

Software Design and Construction 1

SOFT2201 / COMP9201

**Introduction to Software
Construction & Design**

Dr. Xi Wu

School of Computer Science



Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

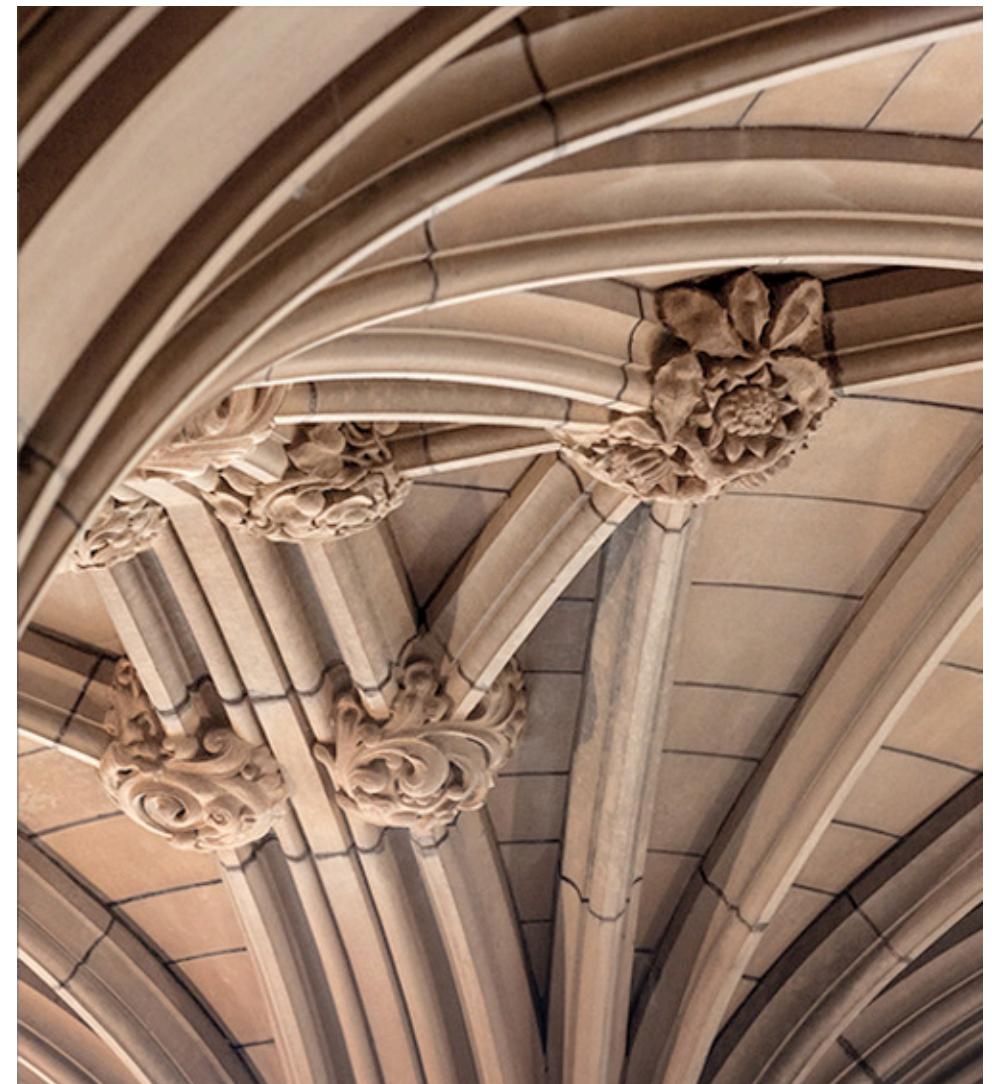
Agenda

- Workplace Health and Safety
- Unit of Study Overview
- Software Engineering
- Software Modeling
- The Unified Process

WHS Induction



THE UNIVERSITY OF
SYDNEY



Keeping our campus COVID safe

- The University is following NSW Government and NSW Health guidance as a minimum standard in our response to the COVID-19 pandemic.
- NSW Government restrictions can change at short notice.
- Check your student email for updates about University operations and COVID safety precautions.
- Visit our website: sydney.edu.au/covid-19

Follow COVID safety precautions



Stay home if you are sick



Wash hands regularly



Avoid physical greetings



Cough or sneeze into your elbow or tissue



Keep 1.5m away from others where possible



Avoid crowding entrances and exits

sydney.edu.au/covid-19



Feeling unwell?

- **Stay at home**
 - if you are feeling unwell with any COVID-19 symptoms
 - If you have been directed to self-isolate
- **Get tested**
 - If you are feeling unwell with COVID-19 symptoms, please get tested as soon as possible
- **Did you test positive?**

Yes? If you have visited campus within the infectious period, i.e. 72 hours before taking the test, you must advise the University via:

 - email covid19.taskforce@sydney.edu.au, or
 - call +61 2 9351 2000 (select option 1)
- **Stay informed**
 - Monitor [the list of confirmed COVID case locations on campus page](#) to check for potential exposure and [follow NSW Health isolation and testing requirements.](#)

COVID-19 support and care

- Most large lectures will be delivered online and accommodations will be made for international students who have not yet returned to Australia.
- If you become infected with COVID-19 during the semester, or need to isolate, please notify your unit of study coordinator, as with any unexpected absence.
- If COVID-19 isolation or illness impacts assessment, use the usual mechanisms, for example, special consideration to arrange reasonable adjustments.
Visit <https://www.sydney.edu.au/covid-19/students/study-information/test-exams-assessment.html#consideration>.
- Further information on student support can be found on the University website at <https://www.sydney.edu.au/students/support.html>
- Other helpful study information can be found on the website at <https://www.sydney.edu.au/covid-19/students/study-information.html>.

Emergency procedures (on campus)

- In the unlikely event of an emergency, we may need to evacuate the building.
- If we need to evacuate, we will ask you to take your belongings and follow the green exit signs.
- We will move a safe distance from the building and maintain physical distancing whilst waiting until the emergency is over.
- In some circumstances, we might be asked to remain inside the building for our own safety. We call this a lockdown or shelter-in-place.
- More information is available at www.sydney.edu.au/emergency.

Tips for students learning online

- Remember that you are still in a space with other students.
- Mute your microphone when not speaking.
- Use earphones or headphones - the mic is better and you'll disturb others less.
- If you have a webcam, please switch it on so we can see you, if you are comfortable doing so.
- Try not to talk over someone else.
- Some classes may use breakout rooms – engaging fully in these is a great way to meet classmates and your teachers.
- Help your teachers know you're there by participating in chat, polls and other activities during class - we're all in this together.

Tips for learning online

- For tips and guides on learning online and the tools you will use, refer to Learning while off campus resources in Canvas. This is especially useful if it's your first time learning online at university.

The screenshot shows the University of Sydney Canvas interface. On the left is a dark sidebar with icons for Account, Help, Dashboard, Courses, Calendar, Inbox, Studio, and OLE. The main navigation bar includes 'UNIV_STUDENT_CANVAS_GUIDE', 'Pages', and 'Learning while off campus'. Below the navigation, there are links for 'Home', 'View All Pages', 'Modules', 'Pages', and 'Recorded Lectures'. The main content area features a title 'Learning while off campus' and a paragraph about the unique challenges of remote learning. It also includes a list of 'On this page:' items and a photograph of a student sitting by a window using a laptop.

This is a unique situation for all of us. The University is working hard to make sure that you are receiving an excellent educational experience despite possibly not being able to learn on campus. Studying online may be an isolating experience - this page has some ideas to help you adjust to learning while off campus.

Remember to stay positive - this too will pass! Look after yourself and those around you, and prioritise your time accordingly. You will have productive and not-so-productive days - that is OK. Remember to snack healthily, take regular breaks, and reward yourself from time-to-time, especially after a challenging task.

On this page:

- [How can I keep up to date with my study?](#)
- [How should I access classes like lectures and tutorials?](#)
- [What should I do in a live-streamed class?](#)
- [How can I communicate with my teachers?](#)
- [How can I communicate with my classmates?](#)

Assistance

- There are a wide range of support services available for students
 - e.g., <http://sydney.edu.au/study/academic-support/disability-support.html>
 - it is advisable to do this as early as possible
- Please make contact, and get help
- You are not required to tell anyone else about this
- If you are willing to inform the unit coordinator, they may be able to work with other support to reduce the impact on this unit
 - eg provide advice on which tasks are most significant

Special Consideration (University policy)

- If your performance on assessments is affected by illness or misadventure
- Follow proper bureaucratic procedures
 - Have professional practitioner sign special USyd form
 - Submit application for special consideration online, upload scans
 - Note you have only a quite short deadline for applying
 - http://sydney.edu.au/current_students/special_consideration/
- Also, notify coordinator by email as soon as anything begins to go wrong
- There is a similar process if you need special arrangements, e.g., for religious observance, military service, representative sports

Academic Integrity (University policy)

- “The University of Sydney is unequivocally opposed to, and intolerant of, plagiarism and academic dishonesty.”
 - Academic dishonesty means seeking to obtain or obtaining academic advantage for oneself or for others (including in the assessment or publication of work) by dishonest or unfair means.
 - Plagiarism means presenting another person’s work as one’s own work by presenting, copying or reproducing it without appropriate acknowledgement of the source.” [from site below]
 - <http://sydney.edu.au/elearning/student/EI/index.shtml>
- Submitted work is compared against other work (from students, internet, etc)
 - Turnitin for textual tasks (through eLearning), other systems for code
- Penalties for academic dishonesty or plagiarism can be severe
- Complete self-education AHEM1001 (required, canvas → assessment info)

What is academic dishonesty?

The following are some behaviours that are academically dishonest:

- **Plagiarism** (this is the most common form)
- **Collusion** or illegitimate co-operation
- **Recycling** (using your own work from previous assessments)
- **Cheating**, including **contract cheating**
 - sharing questions or accessing solutions on online “help sites”
 - receiving coaching from a private tutoring company on how to complete an assignment
 - asking someone else to write your assignment (for payment or not)
- **Exam cheating** (using prohibited materials, working with others)
- **Fabrication** or falsification of sources, data or results

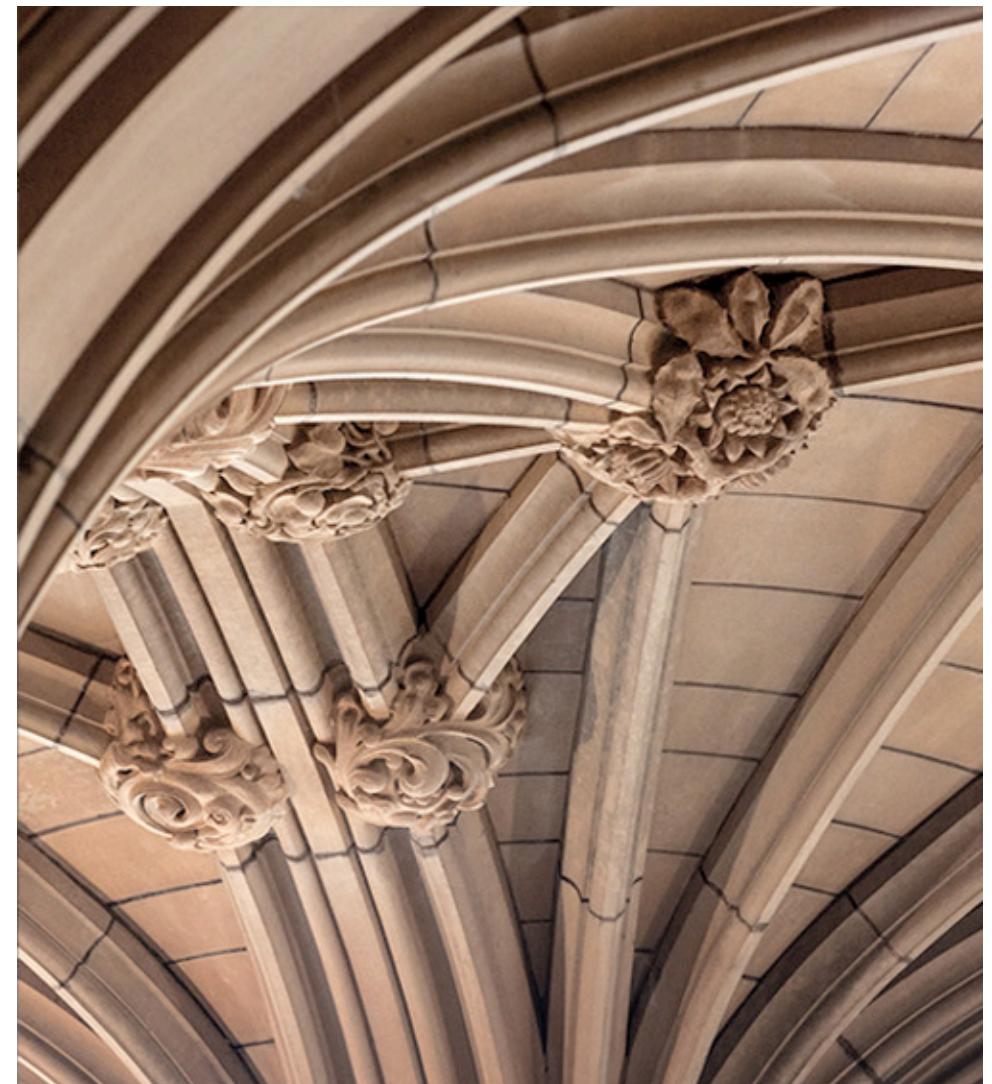
What are the consequences?

- The University has strong mechanisms for detection of potential academic dishonesty.
- Suspected breaches are reported to the faculty educational integrity team for investigation.
- The University is deeply committed to ensuring the integrity of its educational programs and treats integrity breaches seriously. As a result, the **academic consequences** for cheating are numerous.
- You may:
 - need to resubmit a task with a mark penalty or
 - receive a 0 for the assessment or even the unit of study
 - be suspended or even excluded from your studies for serious misconduct

Unit of Study Overview



THE UNIVERSITY OF
SYDNEY



Prerequisites

- This course has the following prerequisites:
 - [INFO1113](#) OR [INFO1103](#) OR [INFO1105](#) OR [INFO1905](#)
- This means that we expect students who enroll in this course to be:
 - Confident Java programmers
 - Familiar with data-structures
- Prohibitions
 - [INFO3220](#) OR [COMP9201](#) (for SOFT2201 students)
 - [INFO3220](#) OR [SOFT2201](#) (for COMP9201 students)

SOFT2201/COMP9201 Lectures

- Online Lecture:
 - Monday, 2pm to 4pm (AEST), Zoom (w1 to w13)
- When you have questions during lectures:
 - if questions are public, feel free to type them on zoom chat and I will read and answer them during the lecture
 - if questions are private, please take a quick note and post it on Ed forum **in private mode after lecture**. I will then answer your private questions there
- How you are suggested to use the Zoom chat:
 - Highly appreciate it that you use the Zoom chat only for the lecture related stuff

SOFT2201/COMP9201 Tutorials

- Tutorial:
 - 2h tutorial, hybrid mode, from Tuesdays to Fridays (w2 to w13)
 - Check your timetable
 - Attend ONE (go to the tutorial you're scheduled for)
 - Great opportunity for interactive and hands-on practical exercises
 - Tutors to supervise your learning and provide guidance
 - Not to debug your code, or solve the problems for you
 - Respect your tutors and value their feedback
 - Respect your classmates

SOFT2201/COMP9201 Staff

Course Coordinator and Lecturer:

- Dr. Xi Wu (xi.wu@sydney.edu.au)
 - Consultation: Monday, 4pm to 5pm (AEST), the same Zoom link for lectures

Tutors:

Abbey Lin	Alex Maher
Daniel Friedrich	Dylan Williams
Hetush Gupta	Liza Fishman
Rayman Tang	Tim Yarkov

Tutor allocation can be found via [Staff and Contact Details](#) on canvas

Language and Tools

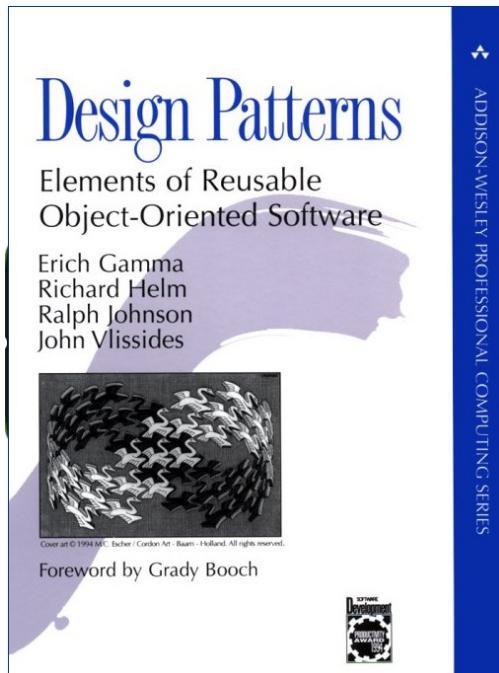
- Your coding will be in Java 17
- You will use IntelliJ (2022.1) for writing your code
- You will use JavaFX for designing your GUIs
- You will use supporting tools such as Gradle, Git, etc.

SOFT2201/COMP9201 Resources

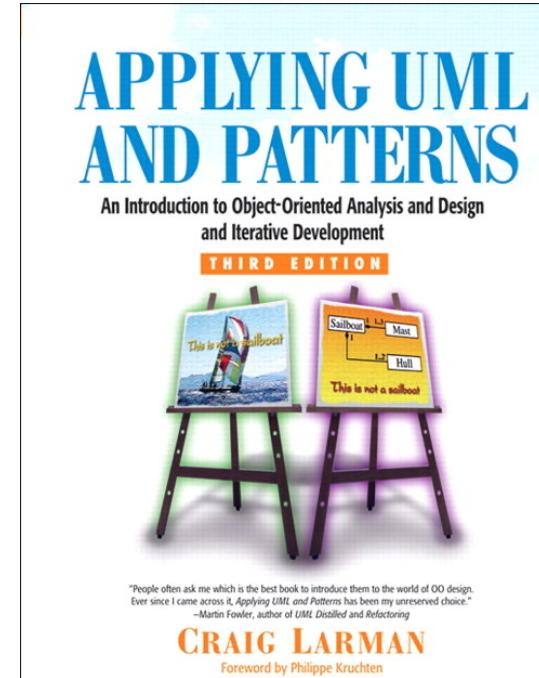
- Canvas (eLearning)
 - Announcements (**important**)
 - Modules: Copies of lecture slides, Tutorial instructions
 - Assignments: Assignment instructions
 - Recorded Lecturers
 - Lecture recordings (usually upload after 1~2 hours of the live lectures)
 - Zoom: Live lectures and tutorials
 - Ed
 - The best place to ask **technical questions** about the course.
 - You will get faster answers here from staff and peers than through email.
 - You can also ask private questions in the private mode

Main Resources

- We recommend the following textbooks



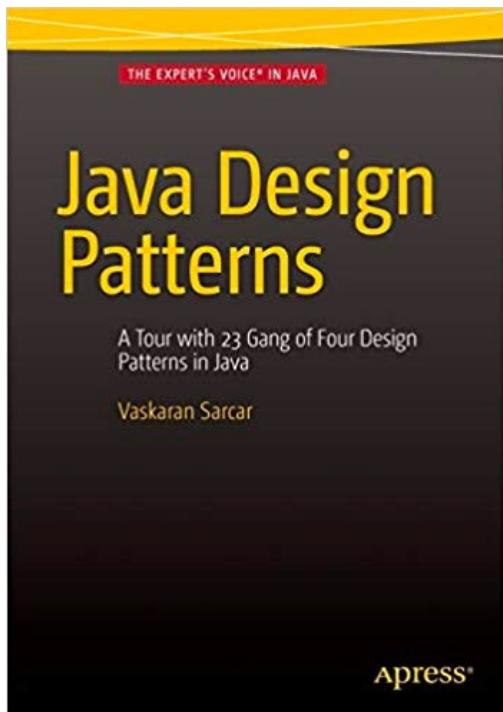
[Link to USYD Library](#)



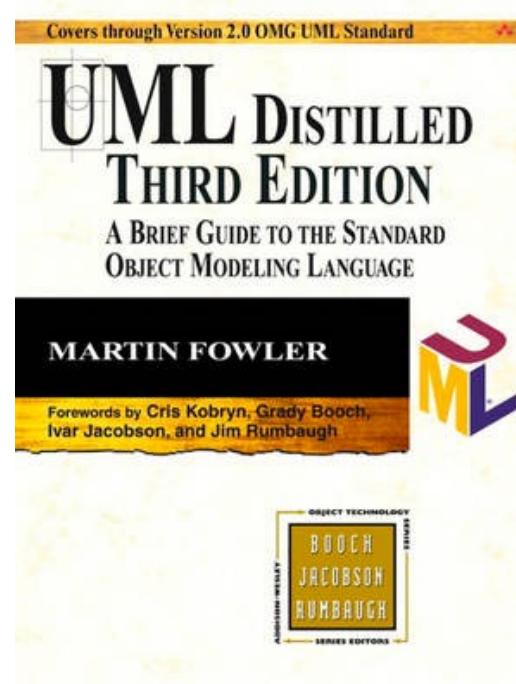
[Link to USYD Library](#)

Additional Resources

- We recommend the following textbooks



[Link to USYD Library](#)



[Link to USYD Library](#)

Topics Overview

Week	Contents		Week
1	Introduction; Software Engineering, Software Modeling, The Unified Process		
2	OO Theory I: Java Knowledge Revisited		
3	UML & Software Modelling Case Studies		
4	OO Theory II: Software Design Principles and Design Smells		
5	Design Pattern: Factory Method & Builder	Design Pattern: Strategy & State	6
7	Testing	Code Review	8
10	Design Pattern: Adapter & Observer	Design Pattern: Prototype & Memento	11
12	Design Pattern: Singleton, Decorator and Facade		
13	Unit Review		

Assessment

What (Assessment)*	When (due)	How	Value
Weekly Exercises	Weekly	Submission on Canvas	10%
Software Construction Stage 1	Week 4	Individual Submission on Canvas	5%
Software Construction Stage 2	Week 8	Individual Submission on Canvas	15%
Software Construction Stage 3	Week 12	Individual Submission on Canvas	20%
Exam	Exam period	Individual exam	50%

Weekly Exercises

- Weekly exercises can be found under each week module on Canvas.
- Exercises cover contents from recent tutorials and lectures
- Timeframe (from week 2 to week 12):
 - Exercises and submission portals are released at 8am (AEST) Tuesday
 - Exercises due and submission portals are closed at 23.59pm (AEST) Saturday
 - You can do the exercise any time after it is open and before its due
 - **However, once it is closed, without SCON, you cannot re-take it if you miss it**
 - **However, only 10 with highest marks out of 11 will be count for final mark**
- Total marks: 10%

Assignment

- Three individual assignments (40% marks):
 1. Design UML diagrams for a given scenario
 - You are going to submit UML diagrams + presentation recording (5 minutes)
 2. Implement a program for a given scenario
 - You are going to submit implementation code with test cases + report about the design principles/design patterns in your implementation
 3. Review and extend somebody else's code
 - You are going to submit implementation code + report about the design principles/design patterns of the receiver code and your extensions

Late Assignments

- Suppose you hand in your assignments after the deadline:
- If you have not been granted special consideration or arrangements
 - A penalty of 5% of the maximal mark will be taken, per day late
 - E.g., your work would have scored 60% and is 1 day late you get 55%
 - **Assignments after 10 calendar days late get 0**
 - Late penalty doesn't apply to weekly exercises; you cannot re-take the exercise if you miss it
- Warning: submission sites get very slow near deadlines
 - Submit early; you can resubmit if there is time before the deadline

Online Final Exam

- 2hrs, will be organized by Exam Office during Exam period
- More details will be announced at the end of the semester (w13)
- Total marks: 50%

Passing this unit

- To pass this unit you must do all of these:**
 - Get a total mark of at least 50%**
 - Get at least 40% for your final exam mark**

SOFT2201/COMP9201 Expectations

- Students attend scheduled classes, and devote an extra 10 hours per week**
 - Prepare and review lecture and tutorial materials**
 - Revise and integrate with ideas**
 - Carry on the required assessments**
 - Practice and self-assess**
- Students are responsible learners**
 - Participate in classes, constructively**
 - Respect for one another (criticize ideas, but not people)**
 - Humility: none of us knows it all; each of us knows valuable things**
 - Check eLearning site very frequently**
- Know the Uni/school policies**

Q&A and Feedback

- A discussion forum is setup:
 - Please **use ED for technical questions** so that everybody can benefit from the questions and answers
- Talk to us:
 - After lectures/tutorials
 - During teaching staff consultation
- Feedback to you will take many forms:
 - Verbally by your tutor
 - As comments accompanying hand marking of your assignment work and exercises work

Advice for doing well in this unit

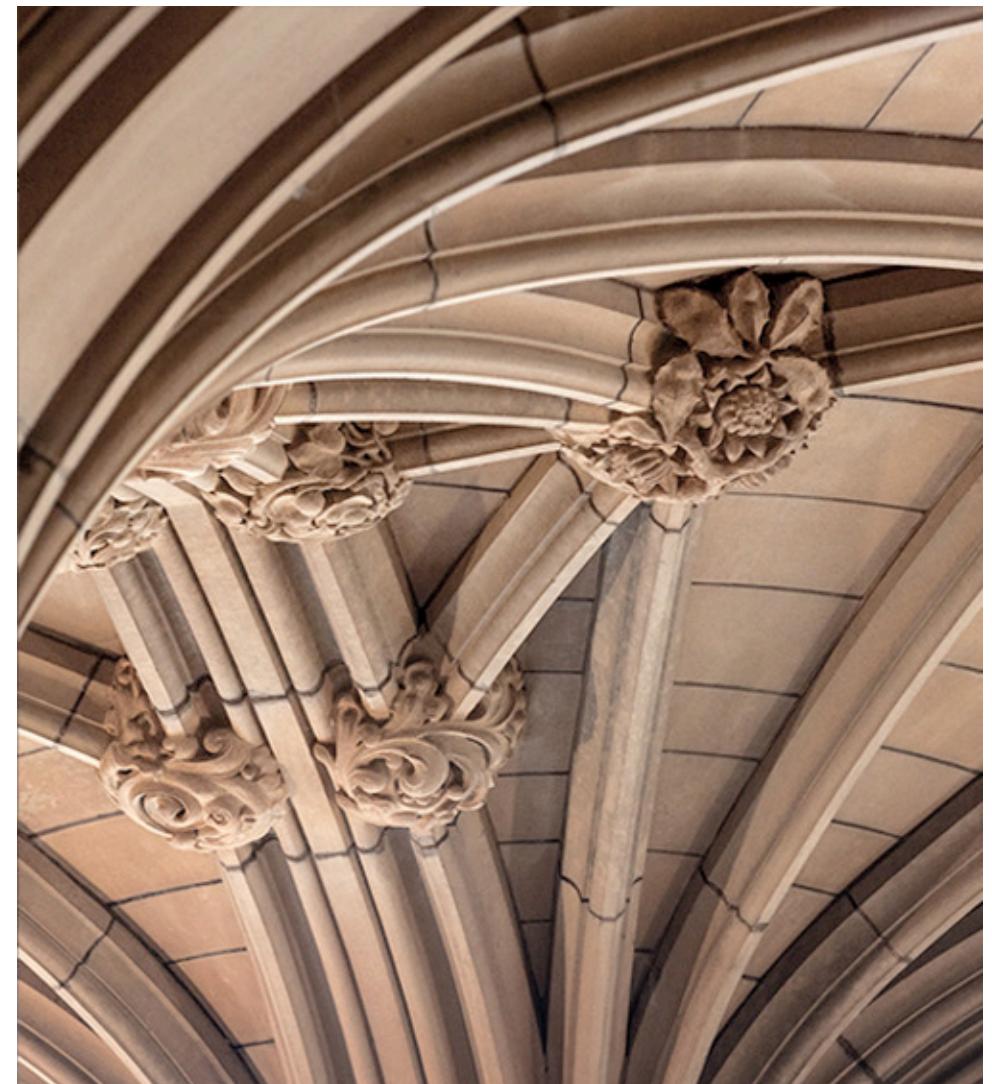
- To do well in this unit you should
 - Organize your time well
 - Devote extra 10 hours a week in total to this unit
 - Read.
 - Think.
 - Practice.

*“Tell me and I forget, teach me and I may remember,
involve me and I learn.” – Benjamin Franklin*

Why Software Engineering?



THE UNIVERSITY OF
SYDNEY



Software is Everywhere!

- Societies, businesses and governments dependent on SW systems
 - Power, Telecommunication, Education, Government, Transport, Finance, Health
 - Work automation, communication, control of complex systems
- Large software economies in developed countries
 - IT application development expenditure in the US more than \$250bn/year¹
 - Total value added GDP in the US²: \$1.07 trillion
- Emerging challenges
 - Security, robustness, human user-interface, and new computational platforms

¹ Chaos Report, Standish group Report, 2014

² softwareimpact.bsa.org

Why Software Engineering?

Need to build high quality software systems under resource constraints

- Social
 - Satisfy user needs (e.g., functional, reliable, trustworthy)
 - Impact on people's lives (e.g., software failure, data protection)
- Economical
 - Reduce cost; open up new opportunities
 - Average cost of IT development ~\$2.3m, ~\$1.3m and ~\$434k for large, medium and small companies respectively³
- Time to market
 - Deliver software on-time

³ Chaos Report, Standish group Report, 2014

Software Failure - Ariane 5 Disaster

What happened?

- European large rocket - 10 years development, ~\$7 billion
- Unmanaged software exception resulted from a data conversion from 64-bit floating point to a 16-bit signed integer
- Backup processor failed straight after using the same software
- Exploded 37 seconds after lift-off



Why did it happen?

- Design error, incorrect analysis of changing requirements, inadequate validation and verification, testing and reviews, ineffective development processes and management

<http://iansommerville.com/software-engineering-book/files/2014/07/Bashar-Ariane5.pdf>

Nissan Recall – Airbag Defect

What happened?

- ~3.53 million vehicles recall of various models 2013-2017
- Front passenger airbag may not deploy in an accident

Why did it happen?

- Software that activates airbags deployment improperly classify occupied passenger seat as empty in case of accident
- Software defect that could lead to improper airbag function (failure)
- No warning that the airbag may not function properly
- Software sensitivity calibration due to combination of factors (high engine vibration and changing seat status)

<http://www.reuters.com/article/us-autos-nissan-recall/nissan-to-recall-3-53-million-vehicles-air-bags-may-not-deploy-idUSKCN0XQ2A8>

<https://www.nytimes.com/2014/03/27/automobiles/nissan-recalls-990000-cars-and-trucks-for-air-bag-malfunction.html>

Software Project Failures

Project	Duration	Cost	Failure/Status
e-borders (UK Advanced passenger Information System Programme)	2007 - 2014	Over £ 412m (expected), £742m (actual)	Permanent failure - cancelled after a series of delays
Pust Siebel - Swedish Police case management (Swedish Police)	2011 - 2014	\$53m (actual)	Permanent failure – scraped due to poor functioning, inefficient in work environments
US Federal Government Health Care Exchange Web application	2013 – ongoing	\$93.7m (expected), \$1.5bn (actual)	Ongoing problems - too slow, poor performance, people get stuck in the application process (frustrated users)

https://en.wikipedia.org/wiki/List_of_failed_and_overbudget_custom_software_projects

Software Engineering - No Silver Bullet⁷

No Silver Bullet - Essence and Accidents in Software Engineering

“There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.” ! - Frederick P. Brooks

- **Essence:** difficulties inherent (or intrinsic) in the nature of SW
- **Accidents:** difficulties related to the production of software
- *Most techniques attack the accidents of software engineering*

⁷ No Silver Bullet - Essence and Accident in Software Engineering - <http://www.itu.dk/people/hesj/BSUP/artikler/no-silver-bullet.pdf>

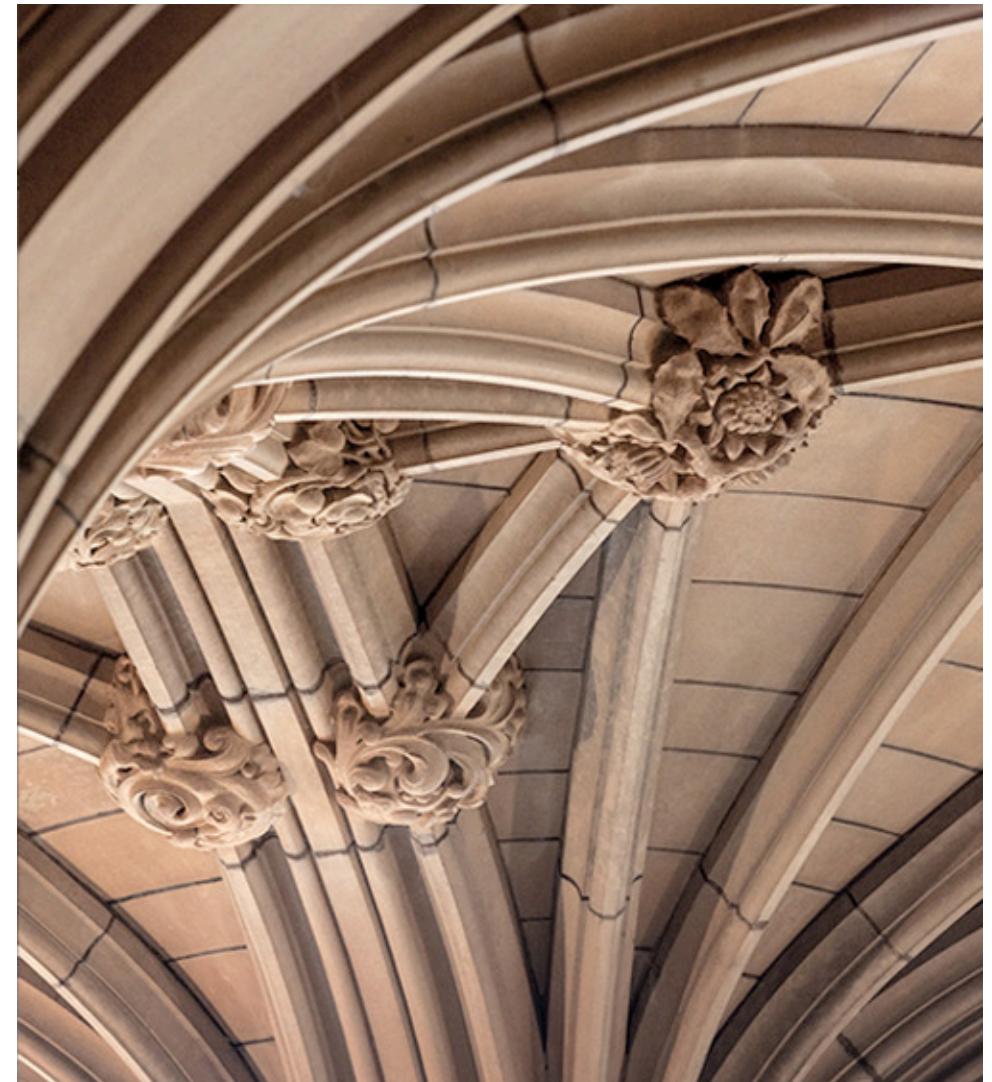
Software Engineering - Essence⁷

- **Complexity**
 - Many diverse software entities - interactions increase exponentially
 - Intrinsic complexity cannot be abstracted - aircraft software, air traffic control
- **Conformity**
 - Arbitrary changes from environment (people, systems) - no unifying principle
- **Changeability**
 - Changing a building model vs. a software
 - Stakeholders understanding of software changes
- **Invisibility**
 - Software is intangible (invisible)
 - Building model vs software models (UML - 13 diagram types)

What is Software Engineering?



THE UNIVERSITY OF
SYDNEY



What is Software Engineering?



<http://www.purplesoft.com.au/wp-content/uploads/2017/03/software.jpg>

Software Engineering

“An engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining it after it is has gone into use.”

