Imperative Programming 3

GUI Design

Peter Jeavons

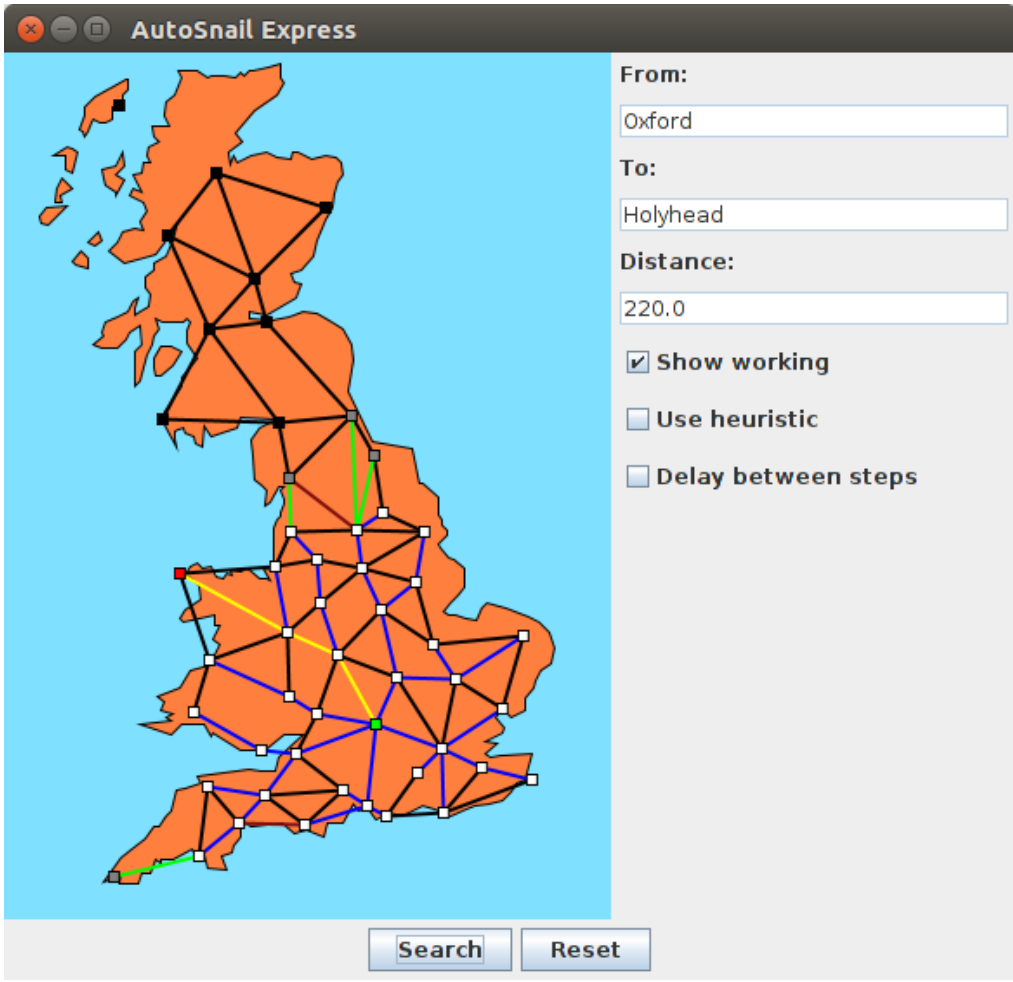
Trinity Term 2019

Second case study

Aim

To use AutoSnail as another case study to explore:

- Architecture
- GUI concepts
- Threads
- Design patterns



Designing AutoSnail

- As with the Ewoks editor we first look at the required functionality:
  - What are the *concerns* of AutoSnail?
  - What *changes* are likely in future?

- Nouns used in description suggest classes
- Aim for loose coupling between classes

“Separation of concerns”

So what are the concerns?

- A ***route planner*** is likely to be concerned with the following things:
  - Storing the details of a map with a collection of towns and roads between them;
  - Showing a picture of the map so that the user can choose the starting and ending towns
  - Showing other controls to specify the kind of search and make it start;
  - Finding a route between the start and end towns by systematically exploring the possible roads from the current position;
  - Displaying the roads being explored and the final route on the map;

So what are the concerns?

