

# An OO Design Exercise; Component Architectures

1 March 2011

CMPT166

Sean Ho

Trinity Western University

# Steps to OO design: wADes

- **Requirements:** system behaviour
  - Use-cases, UI mockups
- **Design:** components / modules
  - Component diagrams, CRC diagrams
  - Sequence diagrams of messages passed
  - State of objects/components
- **Implementation:** from components to classes
  - Class diagrams, relationships
  - Inheritance, composition, etc.

# An OO design exercise

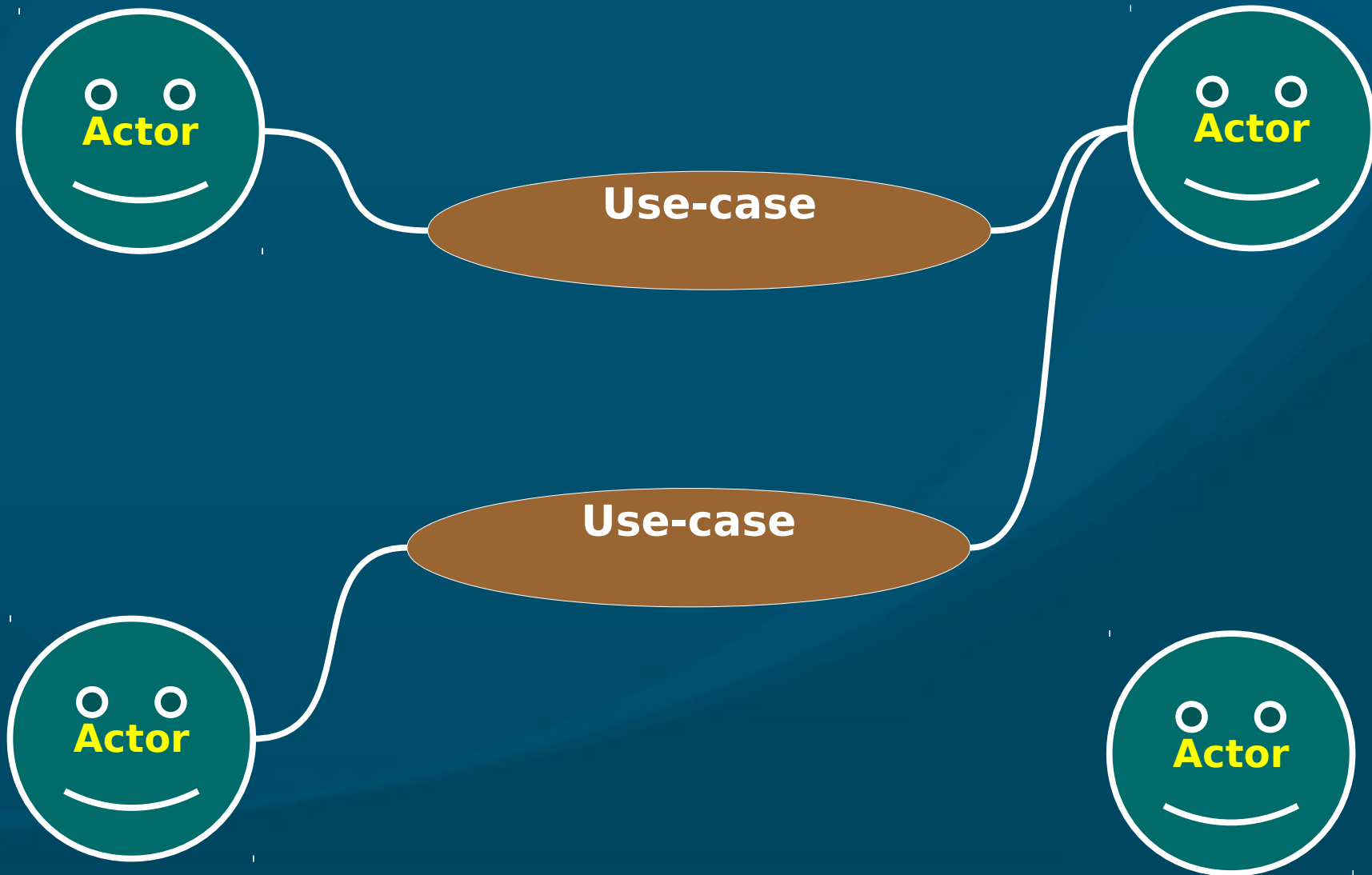
- Problem statement:

Design a **student enrolment database** like we have at TWU

# (1) Actors and actions

- Use-case scenarios: actors and actions
- Who are the actors?
  - *student, registrar, web tech, DB tech , instructors, advisors, TAs*
  - *Finance office*
- What are the actions? Scenarios of use?
  - *Students: .apply, chk pmt, register for course, drop course, chk GPA/transcript, chk course info, list schedule*
  - *Registrar: create new course, modify course, delete course*
  - *Student/Registrar: transfer credits*
  - *Techs: shutdown site, maintain/test site*
  - *DB tech: maintain/modify DB*