- **NOT** programming/coding! a lot more is involved
- Theories, methods and tools for cost-effective software production
- Technical process, project management and development of tools, methods to support software production
- System Engineering (Hardware & Software) - software often dominates costs

Software Engineering

- Theories, methods, tools, techniques and approaches
 - Solve concrete SWEN problems
 - Increase productivity of the engineering process
 - Produce effective software
 - Produce efficient software
 - Control social and technical aspects in the development process
 - Manage complexity, changeability, invisibility and conformity

Software Engineering

“The Roman bridges of antiquity were very inefficient structures. By modern standards, they used too much stone, and as a result, far too much labour to build. Over the years we have learned to build bridges more efficiently, using fewer materials and less labour to perform the same task.”!

- Tom Clancy (*The Sum of All Fears*)

- The art of managing social, economical and technical factors
 - Efficient and effective development processes and management
 - Delivering software on-time and on-budget with all the necessary requirements
- The art of analysing and managing complexity
 - Ability to understand complex systems
 - Apply various abstraction and analytical thinking

Software Engineering Fundamentals

- Software processes for managing and developing of SW systems
 - Waterfall vs. Incremental and agile software development
- Attributes of good software system
 - Maintainability, dependability and security, efficiency and acceptability
- Importance of Dependability and Performance
- Need for specifications and requirements
- Software reuse to save costs
 - Careful consideration – Ariane 5 reused software from Arian 4!

Software Engineering Body of Knowledge

- Software Requirements
- **Software Design / Modelling**
- **Software Construction**
- Software Testing
- Software Maintenance
- Software Configuration Management
- Software Engineering Process
- Software Engineering Tools and Methods
- Software Quality

IEEE® computer society

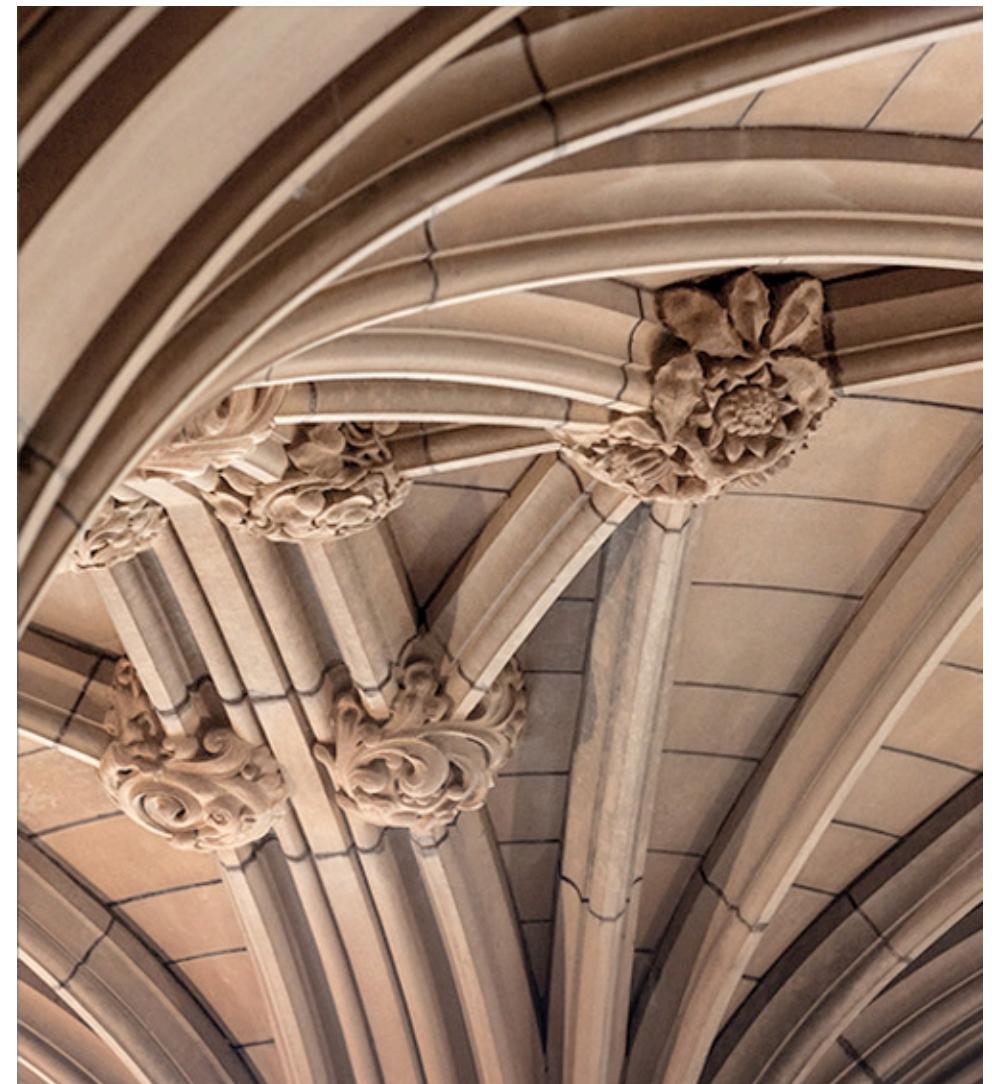


Software Engineering Body of Knowledge (SWEBOK) <https://www.computer.org/web/swebok/>

Software Design/Modelling & Construction



THE UNIVERSITY OF
SYDNEY



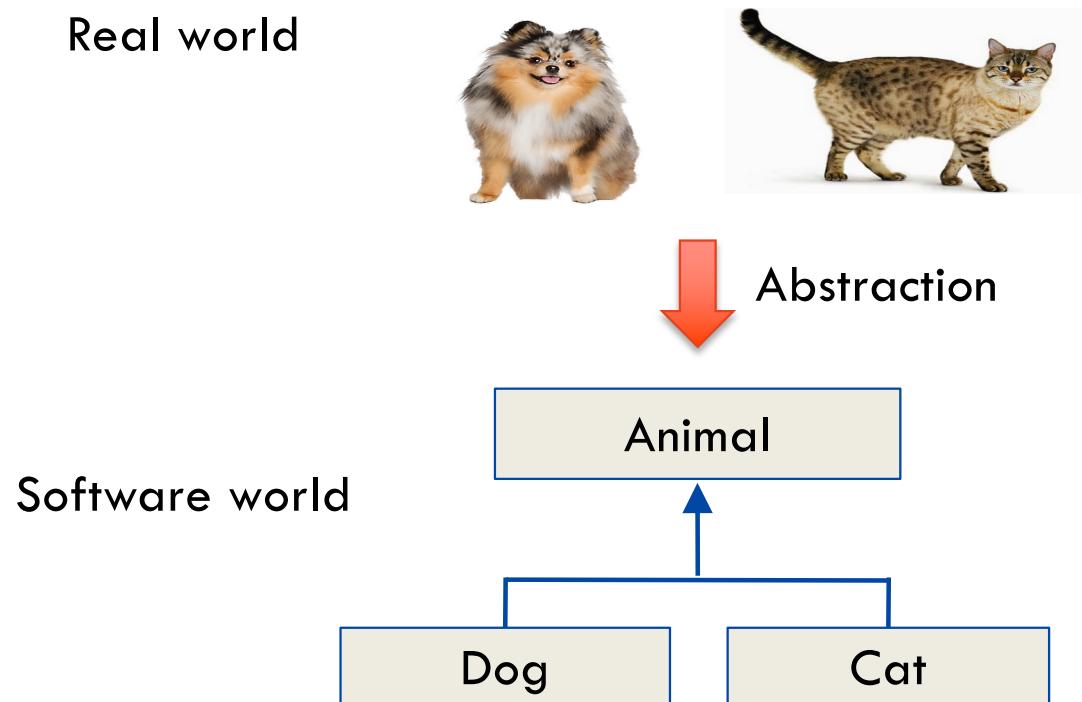
Software Modelling

- The process of developing conceptual models of a system at different levels of abstraction
 - Fulfil the defined system requirements
 - Focus on important system details to deal with complexity
- Object-oriented design approach
 - Concepts in the domain problem are modelled as interacting objects
- Using graphical notations (e.g., UML) to capture different views or perspectives

Software Modelling – The Art of Abstraction

- Conceptual process to derive general rules and concepts from concrete information in a certain context
- Analysis and understanding of domain-specific problems
 - Decompose large problems into smaller understandable piece
 - Language required to break down complexity, i.e., find abstractions
- SW models with different levels of abstraction
 - Focus on certain details in each phase of the SW development process
- SW models with different views/perspectives
 - Time, structural, requirements, etc.

Software Abstraction – Example

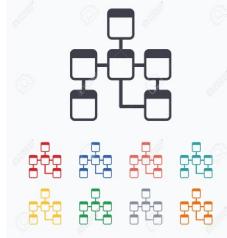


```
1 public class Animal {  
2     public void sleep () {}  
3 }  
4  
5 public class Dog extends Animal {  
6     public void woof {}  
7 }  
8  
9 public class Cat extends Animal {  
10    public void meow {}  
11 }
```

Cat <http://s1.thinpic.com/images/ZP/L4FtpQYKNZzCXV8r34PWhqCF.jpeg>
Dog [https://s7d2.scene7.com/is/image/PetSmart/SV0401_CATEGORY HERO-Dog-Dog-20160818?\\${SV0401\\$](https://s7d2.scene7.com/is/image/PetSmart/SV0401_CATEGORY HERO-Dog-Dog-20160818?${SV0401$)

Data Abstraction – Example

View Table Data - MRCWORKLIB.DMCMP100					
Customer Number	Company Number	Customer Name	Customer Address Line 1	Customer address line 2	
100001	25	BIKER R US	499 GRANVILLE STREET	SUITE 2	
100002	25	AAA-EQUIPMENT BIKE SALES	2412 FAR HILLS AVE.	SUITE 214	
100003	25	ABC0 CORPORATION	1870 S. HIGH ST.	SUITE 5	
100004	25	AIR CYCLE SHOPPE	535 S. FRONT STR.		
100005	25	ACE HARDWARE	999 BROADWAY		
100006	25	STATE OF NEW YORK	163 WEST 2ND ST		
100007	25	A-1 CYCLE CITY	500 W NORTH AVE		
100008	25	BERNARDS SCHWINN CYCLERY	510 FREDERICK ST		
100009	25	WESTFORTH SPORTS INC	4704 ROOSEVELT PL		
100010	25	GRAND CYCLE	1417 CLARK STREET		



View level



Logical level
(conceptual data
model)



Physical level
(data model)



UML Principles



- Graphical notations to visually specify and document design artifacts software systems using OO concepts
 - Industry standard managed by Object Management Group (OMG)
 - Is not OO Analysis and Design (OOA&D) or method!
 - Is a language for OOA&D communicating visual models of the software being designed
 - Is not a skill, but how to design software with different level of abstractions
 - Many software diagramming tools, hand sketches are good too
- Combines techniques from data modeling (ER diagrams), business modeling (workflows), object and component modeling
- Capture system activities (jobs), system components and its interaction, software entities and its interaction, run-time behavior, external interfaces



UML Principles (Cont.)

- UML is not “Silver Bullet”
 - No tool/technique in software will make dramatic order-of-magnitude difference in productivity, reliability or simplicity
 - Design knowledge and thinking in objects is critical
- Three ways to apply UML
 - Sketching to explore part of the problem or solution space
 - Blueprint: detailed design diagrams for:
 - Reverse engineering to visualize and understand existing code
 - Forward engineering (code generation)
 - Programming language (Model Driven Engineering): executable specification of a system
- “Agile Modeling” emphasizes UML as a sketch
 - Not waterfall mindset

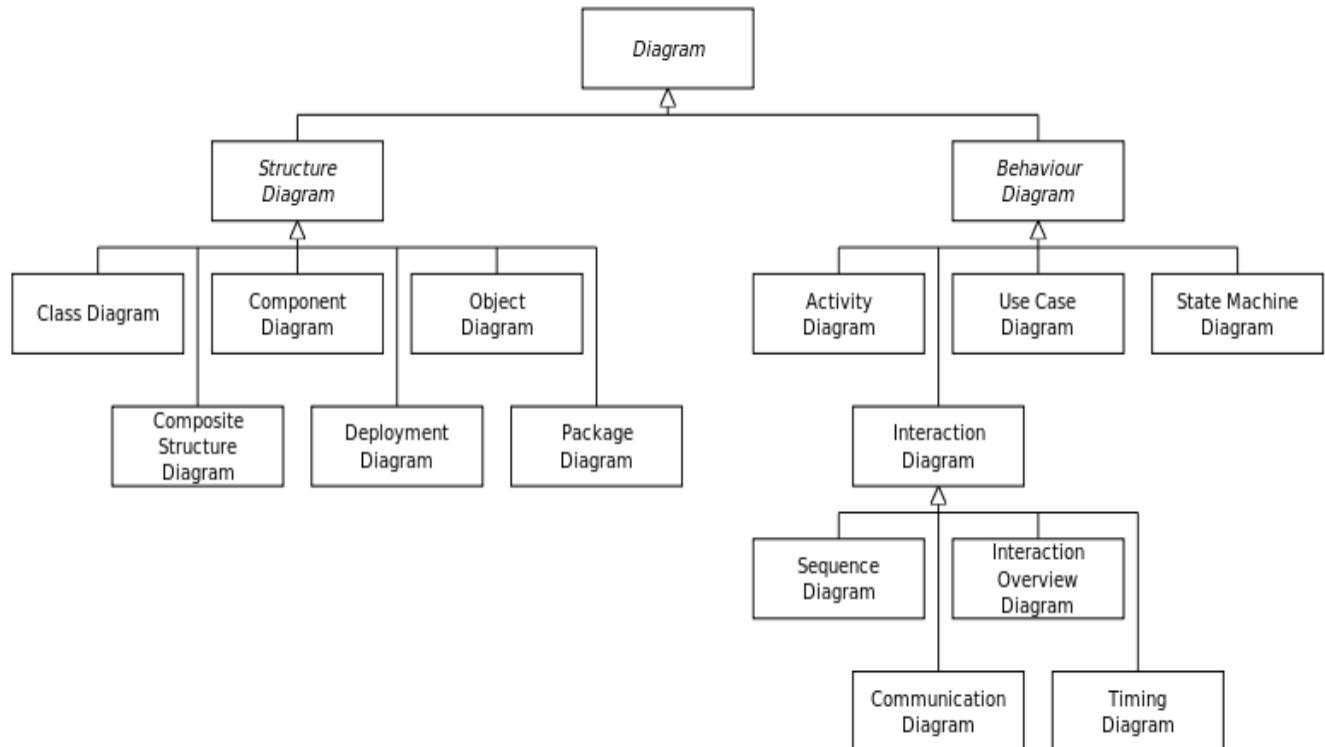
UML Diagrams

Structural (static) View

- Static structure of the system
(objects, attributes, operations and relationships)

Behavioural (dynamic) View

- Dynamic behavior of the system
(collaboration among objects, and changes to the internal states of objects)
- Interaction (subset of dynamic view) - emphasizes flow of control and data

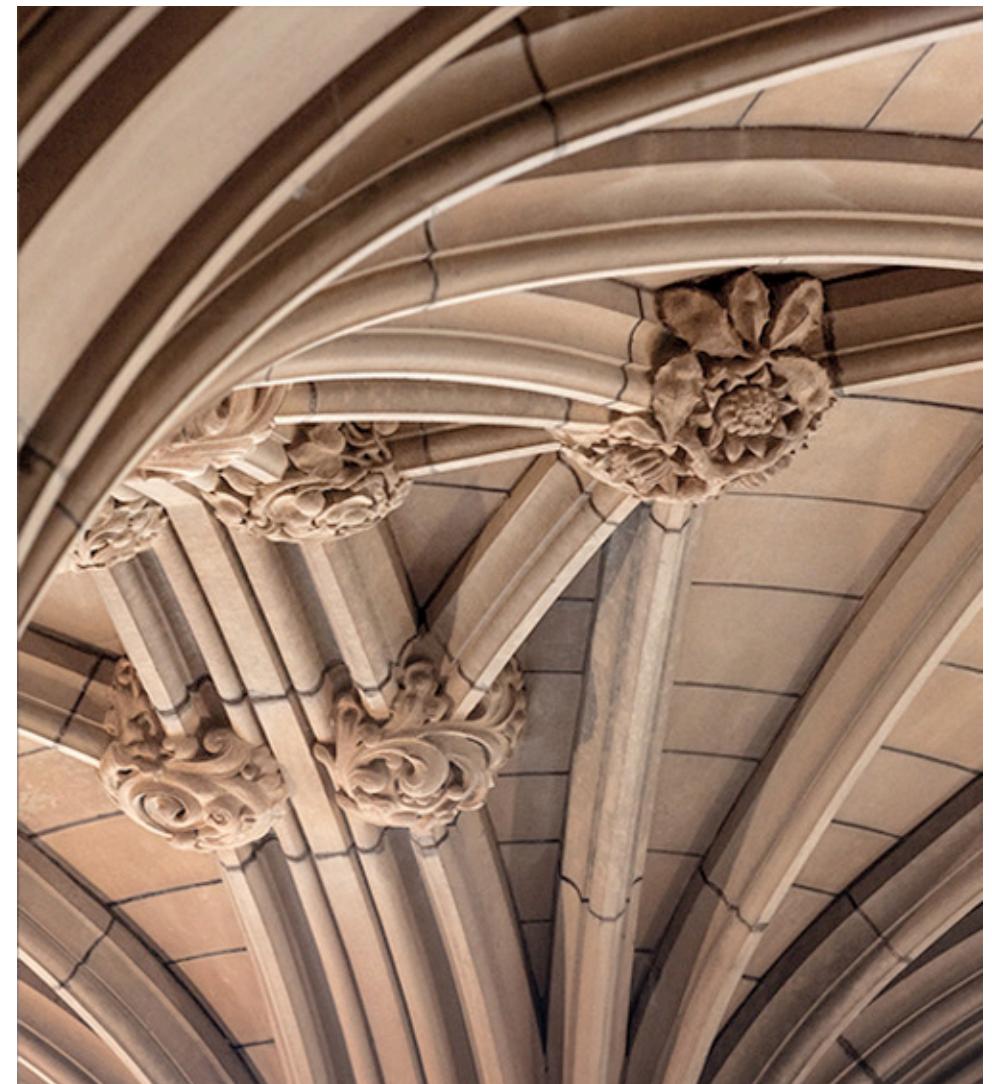


https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/UML

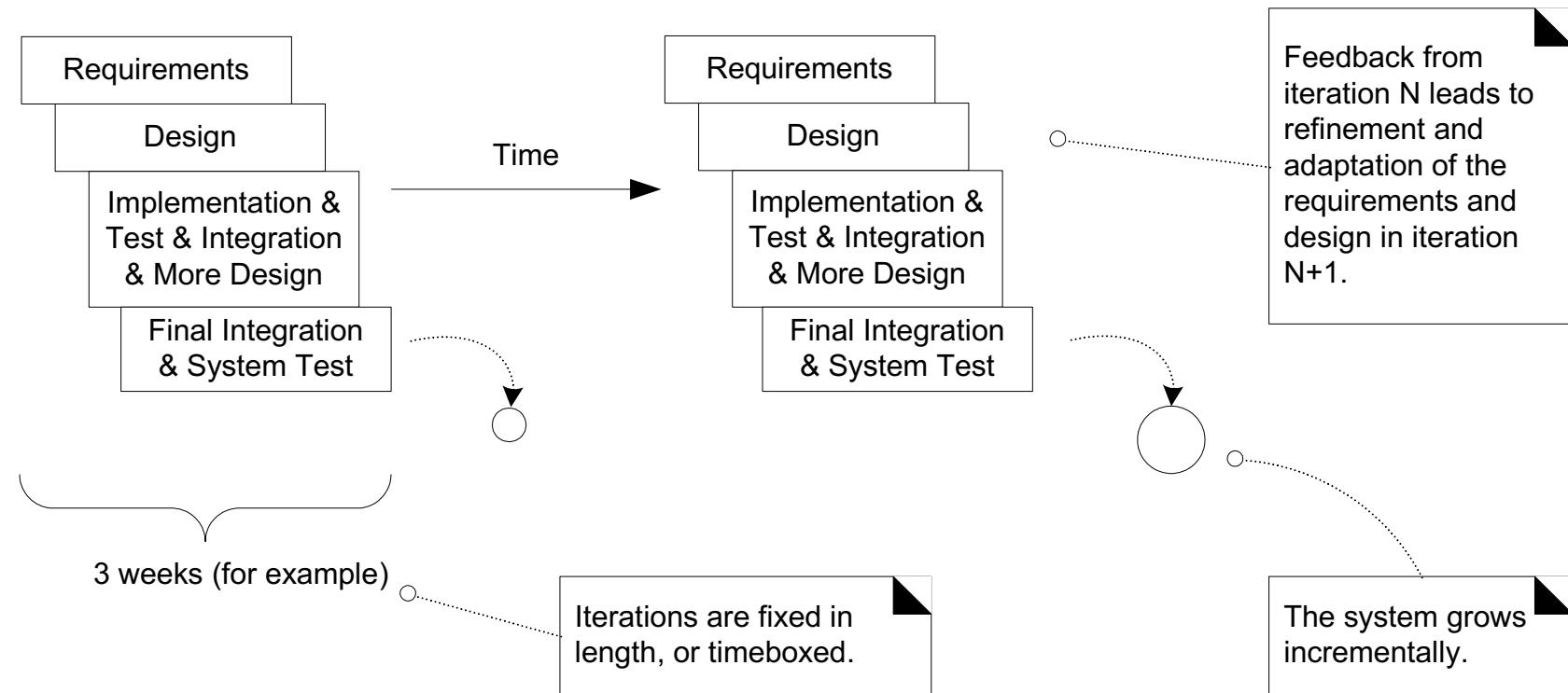
The Rational Unified Process



THE UNIVERSITY OF
SYDNEY

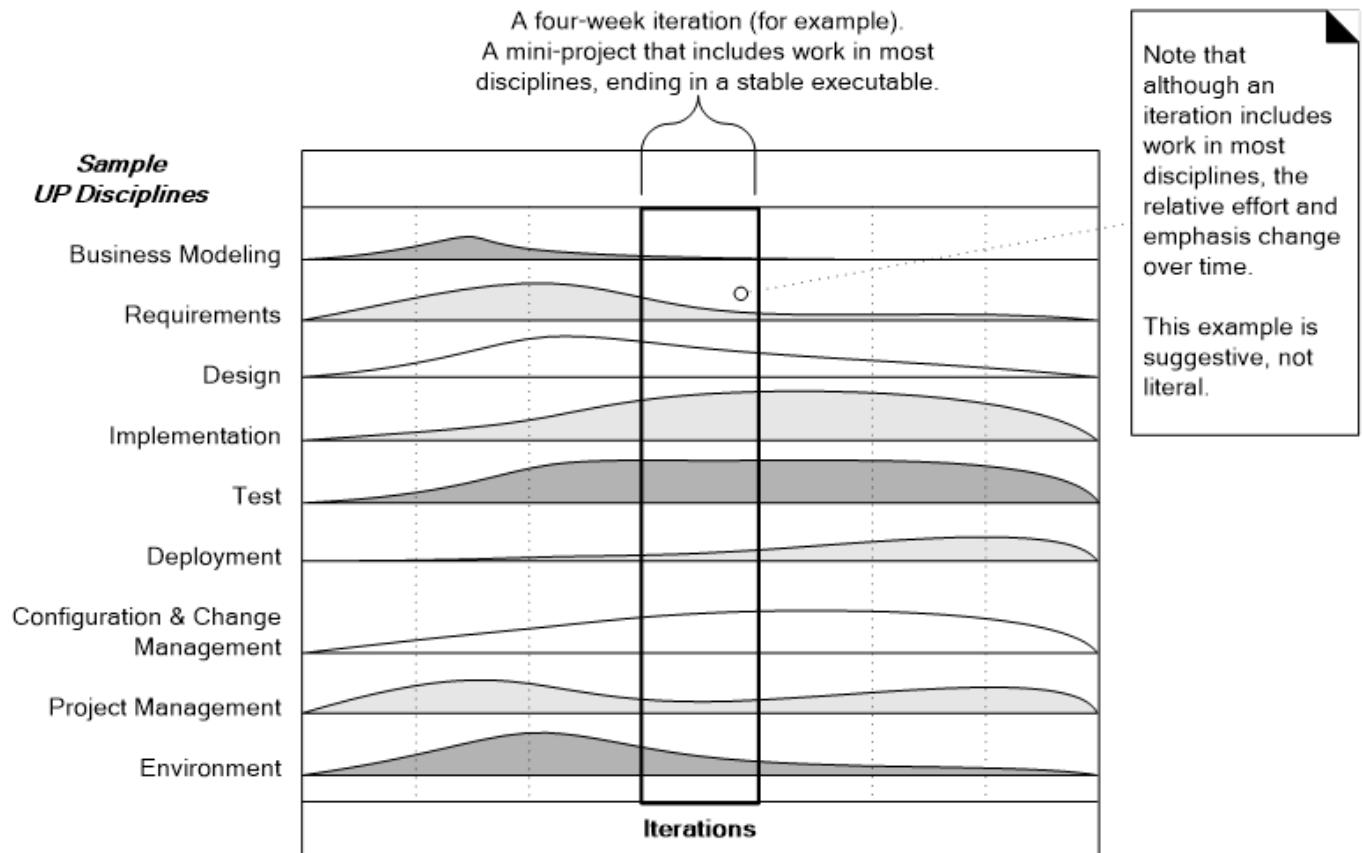


Iterative and Evolutionary Development

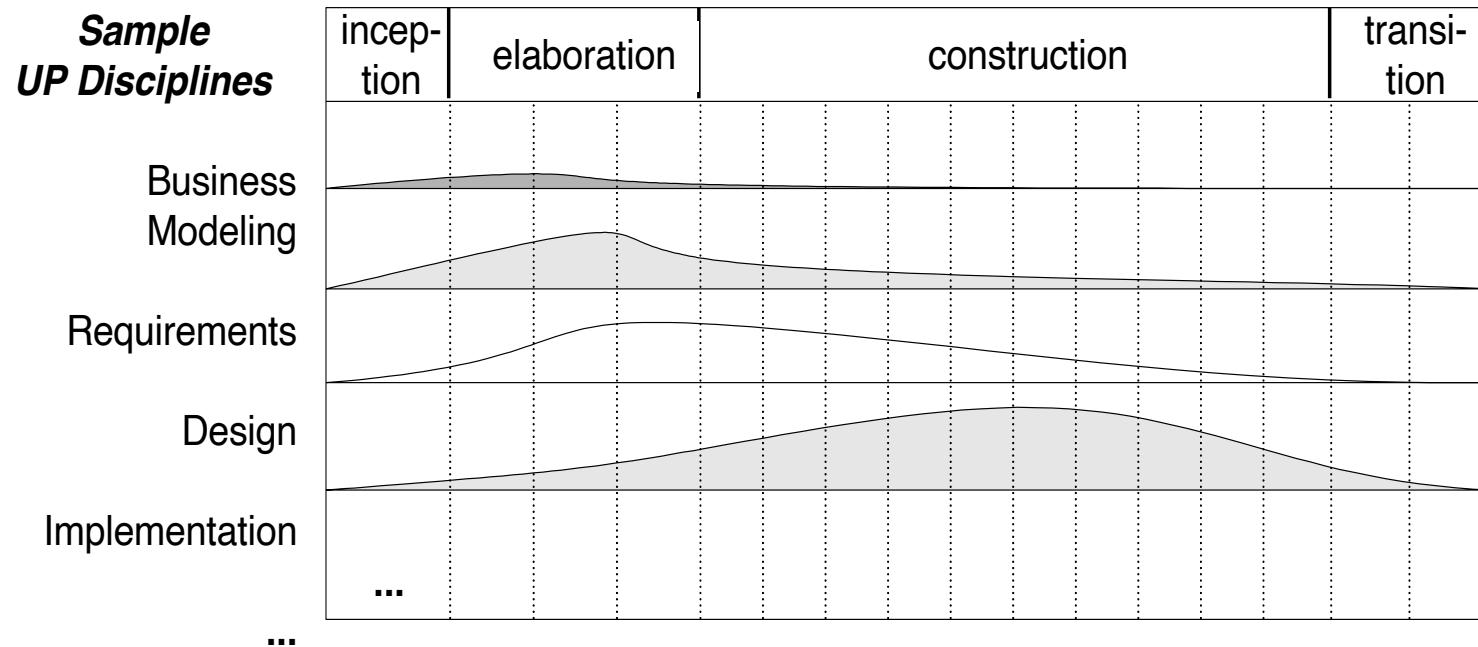


Rational Unified Process (UP)

- Software development process utilizing iterative and risk-driven approach to develop OO software systems
- Iterative incremental development
- Iterative evolutionary development



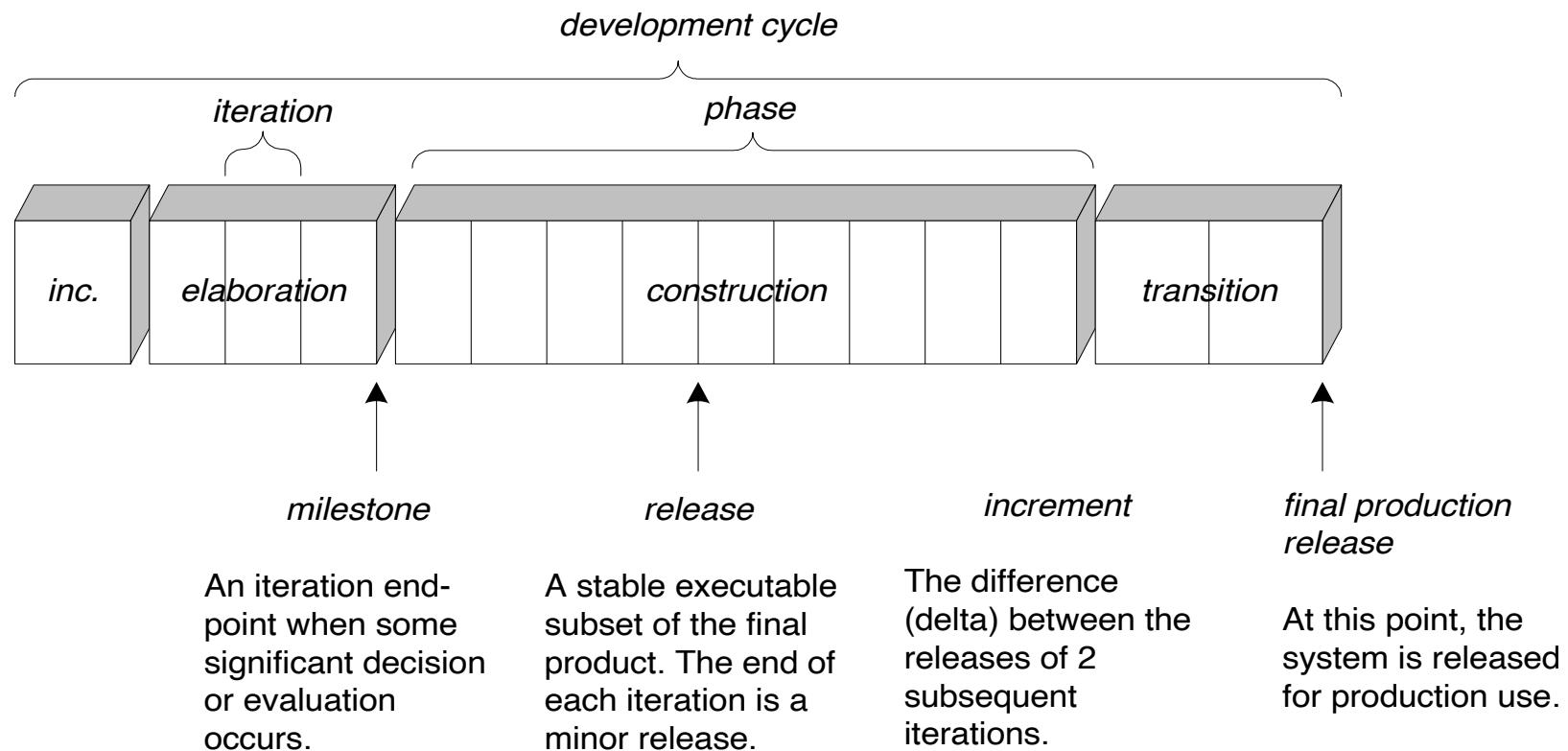
UP Phases and Disciplines



The relative effort in disciplines shifts across the phases.

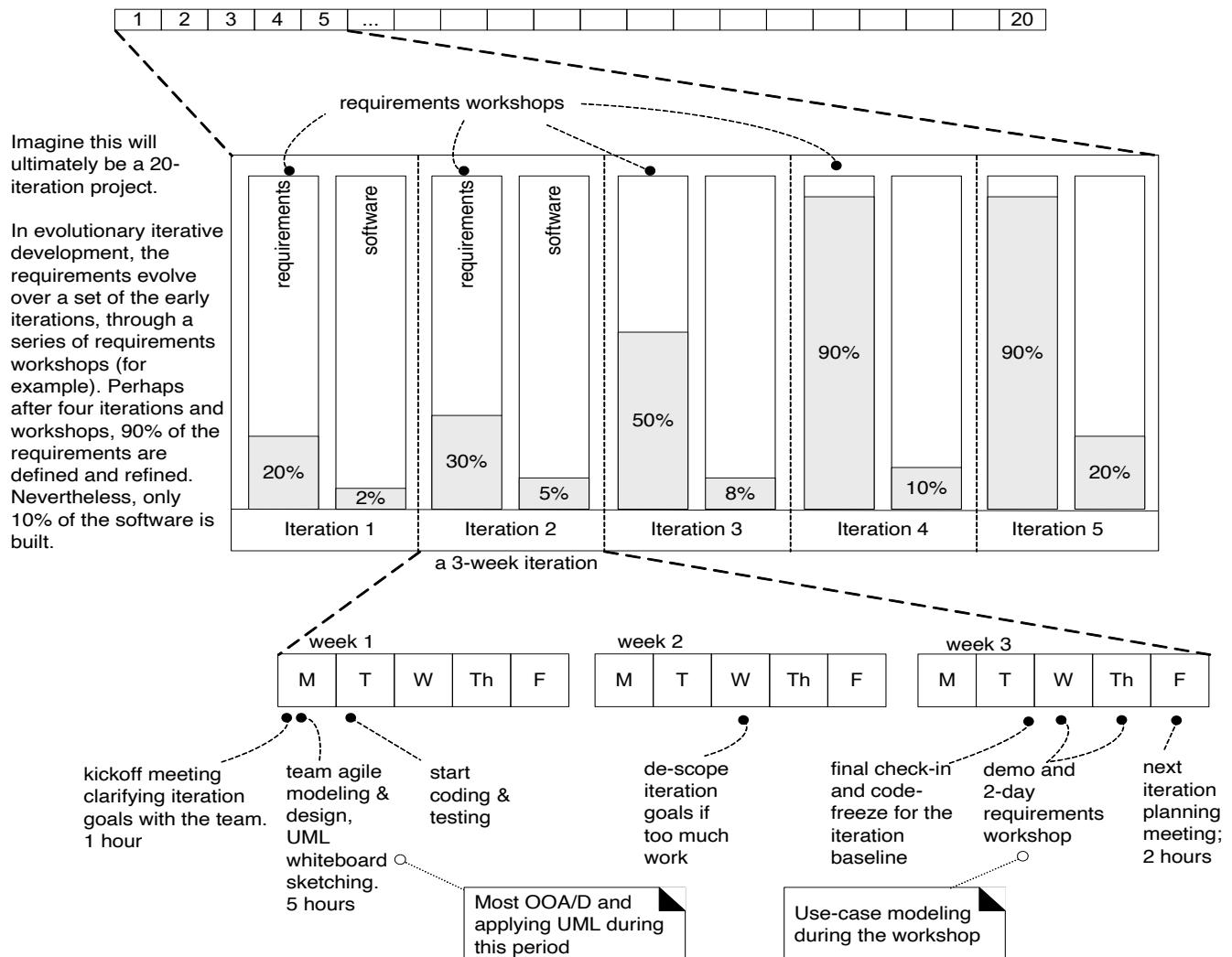
This example is suggestive, not literal.

