysis
  - impact on the *intent* of the design must be considered
  - ownership boundaries (you may not know about all the affected code)

Example: Apply QEA to Observer

- What's the purpose of Observer?
- Remember how it works ...
- And remember what is being abstracted away!

Observer: structure



QEA results

- New ConcreteObserver types are easy to add
- New ConcreteSubject types are easy to add, but a bit more work to make useful
- AbstractObserver and AbstractSubject are very hard to change, especially the event notification protocol
  - Even worse if third party code adds more ConcreteObserver types (not shown in diagram!)
  - Can lead to need to have multiple, specialized event notification protocols: confusion, inefficiency
- New methods can be added without much pain