# (1) Use-case diagrams



# (1) Specify one use case

- **Name:** *Add Course*
- **Actors:** *Student*
- **Goal(s):** *Course is added to student's schedule*
- **Pre-conditions:** *student is current/admitted, currently within registration window, course exists, is offered and open for registration, ... has prereqs, has paid, (audit?), enough space on waitlist*

# (1) Use case: basic flow

## ■ Basic flow:

- *Login*
- *Check if student*
- *Select “course registration”*
- *Display course list (except for already registered courses)*
- *Browse courses, select course*
- *Click on “Add course”*
- *Check for sched conflicts and flag*
- *Add to student's course selection in DB*



# (1) Use case: alternate flows

- What might **not** go according to the basic flow?
  - *Login incorrect: error screen, prompt again*
  - *Not a student*
  - *Sched conflict, no prereqs, no payment*
  - *Course not offered, doesn't exist anymore*
  - *Doesn't actually add course to DB (chk duplicate entries?), DB error*
  - *Course/WL full*
  - *PHP/web bugs, network errors*



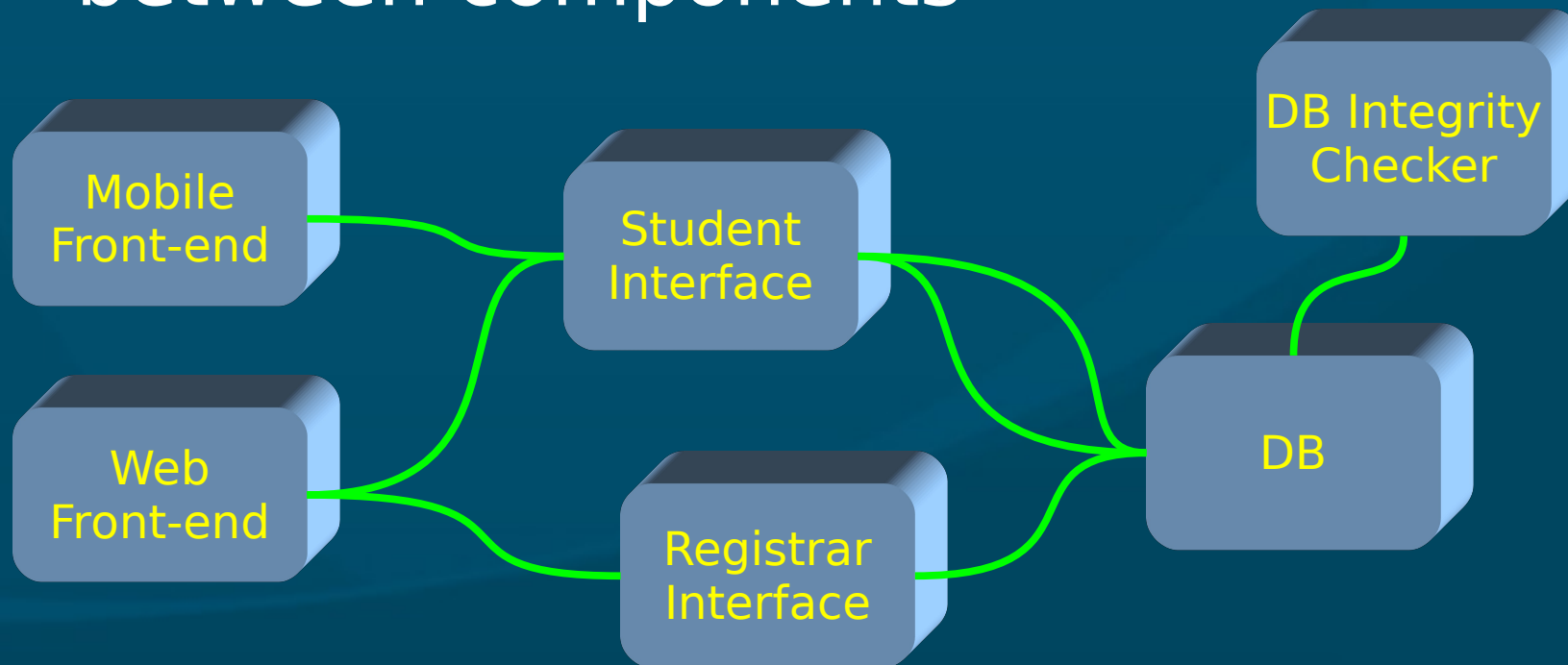
# (1) UI mockup

- For each use-case (action), describe/mockup what the **user interface** will be like:
  - Text Q&A? Windows? Interactivity?