UP Phases and Disciplines



UP - Example

- Iterative and evolutionary analysis and design
 - First five iterations of 20
 - 3-week iteration



Software Construction / Implementation

- Realization of design to produce a working software system
 - Meet customer requirements
- Design and implementation activities often interleaved
 - Agile development to accommodate for changes
- Object-Oriented design and Implementation model
 - Encapsulation
 - Abstraction
 - Reuse
 - Maintenance

Tasks for Week 1

- Install gradle and IntelliJ**
 - Instruction can be found in Ed announcement (Tuesday Morning)**
 - Get help from staffs on Ed forum if you have questions on installation**

What are we going to learn next week?

- OO Theory I: Java Knowledge Revisited
 - Encapsulation
 - Inheritance
 - Variable Binding & Polymorphism
 - Virtual Dispatch
 - Abstract Classes
 - Interfaces

Software Design and Construction 1

SOFT2201 / COMP9201

OO Theory in Java

Dr. Xi Wu

School of Computer Science



**Which feature of Java/OOP do you
think is the most interesting to you?
– share via Zoom chat**

Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Announcement

- Tutorial starts from this week (week 2)
 - Please find your tutorial schedule via your timetable
 - Tutorials will be either on campus or live Zoom session (no recording)
- Weekly exercise will be open at 8am Tuesday and close at 11.59pm Saturday, no late submission except SCON

Overview

- Types
- Encapsulation
- Inheritance
- Variable Binding & Polymorphism
- Virtual Dispatch
- Abstract Classes
- Interfaces

Types in Java

- Overview
 - Java is a strongly typed language
 - Primitive types: built-in types of the virtual machines
 - Cannot be changed by programs
 - Have same meaning across computer platforms
- Compositional types
 - Interfaces
 - Classes

Primitive Types in Java

- Primitive types represent basic data-types
- In-built and cannot be changed in programs
- Integral types
 - byte, short, int, long
- Floating point number types
 - float, double
- Character type
 - char
- Boolean
 - boolean

Integral and Floating Types

Type	Internal	Smallest Value	Largest Value
byte	8 bits	-128	+127
short	16 bits	-32,768	+32,767
int	32 bits	-2,147,483,648	+2,147,483,647
long	64 bits	-2.3E+18	+2.3E+18
float	32 bits	-/+1.4e-45	-/+3.4e+38
double	64 bits	-/+4.9e-324	-/+1.7e+308d

Conversion between Primitive Types

- Implicit widening of integral & floating point number types
 - byte to short, int, long, float, or double
 - short to int, long, float, or double
 - char to int, long, float, or double
 - int to long, float, or double
 - long to float or double
 - float to double
- Everything else requires type casts or is not possible

Casting/Widening Matrix

From/To	byte	char	short	int	long	float	double	boolean
byte								n/a
char	(byte)		(short)					n/a
short	(byte)	(char)						n/a
int	(byte)	(char)	(short)					n/a
long	(byte)	(char)	(short)	(int)				n/a
float	(byte)	(char)	(short)	(int)	(long)			n/a
double	(byte)	(char)	(short)	(int)	(long)	(float)		n/a
boolean	n/a	n/a	n/a	n/a	n/a	n/a	n/a	

Example for Implicit Widening and Casting

```
...
byte x = 10;

/* implicit widening */
short y = x;
long z = x;
float f = x;

/* casting */
byte p = (byte) z;
short q = (short) f;
```

Silent Failure of Types

- Example:

```
...
int i = Integer.MAX_VALUE;
System.out.println(i);
i = i + 1;
System.out.println(i);
...
```

- Integer overflow occurs for variable **i**
- Type system cannot prevent this at compile-time
 - weak notion of correctness

Primitive Types

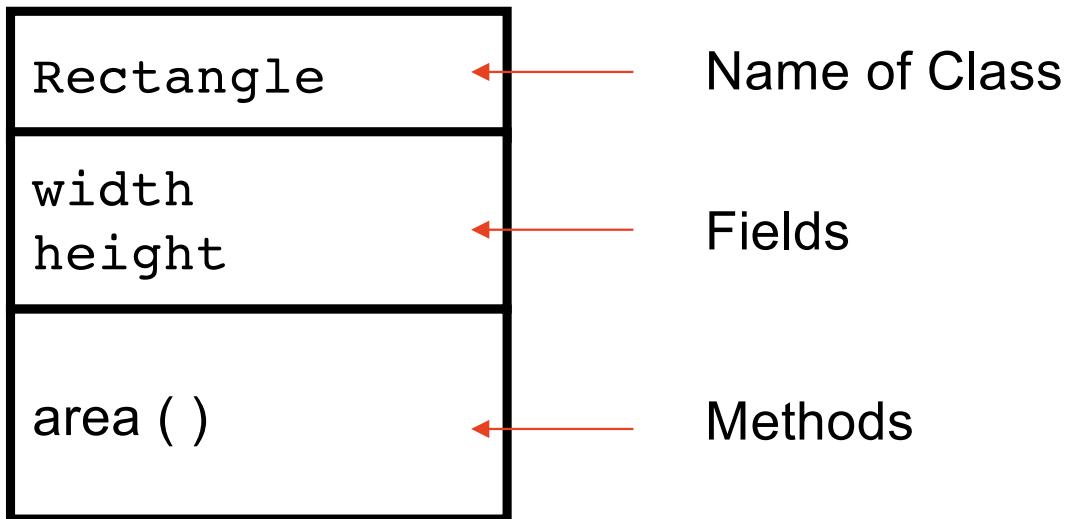
- Basic data-types
 - Behave the same on all platforms running Java's virtual machine
- Primitive Types are not classes nor interfaces
- Primitive Types are building blocks for more complex types
- Beside the storing the actual value there are very little additional overheads
 - Note that classes have extra information to keep of dynamic runtime info
- Operations on primitive data-types are very efficient

Classes in Java

- Classes are types
 - Composite type consisting of fields(=state) and methods(=behavior)
- Class Abstraction
 - Separate class implementation from the use of a class
 - User of a class does not need to know the implementation / just the public methods
 - The implementor provides the implementation of the class
 - The details are encapsulated and hidden from the user
- Class creates objects (instantiate)
- Object communicates with each other via methods

Classes

- A basic class has a name, fields, and methods:



Classes (cont'd)

- The basic syntax for a class is

```
class <class name> [ extends <super class> ]
{
    <field declarations>
    <method declarations>
}
```

- Example:

```
class Rectangle
{
    <field declarations>
    <method declarations>
}
```

Adding Fields: Class Rectangle with Fields

- Add *fields*

```
public class Rectangle {  
    public double width, height;  
}
```

- The fields (data) are also called the *instance* variables
- Fields can be
 - Primitive types (int, bool, etc.)
 - References to other class instances

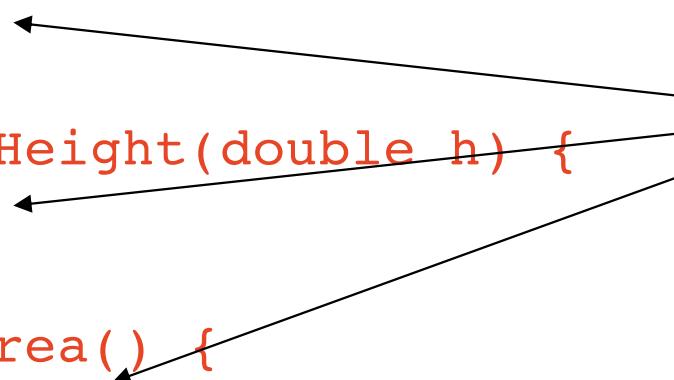
Adding Methods

- To change / retrieve state of a class, methods are necessary
- Methods are declared inside the body
 - Methods are messages in an object-oriented programming sense
 - A return type must be specified
 - A list of arguments with their types must be specified
- The general form of a method declaration is:

```
type Method (argument-list) {  
    Method-body;  
}
```

Adding Methods to Rectangle

```
public class Rectangle {  
    public double width, height; // fields  
  
    public void setWidth(double w) {  
        width = w;  
    }  
    public void setHeight(double h) {  
        height = h;  
    }  
    public double area() {  
        return width * height;  
    }  
}
```



Method Body

Mutators

- Methods that change the state of a class object are called mutators
- Example:

```
public void setWidth(double w) {  
    width = w;  
}  
public void setHeight(double h) {  
    height = h;  
}
```

- Gives clients of the class write access to the state

Accessor

- Methods that read the state of a class object are called accessor
- Example:

```
public double area() {  
    return width * height;  
}
```

- Gives clients of the class read access to the state

Constructors

- Special method in class whose task is to initialize fields of the class
- The name is the same name as class
- Constructors are invoked whenever an object of that class is created
- It is called constructor because it constructs values of data members.
- A constructor cannot return a value
- There can be several constructors for a class
 - Default constructor with no parameters
 - Parameterized constructors

Example: Default Constructor

- Default constructor

```
public class Rectangle {  
    public double width, height;  
    public Rectangle() { // default constructor  
        width = 1.0;  
        height = 1.0;  
    }  
}
```

- Java has a synthesized default constructor when not specified

Example: Parameterized Constructor

- Parameterized constructor:

```
public class Rectangle {  
    public double width, height;  
  
    // parameterized constructor  
    public Rectangle(double w, double h) {  
        width = w;  
        height = h;  
    }  
}
```

Example: Multiple Constructors

- Default and parameterized constructor (overloading)

```
public class Rectangle {  
    public double width, height;  
    // default constructor  
    public Rectangle() {  
        width = 1.0;  
        height = 1.0;  
    }  
    // parameterized constructor  
    public Rectangle(double w, double h) {  
        width = w;  
        height = h;  
    }  
}
```

Creating Instances of a Class

- Objects are created dynamically using the **new** keyword.
- The new operator creates a new object on the heap
 - The object will be selected as a candidate if not further used
 - The object becomes a candidate for automatic **garbage collection**
 - Garbage collection kicks in periodically and releases candidates
 - developers don't need to delete their own objects
- Example:

```
<Class> x = new <Class> (<args>);
```

 - Define a reference variable X of type class that points to a new instance

Example: Creating Instances of a Class

- rec1 and rec2 refer to Rectangle objects

```
rec1 = new Rectangle();    rec2 = new Rectangle(10,1);
```



- First instance calls default constructor
- Second instance calls parameterized constructor

Copy Constructor

- Java has no copy constructor
- We can simulate copy constructor by having a constructor with objects of same type as argument
- All classes are derived from class Object
- Object has the method clone()

Example: Copy Constructor

- Copy constructor

```
public class Rectangle {  
    public double width, height;  
    // copy constructor  
    public Rectangle(Rectangle o) {  
        width = o.width;  
        height = o.height;  
    }  
}
```

Accessing Fields and Methods

- Access is done via the . operator
- Access to field and methods of an instance in the same style

```
<object> . <field>
<object> . method(<args>)
```

- Access modifier is important
 - public/protected/private
 - Controls whether access is permitted at client or member level

Example: Accessing fields / methods

- Using object methods:

```
Rectangle rec = new Rectangle();  
  
rec.width = 1.0;  
rec.height = 100.0;  
double area = rec.area();
```

send ‘message’ area to an instance of type Rectangle which rec is referring to

Access Modifiers

- Access modifiers provide encapsulation for object-oriented programming
- Protect state from client
- Access modifiers work on different levels:
 - Class level
 - Member level
 - Client level

Access Modifier: public

- A class that is labeled as public is accessible to all other classes inside and outside a package

```
package com.foo.goo;

public class test1 {
    public void test();
}
```

```
package com.foo.hoo;
import com.foo.goo;

public class test2 {
    public void test(){
        test1 x = new test1();
    }
}
```

Access Modifier: public

- A class marked as public is accessible to all classes with the import statement
- There is no need for import statement if a public class is used in the same package

Access Modifier: default

- Default access is a class definition with no specified access qualifier
- Classes marked with default access are visible only in package but not outside of package
- Note that protected and private classes are not available for packages

Member access modifiers for methods and fields

- **Public**
 - Accessible for client, class, and sub-classes
- **Protected**
 - Accessible for class, and sub-classes
- **Private**
 - Accessible for class itself only

Example: Member access modifiers

- Example: public

```
public class Rectangle {  
    public double width, height;  
}  
  
// client can access state  
Rectangle x = new Rectangle();  
x.width = 100;  
x.height = 100;
```

Example: Member access modifiers

- Example: protected

```
public class Rectangle {  
    protected double width, height;  
}  
  
// client cannot access state  
Rectangle x = new Rectangle();  
x.width = 100; // access error!  
x.height = 100; // access error!
```

Example: Member access modifiers

- Example: private

```
public class Rectangle {  
    private double width, height;  
    Rectangle(double w, double h) {  
        width = w; // access is ok  
        height = h; // access is ok  
    }  
}  
public class Square extends Rectangle {  
    Square(double w) {  
        super.Rectangle(w,w);  
    }  
    void setWidth(double w) {  
        super.width = w; // access error!  
        super.height = w; // access error!  
    }  
}
```

Encapsulation

- Wrap data/state and methods into a class as a single unit
- Multiple instances can be generated
- Protect state via setter (mutator)/getter (accessor) methods

```
class Rectangle {  
    double length; double width;  
    double getLength() { return length; }  
    double getWidth() { return width; }  
    double area() { return length * width; }  
}
```

Sub-Classes

- Classes permit inheritance
 - Methods and fields from a super-class can be inherited
- Ideal for software-engineering
 - Reuse of code
- Java uses “extends” keyword for extending from existing class
- Single-inheritance paradigm
 - Class hierarchy can be a tree most
- If no super-class is specified, default class Object becomes super-class

Inheritance

- Sub-class inherits from super-class: methods, variables, ...
- Reuse structure and behavior from super-class
- Single inheritance for classes

```
class Rectangle extends Object {  
    double length; double width;  
    double area() { return length * width; }  
}  
  
class ColouredRectangle extends Rectangle {  
    int colour;  
    int colour() { return colour; }  
}
```

Variable Binding, Polymorphism

- Object (on heap) has single type when created
 - Type cannot change throughout its lifetime
- Reference variables point to null or an object
- Type of object and type of reference variable may differ
 - e.g., Shape x = new Rectangle(4,2);
- Understand the difference
 - Runtime type vs compile-time type
- Polymorphism:
 - Reference variable may reference differently typed object
 - Must be a sub-type of

Virtual Dispatch

- Methods in Java permit a late binding
- Reference variable and its type does not tell which method is really invoked
- The type of reference variable and class instance may differ
- Class variables may override methods of super classes
- The method invoked is determined by the type of the class instance
- Binding is of great importance to understand OO

Example: Virtual Dispatch

- Example:

```
public class Shape{ // extends Object
    double area() { }
}

public class Rectangle extends Shape {
    double area() { }
}
...
Shape X = new Shape();
Shape Y = new Rectangle();

double a1 = X.area() // invokes area of Shape
double a2 = Y.area() // invokes area of Rectangle
```

Abstract Classes

- Method implementations are deferred to sub-classes
- Requires own key-word abstract for class/method
- No instance of an abstract class can be generated

```
abstract class Shape extends Object {  
    public abstract double area();  
}  
  
public class Rectangle extends Shape {  
    double width; double length;  
    double area() { return width * length; }  
}
```

Interfaces

- Java has no multi-inheritance
 - Interface is a way-out (introduction of multi-inheritance via the back-door)
- Interfaces is a class contract that ensures that a class implements a set of methods.
- Interfaces can inherit from other interfaces
- Ensures that a class has a certain set of behavior
- Interfaces are specified so that they form a directed acyclic graph
- Methods declared in an interface are always public and abstract
- Variables are permitted if they are static and final only

Example: Interface

```
// definition of interface
public interface A {
    int foo(int x);
}

// class X implements interface A
class X implements A {
    int foo(int x) {
        return x;
    }
}
```

Example: Interface

- Inheritance in interfaces

```
// definition of interface
public interface A {
    int foo(int x);
}

public interface B extends A{
    int hoo(int x);
}
```

- Interface B has methods foo() and hoo()

Task for Week 2

- Submit weekly exercise on canvas before 23.59pm Saturday
- Preview UML related concept from references:
 - Use Case
 - Chap 6 of book “Applying UML and Patterns, 3rd Edition”
 - Chap 9 of book “UML Distilled, 3rd Edition”
 - Class Diagram
 - Chap 16 of book “Applying UML and Patterns, 3rd Edition”
 - Chap 3 and 5 of book UML Distilled

What are we going to learn next week?

- UML Modeling and Case Studies
 - Use Case Diagrams
 - Class Diagrams
 - Interaction Diagrams

Software Design and Construction 1

SOFT2201 / COMP9201

UML Modeling

Dr. Xi Wu

School of Computer Science



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Announcement

- Some tutorials of this week (week 3) may have alternative arrangement
 - Please double check the Ed post/announcement for further action

Agenda

- UML Modeling
 - UML Use Case Diagrams
 - UML Class Diagrams
 - UML Interaction Diagrams
- Case Study: Next Gen Point-of-Sale (POS) System
 - Brief Introduction
 - Self learning details

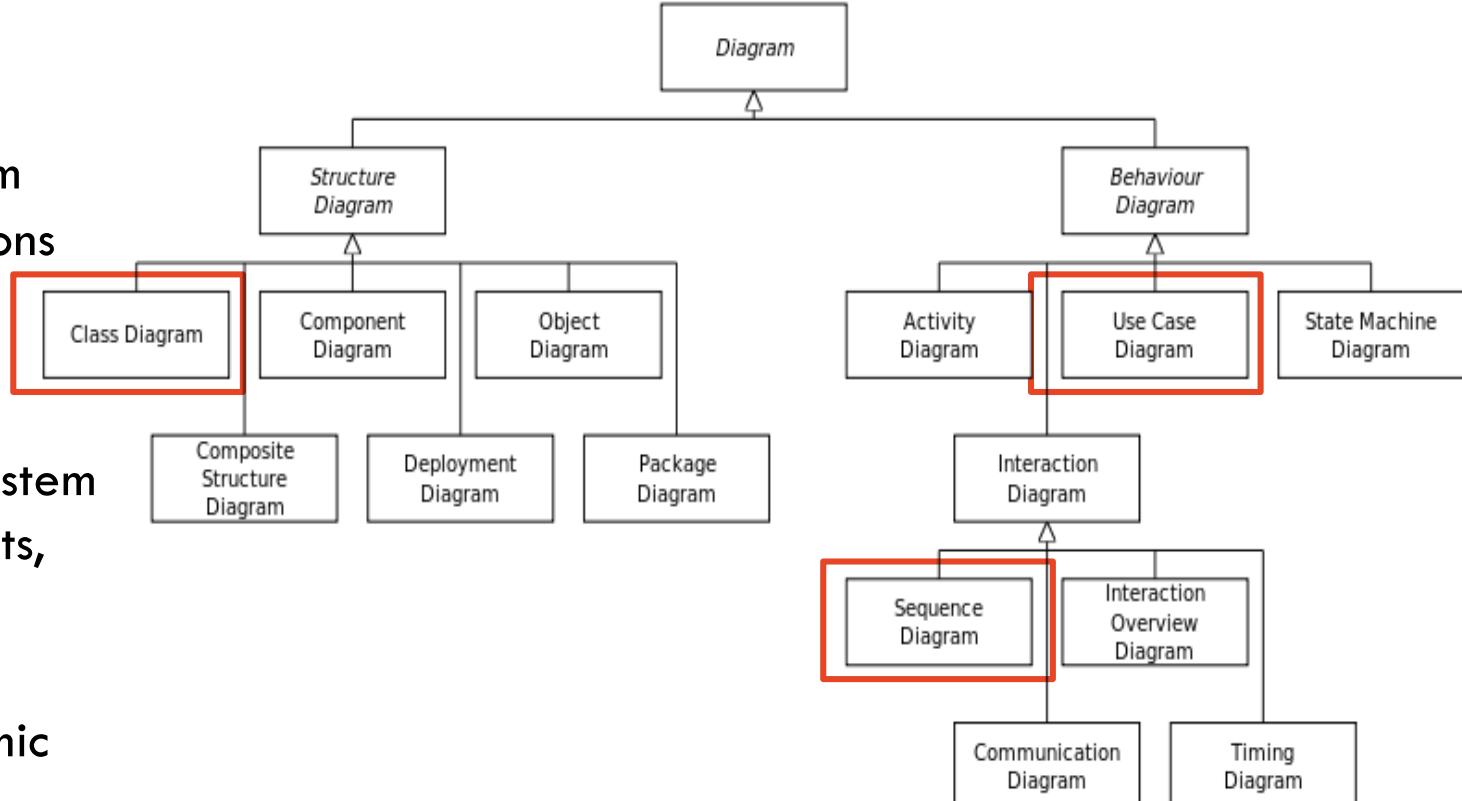
Software Modelling

- The process of developing conceptual models of a system at different levels of **abstraction**
 - Fulfil the defined system requirements
 - Focus on important system details to deal with complexity
- Using graphical notations (e.g., UML) to capture different views or perspectives
 - There are three ways to apply UML
 - Blueprint: detailed design diagrams for:
 - Reverse engineering to visualize and understand existing code
 - Forward engineering (code generation)

UML Diagrams

Structural (static) View

- Static structure of the system (objects, attributes, operations and relationships)



Behavioural (dynamic) View

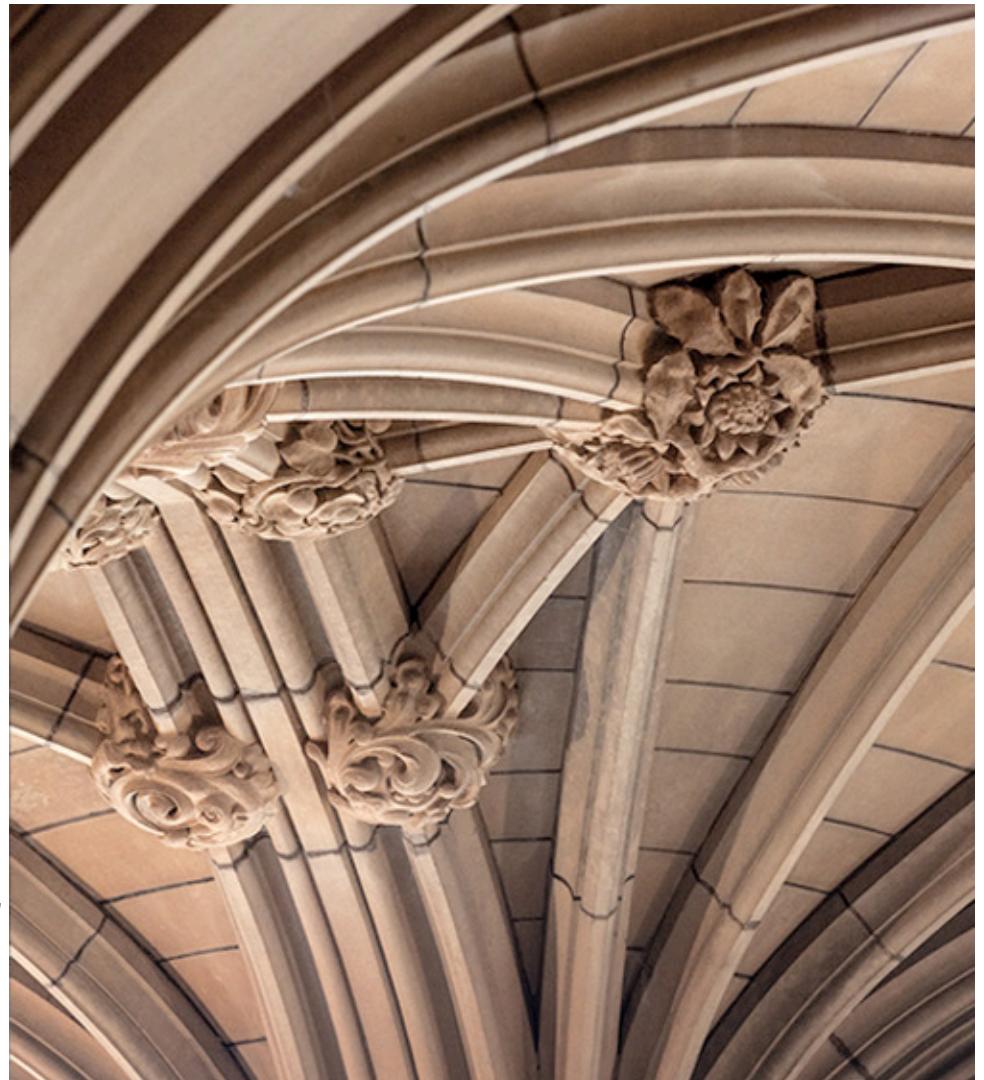
- Dynamic behavior of the system (collaboration among objects, and changes to the internal states of objects)
- Interaction (subset of dynamic view) - emphasizes flow of control and data

https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/UML

Use Case Diagrams

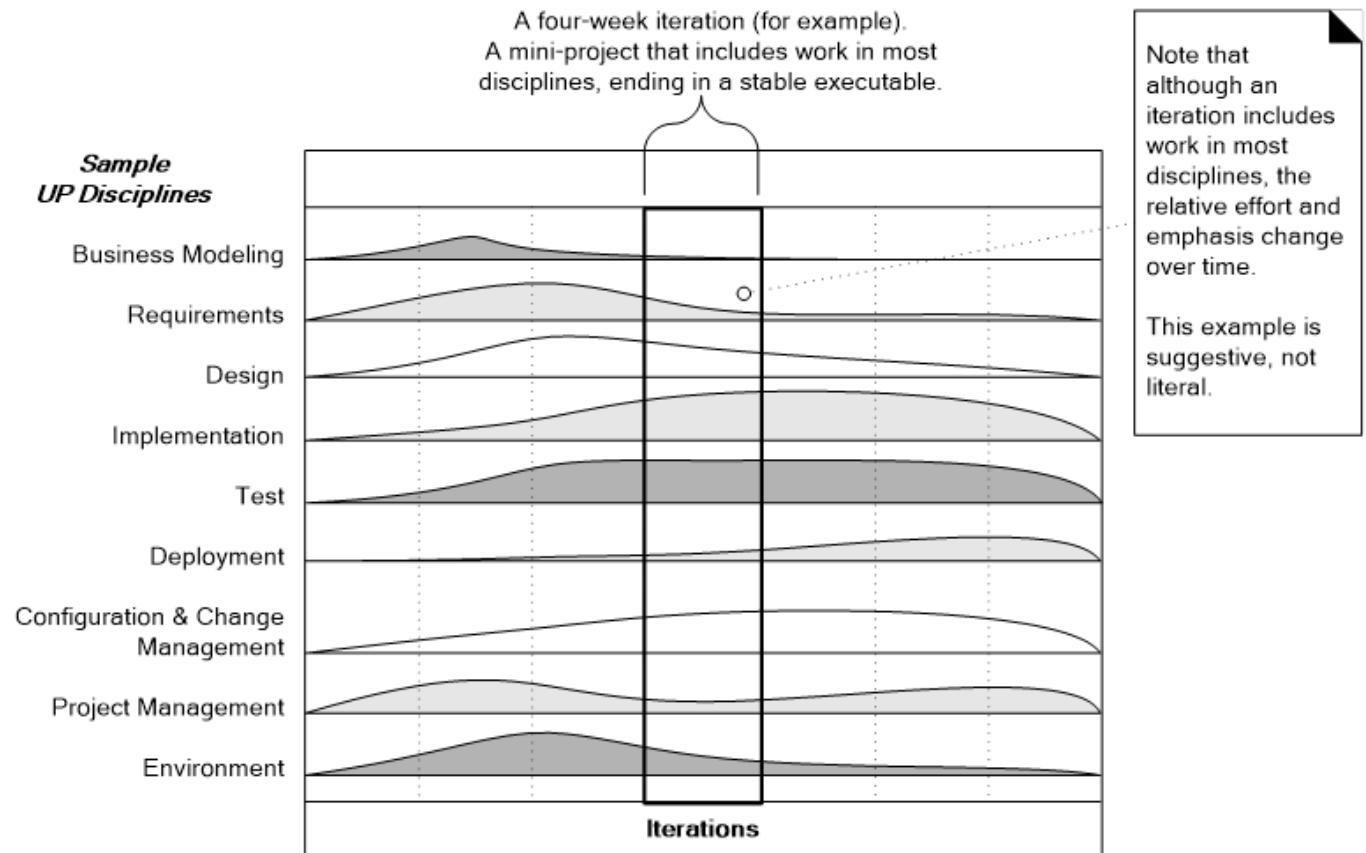
- 1. Why we need Use Case Diagrams?**
- 2. What is Use Case?**
- 3. How can we draw Use Case Diagrams?**

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition).



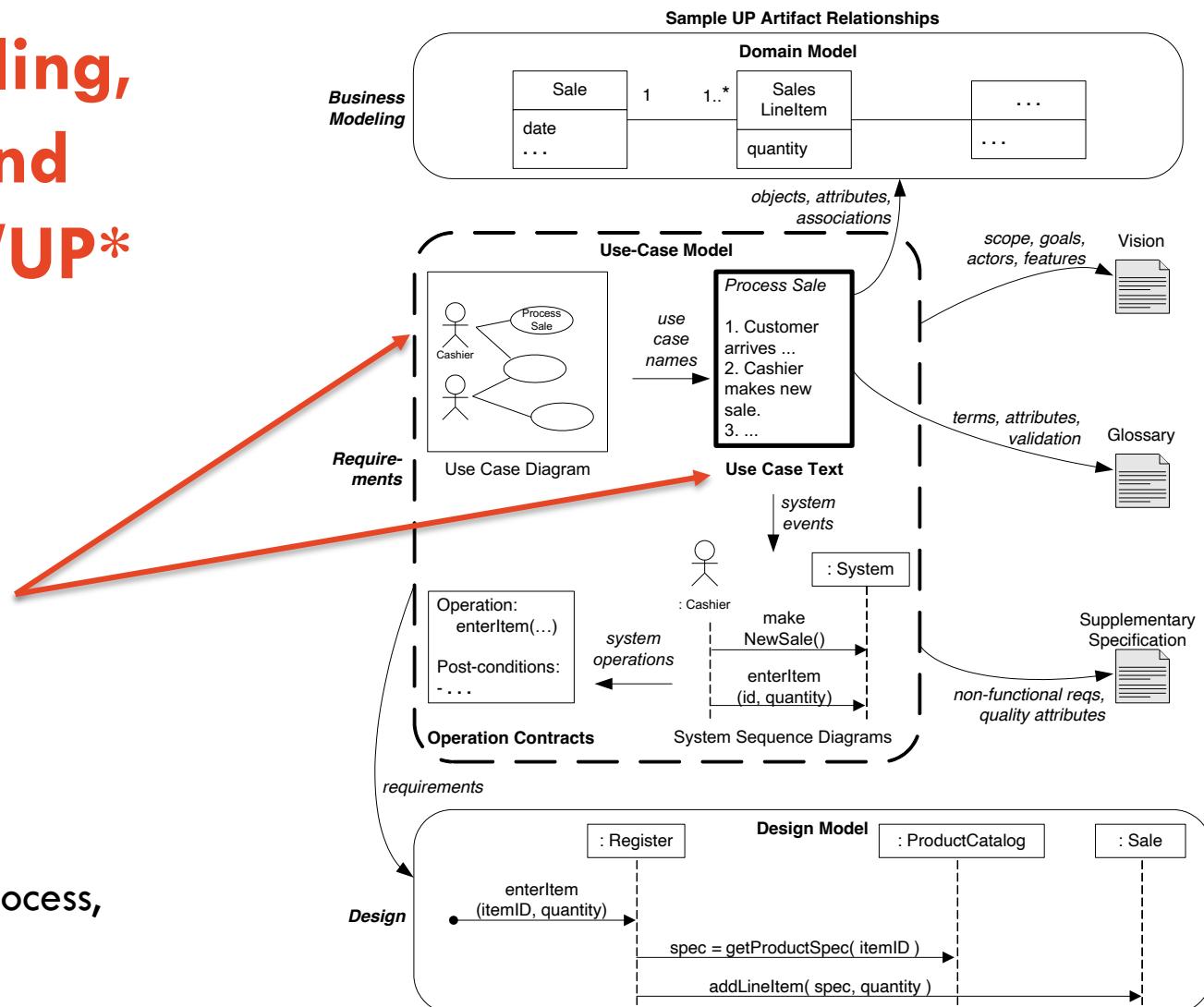
Rational Unified Process (UP)

- Software development process utilizing iterative and risk-driven approach to develop OO software systems
- Iterative incremental development
- Iterative evolutionary development



Business Modelling, Requirements and Designs in RUP/UP*

Use Cases and Use Case Diagrams



*RUP/UP: Rational Unified Process,
Lecture 1 pp. 59--64

Use Cases

- **Use case:** “specifies a set of behaviors performed by a system, which yields an observable result that is of value for Actors or other stakeholders of the system”*
 - It capture what a system supposed to do (system’s requirements)
 - Text documents not diagrams
 - Primary part of the use case model
- **Scenario (use case instance):** specific sequence of action and interactions between actors and the system
 - One particular story of using a system (e.g., successfully purchasing items with cash)
- **Use case model:** set of all written use cases (a model of the system functionality and environment)
 - Defined within the Requirements discipline in the UP
 - Not the only requirement artefact – business rules, supplementary specifications, etc

* OMG Unified Modeling Language, version 2.5.1, Dec. 2017 <https://www.omg.org/spec/UML/2.5.1>

Use Cases – Common Formats

- **Brief:** one-paragraph summary, usually of the main success scenario
 - During early requirements analysis to get quick subject and scope
- **Casual:** Informal paragraph format; multiple paragraphs that cover various scenarios
 - During early requirements analysis to get quick subject and scope
- **Full-dressed:** all steps and variations are written in detail and there are supporting sections; e.g., pre-conditions, success guarantee
 - After many use cases have been identified and written in brief format

* OMG Unified Modeling Language, version 2.5.1, Dec. 2017 <https://www.omg.org/spec/UML/2.5.1>

Use Cases – Full-dressed Template

Use case section	Comment
Use case name	Start with a verb
Scope	The system under design
Level	“user-goal” or sub-function
Primary actor	Calls on the system to deliver its services
Stakeholders and interests	Who cares about this UC and what do they earn?
Preconditions	What must be true on start
Success guarantee	What must be true on successful completion
Main success scenario	Typical unconditional happy path scenario
Extensions	Alternate scenarios of success or failure
Special requirements	Related non-functional requirement
.....	

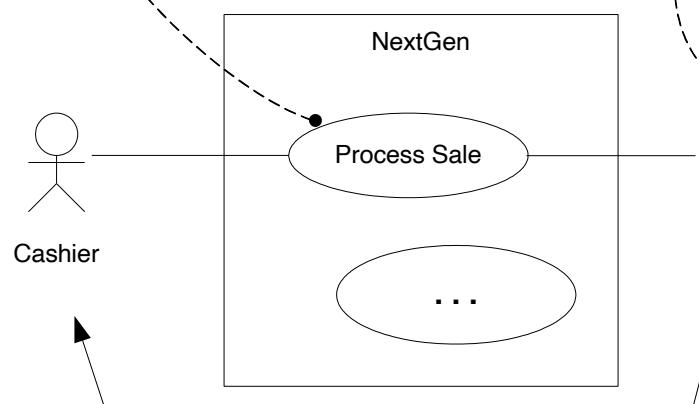
Use Case Diagrams

- UML graphical notations that help to capture use cases (system's boundaries and interactions and relationships)
 - **Subject:** system under consideration to which the use case applies
 - **Actor:** role that interact with the subject/system (e.g., end user, customer, supplier, another system)
 - **Use case:** describes functionality of the system
 - **Association:** relationship between an actor and a use case (an actor can use certain functionality of the system)
 - «include» indicates the behavior of the included use case is included in the behavior of the including use case

* OMG Unified Modeling Language, version 2.5.1, Dec. 2017 <https://www.omg.org/spec/UML/2.5.1>

Use Case Diagram – UML Notations

For a use case context diagram, limit the use cases to user-goal level use cases.