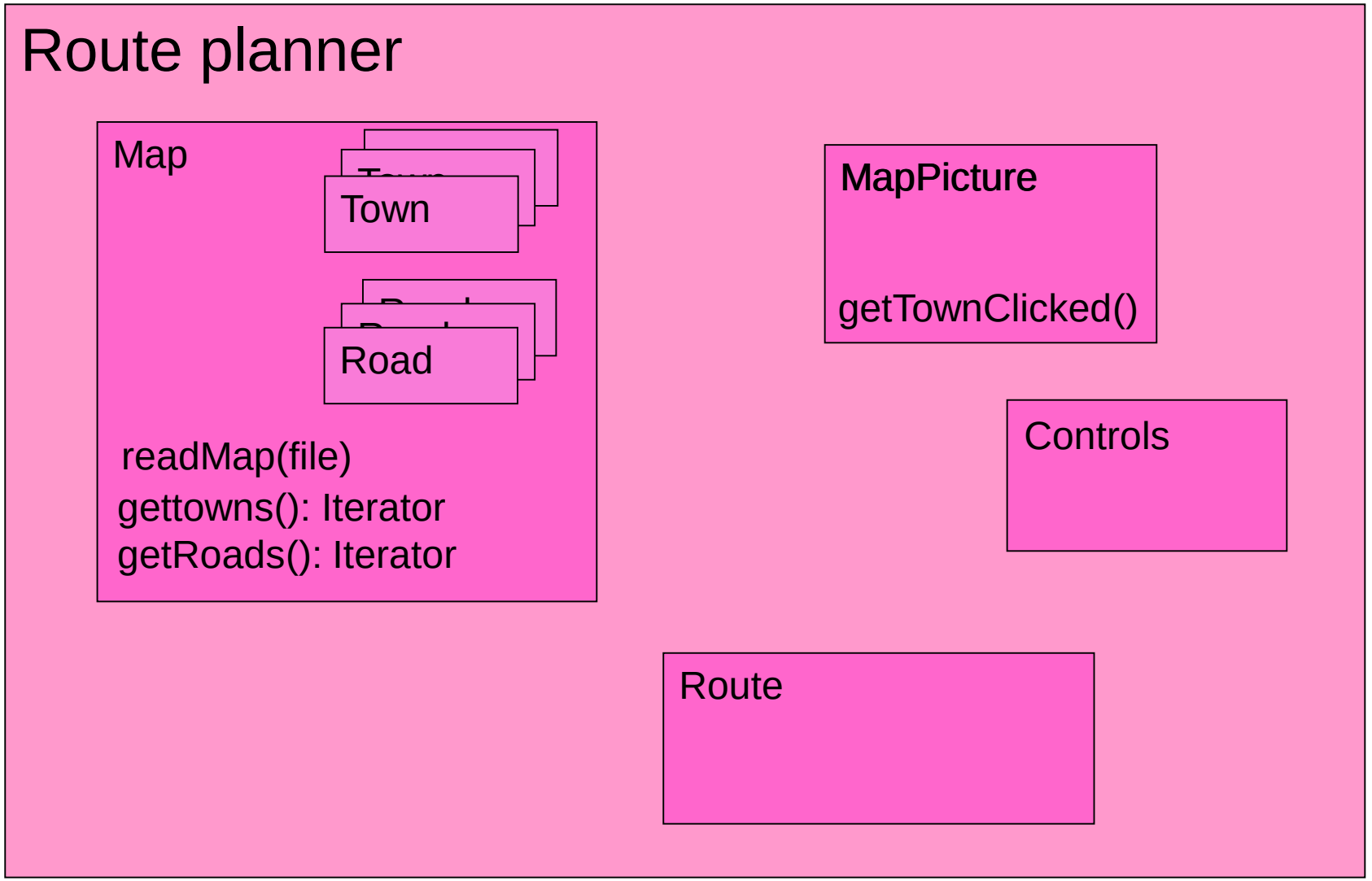
- A ***route planner*** is likely to be concerned with the follo
  - Storin on of towns and ro
  - Showing a picture of the map so that the user can choose the starting and ending towns
  - Showing other controls to specify the kind of search and make it start;
  - Finding a route between the start and end towns by systematically exploring the possible roads from the current position;
  - Displaying the roads being explored and the final route on the map;

Question: What are the nouns in this description?