# (2) Component design

- This is often the **hardest** part!
- Components need **not** be classes
- **Thinly** coupled: describe all **interfaces** between components



# Component: ....

- Name: ...
- Description: ...
- Interface to (*component*):
  - ...
- Interface to (*component*):
  - ...

# (3) Components to classes

- Each component may need several classes to implement it
- Component: ...
  - Class: ...
    - ◆ Attributes: ...
    - ◆ Methods: ...
  - Class: ...
    - ◆ Attributes: ...
    - ◆ Methods: ...

# What are components?

- Pre-fabricated, **reusable** building **blocks** for software systems
- Allow rapid **development** and consistent **reuse**
- Not tying together **libraries** (chunks of code)
  - But **coordinating** running code (dynamic cooperation of live objects with active state)
- **Bigger** than single objects, may be **combinable**
- Relate “**peer-to-peer**” rather than in

# Component concepts

- Applications use a **palette** of components
  - The programmer **composes** or “wires” them together to make a complete **application**
- Requires:
  - Ways to define **new** components
  - Standards to specify component **interfaces**
  - Each component has “**hooks**” (methods) by which other components interact with it
- Compare to **hardware** components:
  - Transistors, integrated chips, etc.

# Components vs. code

- **Hardware** components are black boxes with **spec** sheets: **wires** connect them together
- **Software** components are represented as black boxes with **interfaces**: you write **code** to connect them up
- Software companies may sell components as **binaries** (black boxes) with API **documentation**
  - Need not sell the actual **source** code
  - e.g., **NVIDIA** binary graphics drivers for Linux
  - e.g., Scantron **ClassClimate** and **myTWU** portal

# Component-based apps

- Components are **assembled** into containers
  - The finished assembly is the **application**
- May also take **document-centric** view:
  - e.g., a **container** document may hold **text, images, videos, buttons**, etc.
  - **Editing** any item passes control to the appropriate component:  
**text editor, image editor**, etc.
  - The **document** *is* the **application!**
  - **Peer-to-peer**: no one component is “boss”



# Component-based develop.

- Delivering solutions by **building** or **buying** interoperable components
- Don't **reinvent** the wheel: **write** once, **deploy** many times (server, desktop, handheld, ...)
- Rigid adherence to software **infrastructure**:
  - **Standards** of how components work together
- Fits naturally with **distributed**, multi-**language**, multi-**platform** heterogeneous environments:
  - Don't care what **language** it's written in
  - Don't care **where** it runs

# People use components to...

- Tie together **departments** within a company (enterprise resource software):
  - Accounting, invoicing, human resources
- Connect data from **mergers** of **banks**, **hospitals**
- Leverage rich, complex **data stores**:
  - Data mining, pattern recognition, image analysis, genomics, StatsCan, ...
- Adding **multimedia** to a field **salesperson's** laptop/handheld



# Layering



- Sometimes component architecture is deployed as “**middleware**”:
  - A set of components that allow a variety of **database stores** or **applications** to be manipulated by a common **interface**
  - Other applications **must** go through the middleware in order to access the data store
- **Security**, ease of **debugging**, **simplicity**
- Allows format of **back-end** database to change while preserving the **front-end** UI for users

# Examples of component arch.

- Application **plug-in** interface: **Firefox, Eclipse**
- **LAPACK/BLAS**: std. **linear algebra** library
- **ActiveX/COM**: **interoperation** of MS apps
  - e.g., graphics, outlining, cut-and-paste
- **.NET**: **Microsoft's** (2002) component arch.
  - **CLR** (Common Language Runtime) is the equivalent of the JVM
  - **C#**, but may use other languages, too
- **JavaEE JavaBeans**: components for **Java**

# Example: JavaEE JavaBeans

- **Builder**: IDE to assemble components
  - **JavaStudio**, **NetBeans**, etc.
- Components advertise what **features** are available to builder (methods, events, ...)
  - **Introspection**: a “JavaBeans-enabled” builder can examine a Bean to learn its features
  - Beans **fire** (send) or **handle** (receive) events
- **Swing** components are all JavaBeans!
- **Drag-and-drop** application development
- **Persistence**: Beans can save/restore state

# Example: ODBC

## ■ Open DataBase Connectivity

- Standard **API** to many database systems: **MS-SQL, Oracle, DB/2, mySQL, PostgreSQL, ...**
- Simplifies use of **standard** SQL commands
  - ◆ Structured Query Language: query/edit the DB
  - ◆ Can also access **vendor-specific** commands
- Cross-**platform**, cross-**language**
  - ◆ Although **Java** also has its own: **JDBC**
- Sybase ACA (Architecture for Competitive Advantage): similar, using Transact-SQL