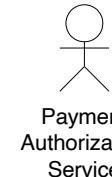
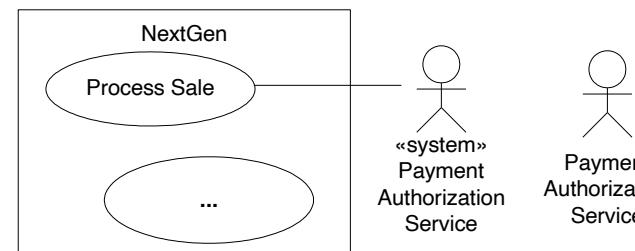


primary actors on
the left

Show computer system actors
with an alternate notation to
human actors.



supporting actors
on the right

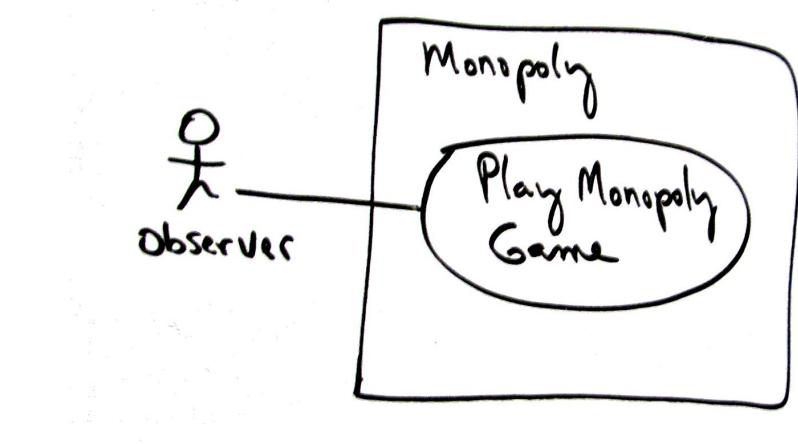
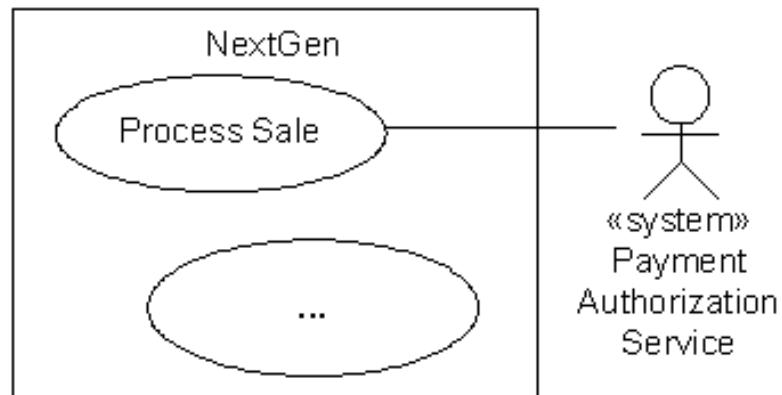


Some UML alternatives to
illustrate external actors that
are other computer systems.

The class box style can be
used for any actor, computer or
human. Using it for computer
actors provides visual
distinction.

Use Case Diagrams – Tools

- There are many tools to aid drawing UML diagrams
 - Tools are means to make your life easier
 - You can also sketch diagrams using pen-and-paper or white board

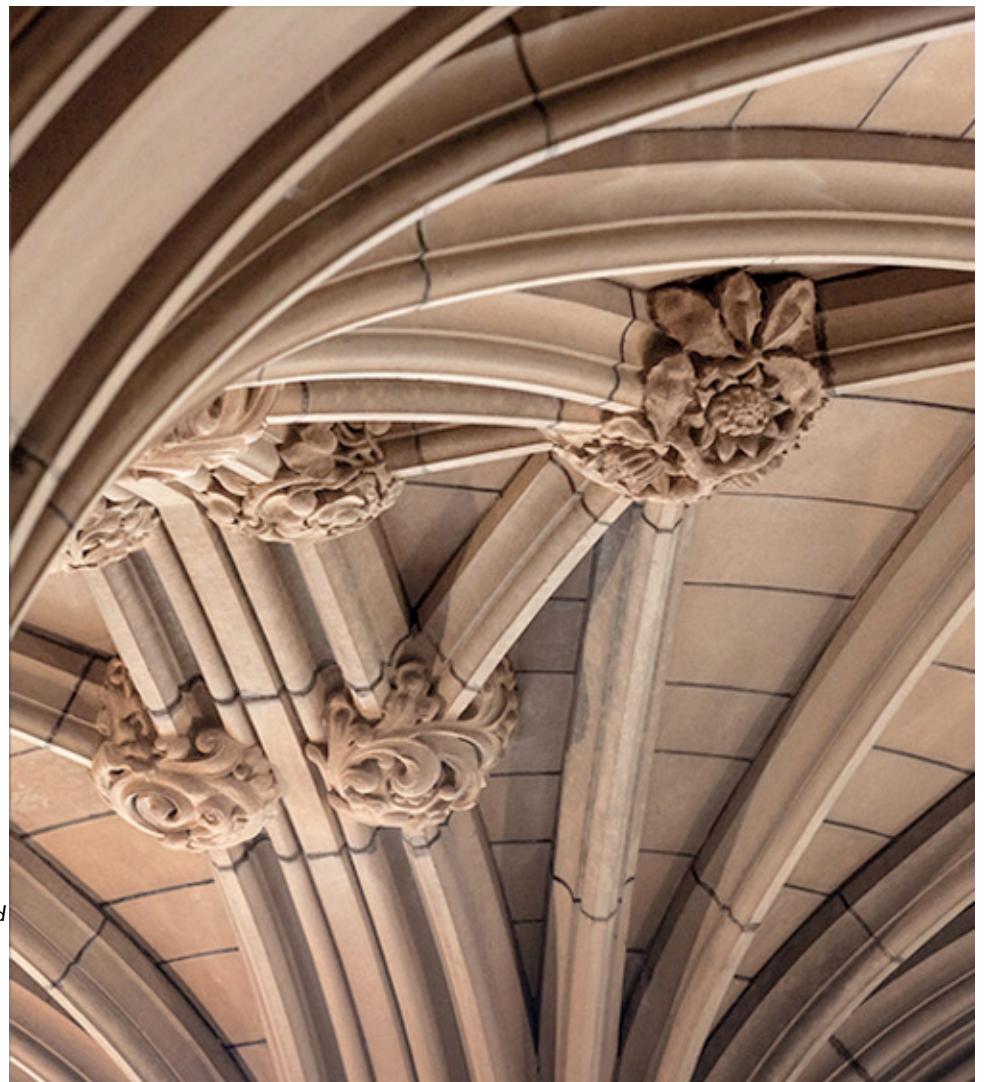


https://www.lucidchart.com/pages/examples/uml_diagram_tool

Class Diagrams

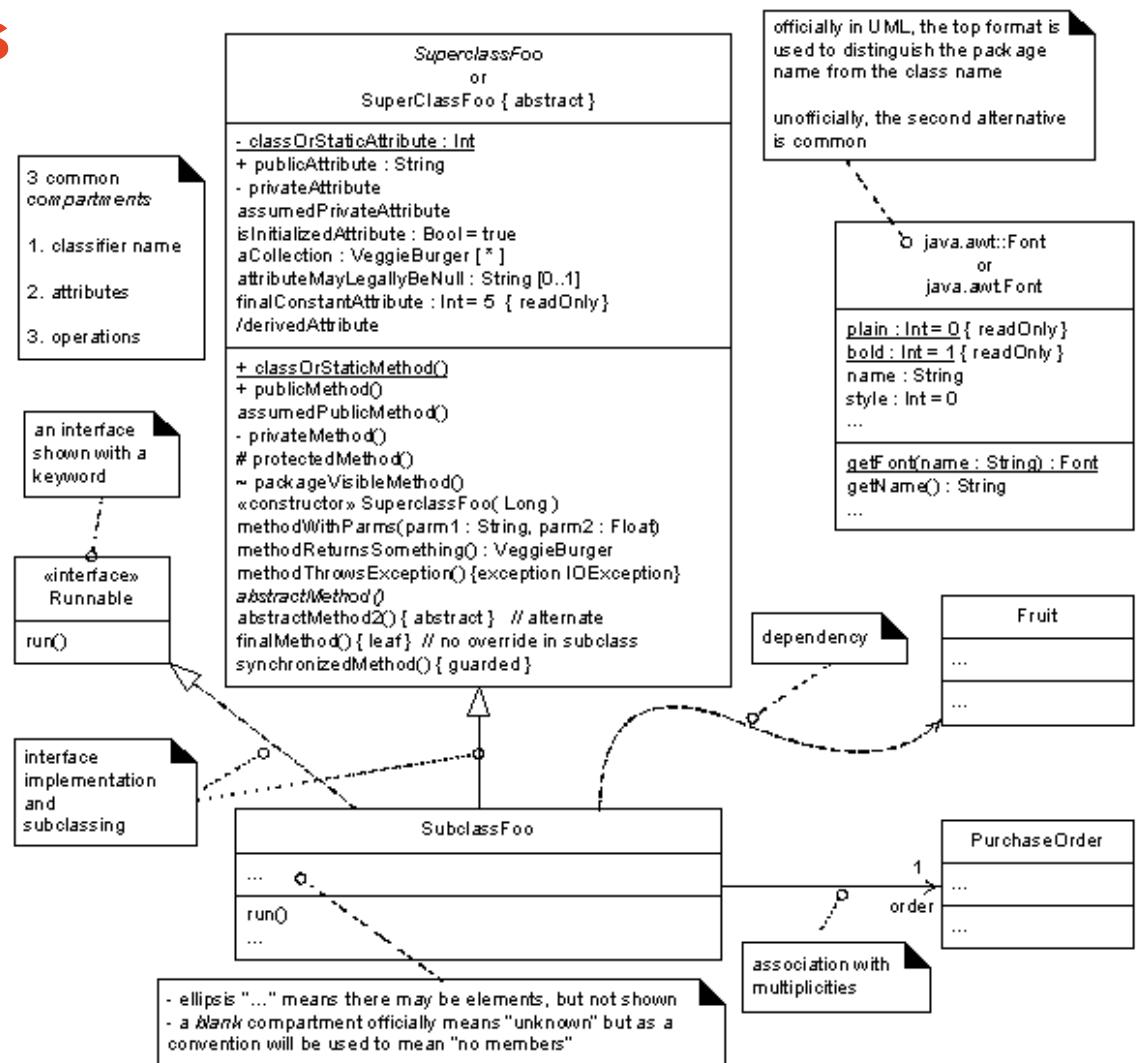
Structural Diagrams

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition).



Class Diagram – Notations

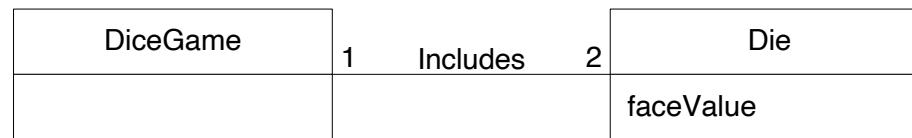
- Common compartments: classifier name, attributes and operations
 - Package name
 - <<interface>>
- Class hierarchy – inheritance
- Dependency
- Association and multiplicity
- Optional and default elements



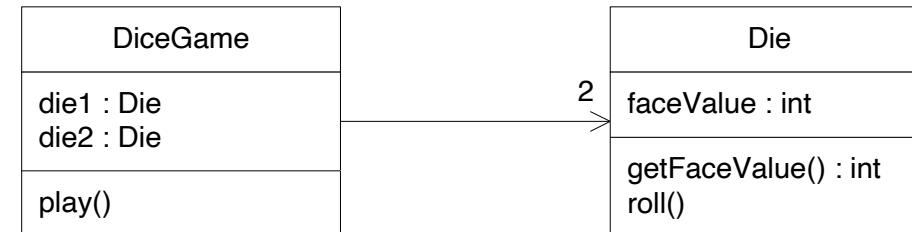
Class Diagram – Perspectives



- **Conceptual:** describes key concepts in the problem domain. Use in business modeling for OO analysis
- **Specification:** describes software components with specification and interfaces
- **Implementation:** describes software implementation in a particular programming language (e.g., Java)



Conceptual Perspective
(domain model)



Specification or
Implementation
Perspective
(design class diagram)

Class Diagrams – UML Attributes

Attribute Text

Visibility : type {property string}

Visibility + (public), - (private)

Attributes are assumed private if no visibility sign shown

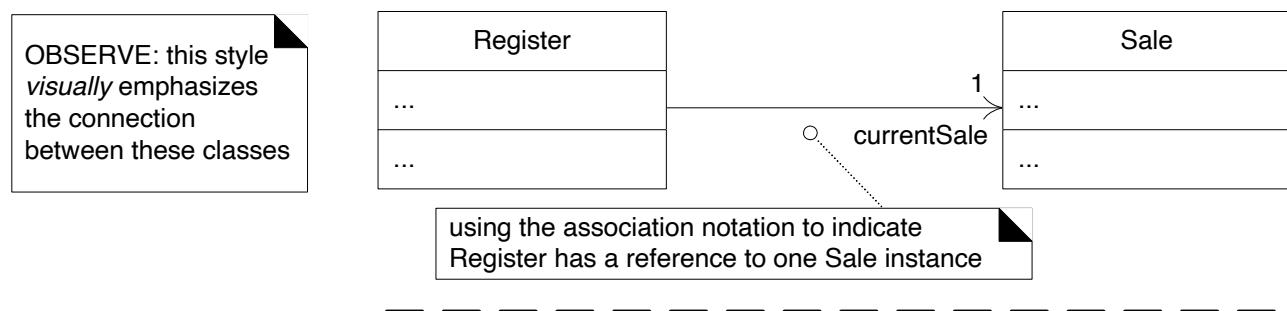


Attribute-as-association

Arrow pointing from the source to the target

Multiplicity and 'rolename' (*currentSale*) at the target

No association name

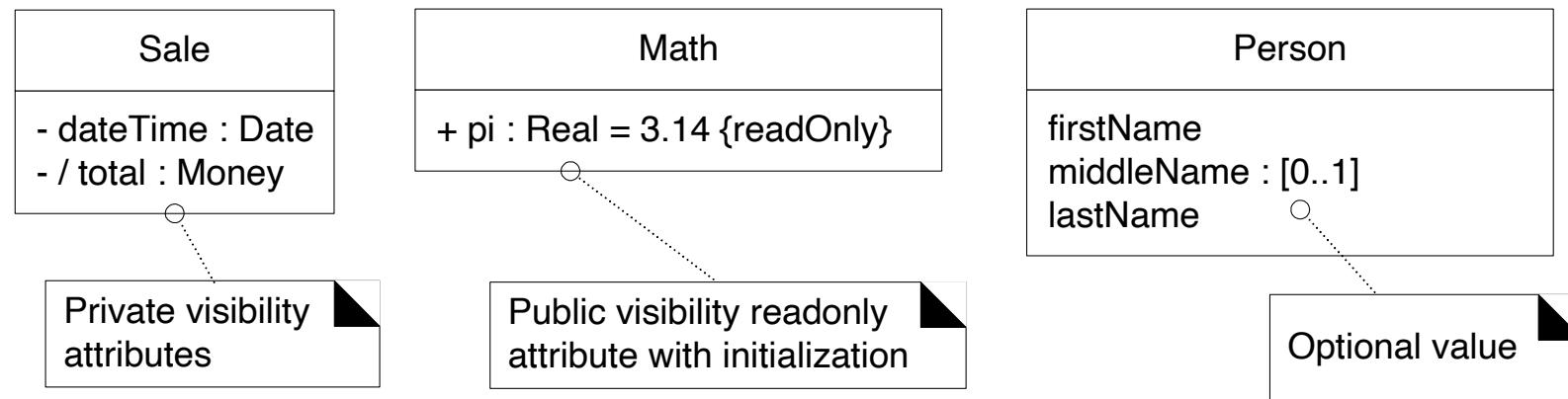


Attribute text and as-association

- Not popular



Class Diagrams – Attribute Examples



Read-only Attributes with initialization, and optional values

Class Diagrams – Operations

Visibility Name (parameter-list) : return-type {property-string} (UML 1)

Visibility Name (parameter-list) {property-string} (UML 2)

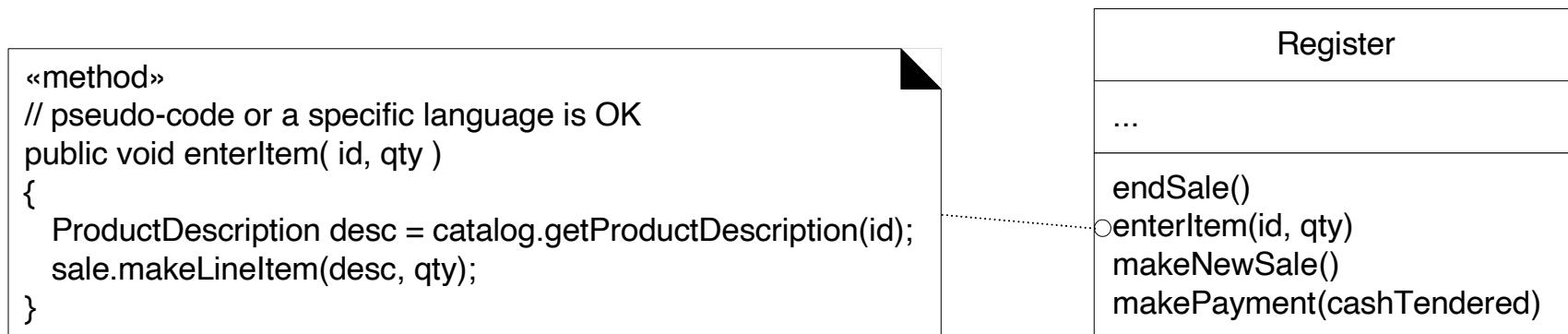
- Not a method, but declaration with a name, parameters, return type, exception list, and possibly a set of constraints of pre-and post-conditions
- Operations are public by default, if no visibility shown
- Operation signature in a programming language is allowed, e.g.,

+ getPlayer (name : String) : Player {exception IOException}

Public Player getPlayer(String name) throws IOException

Class Diagrams – Methods

- Implementation of an operation, can be specified in:
 - Class diagrams using UML note symbol with stereotype symbol «method»
 - Mixing static view (class diagram) and dynamic view (method implementation)
 - Good for code generation (forward engineering)
 - Interaction diagrams by the details and sequence of messages



UML Keywords

- Textual adornment to categorize a model element
 - Using «» or { }
 - UML 2 the brackets («») are used for keywords and stereotype

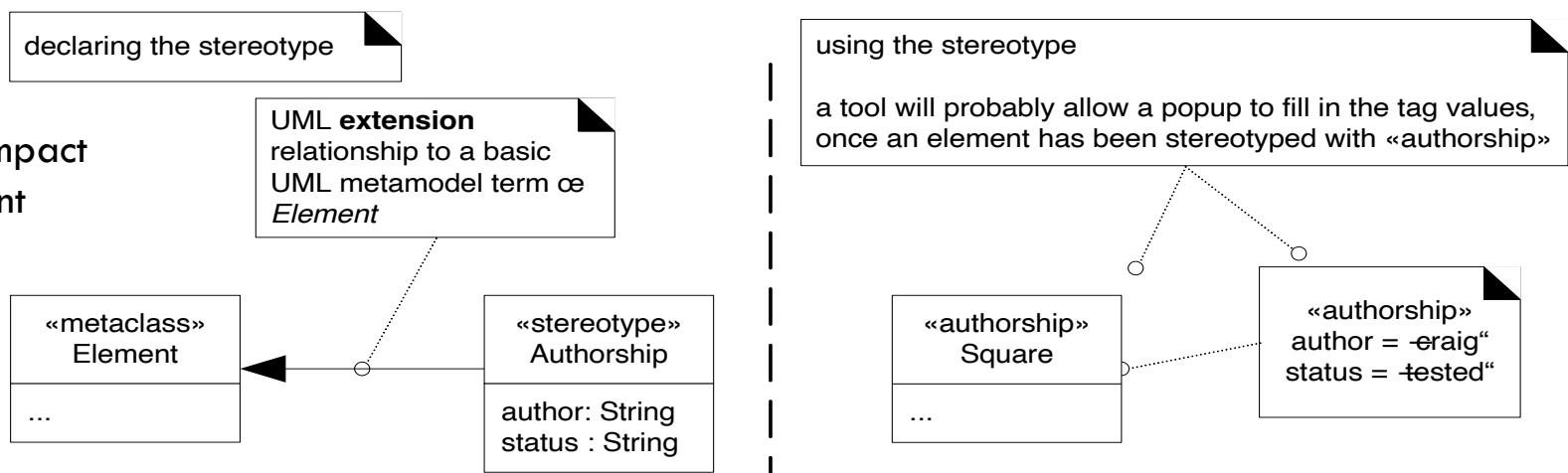
Keyword	Meaning	Example usage
«interface»	Classifier is an interface	In class diagram, above classifier name
{abstract}	Abstract element; can't be instantiated	In class diagrams, after classifier name or operation name
{ordered}	A set of objects have some imposed ordered	In class diagrams, at an association end

UML Stereotypes

- **Stereotypes** allow refinement (extension) of an existing modeling concept
 - Defined in UML Profile
- **UML profile:** group of related model elements allow customizing UML models for a specific domain or platform
 - Extends UML's «metaclass» *Element*

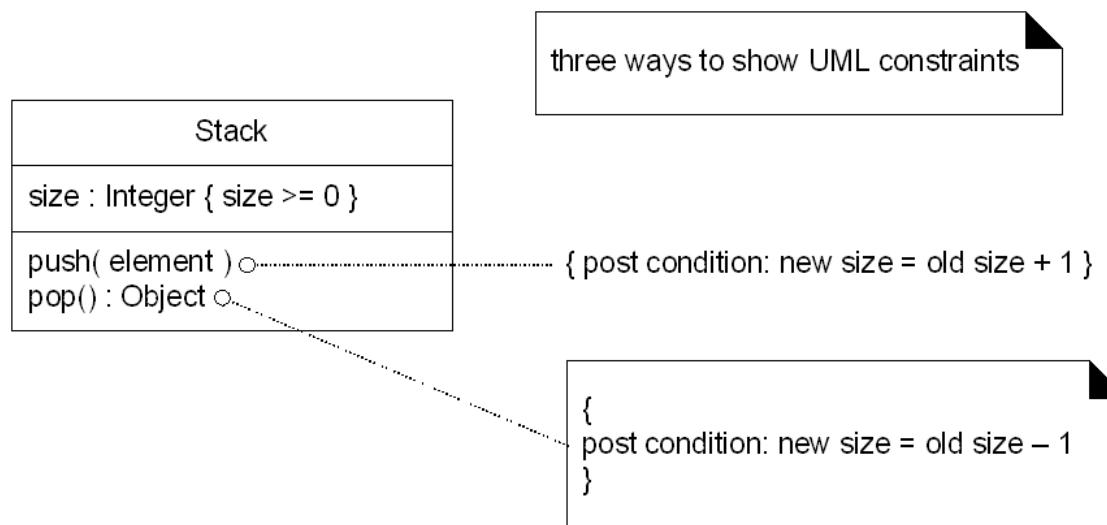
- **UML note symbol**

- Has no semantic impact
- Specify a constraint



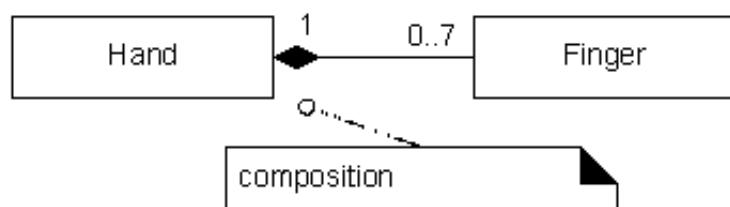
Constraints

- Restriction/condition on a UML elements described in a natural or a formal language (Object Constraint Language (OCL))
- Different ways to represent constraints



Composition

- Composition, or composite aggregation, relationship implies:
 - Instance of the part (e.g., Square) belongs to only one composite instance at a time (e.g., one board)
 - The part must always belong to a composite
 - The composite is responsible for the creation and deletion of its parts (by itself or by collaborating with other objects)



composition means
-a part instance (Square) can only be part of one composite (Board) at a time
-the composite has sole responsibility for management of its parts, especially creation and deletion



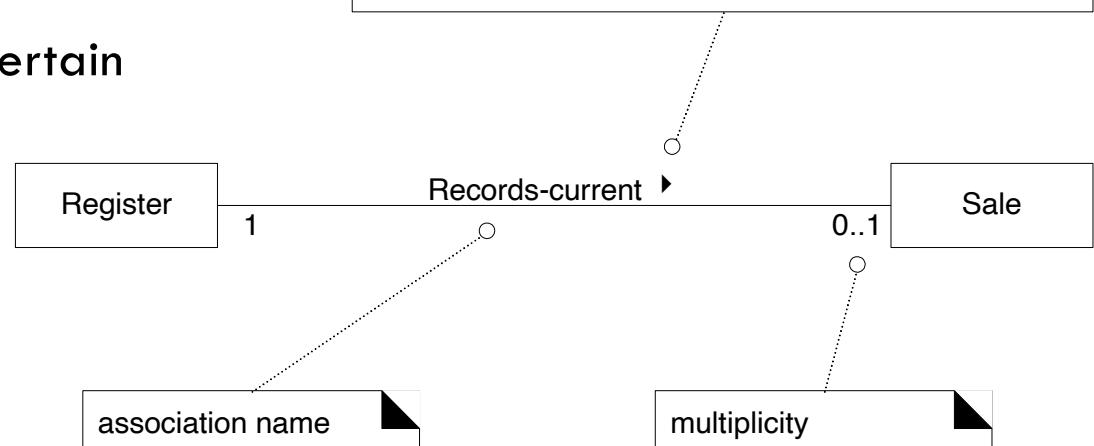
Associations

- Relationship between classifiers where logical or physical link exists among classifier's instances
- May implemented differently; no certain construct linked with association

- Notations:

- Association name (meaningful)
- Multiplicity
- Direction arrow

-"reading direction arrow"
-it has *no* meaning except to indicate direction of reading the association label
-often excluded



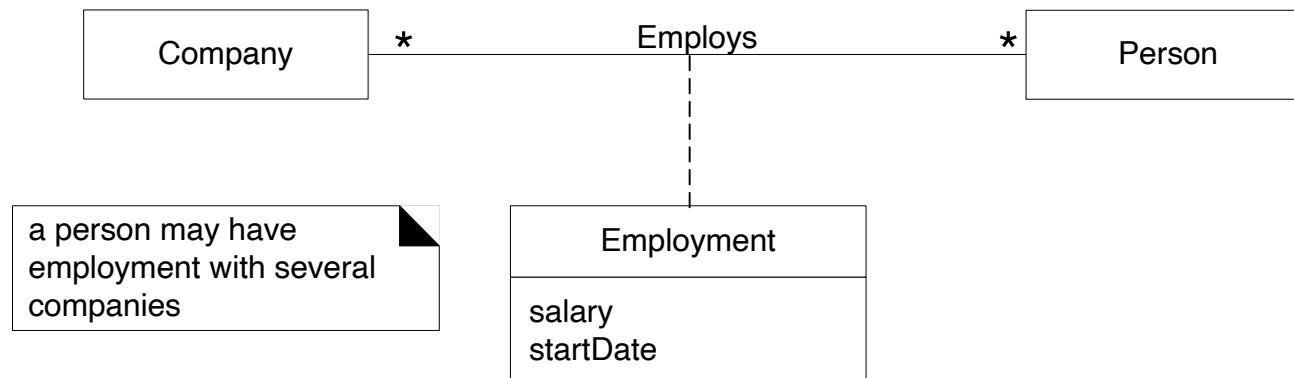
Associations – Multiplicity

- Multiplicity: number of instances involved in the relationship (association)
- Communicates domain constraints that will be implemented
- Multiplicity focus on the relationship at a particular moment, rather than over a span of time
 - “In countries with monogamy laws, a person can be *Married-to* only one other person at any particular moment, even though over a span of time, that same person may be married to many persons.”

Multiplicity	Meaning (number of participating instances)
*	Zero or more; many
0..1	Zero or one
1..*	One or more
1..n	One to n
n	Exactly n
n, m, k	Exactly n, m or k

Association Class

- Modeling an association as a class (with attributes, operations & other features)
 - A Company *Employs* many Persons
 - *Employs* → *Employment* class with attributes salary and startDate



Dependency

- A dependency exists between two elements if changes to the definition of one element (the supplier) may cause changes to the other (the client)
- Various reason for dependency
 - Class send message to another
 - One class has another as its data
 - One class mention another as a parameter to an operation
 - One class is a superclass or interface of another

When to show dependency?

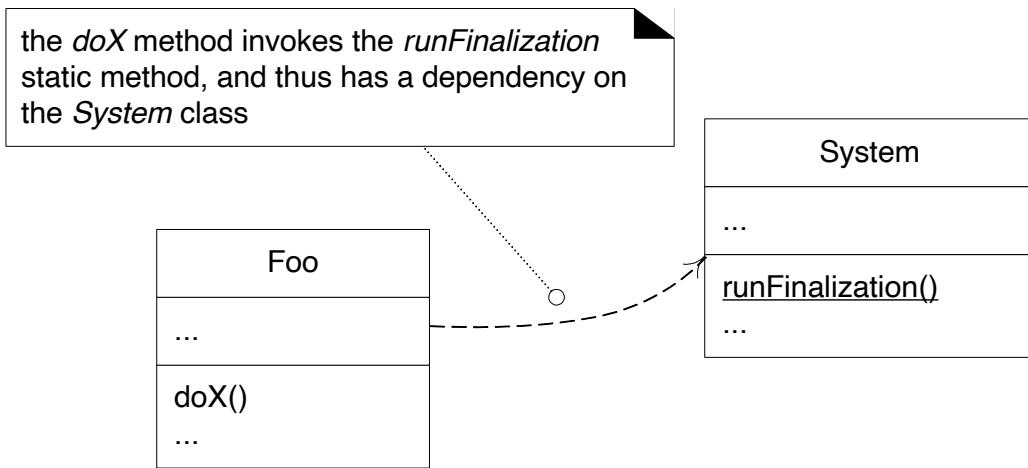
- Be selective in describing dependency
- Many dependencies are already shown in other format
- To depict global, parameter variable, local variable and static-method
- To show how changes in one element might alter other elements
- There are many varieties of dependency, use keywords to differentiate them
- Different tools have different sets of supported dependency keywords:

<<call>> the source calls an operation in the target

<<use>> the source requires the targets for its implementation

<<parameter>> the target is passed to the source as parameter.

Dependency Example

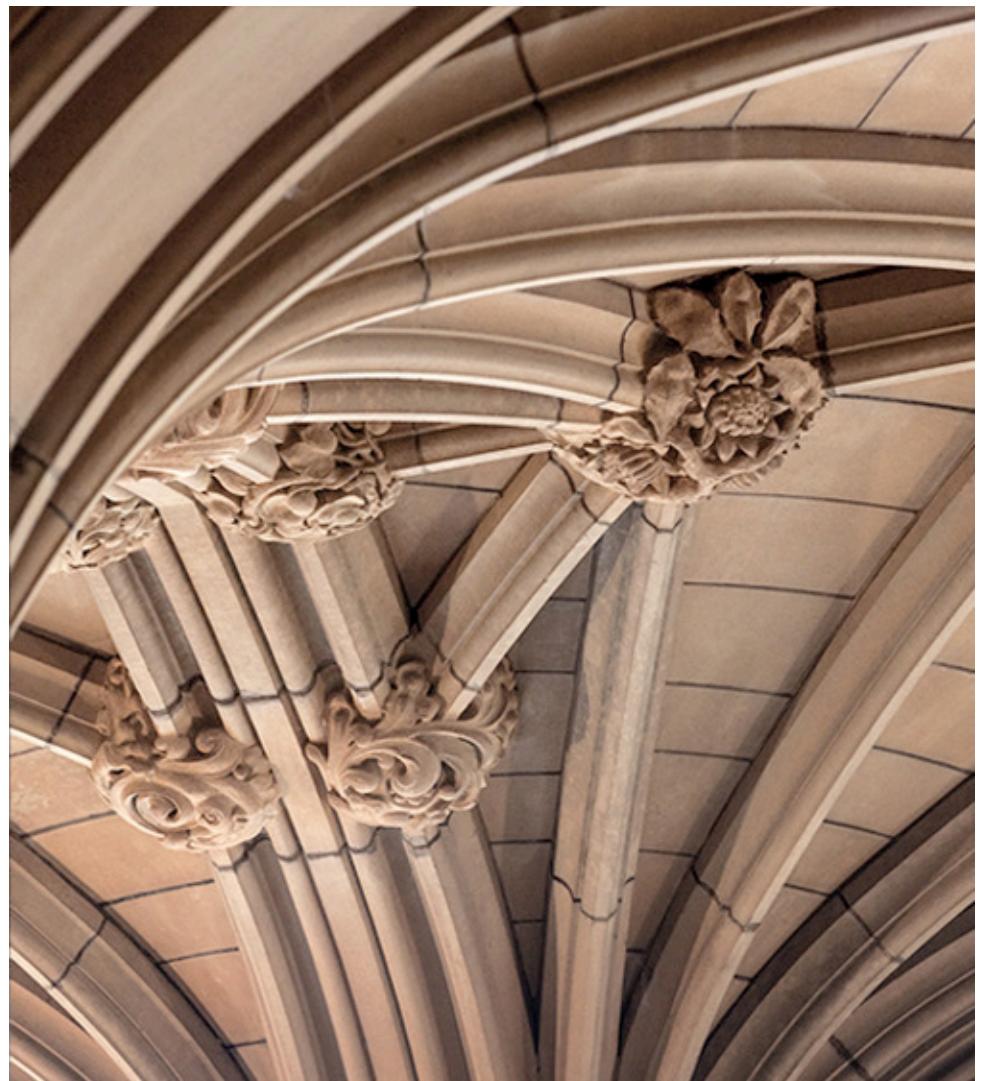


```
1 public class Foo{  
2     public void doX(){  
3         System.runFinalization();  
4         //...  
5     }  
6 }  
7 }
```

UML Interaction Diagrams

Dynamic (Behavioural) Diagrams

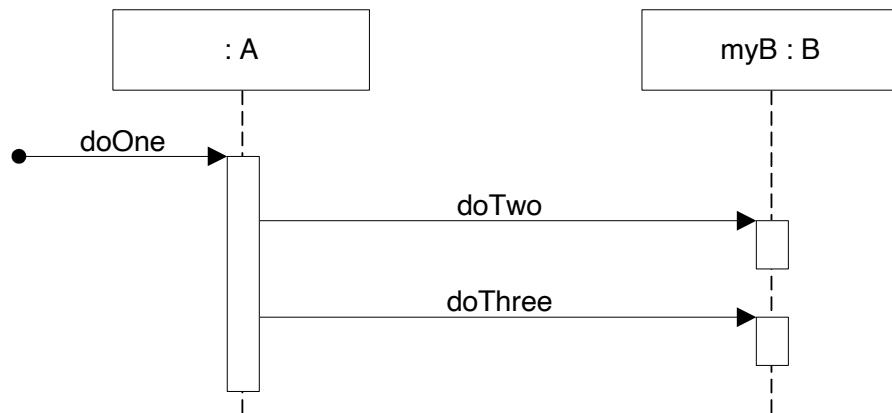
Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition).