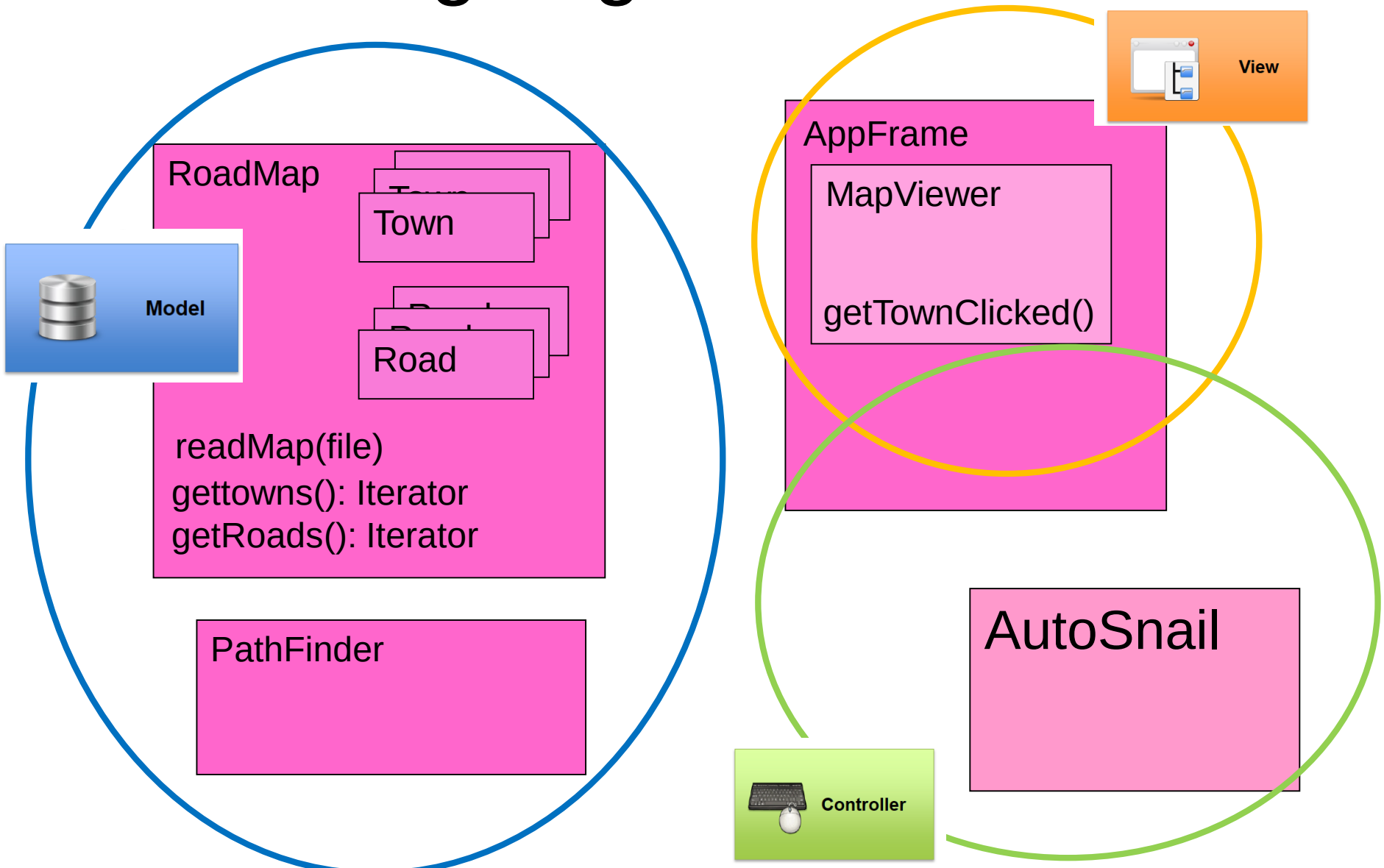
Changes

Question: What are the likely changes?

