# Review & Misc.

# More On Exceptions

**Demo**

# Student Code Reviews

# Covered In Class – Real World Design Patterns

- Rob's composite menu example.  No judging imperfections!
  - **Pros:**
    - It ain't pretty but it works!  XML content created by business people can change the structure of menus in mobile apps running out in the wild.
    - Descriptor (Rob's bad name for mementos) pattern / system works well for restoring last position on next app load
  - **Cons:**
    - Lots of big long if..else / switch logic vs. creational patterns / behavioural patterns
    - Lack of abstraction in these areas made adding new menu items a pain in the ass
    - Coupled too tightly to the way Android views (activities) are created

# Assignment 1 – Test Driven Development

# Answers for Assignment 2

# Group Discussion: Patterns In Your Projects

# Projects – Understanding Marking

- From the syllabus:
  - 50% Individual Mark
    - 40% quality of each individual's coding contribution
    - 10% individual reflection in project report
  - 50% Group Mark
    - 10% design document (complete)
    - 5% project plan (complete)
    - 5% milestone 1, continuous integration (complete)
    - 5% milestone 2, error handling & logging (due tomorrow in lab)
    - 5% class presentation
    - 5% project report (group portion)
    - 15% overall quality of process & final implementation

# Projects – Understanding Marking

- Individual Mark:
  - 40% quality of each individual's coding contribution
    - This is the bulk of your group project mark
    - Graded on two measures:
      - How much code did you write, it should be **a lot of code**, more than I wrote!
      - How **good** is the code, does it follow everything we learned in class?
  - 10% individual reflection in project report
    - From syllabus: 1 page statement from each group member about what they will take away from this experience
    - What went well?
    - What didn't go well?  What did you do about it?
    - What did you learn?
    - Honest, well-reasoned reflection gets you 10%

# Projects – Understanding Marking

- 50% Group Mark
  - Complete: 10% design document, 5% project plan, 5% milestone 1
  - 5% milestone 2, error handling & logging (due tomorrow in lab)
    - You must introduce an error, or set up a situation where an error will occur (for example rename a table or stored procedure in your database)
    - Demonstrate that your code **CATCHES** the error, handles it if possible, and **LOGS** that the error happened.
    - Logs must be **easily** accessible to the TAs (1 – 2 minutes to see the logging in action)
  - 5% class presentation (10 – 15 minutes)
    - Group presents your project working in your production environment
    - Discuss your process: how did you follow agile? How did you implement gitflow?
  - 15% overall quality of process & final implementation
    - Did your group deliver what you said you were going to deliver in your project plan.
    - Did your group as a whole follow the lessons taught in the course (agile, CI, gitflow

# Projects – Understanding Marking

- 5% project report (group portion):
  - How you implemented continuous integration
    - Tools used
    - Did you have hardcoded configuration? Why? How might you have solved that in the real world?
  - List of design patterns used, and why they were used
  - How you achieved separation of presentation / business / data layers
  - What naming convention / spacing convention you agreed on and why
  - Examples of any refactorings you performed
  - List of what is considered "technical debt" and how it would be resolved (more on this in the refactoring class)
  - Assemble each group member's list of contributions and 1 page reports and include in the final combined report.

# Quick Summary Of Concepts Used To Evaluate Whether Your Code Is "good" Code

- Everything we've learned in class both written on slides and discussed in lectures will be used for grading
- Get out a pen and paper and make a checklist as I go through them now…
  - For more information on each item revisit the lecture slides and handouts from class

# "good" Code Evaluation:

- **#1 – The project compiles**. If I update a file in your development branch via github I see it get re-deployed.
- **Test Driven Development**
  - All code not in the presentation is 100% covered by unit tests in separate files, with mock objects as necessary.
  - All tests pass
  - Tests have quality of their own (no "omega tests", no testing inside the black box)
- **S.O.L.I.D.:**
  - All modules and classes adhere (do not violate) to S.O.L.I.D. principles