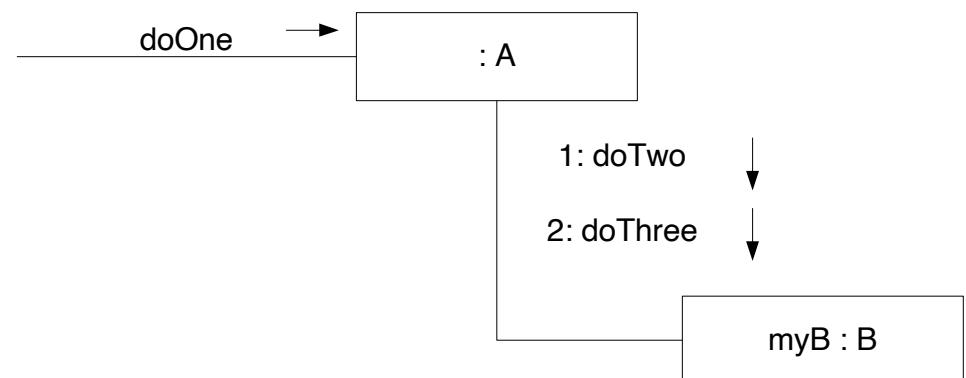
UML Interaction Diagrams

- One of the dynamic (behavioral) diagrams which consists of diagrams including Sequence and Communication diagram

Sequence diagrams: illustrate sequence/time-ordering of messages in a fence format (each object is added to the right)

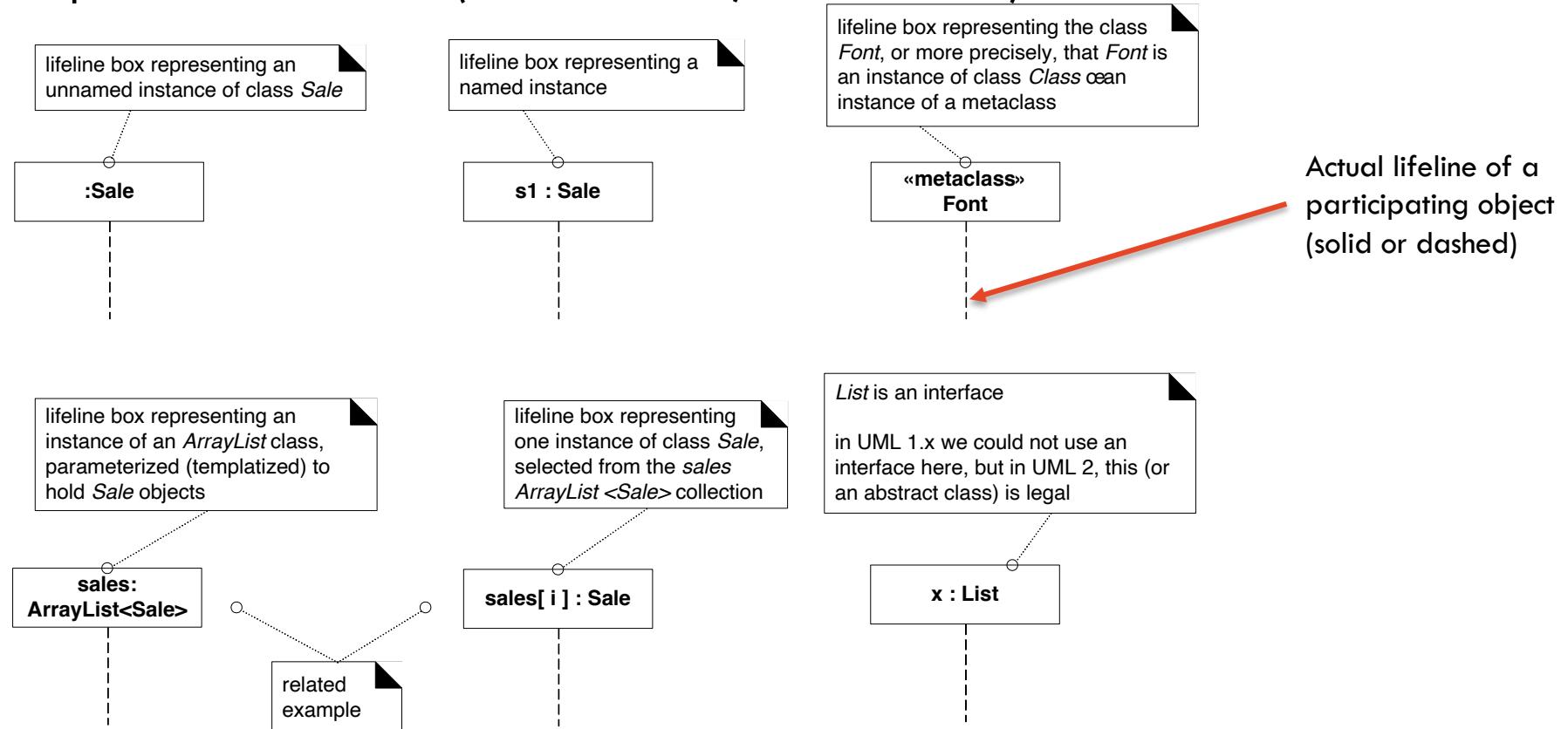


Communication diagrams: objects' interactions are illustrated in a graph/network format



Sequence Diagrams: Classes/Objects

- Participants in interactions (class instances, lifeline boxes)



Sequence Diagrams: Messages

- Standard message syntax in UML

ReturnVar = message (parameter : parameterType) : returnType

- Some details may be excluded.

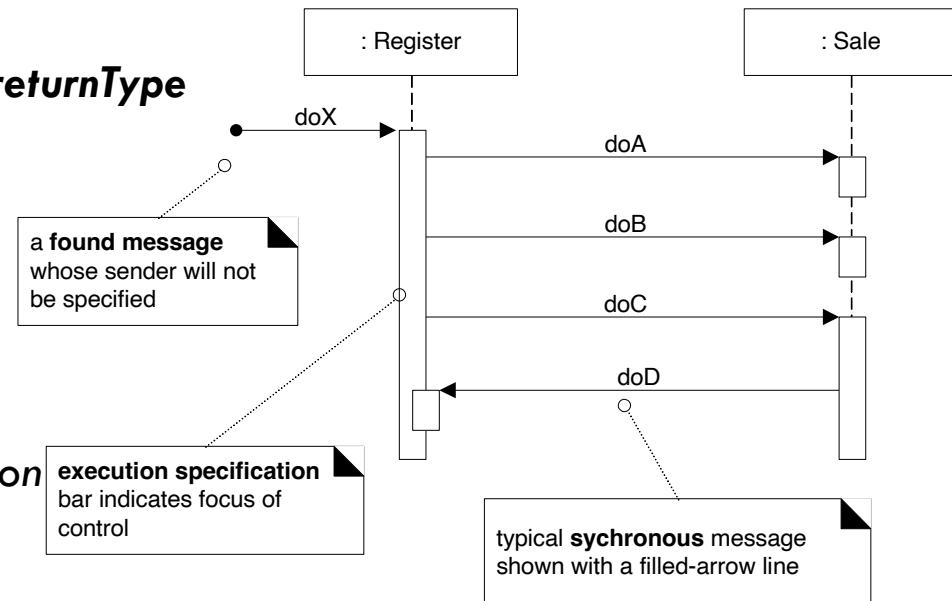
Examples:

Initialize(code)

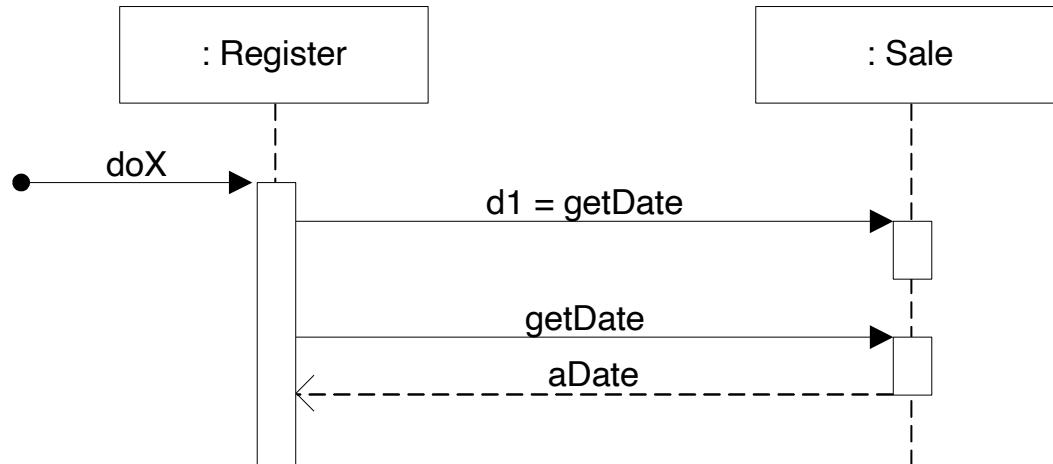
desrp = getProductDescription(id)

desrp = getProductDescription(id) : ProductDescription

- The time ordering from top to bottom of lifelines



Sequence Diagrams: Messages

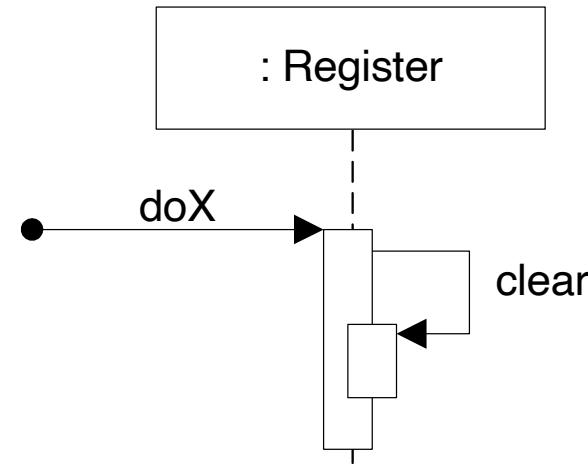


Message reply/return

1. Standard reply/return message syntax

returnVar = message (parameter)

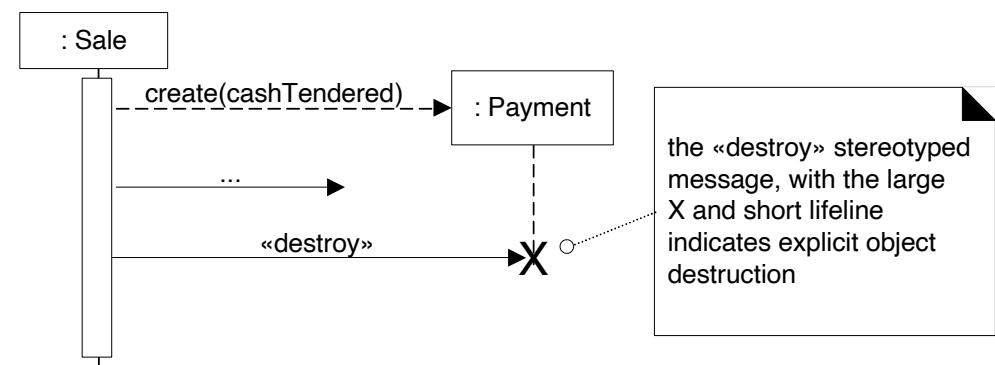
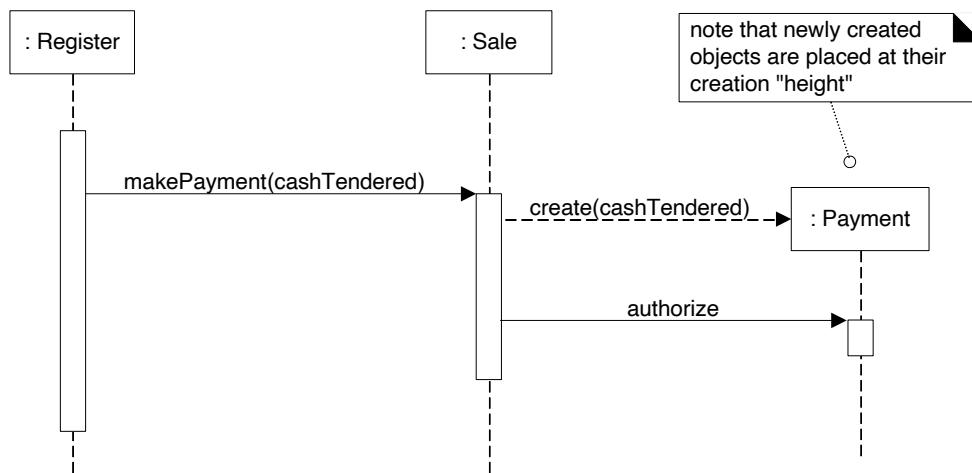
2. Reply/return message line at the end of execution bar



Messages to “Self”

Using nested execution bar

Sequence Diagrams: Objects Creation/Destruction



Object Creation

Read as:

"Invoke the new operator and call the constructor"

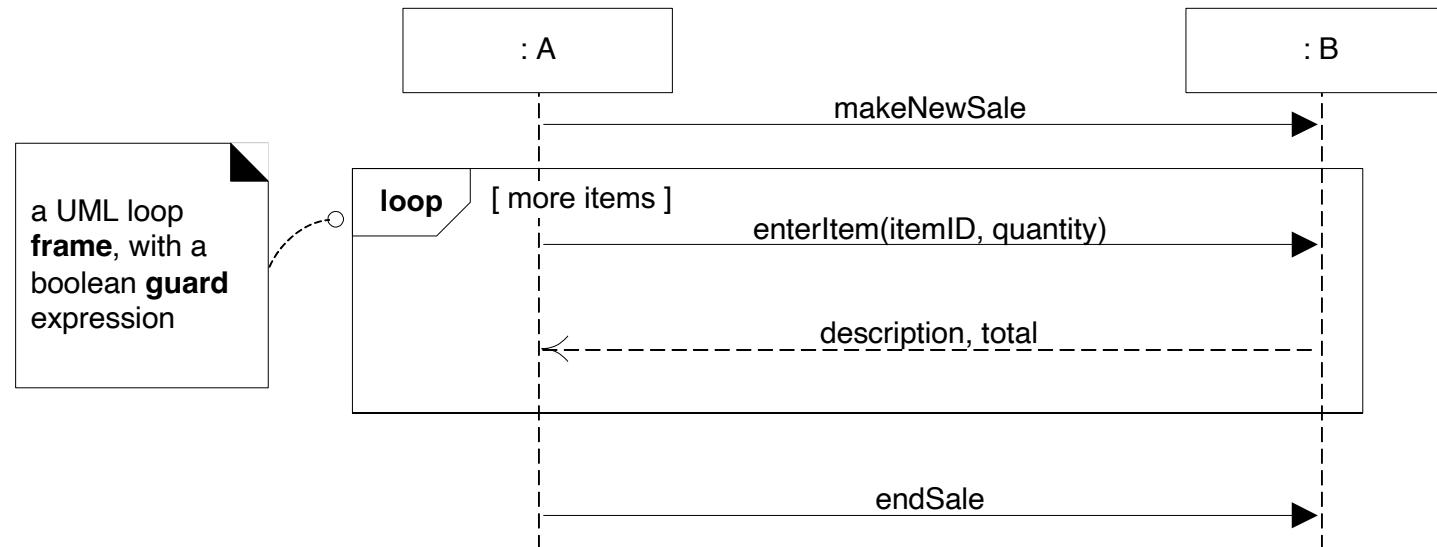
Message name is optional

Object Destruction

Explicit destruction to indicate object is no longer useable (e.g., closed database connection)

Sequence Diagrams: Frames

- Diagram frames in UML sequence diagrams
 - Support conditional and looping construct



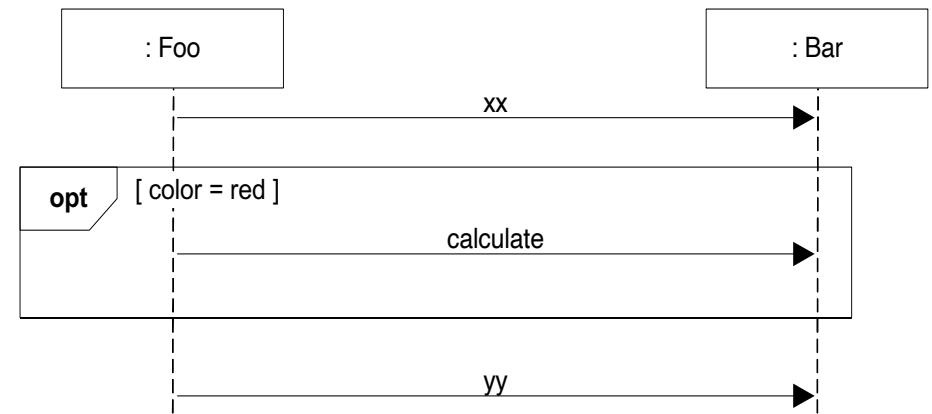
Sequence Diagrams: Frames

- Common frame operators

Frame operator	Meaning
Alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards
Loop	Loop fragment while guard is true
Opt	Optional fragment that executes if guard is true
Par	Parallel fragments that execute in parallel
Region	Critical region within which only one thread can run

Sequence Diagrams: Conditional Messages

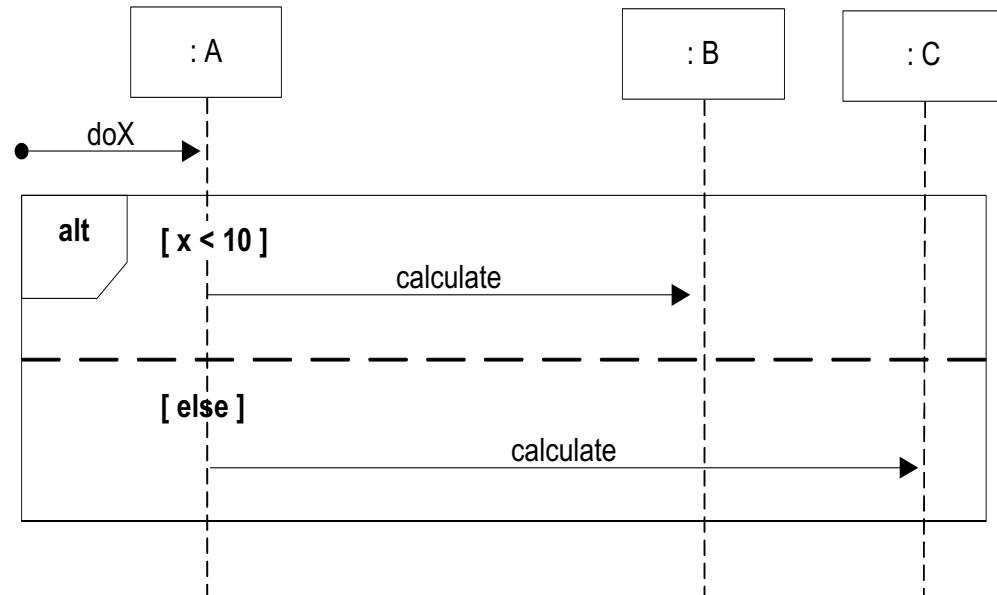
```
public class Foo {  
    Bar bar = new Bar ( );  
    ...  
    public void ml ( ) {  
        bar.xx( );  
        if (color.equals("red"))  
            bar.calculate( );  
        bar.yy( );  
    }  
}
```



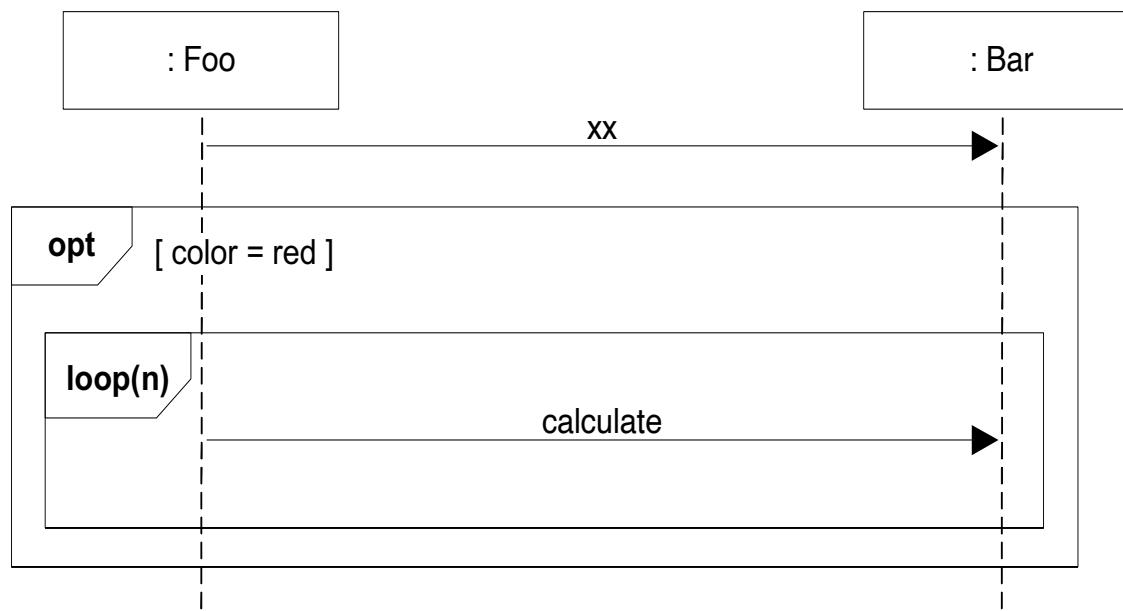
Sequence Diagrams: Conditional Messages

Mutually exclusive conditional messages

```
1 public class A{  
2     B b = new B();  
3     C c = new C();  
4     public void doX(){  
5         ...  
6         if (x < 10)  
7             b.calculate();  
8         else  
9             c.calculate();  
10    }  
11 }  
12 }
```

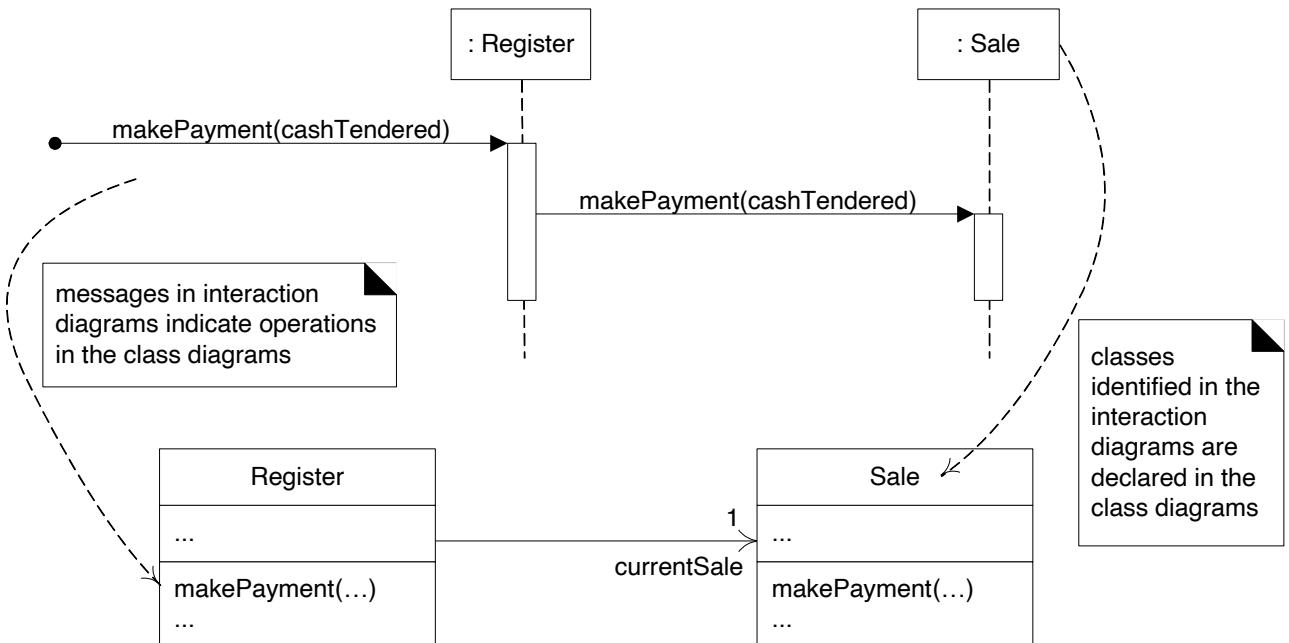


Sequence Diagrams: Nesting of Frames



Class Diagrams: Relationship to Interaction Diagrams

- Interaction diagrams illustrates how objects interact via messages (dynamic behavior)
 - Classes and their methods can be derived
 - E.g., **Register** and **Sale** classes from *makePayment* sequence diagram
- Agile modeling practice:
draw diagrams
concurrently as dynamic
and static views
complement each other
during the design process



Software Modelling Case Study

NextGen POS software modeling

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition).



Next Gen Point-of-Sale (POS) System

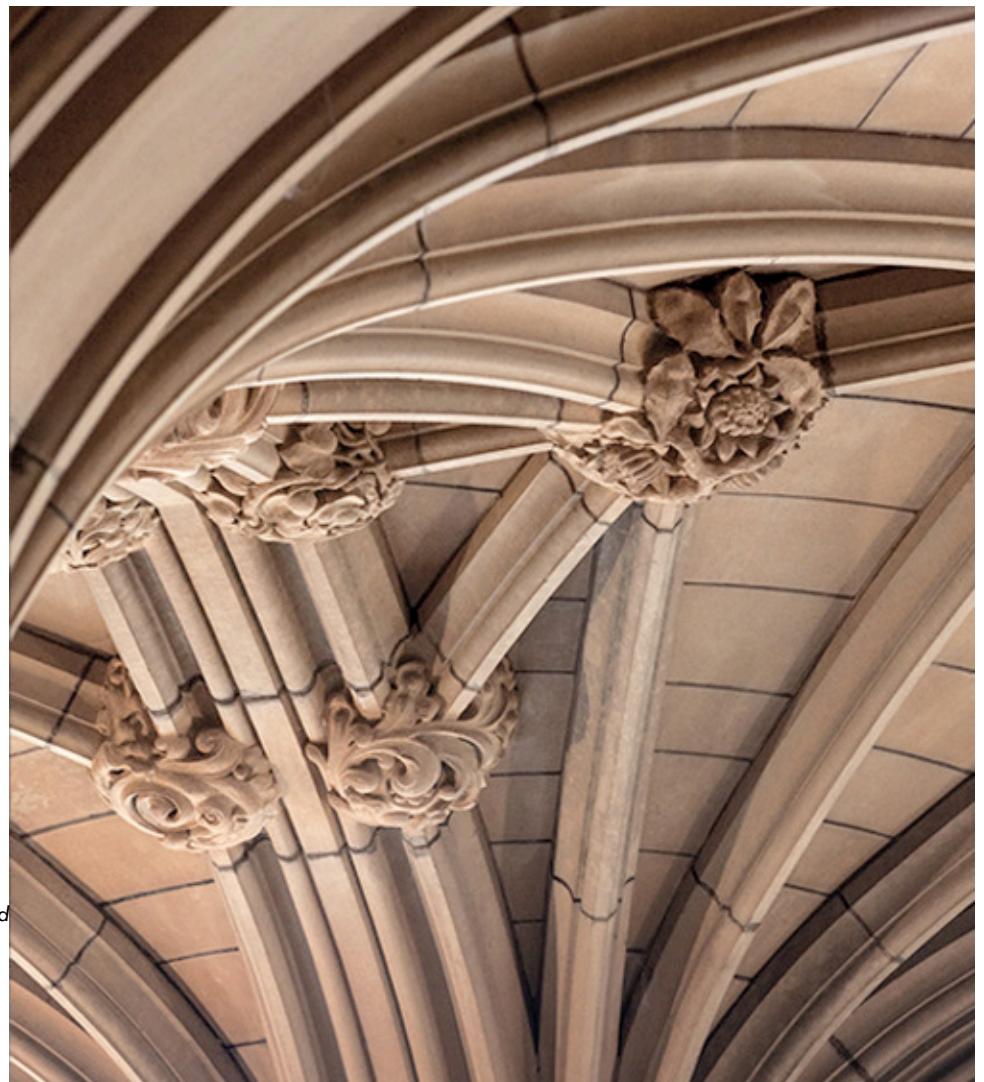
- A POS is a computerized application used (in part) to record sales and handle payments
 - Hardware: computer, bar code scanner
 - Software
 - Interfaces to service applications: tax calculator, inventory control
 - Must be fault-tolerant (can capture sales and handle cash payments even if remote services are temporarily unavailable)
 - Must support multiple client-side terminals and interfaces; web browser terminal, PC with appropriate GUI, touch screen input, and Wireless PDAs
 - Used by small businesses in different scenarios such as initiation of new sales, adding new line item, etc.



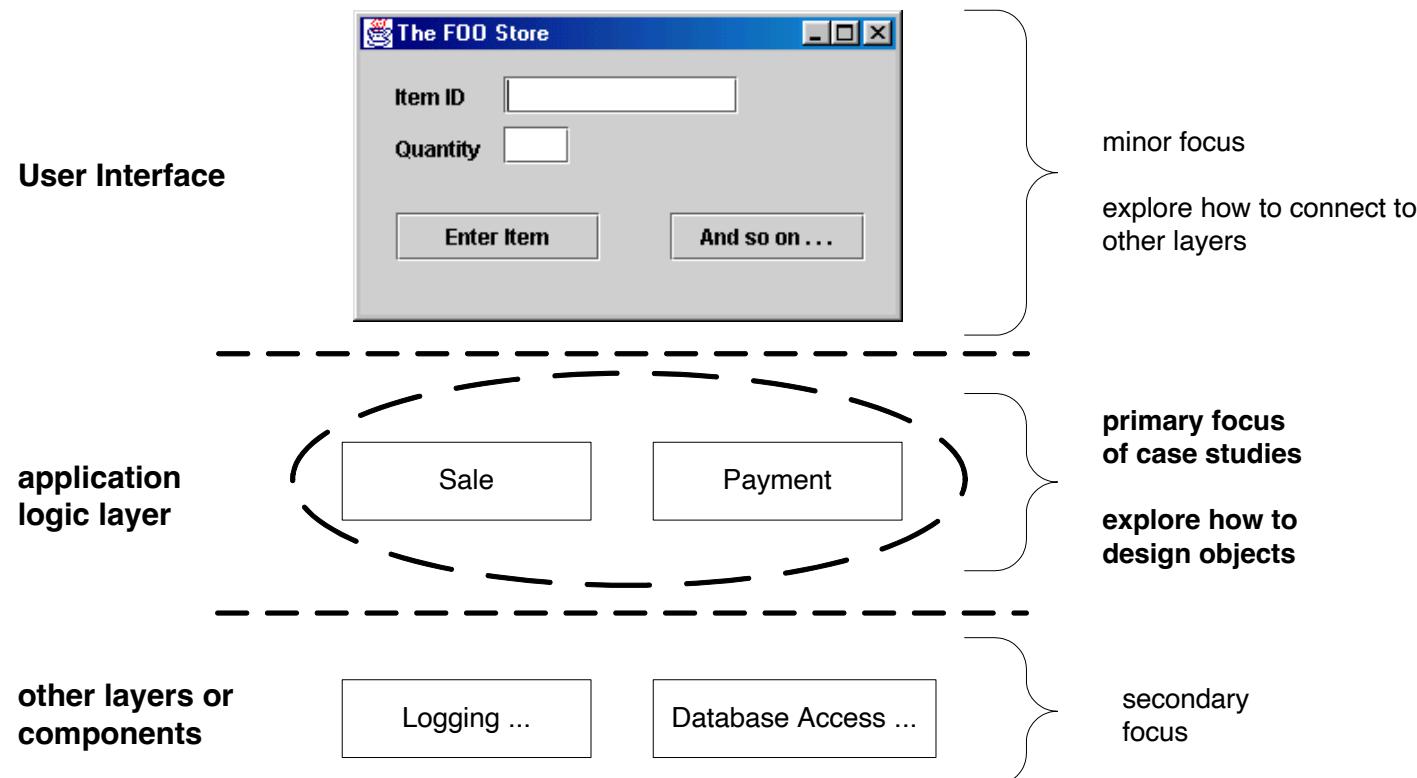
Next Gen POS Analysis

Scope of OOA & D and Process Iteration

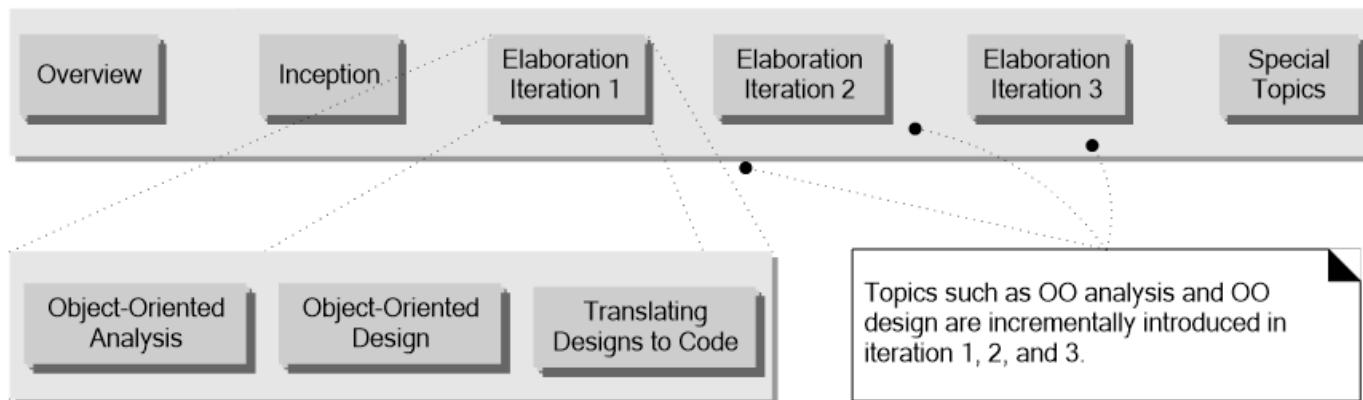
Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition).



Next Gen POS – Scope (Analysis & Design)



Iteration and Scope – Design and Construction



Iteration 1

Introduces just those analysis and design skills related to iteration one.

Iteration 2

Additional analysis and design skills introduced.

Iteration 3

Likewise.

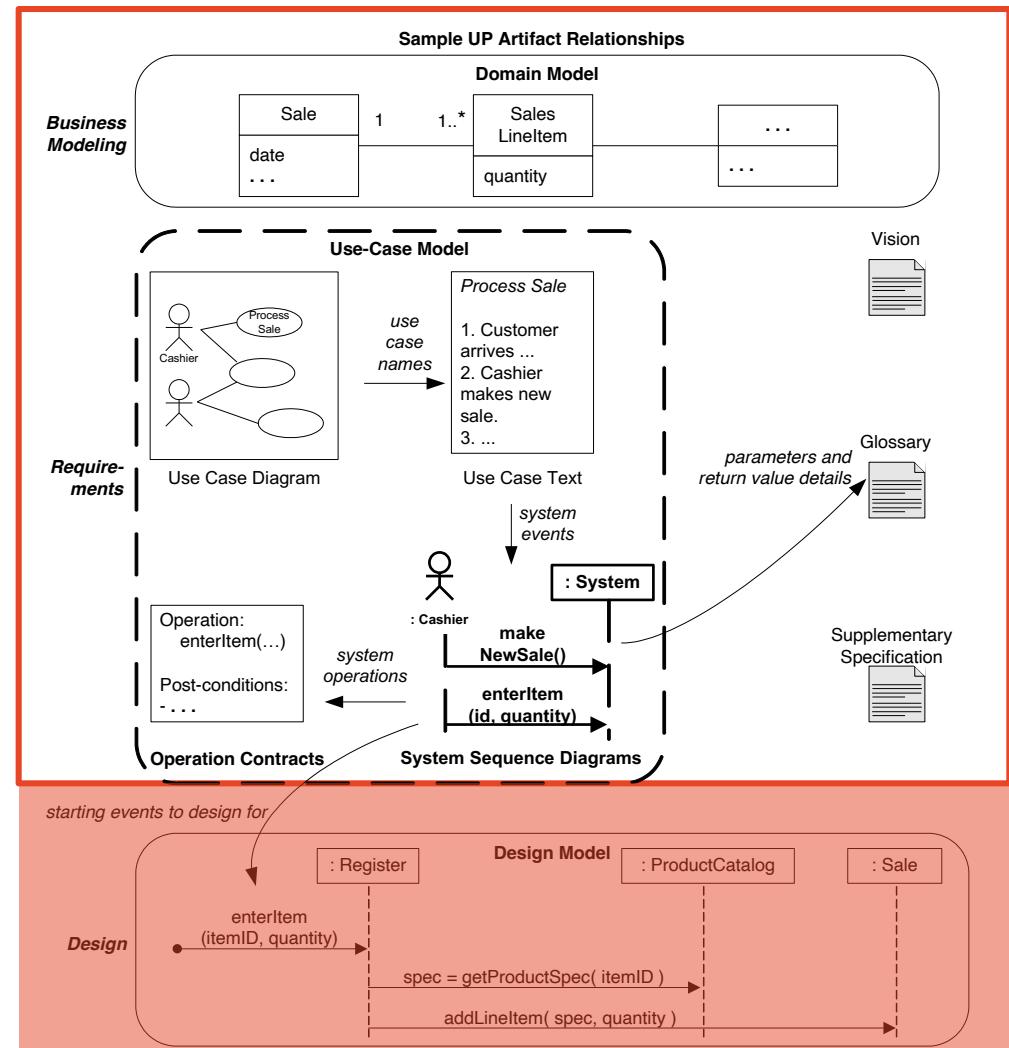
Next Gen POS: From Analysis to Design

Requirements Analysis (OOA)

Business modelling – domain models
 Use case diagrams
 Use case description
 System Sequence Diagrams

Design (OOD)

Sequence diagrams
 Class diagrams



Self Learning Case Study

1. Use Cases
2. Use Case Diagram
3. Domain Model
4. Class Diagram
5. Interaction Diagram

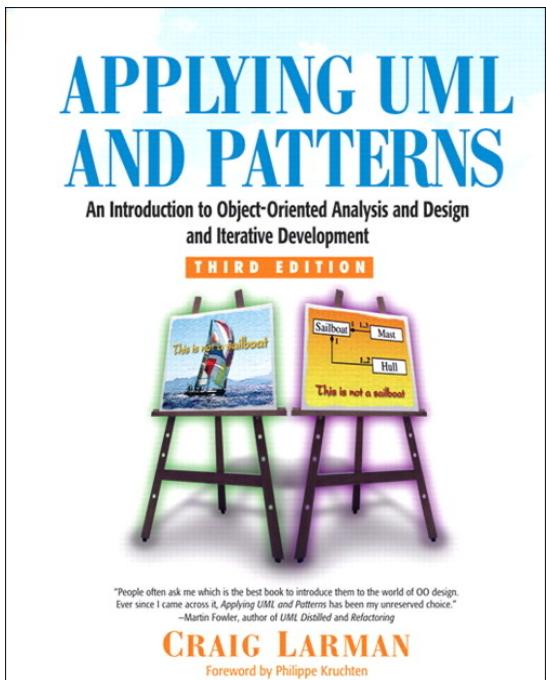
Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition).