Designing the Classes



Designing the Classes



CRC card method

From *Agile Programming* based on *user stories*

- For each “noun” write an index card
  - Class name
  - Responsibilities (things done by class)
  - Collaborators (other classes needed for class to provide functionality)

- Reduce responsibilities => high cohesion
- Reduce collaborators => loose coupling
- Turn into UML, traits, classes, … code

RoadMap CRC card

RoadMap *Responsibilities*: • Stores static information about the map (not changing during a search) • Creates map from text file	*Collaborations*: • Uses immutable `Town` and `Road` classes, and standard collection classes for lists of towns and roads • Provides access to all towns and all roads via iterators

MapViewer CRC card

MapViewer	*Collaborations*:
Responsibilities: • Displays the roadmap • Converts mouse clicks into clicks on towns	• Publishes `TownClicked` events

PathFinder CRC card

PathFinder	*Collaborations*:
Responsibilities: • Conducts a search for shortest path using Dijkstra's algorithm • Can be run in separate thread	• Uses map info from `RoadMap`, but adds dynamic information about each town • Can be linked to GUI classes in Subject/Observer relationship

RoadMap CRC card

RoadMap

Responsibilities:

- Stores static information about the map (not changing during a search)
- Creates map from text file

Collaborations:

- Uses immutable `Town` and `Road` classes, and standard collection classes for lists of towns and roads
- Provides access to all towns and all roads via iterators *and all roads leaving a given town*

MapViewer CRC card

MapViewer	*Collaborations*:
Responsibilities: • Displays the roadmap *and ongoing progress of pathfinder* • Converts mouse clicks into clicks on towns	• Publishes `TownClicked` events

