Design Smells

- Recall the code smells we talked about during the refactoring lecture:
  - unchecked growth, focus failure, data on the run, code crowds etc
- The smell can also be at a higher level, i.e., at the design level, leaving us with Design Smells.
- As with code smells, design smells can be there from the start or build over time as the project evolves.
- Design smells will almost always lead to code smells as well.
  - To fix the code smells, it is often a good idea to step back and look at the overall design.
- Rigidity
  - the design is hard to change; changes to one part of the system forces changes in other parts of the system.
- Fragility, a.k.a. brittleness
  - the design is easy to break - a change in one place triggers bugs in other conceptually unrelated parts of the system.
Design Smells

› Immobility
  › the design is hard to reuse - components are entangled with other components preventing reuse of the components in isolation.

› Needless complexity
  › over-design, code added without any real need for it.

› Needless repetition
  › DRY-violation, copy/paste coding or design, failure to refactor as need arises, can occur in other places than code.

› Opacity
  › disorganised expression, convoluted hard-to-read code, the opposite would be clear and expressive.

› Viscosity
  › it's easier to do the wrong thing than the right thing, preserving design is harder than quick fixes.
Design Principles

‣ Code standards provide a set of guidelines for how code should be written, named and formatted.

‣ As such, it makes it easier to understand and maintain code.

‣ In a similar fashion, it is possible to have standards, or principles, for design, i.e., a set of principles which when followed leads to great and beautiful programs.

‣ Good design will lead to good programs and less smell.


‣ Using these principles will lead to code that is
  ‣ easy to test
  ‣ easy to maintain
  ‣ easy to reuse
Basic Design Principles

‣ SRP - Single Responsibility Principle
  ‣ Design classes so they have a single conceptual responsibility.
  ‣ They will only have one reason to change, i.e., when changes related to that responsibility changes.
  ‣ The class will be highly focused.
  ‣ The first half of the classic OO mantra *high cohesion, low coupling*.

‣ ISP - Interface Segregation Principle
  ‣ Related to SRP, but for interfaces
  ‣ A class might have several related responsibilities, but it should not expose them all to other parts of the system
  ‣ Design focused interfaces that have a focused responsibilities
  ‣ Failure leads to dependency between unrelated parts of the system
Basic Design Principles

- LSP - Liskov Substitution Principle
  - Let S be a sub class of T.
  - It should be possible to use instances of S instead of T without any unexpected behaviour from the program, i.e., no nasty surprises!
  - The dangers of inheritance..
  - Sub classes should not require more or promise less than their super class.

- DIP - Dependency Inversion Principle
  - It should be possible to inject behaviour at runtime rather than fixing it a compile time.
  - Use interfaces to describe the desired properties of objects you want to handle.
  - The user of the interface owns it and thus sets the requirements - you might need to write an adapter to adjust
  - Leads to easily testable code.
Basic Design Principles

‣ OCP - Open Closed Principle
  ‣ Open for extension, closed for modification
  ‣ Behaviour can be changed or added with needing to change source code
  ‣ New behaviour leads to new classes and thus less risk to existing code and uses of it.
  ‣ Leads to easily testable code.
Principles, Tests, Reuse

- Well design code, following sound principles, leads to testable code.
- Testable code leads to reusable code.
- Automatic tests upholds the quality of the code.
- Reused code needs less testing and debugging since it has already been used.
- Reuse leads to less risk.
- We become more productive!
- Also, see more on the subject of Design Patterns.
Faster Ray Tracing

- What is the best way of speeding up the ray tracer?
  - Avoid doing hit point calculations!
  - We do a very large number of hit point calculations, most of them unnecessarily.
  - How can we avoid doing those, or at least some, that we don’t need?
  - Bounding objects - notes went to blackboard.