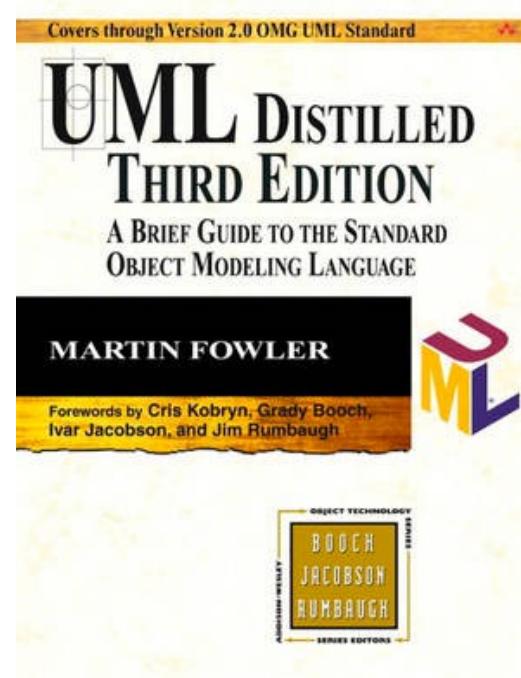
Task for Week 3

- Submit weekly exercise on canvas before 23.59pm Saturday
- Self learning on Next Gen POS system
- If you haven't heard about JSON format, please start to learn JSON by find resources online (e.g., [JSON tutorials](#))

References



[Link to USYD Library](#)



[Link to USYD Library](#)

What are we going to learn next week?

- Software Design Principles
 - Design Smells
 - SOLID
 - GRASP

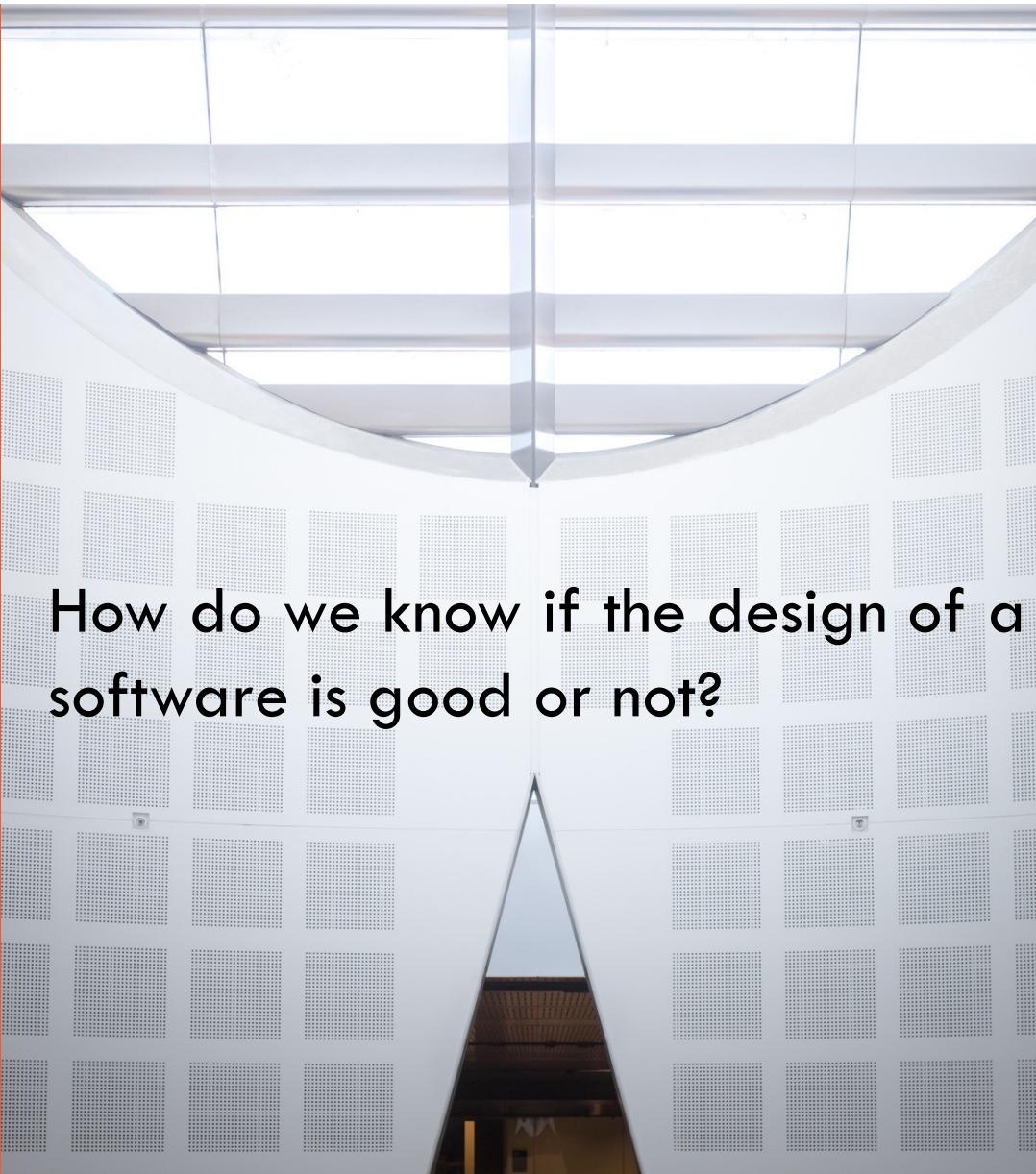
Software Design and Construction 1

SOFT2201 / COMP9201

Software Design Principles

Dr. Xi Wu

School of Computer Science



How do we know if the design of a software is good or not?

Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

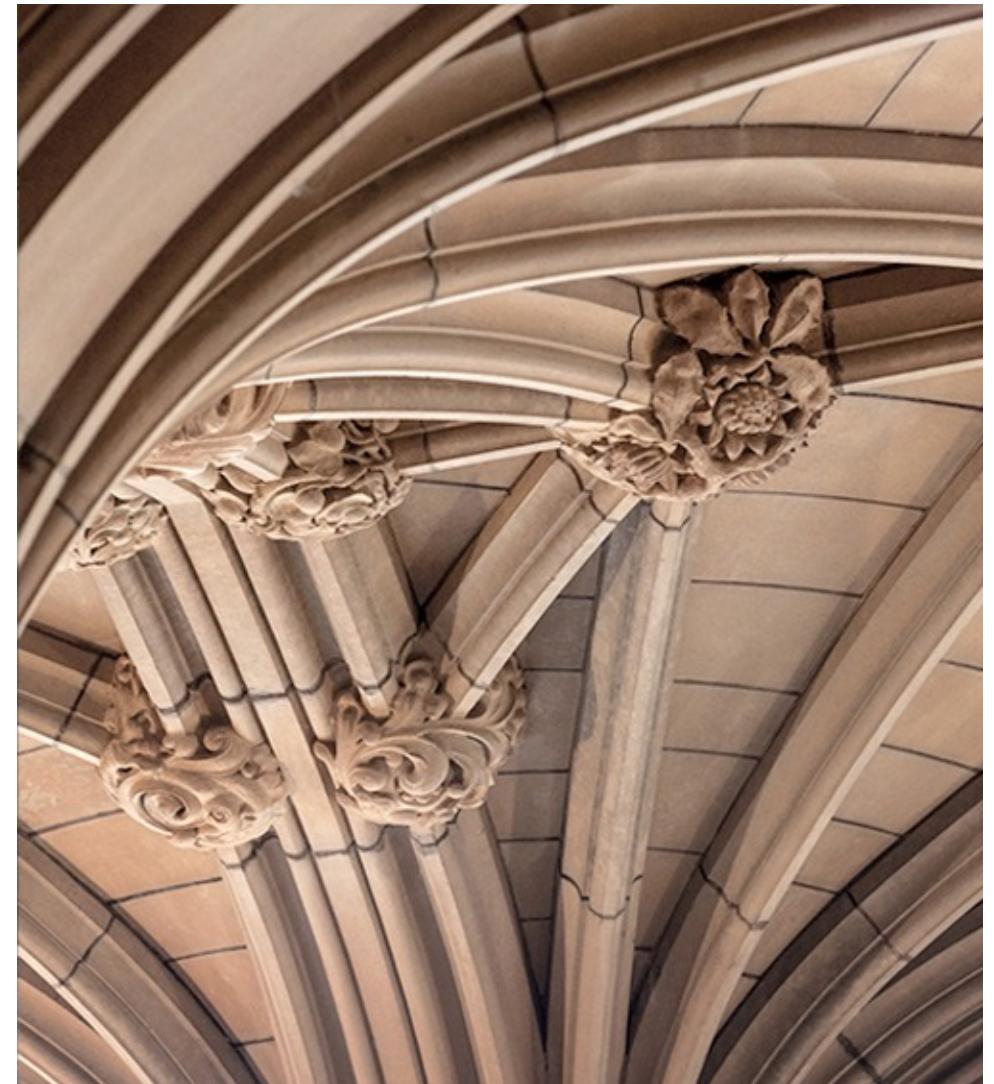
Agenda

- Design Smells
- SOLID
 - A list of excellent design principles
- GRASP
 - Designing objects with Responsibilities

Design Smells



THE UNIVERSITY OF
SYDNEY



Design Smells

“Structures in the design that indicate violation of fundamental design principles and negatively impact design quality” — Girish Suryanarayana et. Al. 2014

- Poor design decision that make the design fragile and difficult to maintain
- Bugs and unimplemented features are not accounted



Smell bad.

<http://www.codeops.tech/blog/linkedin/what-causes-design-smells/>

Girish Suryanarayana, et. al. (2014). "Refactoring for software design smells: Managing technical debt"

Common Design Smells

- **Missing abstraction:** bunches of data or encoded strings are used instead of creating an abstraction
- **Multifaceted abstraction:** an abstraction has multiple responsibilities assigned to it
- **Insufficient modularization:** an abstraction has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both
- **Cyclically-dependent modularization:** two or more abstractions depend on each other directly or indirectly (creating tightly-coupling abstractions)
- **Cyclic hierarchy:** a super-type in a hierarchy depends on any of its subtypes

Girish Suryanarayana, et. al. (2014). "Refactoring for software design smells: Managing technical debt"

Symptoms Design Smells

- **Rigidity (difficult to change):** the system is hard to change because every change forces many other changes to other parts of the system
 - A design is rigid if a single change causes a cascade of subsequent changes in dependent modules
- **Fragility (easy to break):** Changes cause the system to break in places that have no conceptual relationship to the part that was changed
 - Fixing those problems leads to even more problems
 - As fragility of a module increases, the likelihood that a change will introduce unexpected problems approaches certainty
- **Immobility (difficult to reuse):** It is hard to detangle the system into components that can be reused in other systems

Symptoms Design Smells

- **Viscosity (difficult to do the right thing)** : Doing things right is harder than doing things wrong
 - *Software*: when design-preserving methods are more difficult to use than the others, the design viscosity is high (easy to do the wrong thing but difficult to do the right thing)
 - *Environment*: when development environment is slow and inefficient.
 - *Compile times are very long, developers try to make changes that do not force large recompiles, even such changes do not preserve the design*
- **Needless Complexity**: when design contains elements that are not useful.
 - When developers anticipate changes to the requirements and put facilities in software to deal with those potential changes.

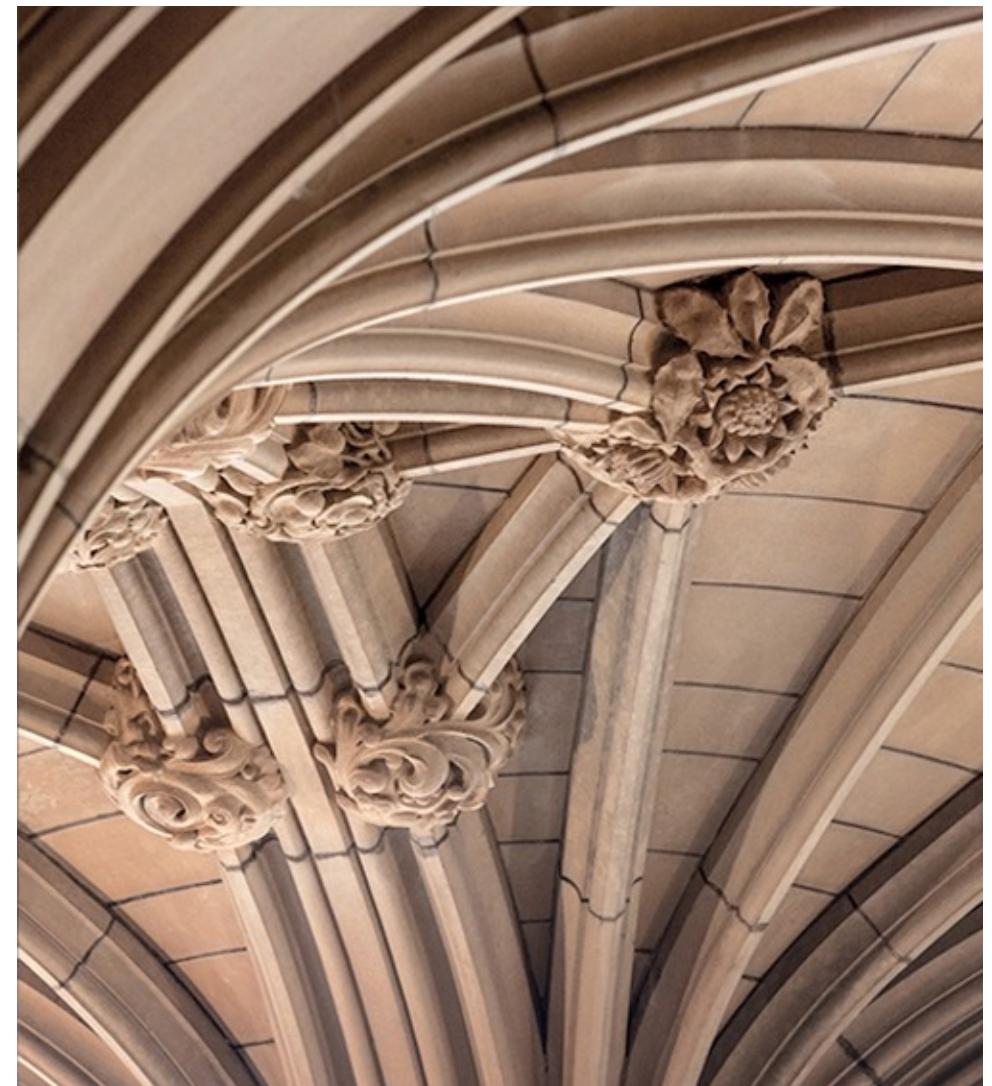
Symptoms Design Smells

- **Needless Repetition:**
 - Developers tend to find what they think relevant code, copy and paste and change it in their module
 - Code appears over and over again in slightly different forms, developers are missing an abstraction
 - Bugs found in repeating modules have to be fixed in every repetition
- **Opacity:** tendency of a module to be difficult to understand
 - Code written in unclear and non-expressive way
 - Code that evolves over time tends to become more and more opaque with age
 - Developers need to put themselves in the reader's shoes and make appropriate effort to refactor their code so that their readers can understand it

SOLID Design Principles

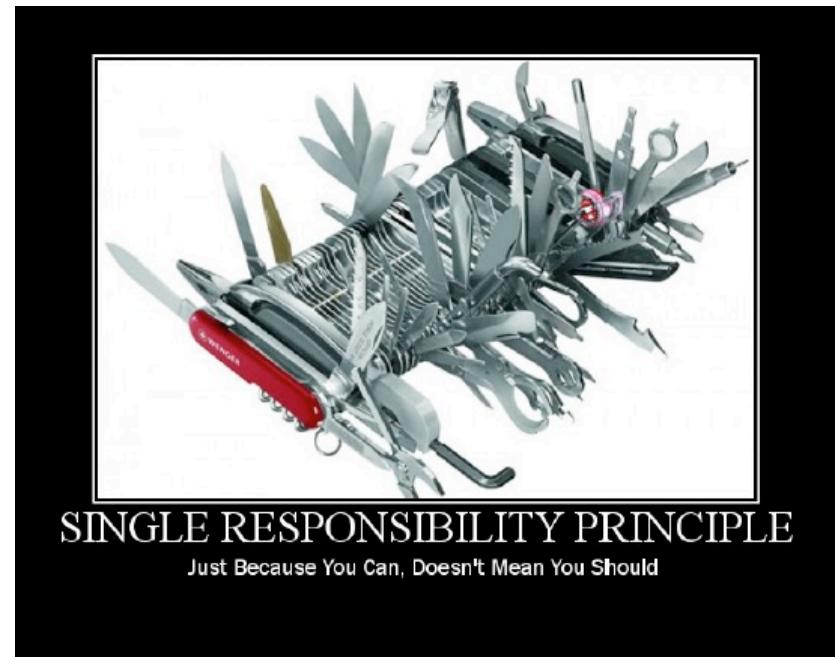


THE UNIVERSITY OF
SYDNEY



SOLID: Single Responsibility

Every class should have a single responsibility and that responsibility should be entirely met by that class



SOLID: Open/Closed

Have you ever written code that you don't want others to mess with?

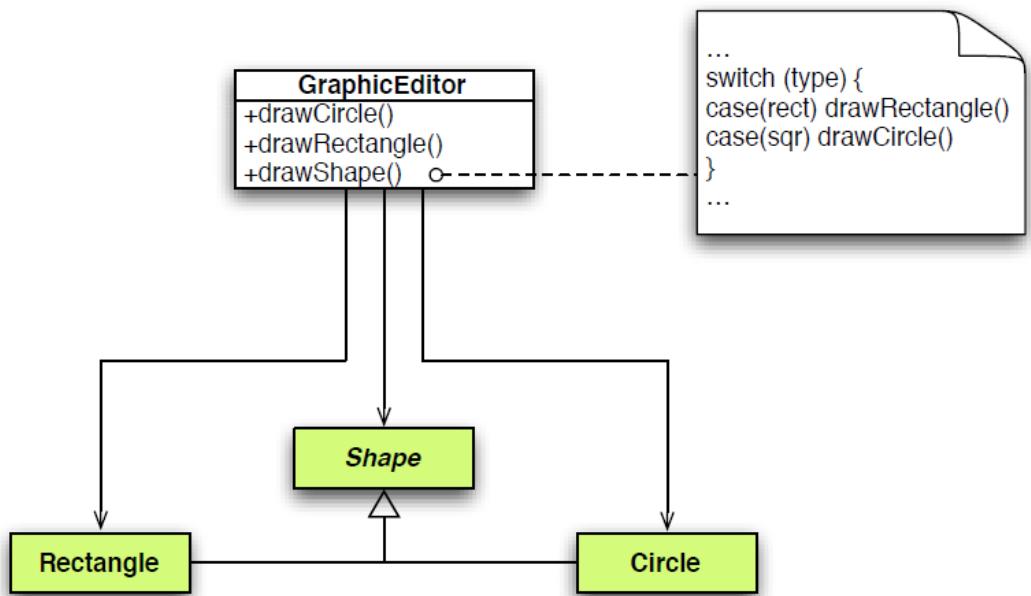
Have you ever wanted to extend something that you can't?

Open for extension but not for modification

The Open/Closed principle is that you should be able to extend code without breaking it. That means not altering superclasses when you can do as well by adding a subclass.

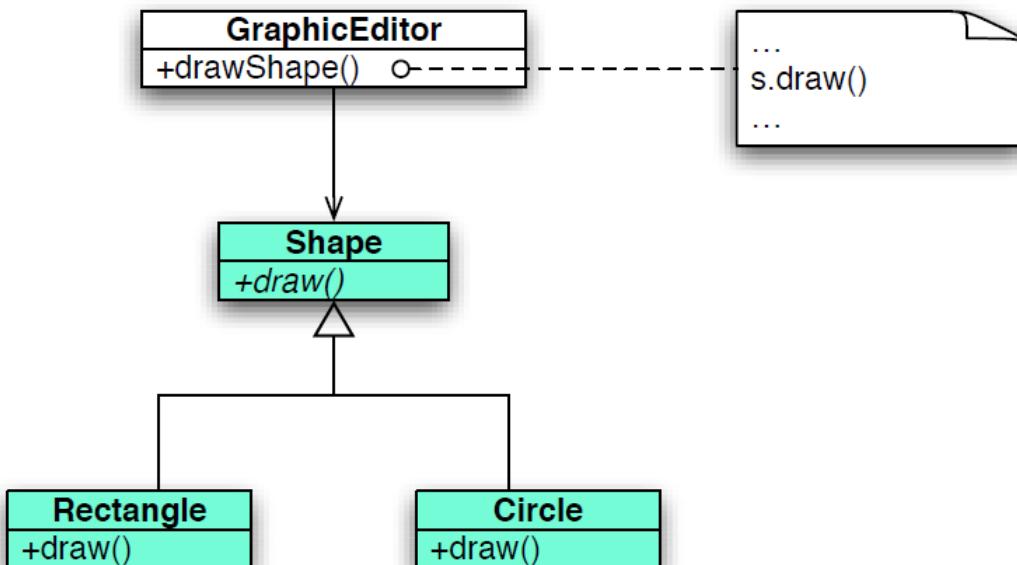
New subtypes of a class should not require changes to the superclass

SOLID: Open/Closed



This really isn't a good design: every time a new object is introduced to be drawn, the base class has to be changed

SOLID: Open/Closed



This is much better: each item has its own draw method which is called at runtime through polymorphism mechanisms

SOLID: Liskov Substitution Principle

- In 1987 Barbara Liskov introduced her idea of strong behavioral subtyping, later formalized in a 1994 paper with Jeannette Wing and updated in a 1999 technical report as

Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be provable for objects y of type S where S is a subtype of T .

- This defines required behaviours for mutable(changeable) objects: if S is a subtype of T , then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.
- What's "desirable"? One example is correctness. . .

```
Shape x = new Shape(); // property q: q(x) = true
```

Assume Square extends Shape

```
Square y = new Square(); // property q: q(y) = true
```

All places where we use x can be replaced by y, no side effect can be observed from outside of the system

Substitutability

- Suppose we have something like this in our code: after a definition of `someRoutine()` we have
- But now we want to replace `someRoutine` with `someNewRoutine` with a guarantee of no ill-effects, so now we need the following: `someNewRoutine();`

ALGORITHM

```
// presomeRoutine must be true here
    someRoutine()
// postsomeRoutine must be true here
```

```
// presomeRoutine must be true here
// presomeNewRoutine must be true here
    someNewRoutine()
// postsomeNewRoutine must be true here
// postsomeRoutine must be true here
```

This means, for a routine r substituted by s

$$\text{pre}_r \Rightarrow \text{pre}_s \quad \text{and} \quad \text{post}_s \Rightarrow \text{post}_r$$

Substitutability

- In other words;
 - Pre-conditions cannot get stronger, Post-conditions cannot get weaker
- The aim is not to see if a replacement can do something the original couldn't but can the replacement do everything the original could under the same circumstances.
- Substitutability is asking the question 'Can one substitute one type for another with a guarantee of no ill-effects? We might need to consider substitutability in cases:
 - Refactoring
 - Redesign
 - Porting
- The context is 'changing something in existing use'

SOLID: Interface Segregation

You should not be forced to implement interfaces you don't use!



SOLID: Dependency Inversion

No complex class should depend on simpler classes it uses; they should be separated by interfaces (abstract classes)

The details of classes should depend on the abstraction (interface), not the other way around



Summary of SOLID Principles

Single Responsibility: Every class should have a single responsibility and that responsibility should be entirely met by that class;

Open/Closed: Open for extension but closed for modification; inheritance is used for this, e.g. through the use of inherited abstract base classes;

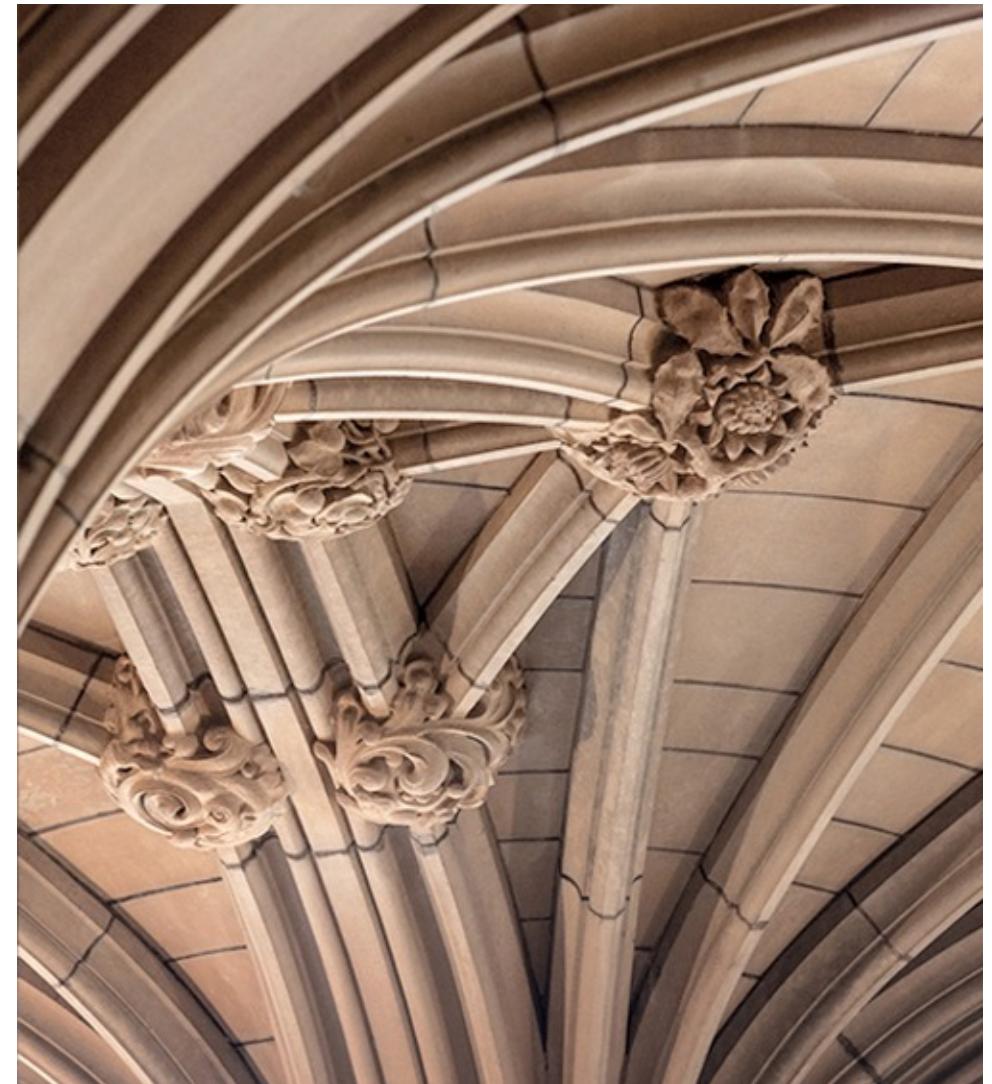
Liskov Substitutability: If $S <: T$ ("S is a subtype of T") then a T object can be replaced with an S object and no harm done;

Interface Segregation: Client code should not have to implement interfaces it doesn't need;

Dependency Inversion: High level, complex modules should not depend on low-level, simple models — use abstraction, and implementation details should depend on abstractions, not the other way around.

General Responsibility Assignment Software Pattern (GRASP)

Designing objects with responsibilities



Object Design

- “Identify requirements, create a domain model, add methods to the software classes, define messages to meet requirements...”
- Too Simple!
 - What methods belong where?
 - How do we assign **responsibilities** to classes?
- The critical design tool for software development is a mind well educated in design **principles** and **patterns**.

Responsibility Driven Design

- Responsibility is a contract or obligation of a class
- What must a class “know”? [knowing responsibility]
 - Private encapsulated data
 - Related objects
 - Things it can derive or calculate
- What must a class “do”? [doing responsibility]
 - Take action (create an object, do a calculation)
 - Initiate action in other objects
 - Control/coordinate actions in other objects
- Responsibilities are assigned to classes of objects during object design

Responsibilities: Examples

- “A *Sale* is responsible for creating *SalesLineItems*” (**doing**)
- “A *Sale* is responsible for knowing its total” (**knowing**)
- **Knowing** responsibilities are related to attributes, associations in the domain model
- **Doing** responsibilities are implemented by means of methods.

GRASP: Methodological Approach to OO Design

General Responsibility Assignment Software Patterns

The five basic principles:

- Creator
- Information Expert
- High Cohesion
- Low Coupling
- Controller

GRASP: Creator Principle

Problem

Who creates an A object

Solution

Assign class B the responsibility to create an instance of class A if one of these is true

B “contains” A

B “records” A

B “closely uses” A

B “has the Initializing data for” A



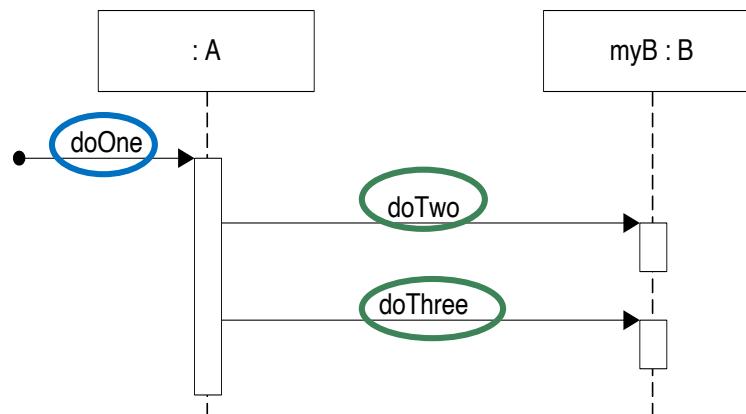
GRASP: Information Expert Principle

Problem

What is a general principle of assigning responsibilities to objects

Solution

Assign a responsibility to the class that has the information needed to fulfill it



Dependency

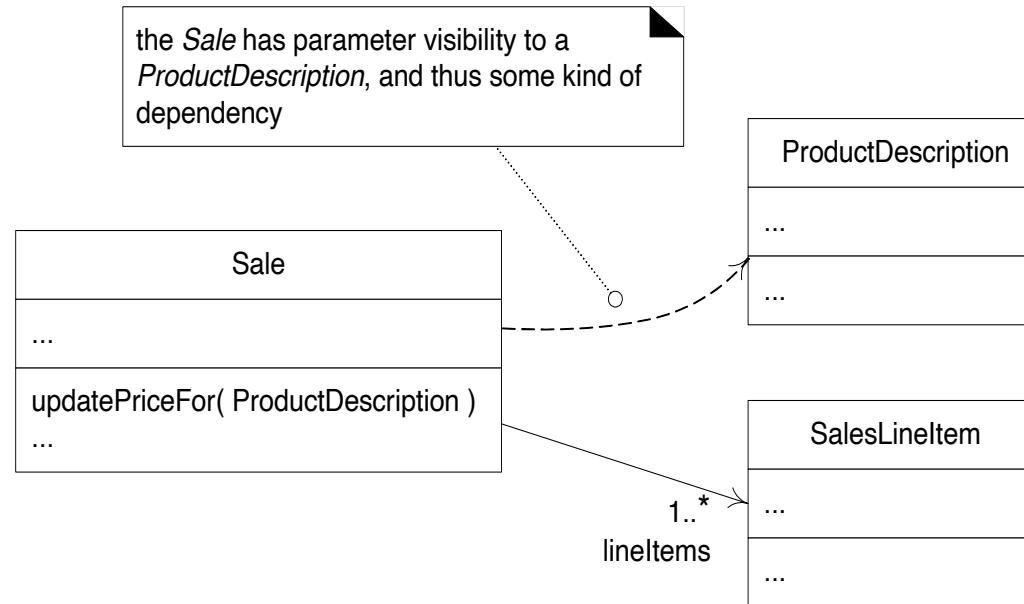
- A dependency exists between two elements if changes to the definition of one element (**the supplier**) may cause changes to the other (**the client**)
- Various reason for dependency
 - Class send message to another
 - One class has another as its data
 - One class mention another as a parameter to an operation
 - One class is a superclass or interface of another

When to show dependency?