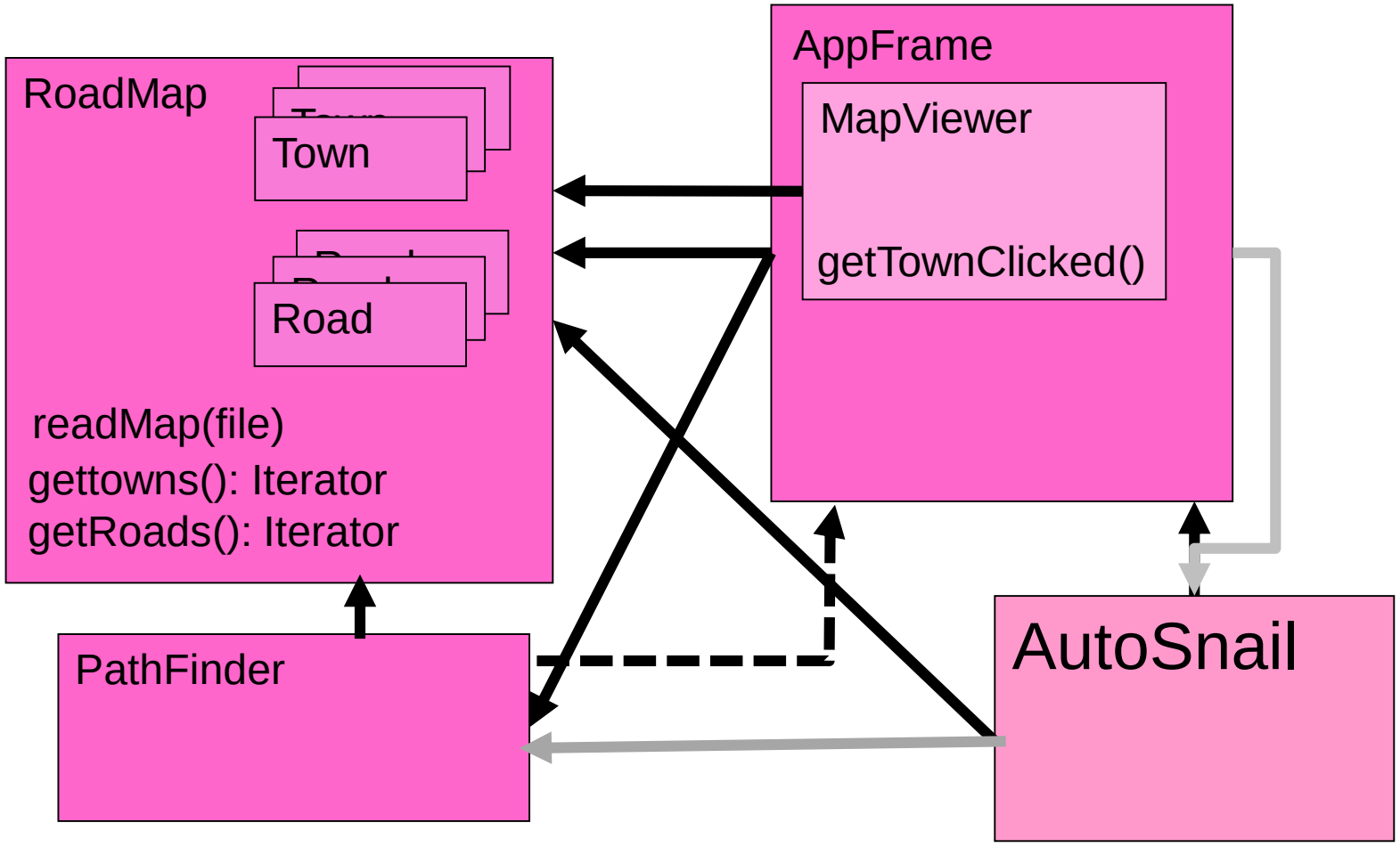
AppFrame CRC card

AppFrame	*Collaborations*:
Responsibilities: • Aggregates all GUI elements into one window	• Invokes commands on `AutoSnail` • Can act as Observer to a `PathFinder`