# "good" Code Evaluation:

- **Cohesion: Your modules are cohesive (code belongs together)**
  - Not just modules, but classes too (data and methods belong together)
  - Use release reuse equivalence principle, common closure principle, common reuse principle
- **Coupling: BAD**
  - Your code is not unnecessarily coupled to other modules / classes
  - Data coupling, temporal coupling, stamp coupling, control coupling, external coupling, common coupling, content coupling
  - Principles: acyclic dependency principle, stable dependencies principle, stable abstractions principle

# "good" Code Evaluation:

- **Clean code:**
  - Indentation:  Your team picks one thing and follows it consistently (tabs)
  - Code follows standard conventions for your language
  - KISS:  Keep it simple stupid.  No unnecessarily complicated code.
  - Configurable data at high levels
  - Prefer polymorphism to if / else / switch statements (for big ones anyways)
  - Separate multi-threading code and single threading code
  - Prevent over configuration
  - Use dependency injection
  - Follow "Law of Demeter"
  - Be consistent (things done in similar ways, especially within same module, e.g. Save() + Load(), Open() + Close(), ReadFile() + WriteFile()

# "good" Code Evaluation:

- **Clean code (continued):**
  - Hardcode NOTHING
  - Use explanatory variables over comments
  - Encapsulate boundary conditions
  - Prefer dedicated value objects to primitive types where it makes sense
  - Avoid logical dependency (depending on pre-conditions)
  - Avoid negative conditionals:
    - E.g.:
      - If (user.isAuthorized()) { giveAccess(); } vs. if (!user.isNotAuthorized()) { giveAccess(); }
      - If (user.isAuthorized()) { giveAccess(); } vs. if (!user.isAuthorized()) { denyAccess(); }
      - If (somethingDidHappen) { DoA() } else { DoB() } vs. if (somethingDidNotHappen) { DoB() } else { DoA() }

# "good" Code Evaluation:

- **Clean code (continued):**
  - Code is explicit, not implicit
    - No one line if's
    - No relying on implicit conversions, specify
    - Avoid reflection
    - Don't make assumptions, assert on pre-conditions
  - Naming convention: Team picks one standard and applies it consistently across all code in the project (including the DB)
    - Names follow Uncle Bob guidelines, long names for long scopes, etc.
  - Comments:
    - No commented out code
    - No unecessary / irelevant comments
    - No noise (e.g. #region's)

# "good" Code Evaluation:

- **Clean code (continued):**
  - Functions:
    - Small
    - Do one thing. Do it well. Do it only.
    - Use descriptive names
    - Prefer fewer arguments
    - No side effects
    - No flag / control arguments (booleans)

# "good" Code Evaluation:

- **Design Patterns:**
  - Your modules are implemented with design patterns where appropriate
    - E.g. creational patterns / structural patterns in your content areas, behavioural patterns where your algorithms / logic is implemented.
    - You never reinvent the wheel pattern-wise
  - The patterns are implemented correctly
  - If I see spots in your modules where a pattern would have been useful you will lose points, scan your code and look for opportunities
    - Don't force them in just to have them though
- **Code Smells:**
  - Check the checklist I handed out, make sure your code does not smell
  - At a minimum if you have major smells / problems they better be recognized by you and on your list of technical debt with explanations on how you would fix them.

# "good" Code Evaluation:

- **Boundaries:**
  - Your code does not violate presentation / business logic / data layer boundaries
  - If you have business logic in your DB you have a long explanation of why you chose to put it there and that explanation makes sense.
- **Logging:**
  - All your major areas are covered by some form of logging. (Business / data layers, not presentation)
- **Error Handling:**
  - Things that CAN fail are protected with error handling (e.g. all of you have DB logic that can fail)
  - Assert pre-conditions and post-conditions
  - Use exceptions
  - Use predefined exception types
  - Do not use exceptions for control flow
  - Throw early, catch late
  - Do resource cleanup in finally blocks