- Be selective in describing dependency
- Many dependencies are already shown in other format
- Use dependency to depict global, parameter variable, local variable and static-method.
- Use dependencies when you want to show how changes in one element might alter other elements

Dependency: Parameter Variable

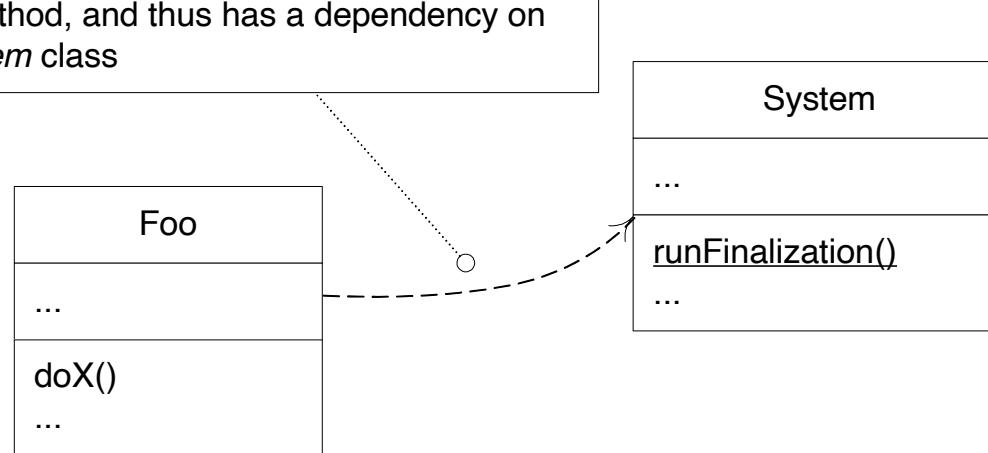
```
public class Sale{  
    public void updatePriceFor (ProductDescription description){  
        Money basePrice = description.getPrice();  
        //...  
    }  
}
```



Dependency: static method

```
public class Foo{  
    public void doX(){  
        System.runFinalization();  
        //..  
    }  
}
```

the *doX* method invokes the *runFinalization* static method, and thus has a dependency on the *System* class

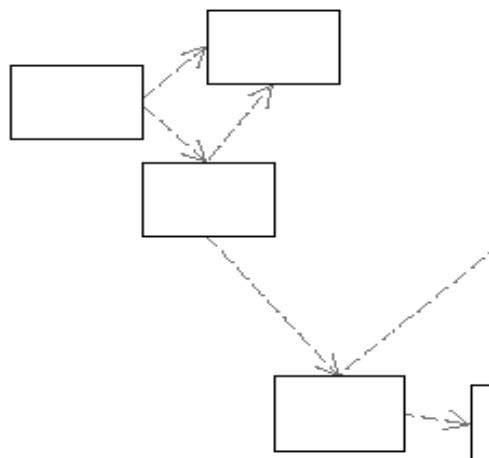


Dependency labels

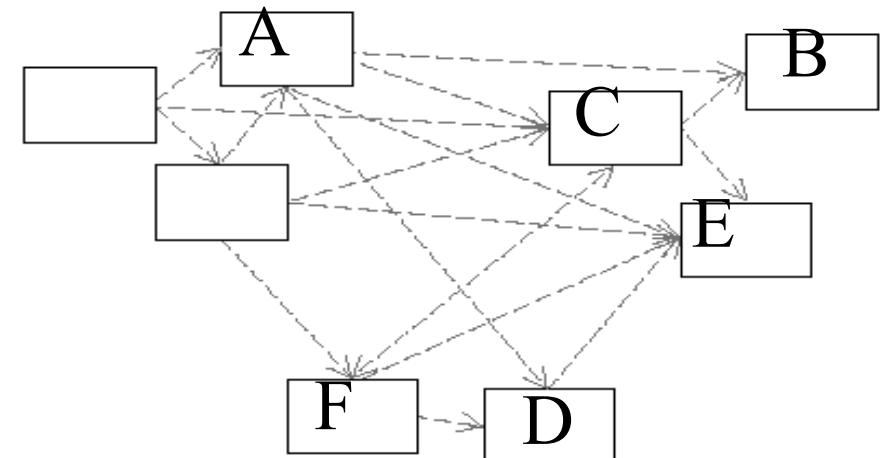
- There are many varieties of dependency, use keywords to differentiate them
- Different tools have different sets of supported dependency keywords.
 - <<call>> the source calls an operation in the target
 - <<use>> the source requires the targets for its implementation
 - <<parameter>> the target is passed to the source as parameter.

Coupling

- How strongly one element is connected to, has knowledge of, or depends on other elements
- Illustrated as **dependency** relationship in UML class diagram



Low coupling



High coupling

GRASP: Low Coupling Principle

Problem

How to reduce the impact of change, to support low dependency, and increase reuse?

Solution

Assign a responsibility so that coupling remains low

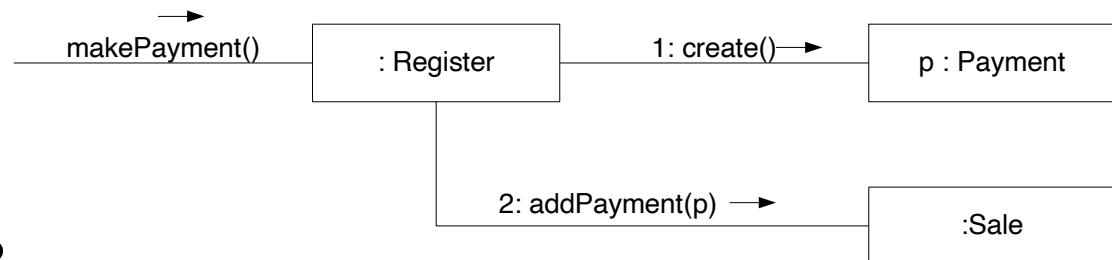
Coupling – Example (NextGen POS)

We need to create a *Payment* instance and associate it with the *Sale*.

What class should be responsible for this?



Since *Register* record a payment in the real-world domain, the Creator pattern suggests register as a candidate for creating the payment

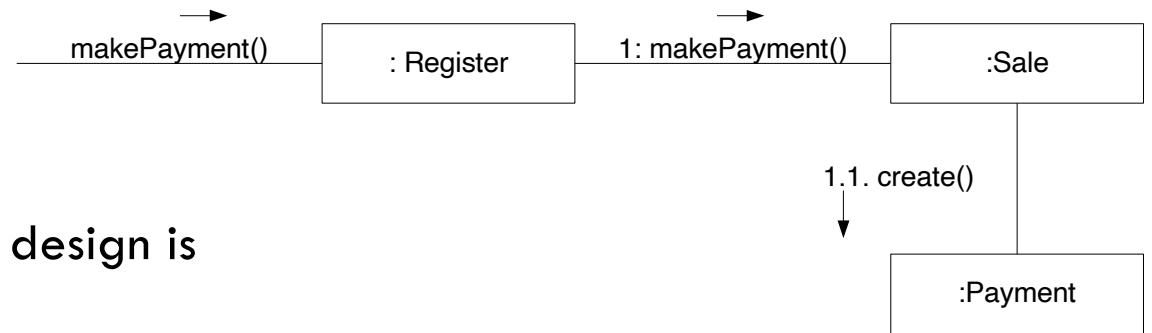


This assignment couple the *Register* class to knowledge of the *Payment* class which increases coupling

Coupling – Example (NextGen POS)

Better design from coupling point of view

It maintains overall lower coupling



From creator point of view, the previous design is better.

In practice, consider the level of couple along with other principles such as Expert and High Cohesion

Cohesion

- How strongly related and focused the responsibilities of an element are
- Formal definition (calculation) of cohesion
 - Cohesion of two methods is defined as the intersection of the sets of instance variables that are used by the methods
 - If an object has different methods performing different operations on the same set of instance variables, the class is cohesive

High Cohesion

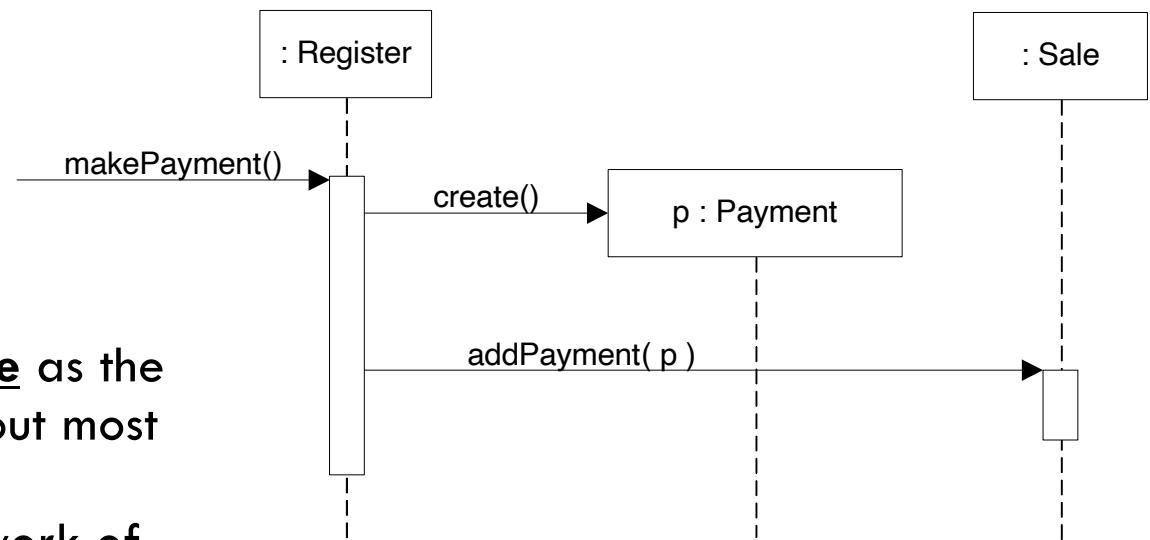
- **Problem**
 - How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
- **Solution**
 - Assign responsibilities so that cohesion remains high

Cohesion – Example (NextGen POS)

We need to create a (cash) *Payment* instance and associate it with the *Sale*.
What class should be responsible for this?

Since *Register* records a payment in the real-world domain, the Creator pattern suggests *register* as a candidate for creating the payment

Acceptable but could become incohesive as the *Register* will increasingly need to carry out most of the system operations assigned to it e.g., *Register* responsible for doing the work of 20 operations (overburden)

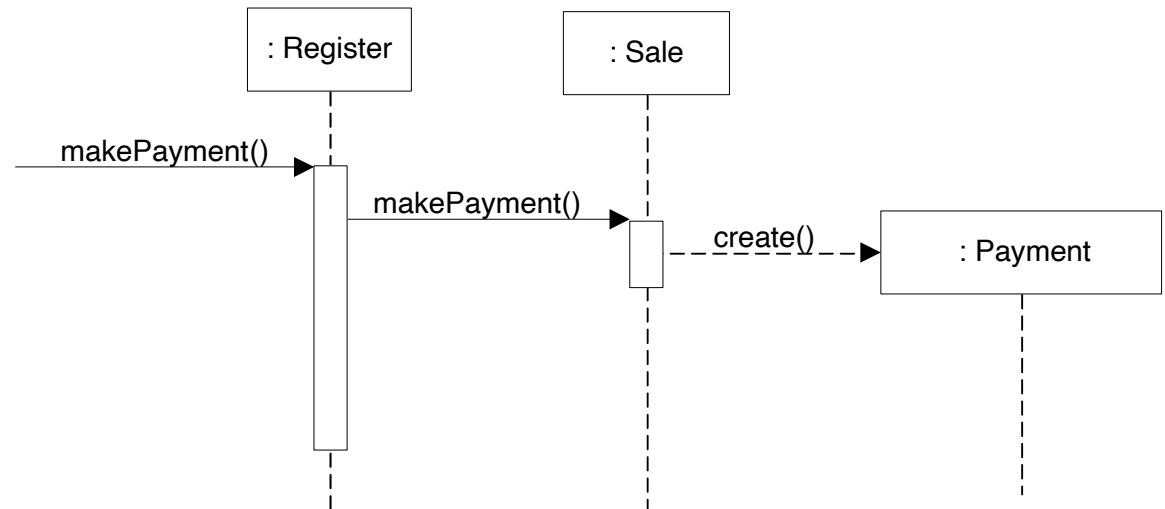


Cohesion – Example (NextGen POS)

Better design from cohesion point of view

The *payment creation responsibility* is delegated to the *Sale* instance

It supports high cohesion and low coupling



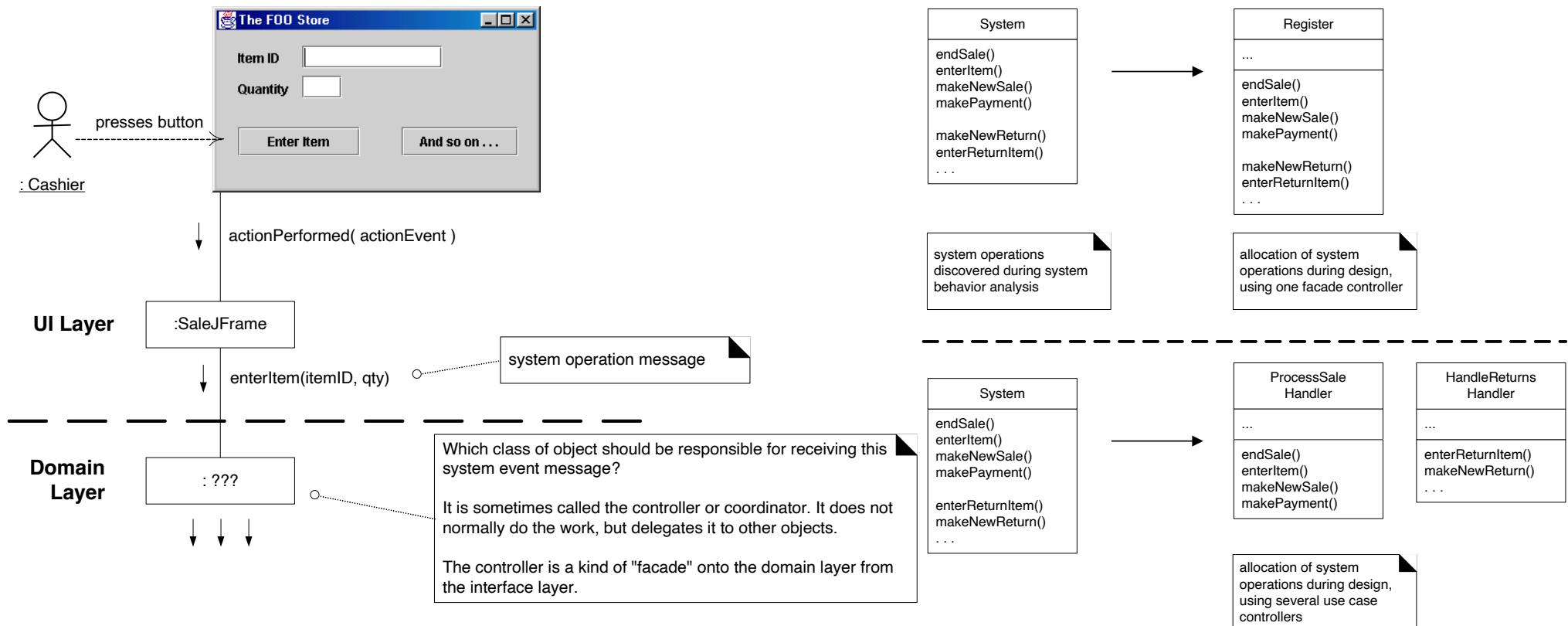
Coupling and Cohesion

- Coupling describes the inter-objects relationship
- Cohesion describes the intra-object relationship
- Extreme case of “coupling”
 - Only one class for the whole system
 - No coupling at all
 - Extremely low cohesion
- Extreme case of cohesion
 - Separate even a single concept into several classes
 - Very high cohesion
 - Extremely high coupling
- Domain model helps to identify concepts
- OOD helps to assign responsibilities to proper concepts

Controller

- Problem
 - What first object beyond the UI layer receives and coordinates (“controls”) a system operation
- Solution
 - Assign the responsibility to an object representing one of these choices
 - Represents the overall system, root object, device or subsystem (a façade controller)
 - Represents a use case scenario within which the system operations occurs (a use case controller)

Controller



Task for Week 4

- Submit weekly exercise on canvas before 23.59pm Saturday
- Self learning on Next Gen POS system (Extended Version)
- Submit assignment 1 on canvas before its due.
 - **All assignments are individual assignments**
 - Please note that: work must be done individually without consulting someone else's solutions in accordance with the University's "**Academic Dishonesty and Plagiarism**" policies

References

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- Robert Cecil Martin. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- Girish Suryanarayana et al. 2015. Refactoring for software design smells : managing technical debt. Waltham, Massachusetts ;: Morgan Kaufmann. Print.

What are we going to learn next week?

- Design Patterns
 - GoF Design Patterns
- Creational Design Patterns
 - Factory Method Pattern
 - Builder Pattern

Software Design and Construction 1

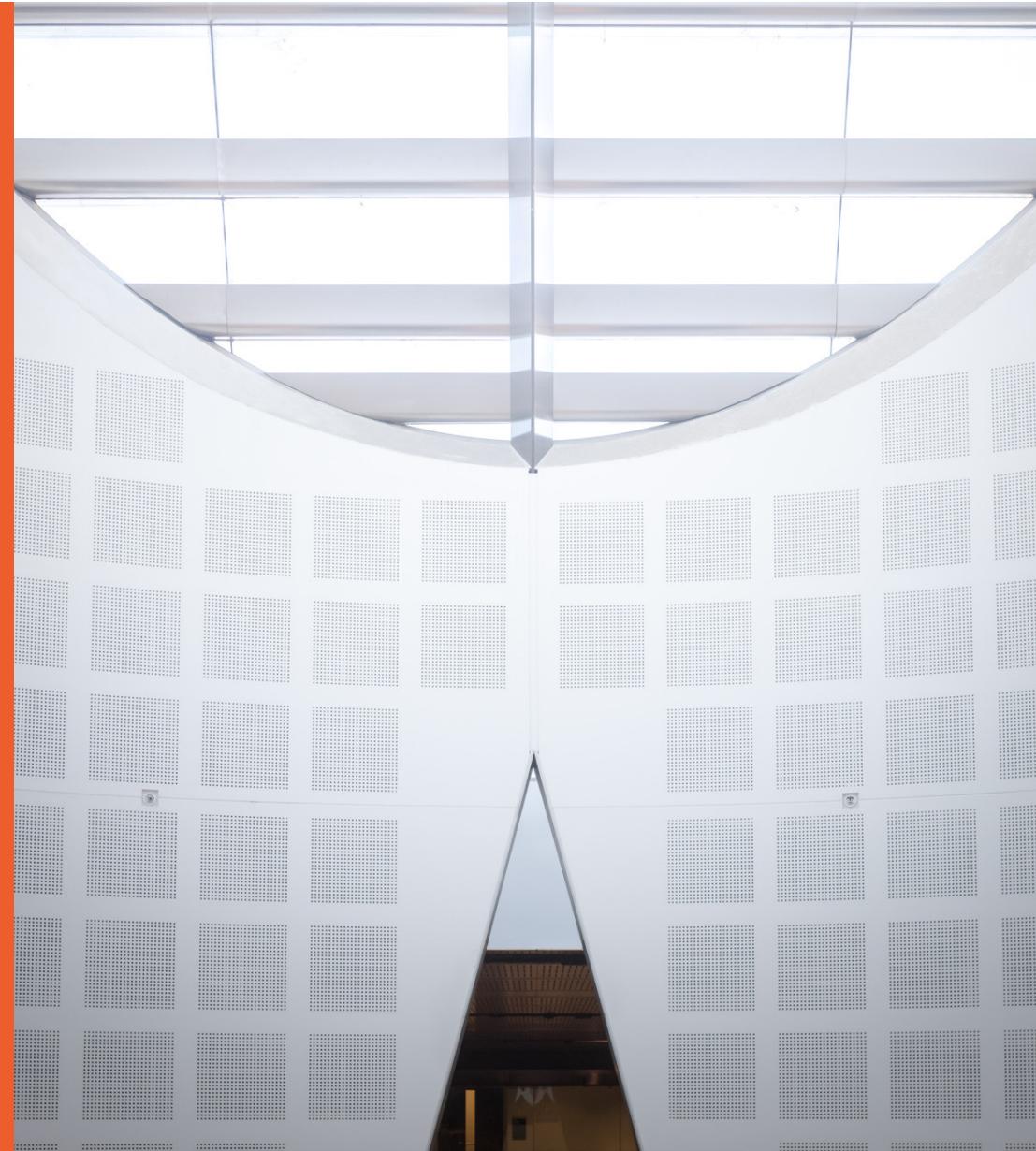
SOFT2201 / COMP9201

**Introduction to Design
Patterns**

Dr. Xi Wu



School of Computer Science



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

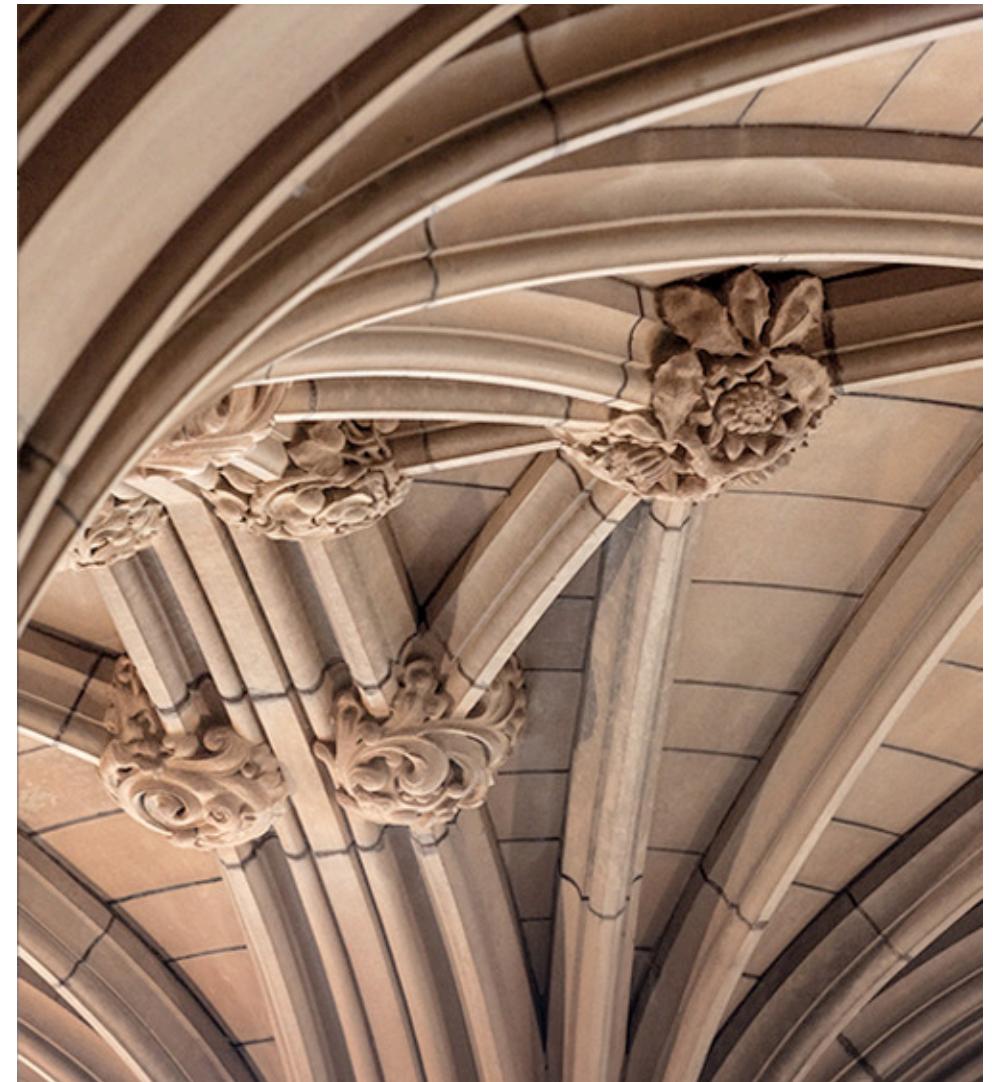
Agenda

- Design Patterns
 - GoF Design Patterns
- Creational Design Patterns
 - Factory Method Pattern
 - Builder Pattern

What is Design Pattern?



THE UNIVERSITY OF
SYDNEY



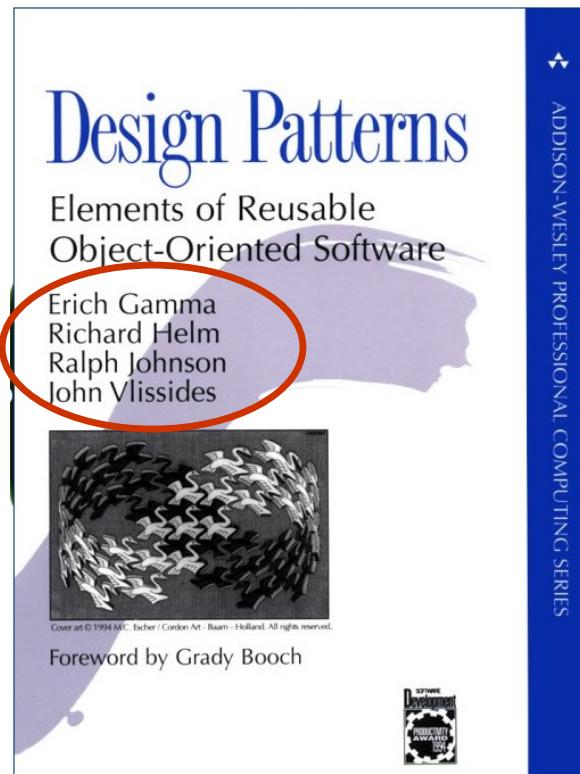
Design Patterns

- A pattern is a description of a problem and its solution
- Tried and tested ideas for recurring design problem
- Not readily coded solution, but rather the solution path to a common programming problem
- Design or implementation **structure** that achieves a particular purpose

Essential Components of a Pattern

- **The pattern name**
 - e.g., Factory Method
- **The problem**
 - The pattern is designed to solve (i.e., when to apply the pattern)
- **The solution**
 - The components of the design and how they related to each other
- **Consequence**
 - The results and trade-offs of applying the pattern
 - Advantages and disadvantages of using the pattern

Gang of Four Patterns (GoF)



- Official design pattern reference
- Famous and influential book about design patterns
- Recommended for students who wish to become experts
- We will cover the most widely - used patterns from the book
- 23 patterns in total - our unit will focus on 11
- GoF Design Patterns → Design Patterns as short

Design Patterns – Classification based on purpose

Scope	Creational	Structural	Behavioural
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object)	Chain of Responsibility
		Bridge	Command
		Composite	Iterator
		Decorator	Mediator
		Façade	Memento
		Flyweight	Observer
		Proxy	State
	Strategy Visitor		Strategy
			Visitor

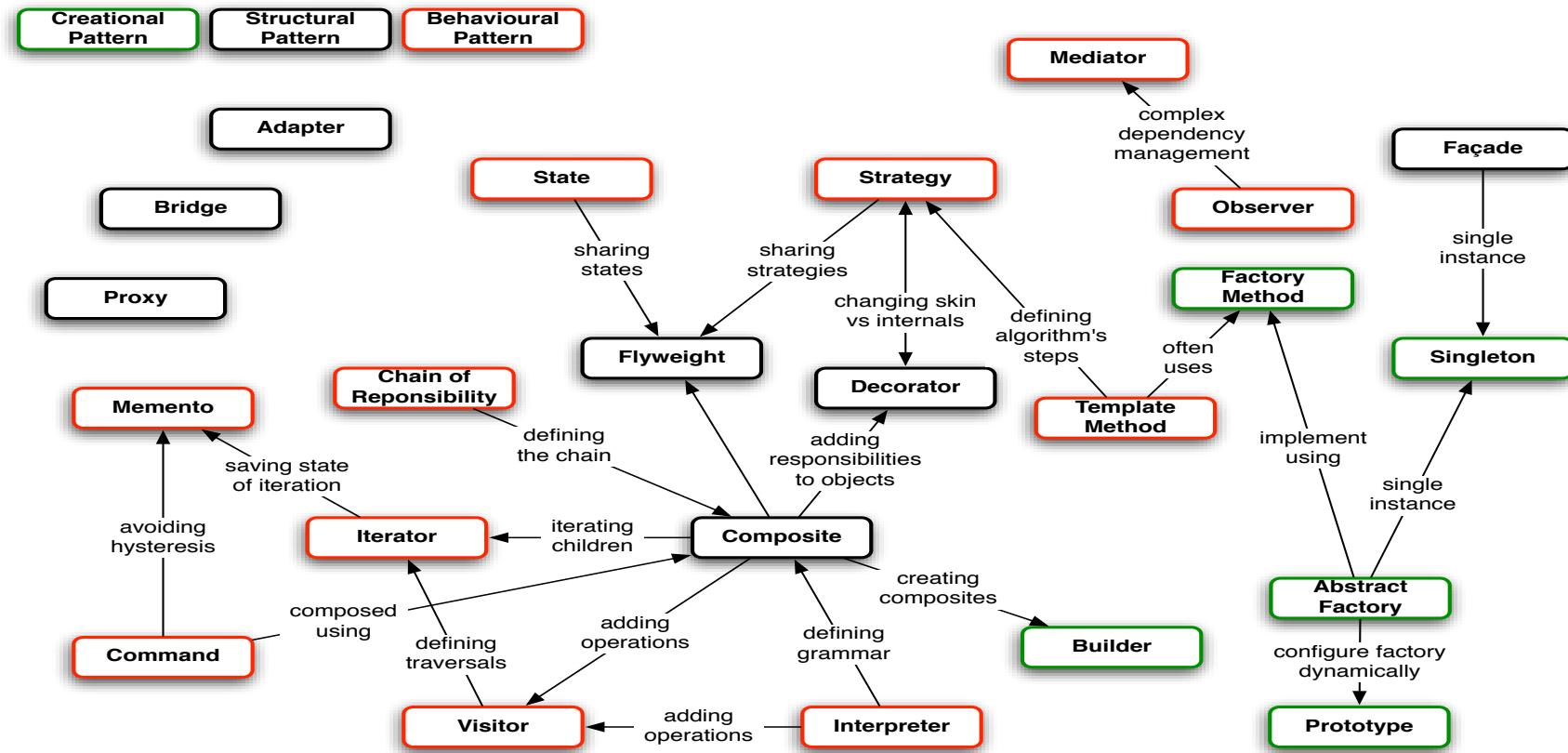
Design Patterns – Classification based on purpose

- **Creational patterns**
 - Abstract the instantiation process
 - Make a system independent of how its objects are created, composed and represented
- **Structural patterns**
 - How classes and objects are composed to form larger structures
- **Behavioral patterns**
 - Concerns with algorithms and the assignment of responsibilities between objects

Design Patterns – Classification based on relationships

- Patterns often used together
 - E.g., Composite often used with Iterator or Visitor
- Alternative Patterns
 - E.g., Prototype often alternative to Abstract Factory
- Patterns results in similar designs
 - E.g., Structure diagram of composite and Decorator are similar

Design Patterns – Classification based on relationships



* Adapted from the GoF book: Design Patterns

Selecting an Appropriate Design Pattern

It is useful to have some guidelines of how to select one. Here are some thoughts on how you might do that:

- Consider the ways in which the design patterns solve problems.
 - We will go into details on this for several design patterns
- Decide on what the intent of each design is.
 - Without knowing what the motivation of the design pattern is, it will be hard to know whether it is right for you
- Look at the relationships among patterns
 - It makes sense to use patterns that have a clear relationship, rather than one that you have manufacture ad hoc

Selecting an Appropriate Design Pattern

- Consider patterns with similar purpose
 - Creational, Structural and Behavioral are quite different purposes so you should consider which one you need
- Look at why redesign might be necessary
 - Knowing why a redesign might be needed will help you select the right design to avoid having to redesign later
- Why can vary?
 - Your design should be open to variation where necessary
 - Choose a design that will not lock you into a particular one, and enable you to make variations without changing your design

Design Aspects Can be Varied by Design Patterns

Purpose	Pattern	Aspects that can change
Creational	Abstract Factory	families of product objects
	Builder	how a composite object gets created
	Factory Method	subclass of object that is instantiated
	Prototype	class of object that is instantiated
	Singleton	the sole instance of a class
Structural	Adapter	interface to an object
	Bridge	implementation of an object
	Composite	structure and composition of an object
	Decorator	responsibilities of an object without subclassing
	Façade	interface to a subsystem
	Flyweight	storage costs of objects
	Proxy	how an object is accessed; its location

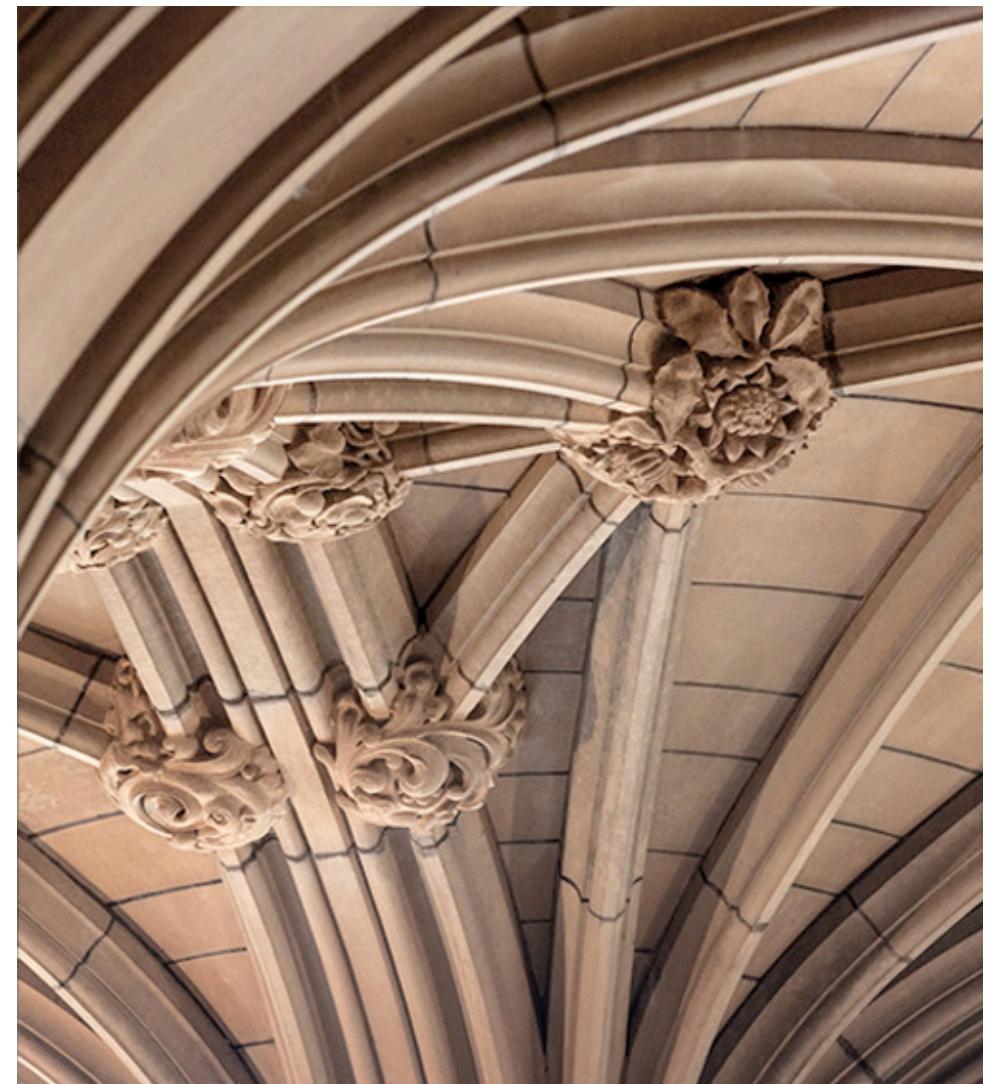
Design Aspects Can be Varied by Design Patterns

Purpose	Pattern	Aspects that can change
Behavioral	Chain of Responsibility	object that can fulfill a request
	Command	when and how a request is fulfilled
	Interpreter	grammar and interpretation of a language
	Iterator	how an aggregate's elements are accessed, traversed
	Mediator	how and which objects interact with each other
	Memento	what private info. is stored outside an object, & when
	Observer	number of objects that depend on another object; how the dependent objects stay up to date
	State	states of an object
	Strategy	an algorithm
	Template Method	steps of an algorithm
	Visitor	operations that can be applied to object(s) without changing their class(es)

Creational Patterns



THE UNIVERSITY OF
SYDNEY



Creational Patterns

- Abstract the instantiation process
- Make a system independent of how its objects are created, composed and represented
 - Class creational pattern uses inheritance to vary the class that's instantiated
 - Object creational pattern delegates instantiation to another object
- Provides flexibility in *what gets created, who creates it, how it gets created and when*

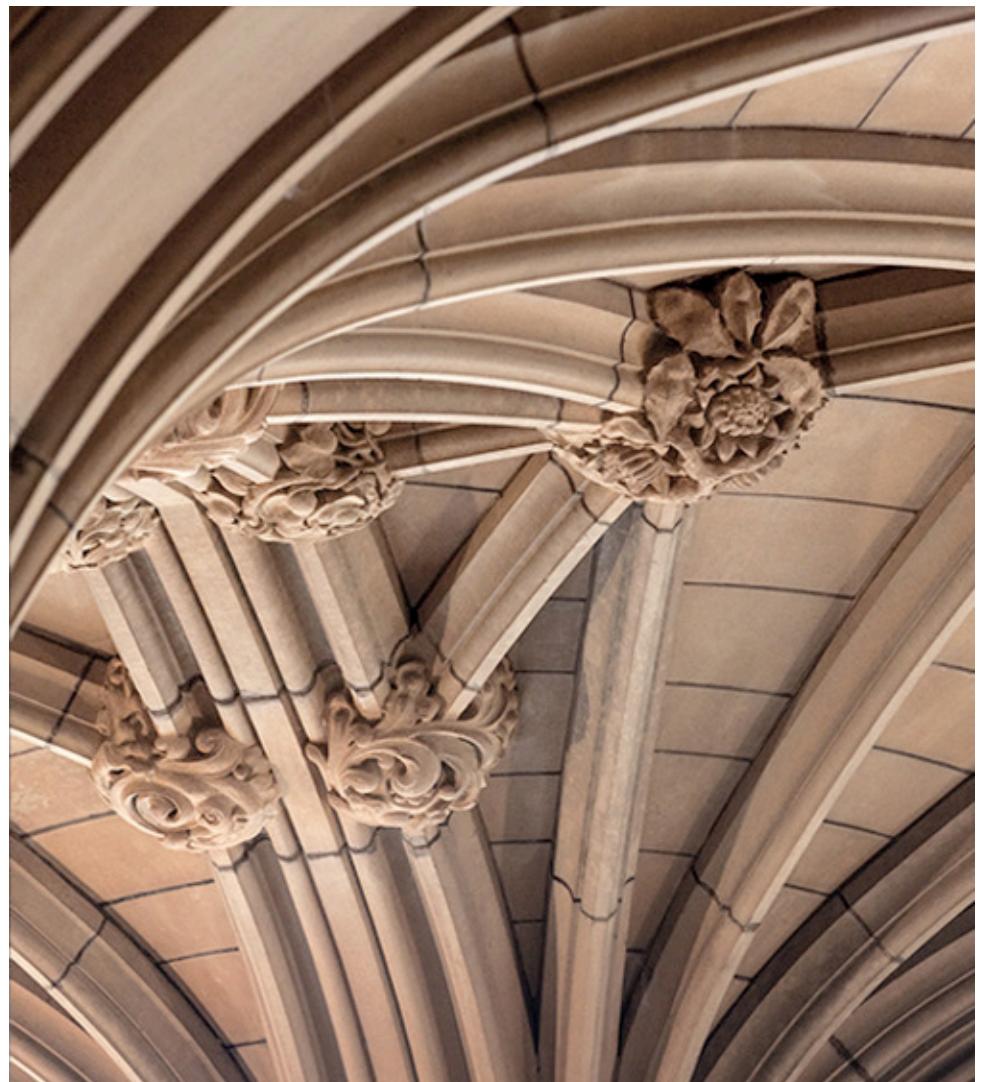
Creational Patterns

Pattern Name	Description
Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes
Singleton	Ensure a class only has one instance, and provide global point of access to it
Factory Method	Define an interface for creating an object, but let sub-class decide which class to instantiate (class instantiation deferred to subclasses)
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations
Prototype	Specify the kinds of objects to create using a prototype instance, and create new objects by copying this prototype

Factory Method

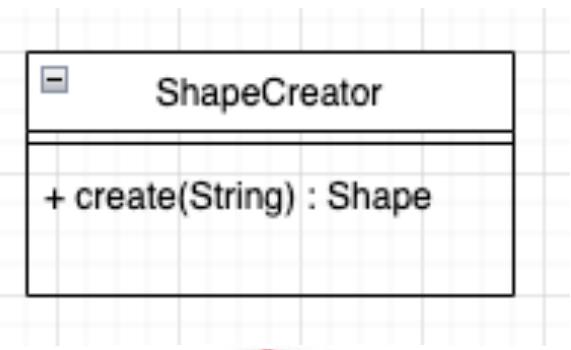
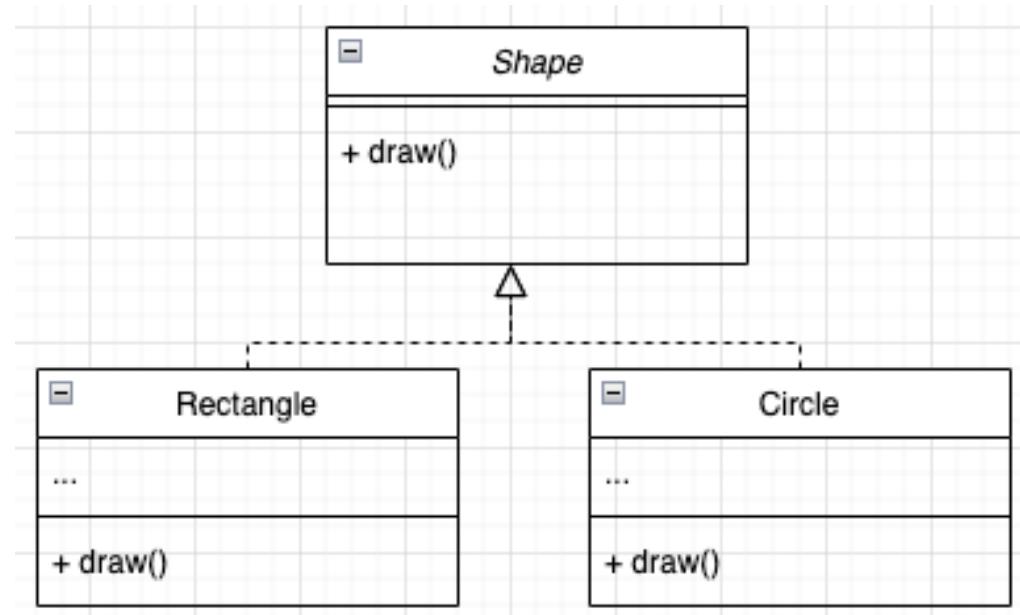
Class Creational Pattern

An interface for creating an object



Motivated Scenario

- Suppose you have a general shape creator that can create a variety of different shapes.