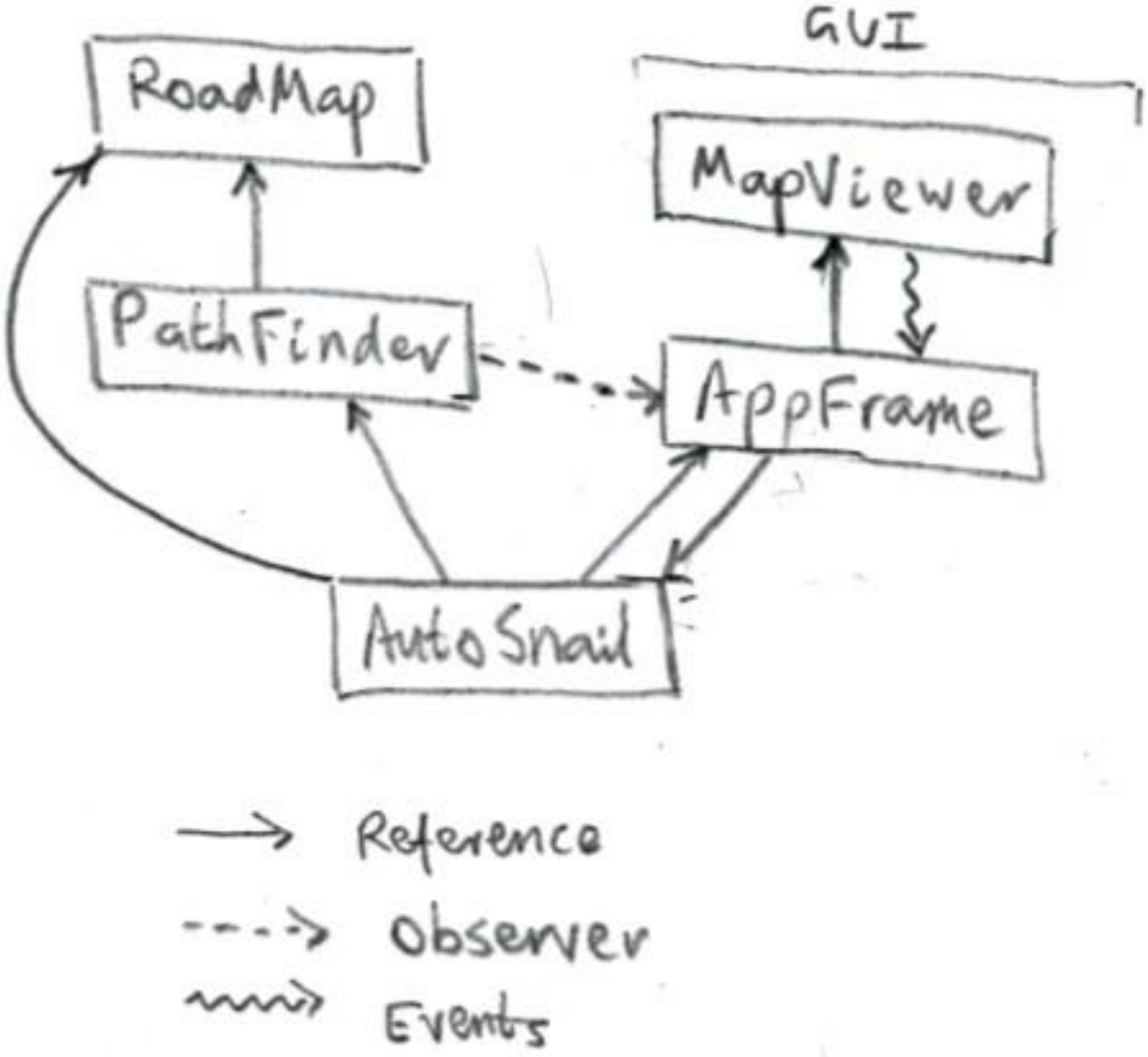
AutoSnail CRC card

AutoSnail	
Responsibilities: • Supervises protocol for user interaction • Initiates search when ready	*Collaborations*: • Creates `PathFinder` instances for searching and connects them with display objects

Designing the Classes



Classes in AutoSnail*



*Genuine artefact from Mike Spivey's notebook

Decoupling

- `PathFinder` and `RoadMap` have no dependencies on the other classes
  - Can be tested independently (`Benchmark.scala`)
  - Can be put “server-side” with a GUI client

- Main object AutoSnail accesses GUI only via AppFrame
  - Façade pattern: simple interface to larger code
  - Different GUI designs easily swapped in

Classes and threads

RoadMap

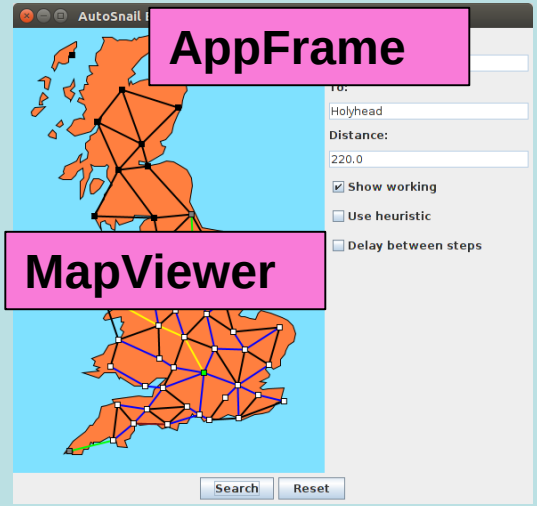
PathFinder

Worker thread

paintComponent

EDT

AppFrame



MapViewer

reactions

AutoSnail

Main thread

Concurrency

- The `PathFinder` object does work as a separate thread
- The extra thread only exists while a search is going on

```
class AutoSnail(map: RoadMap) {
  ...
  /** If all is ready, carry out a search */
  def search(heuristic: Boolean) {
    require(from != RoadMap.noTown || to != RoadMap.noTown)
    pathfinder = new PathFinder(map, from, to, heuristic)
    pathfinder.addObserver(frame)
    val thread = new Thread(pathfinder)
    thread.start()
  }
}
```

```
class PathFinder(map: RoadMap, from: RoadMap.Town, to: RoadMap.Town,
                                                    heuristic: Boolean)
        extends Runnable with Observable[PathFinder] {
    ...
    def run() {//Performs a single search...}
}
```

Subject-observer in traits

```
/** Mixin for classes that can update observers */
trait Observable[T] {
  ...
  /** Notify all observers that the subject has changed */
  def notifyObservers() {
    for (o <- observers) o.refresh(this);
  }
}
```

```
class PathFinder ... extends Runnable with Observable[PathFinder] {
  ...
  def run() {
     notifyObservers()
     while (dest.status != TownStatus.KNOWN) {
       val t = findMin()
       ...
       visitNeighbours(t)
       notifyObservers()
     }

     highlightPath()
     done = true
     notifyObservers()
  }
}
```

Subject-observer in traits

```
trait Observer[T] {
  def refresh(subject: T)
}
```

When `PathFinder` does `notifyObservers`:

- `AppFrame` gets a `refresh` call
- `MapViewer` is passed a `refresh` which posts a repaint
- `AppFrame` updates buttons etc. on the EDT

```
class AppFrame(map: RoadMap, app: AutoSnail)
        extends MainFrame with Observer[PathFinder] {

  def refresh(pathfinder: PathFinder) {
    viewer.refresh(pathfinder)
    Swing.onEDT { updateDisplay(pathfinder) }
  }
```

GUI

- `AppFrame` uses nested containers for layout
  - Responds (to mouse etc.) by calling methods of `AutoSnail`
  - Could have looser coupling by sending events
- `MapViewer` draws map, responds to town clicks and does mouse movement tooltips
  - Clicks and tooltips use proximal geometry
  - Other classes don't need to know about geometry
  - Tooltips can generate a lot of `MouseMoved` events…

Search Algorithm

Add some attributes to towns and roads:

- `t.dist` (stores town's distance from source)
- `t.status` {UNSEEN, ACTIVE, KNOWN}
  (coloured black, grey, white)
- `t.link` (the road used to record best route)
- `r.status` {UNSEEN, ACTIVE, DEAD, TREE, PATH}
- `r.prev` (the town used to record best route)

Two options:

1. Give each town additional instance variables (intuitive)
2. Use an attribute table (respects encapsulation)

Use hash table (with standard hash and default object equality) to map towns/roads to attributes

Performance

Tests of path-finding on fixed large random network shows that the inner loop executed 25 million times

```
protected def findMin() = {
  var d = INFINITY; var t: TownData = null
  for (x <- map.towns) {
    val u = townData(x)
    if (u.status != TownStatus.KNOWN && u.dist < d) {
      t = u; d = u.dist
  }}
  t
}
```

This takes 38 seconds – too slow

Performance

Tests of path-finding on fixed large random network shows that the inner loop executed 25 million times

- Switch from attribute table to attributes in classes
  - 38 sec -> 10 sec
- Switch from search all towns to a priority queue of only grey (ACTIVE) towns
  - 38 sec -> 2.6 sec

Polymorphism allows tuning

"Program to the interface"

```
trait PriorityQueue[T]{
  def insert(x:T)
  def delMin():T
  def isEmpty:Boolean
}
```

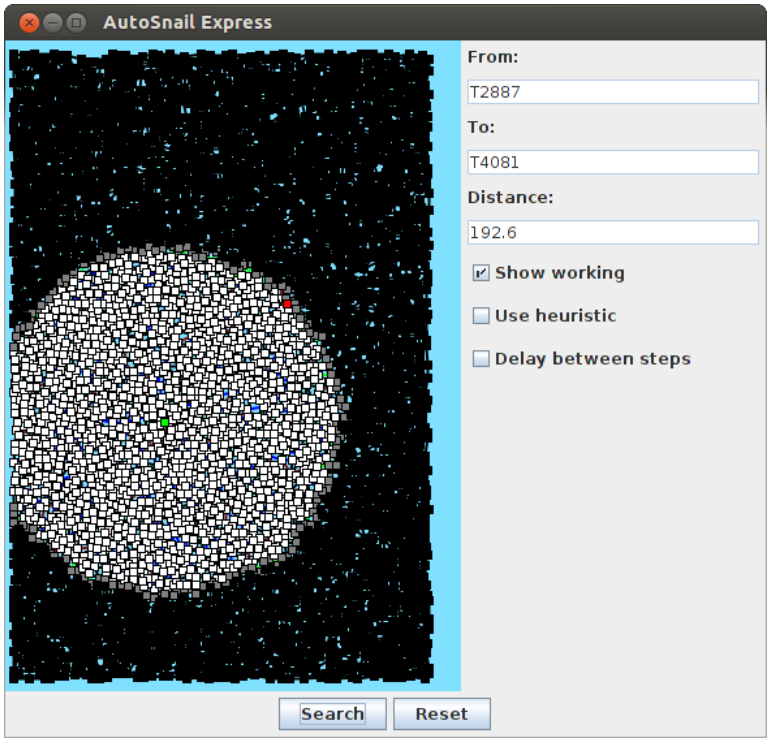
New we can plug in implementations with the factory pattern

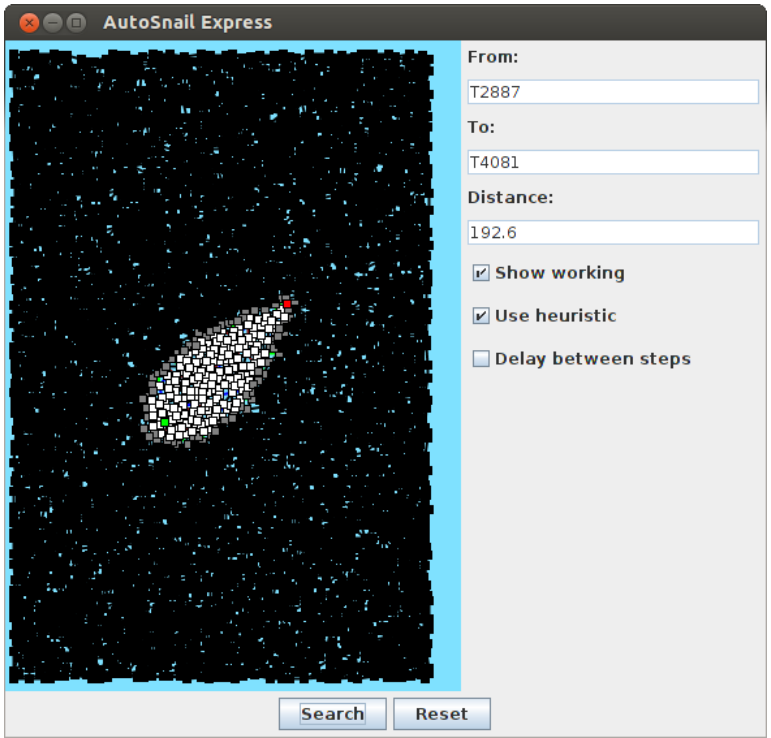
```
trait Factory{
  def makeQueue[T <% Ordered[T]]() : PriorityQueue[T]
}

val factory = new  PriorityQueue.Factory(){
  def makeQueue[T <% Ordered[T]]() = new HeapPriorityQueue[T]
}
```

A* algorithm heuristic

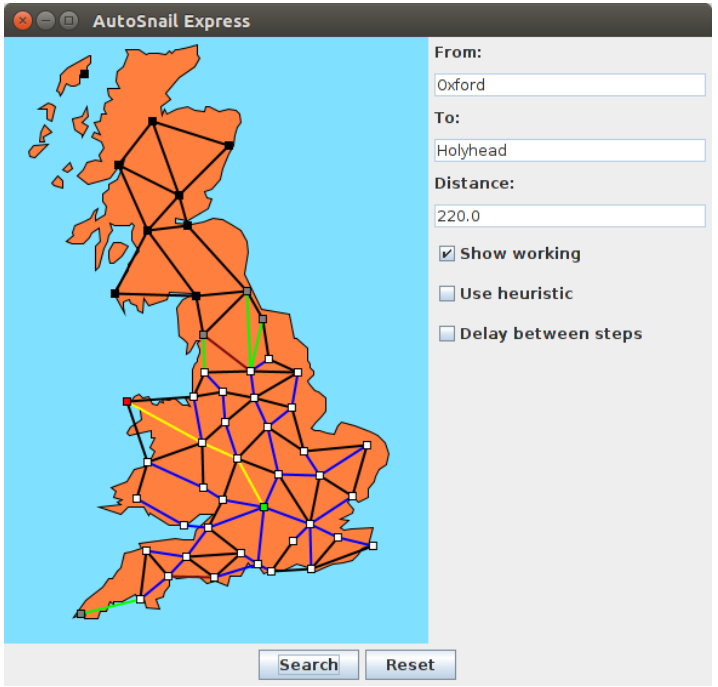
- Explore in the general direction of the destination
- Each town has penalty (Euclidean distance to destination)
- When visiting a neighbour `u` of town `t` adjust its distance priority by `penalty(t) - penalty(u)` so that a `u` which is closer to goal than `t` gets lower/better priority





Summary

- Architecture of AutoSnail
- CRC card exercise
- GUI & threads in concrete setting
- Design patterns
  - Façade
  - Subject/Observer
  - Factory
- Efficiency



See also *Head First Design Patterns:* Chapter 4