Motivated Scenario

```
public class ShapeCreator {  
    1 usage  
    public Shape create(String type) {  
        Shape shape = null;  
        switch (type) {  
            case "rectangle":  
                shape = new Rectangle();  
                break;  
            case "circle":  
                shape = new Circle();  
                break;  
        }  
        return shape;  
    }  
}
```

```
public interface Shape {  
    1 usage 2 implementations  
    public void draw();  
}
```

```
public class Circle implements Shape{  
    1 usage  
    public void draw() { System.out.println("I am drawing a circle"); }  
}
```

```
public class Rectangle implements Shape {  
    1 usage  
    public void draw() { System.out.println("I am drawing a rectangle"); }  
}
```



Client Perspective:

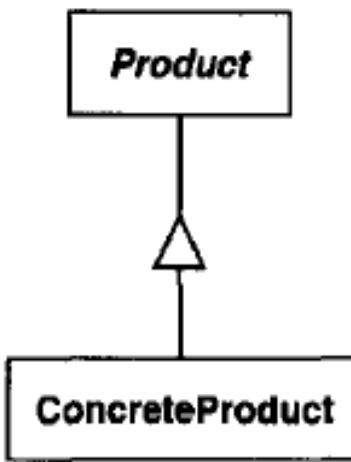
```
ShapeCreator creator = new ShapeCreator();  
Shape shape = creator.create("circle");  
shape.draw();
```

Factory Method Pattern

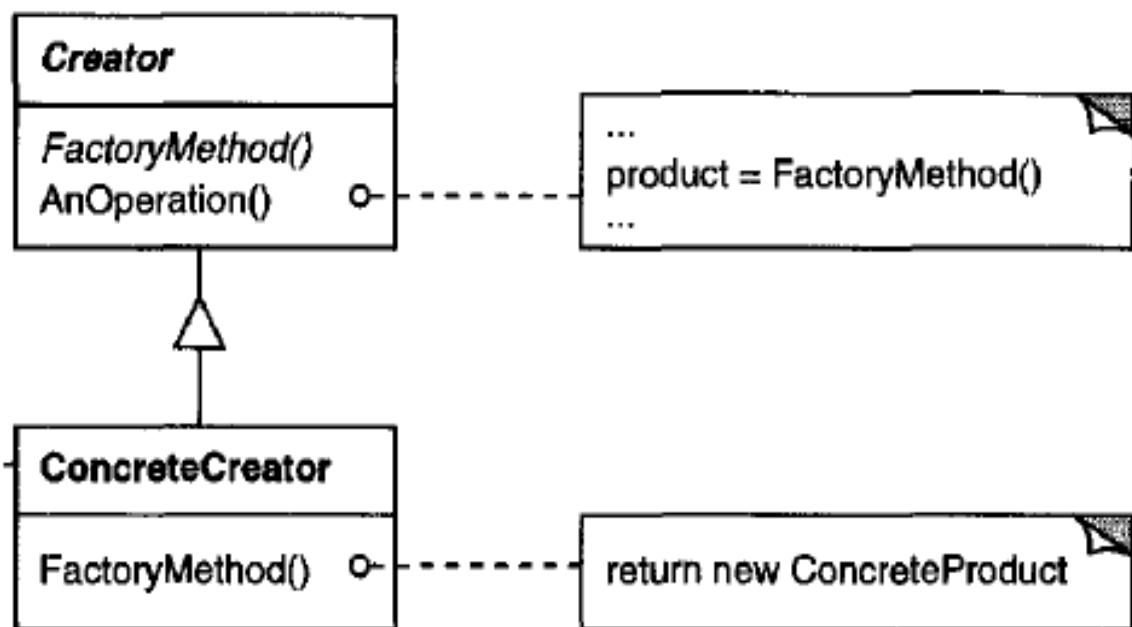
- Purpose/Intent
 - Define an interface for creating an object
 - Let subclasses decide which class to instantiate.
 - Let a class defer instantiation to subclasses
- Also known as
 - Virtual Constructor

Factory Method Structure

Defines the interface of objects the factory method creates



Declares the factory method, which returns an object of type Product.



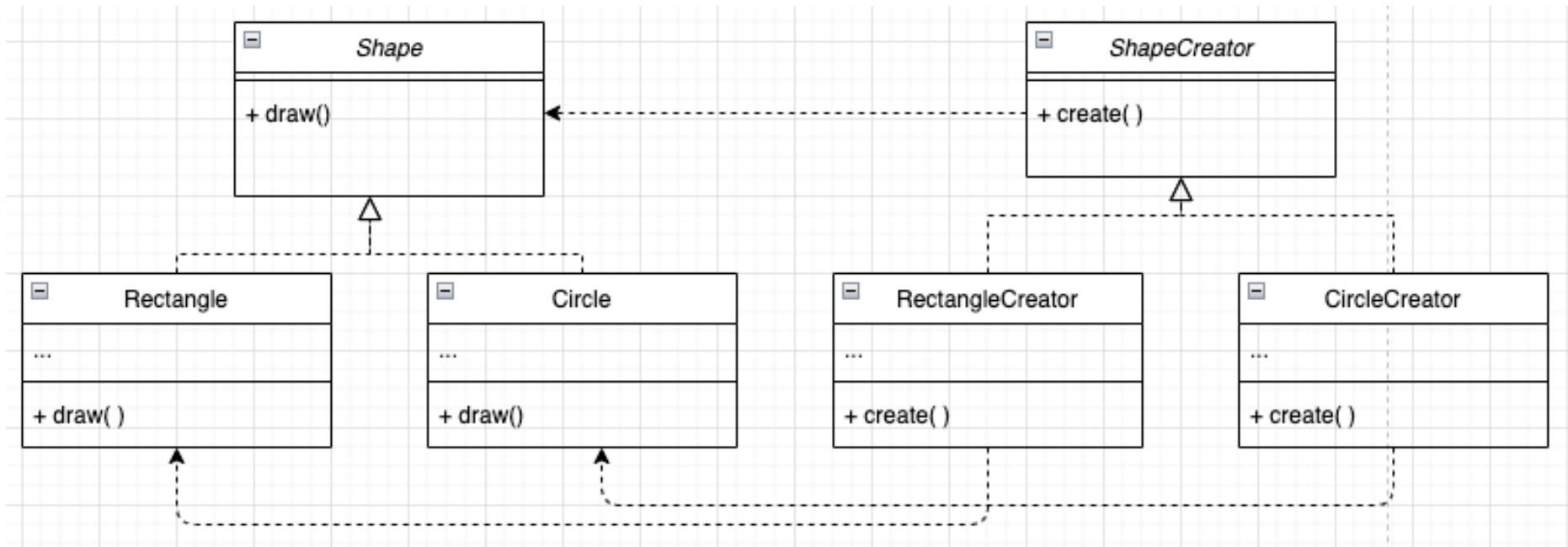
Implements the Product interface

Overrides the factory method to return an instance of the ConcreteProduct

Factory Method Participants

- **Product**
 - Defines the interface of objects the factory method creates
- **ConcreteProduct**
 - Implements the Product interface
- **Creator**
 - Declares the factory method, which returns an object of type Product.
- **ConcreteCreator**
 - Overrides the factory method to return an instance of the ConcreteProduct

Revisit the Motivated Example



Revisit the Motivated Example

```
public interface ShapeCreator {  
    1 usage 2 implementations  
    public Shape create();  
}
```

```
public class CircleCreator implements ShapeCreator {  
    1 usage  
    @Override  
    public Shape create() {  
        return new Circle();  
    }  
}
```

```
public class RectangleCreator implements ShapeCreator{  
    1 usage  
    @Override  
    public Shape create() {  
        return new Rectangle();  
    }  
}
```

Client Perspective

```
ShapeCreator creator = new CircleCreator();  
Shape shape = creator.create();  
shape.draw();
```

Factory Method Pattern

- Applicability
 - A class cannot anticipate the class objects it must create
 - A class wants its subclasses to specify the objects it creates
 - Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate
- Benefits
 - Flexibility: subclasses get a hook for providing an extension to an object; connects parallel class hierarchies
- Limitations
 - Can require subclassing just to get an implementation

Two varieties of Factory

- Variety 1: the Creator class is abstract
 - This requires subclasses to be made because there is no reasonable default value.
 - On plus side, this avoids the problem of dealing with instantiating unforeseeable classes
- Variety 2: the Creator class is concrete
 - Creator may also define a default implementation of the factory method that returns a default *ConcreteProduct* object
 - This provides reasonable default behaviors, and enables subclasses to override the default behaviors where required

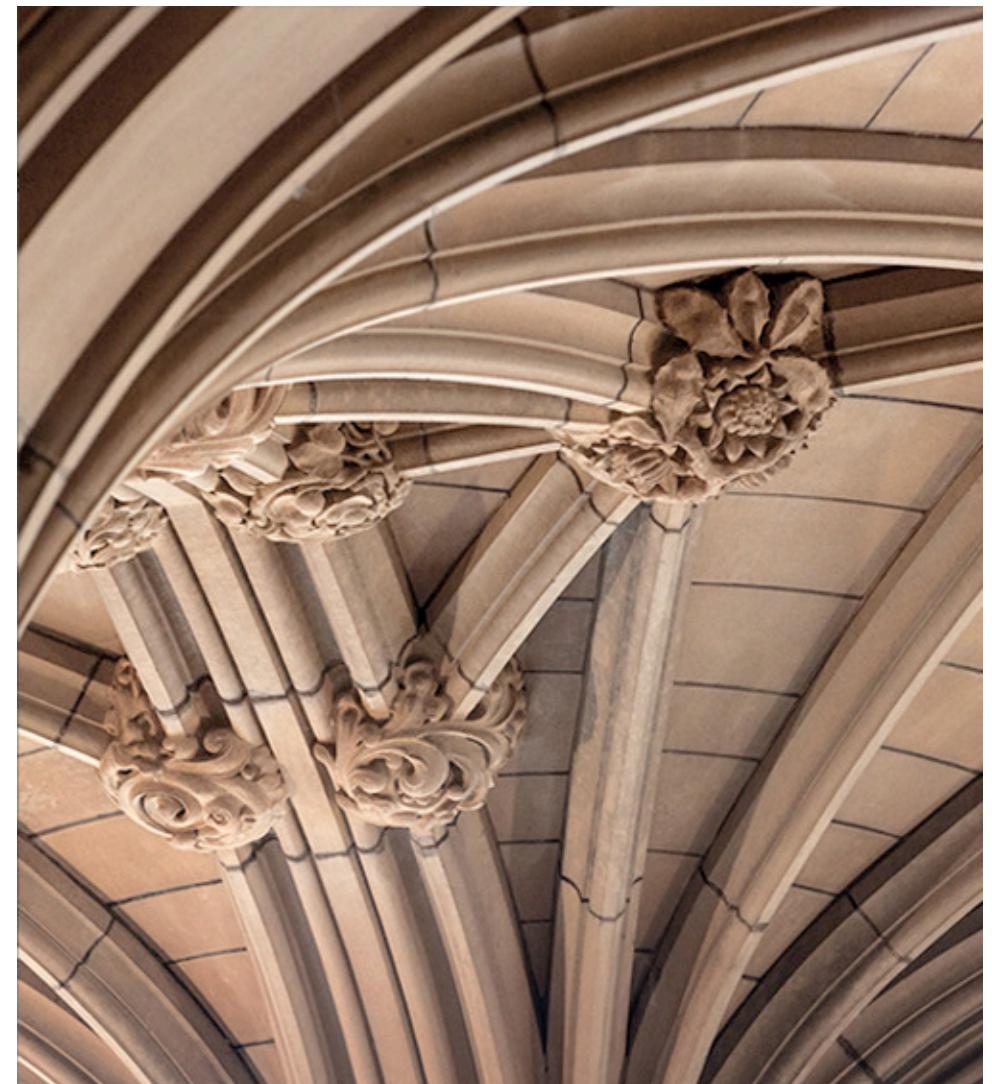
Factory Registry – Selecting a Factory Method Source

- In essence, a mapping from identifier to implementation
- Responsibility for choice of Factory Method concentrated in one location

```
public class FactoryRegistry {  
    private Map<Identifier, Factory> factories = new TreeMap<>();  
    public void registerFactory(Factory factory, Identifier identifier) {...}  
    public Factory getFactory(Identifier identifier) {...}  
}
```

Builder

Object Creational Pattern



Motivated Scenario

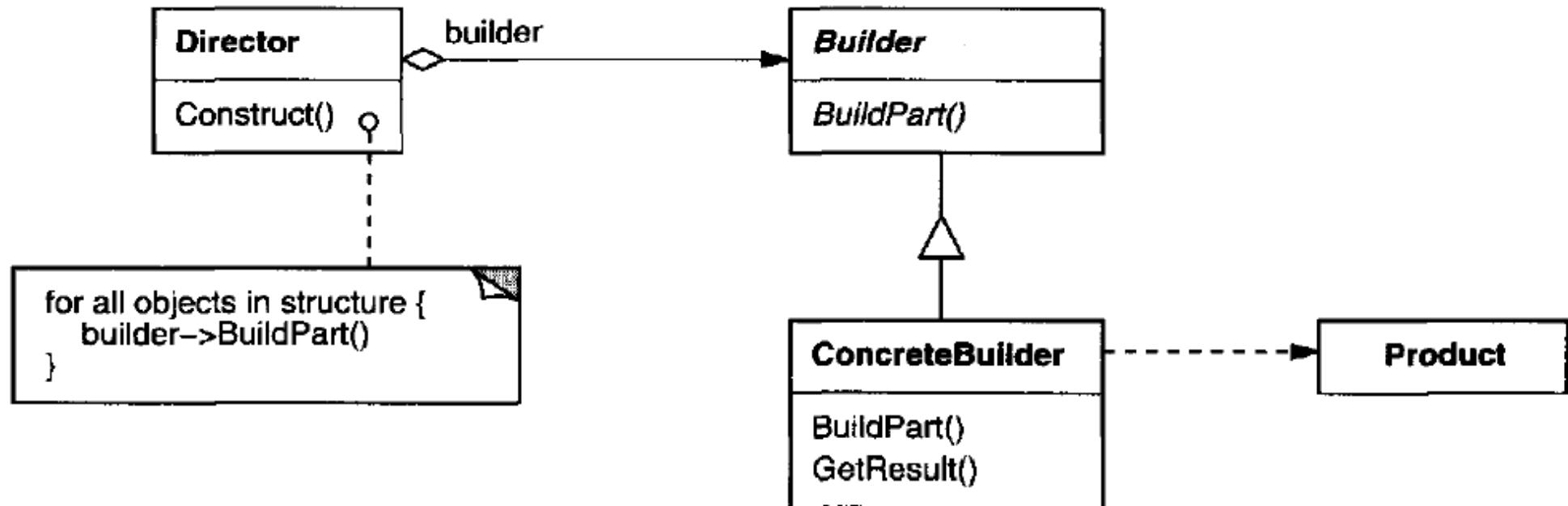
- Suppose the design team is going outside to have a coffee break.
 - John would like a CoffeeA that contains no milk and no sugar
 - Xi would like a CoffeeB that contains no milk but full sugar
 - Sue would like a CoffeeC that contains full milk and full sugar



Builder

- Purpose/Intent
 - Separate the construction of a complex object from its representation so that the same construction process can create different representations
 - You have a range of implementations that might expand, so must be flexible, or you want to create instances of complex objects

Builder -- Structure



Builder – Participants

- **Builder**
 - Specifies an abstract interface for creating parts of a Product object
- **ConcreteBuilder**
 - Constructs and assembles parts of the product by implementing the Builder interface
 - defines and keeps track of the representation it creates.
 - provides an interface (GetResult) for retrieving the product

Builder – Participants

- **Director**
 - Constructs an object using the Builder interface
- **Product**
 - Represents the complex object under construction.
 - ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled
 - Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

Revisit the Motivated Example

“abstract” builder

one concrete builder as an example



```
public interface CoffeeBuilder {  
    1 usage 3 implementations  
    public void pickupCoffeeBean();  
    1 usage 3 implementations  
    public void grindCoffeeBean();  
    1 usage 3 implementations  
    public void addWater();  
    1 usage 3 implementations  
    public void filterCoffee();  
    1 usage 3 implementations  
    public void addMilk();  
    1 usage 3 implementations  
    public void addSugar();  
}
```

```
public class CoffeeABuilder implements CoffeeBuilder{  
    1 usage  
    @Override  
    public void pickupCoffeeBean() {System.out.println("We pick up the proper Coffee bean for CoffeeA");}  
    1 usage  
    @Override  
    public void grindCoffeeBean() {System.out.println("CoffeeA bean needs to be grind for 1 minute");}  
    1 usage  
    @Override  
    public void filterCoffee() {System.out.println("We use filterA for CoffeeA");}  
    1 usage  
    @Override  
    public void addWater() {System.out.println("CoffeeA needs 100 C water");}  
    1 usage  
    @Override  
    public void addMilk() {System.out.println("No milk added for CoffeeA");}  
    1 usage  
    @Override  
    public void addSugar() {System.out.println("No sugar added for CoffeeA");}  
}
```

Revisit the Motivated Example



director

```
public class CoffeeDirector {  
    7 usages  
    private CoffeeBuilder cb;  
  
    1 usage  
    public CoffeeDirector(CoffeeBuilder cb) {  
        this.cb = cb;  
    }  
  
    1 usage  
    public void construct() {  
        cb.pickupCoffeeBean();  
        cb.grindCoffeeBean();  
        cb.filterCoffee();  
        cb.addWater();  
        cb.addMilk();  
        cb.addSugar();  
    }  
}
```

client perspective

```
CoffeeBuilder b = new CoffeeABuilder();  
CoffeeDirector director = new CoffeeDirector(b);  
director.construct();
```

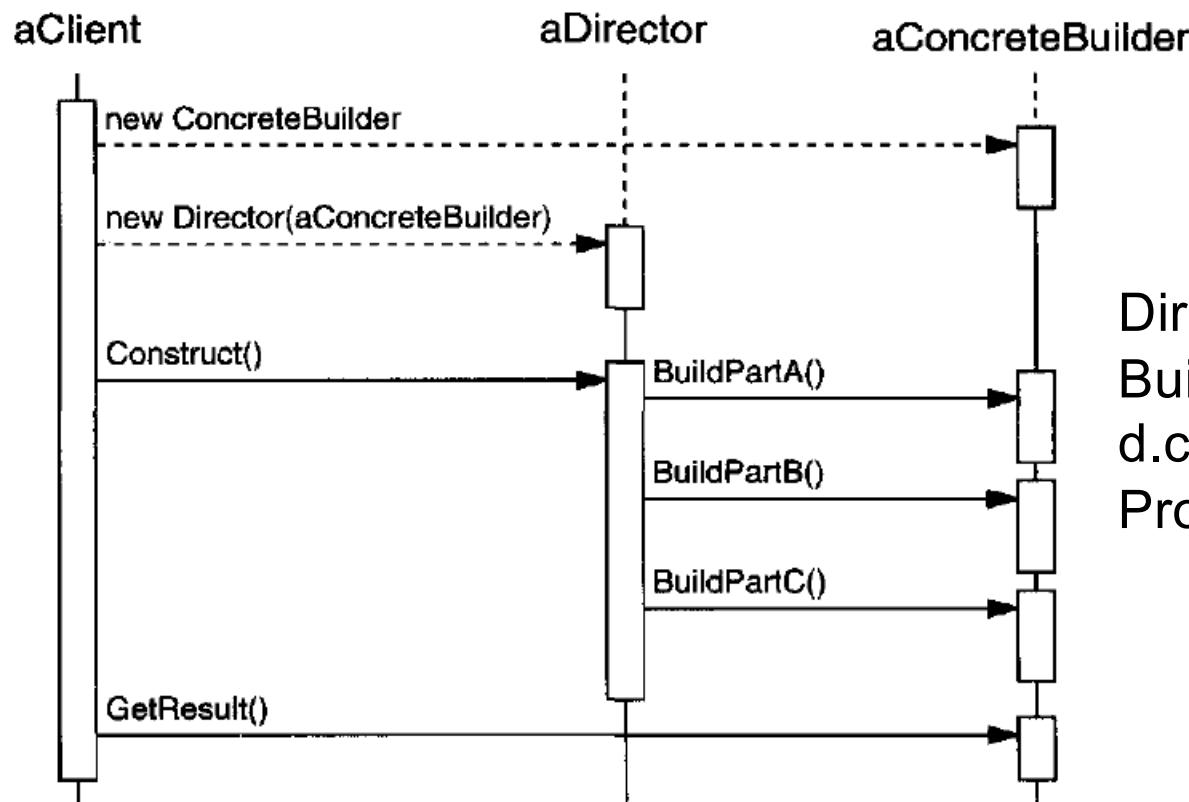
result:

```
We pick up the proper Coffee bean for CoffeeA  
CoffeeA bean needs to be grind for 1 minute  
We use filterA for CoffeeA  
CoffeeA needs 100 C water  
No milk added for CoffeeA  
No sugar added for CoffeeA
```

Builder

- Applicability
 - The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
 - The construction process must allow different representations for the object that's constructed
- Benefits
 - Gives flexibility that can be extended by implementing new subclasses of the Builder, and isolates code of construction from implementation
- Limitations
 - Not completely generic, less useful in situations where variety of implementations is not high

Builder – Collaboration



```
Director d = new Director();
Builder b = new ConcreteBuilder1();
d.construct(b);
Product p = b.GetResult();
```

Builder – Consequences (1)

- Varying product's internal representation
 - Because the product is constructed through an abstract interface, all you have to do to change the product's internal representation is define a new kind of builder
- Isolation of code construction and representation
 - Each ConcreteBuilder contains all the code to create and assemble a particular kind of product.
 - Different Directors can reuse it to build Product variants from the same set of parts

Builder – Consequences (2)

- Finer control over the construction process
 - The builder pattern constructs the product step by step under the director's control
 - Only when the product finished does the director retrieve it from the builder
 - The builder interface reflects the process of constructing the product more than other creational patterns

Task for Week 5

- Submit weekly exercise on canvas before 23.59pm Saturday
- Prepare questions and ask during tutorial this week

What are we going to learn next week?

- Behavioral Design Patterns
 - Strategy Pattern
 - State Pattern

References

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition). Prentice Hall PTR, Upper Saddle River, NJ, USA.

Software Design and Construction 1

SOFT2201 / COMP9201

Behavioural Design Patterns

Dr. Xi Wu

School of Computer Science



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Announcement

- Topics in the following weeks

Week	Contents		Week
7	Design Pattern: Adapter & Observer	Code Review	8
10	Design Pattern: Prototype & Memento	Testing	11
12	Design Pattern: Singleton, Decorator and Facade		
13	Unit Review		

- Slides with demo example will be updated to Canvas after each lecture

Agenda

- Behavioral Design Patterns
 - Strategy
 - State

Behavioural Design Patterns



THE UNIVERSITY OF
SYDNEY



Behavioural Patterns

- Concerned with algorithms and the assignment of responsibilities between objects
- Describe patterns of objects and class, and communication between them
- Simplify complex control flow that's difficult to follow at run-time
 - Concentrate on the ways objects are interconnected
- **Behavioural Class Patterns (SOFT3202)**
 - Use inheritance to distribute behavior between classes (algorithms and computation)
- **Behavioural Object Patterns**
 - Use object composition, rather than inheritance. E.g., describing how group of peer objects cooperate to perform a task that no single object can carry out by itself
 - Question: how peer objects know about each other?

Design Patterns – Classification based on purpose

Scope	Creational	Structural	Behavioural
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object)	Chain of Responsibility
		Bridge	Command
		Composite	Iterator
		Decorator	Mediator
		Façade	Memento
		Flyweight	Observer
		Proxy	State
			Strategy
			Visitor

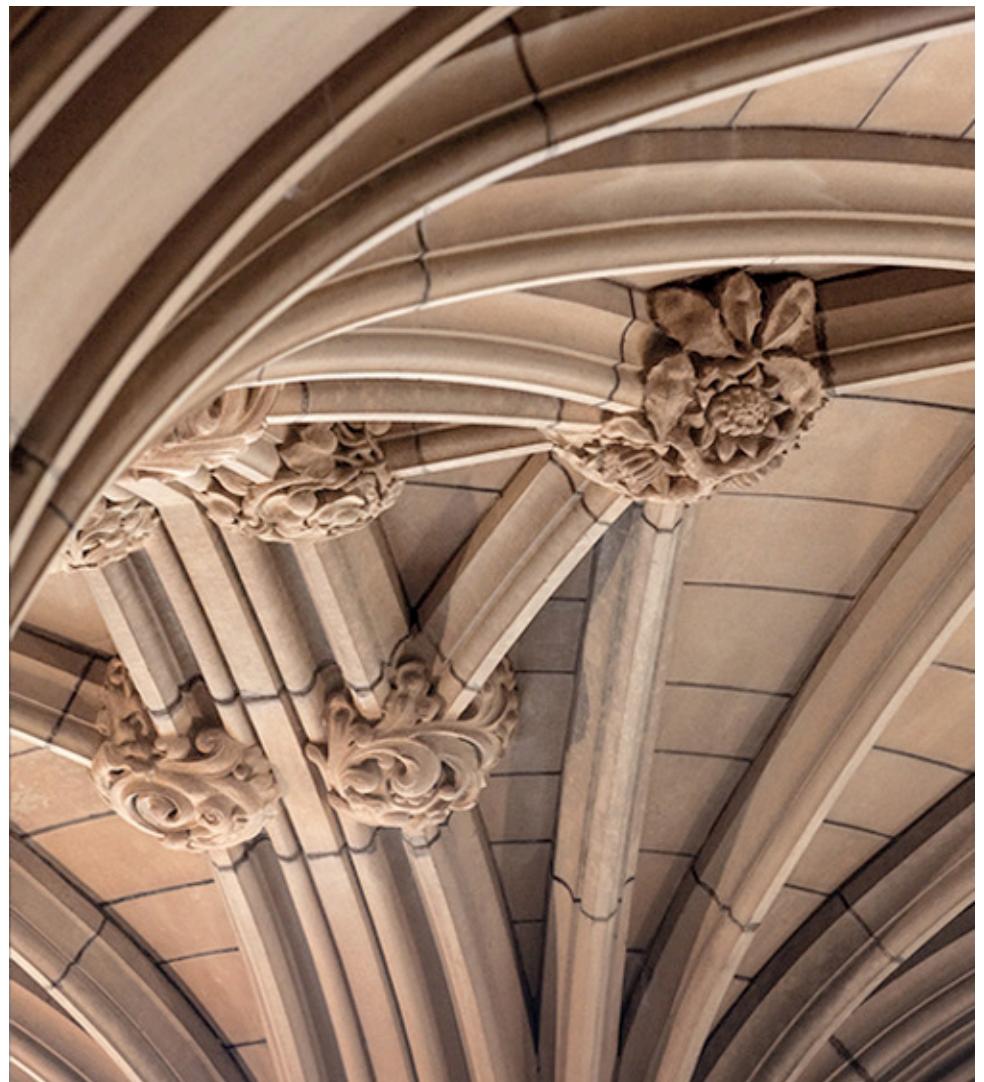
Behavioural Patterns (GoF)

Pattern Name	Description
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable (let algorithm vary independently from clients that use it)
Observer	Define a one-to-many dependency between objects so that when one object changes, all its dependents are notified and updated automatically
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
State	Allow an object to alter its behaviour when its internal state changes. The object will appear to change to its class
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates
Other patterns	Interpreter, Iterator, Mediator, Chain of Responsibility, Template Method

Strategy Design Pattern

Object Behavioural Pattern

Algorithm design through encapsulation



Motivated Scenario

- Suppose you visit a store regularly to buy necessary things
 - Get the normal total price during the weekday
 - Get a discount on the total price during the weekend
 - Get a cashback on the total price during mid-year session



```
public class SalePricing {  
    1 usage  
    public double getTotal (int quantity, double price) {  
        return price * quantity;  
    }  
}
```

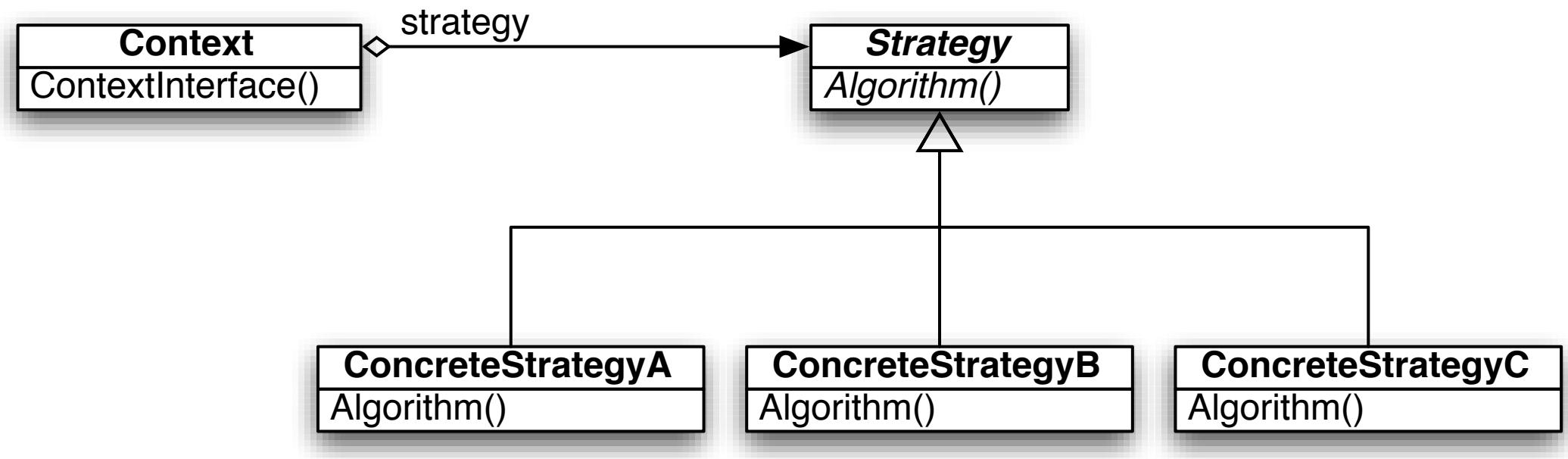
```
public class SalePricingWithDiscount {  
    1 usage  
    private double discount = 0.0;  
  
    public double getTotal (int quantity, double price) {  
        return price * quantity * discount;  
    }  
}
```

```
public class SalePricingWithCashBack {  
    1 usage  
    private double discount = 0.0;  
    1 usage  
    private double threshold = 0.0;  
  
    public double getTotal (int quantity, double price) {  
        double total = price * quantity;  
        if (total >= threshold) return total-discount;  
        else return total;  
    }  
}
```

Strategy Design Pattern

- **Purpose/Intent**
 - Define a family of algorithms, encapsulate each one, and make them interchangeable
 - Let the algorithm vary independently from clients that use it
 - Design for varying but related algorithms that are suitable for different contexts
 - Ability to change these algorithms
- **Known as**
 - Policy

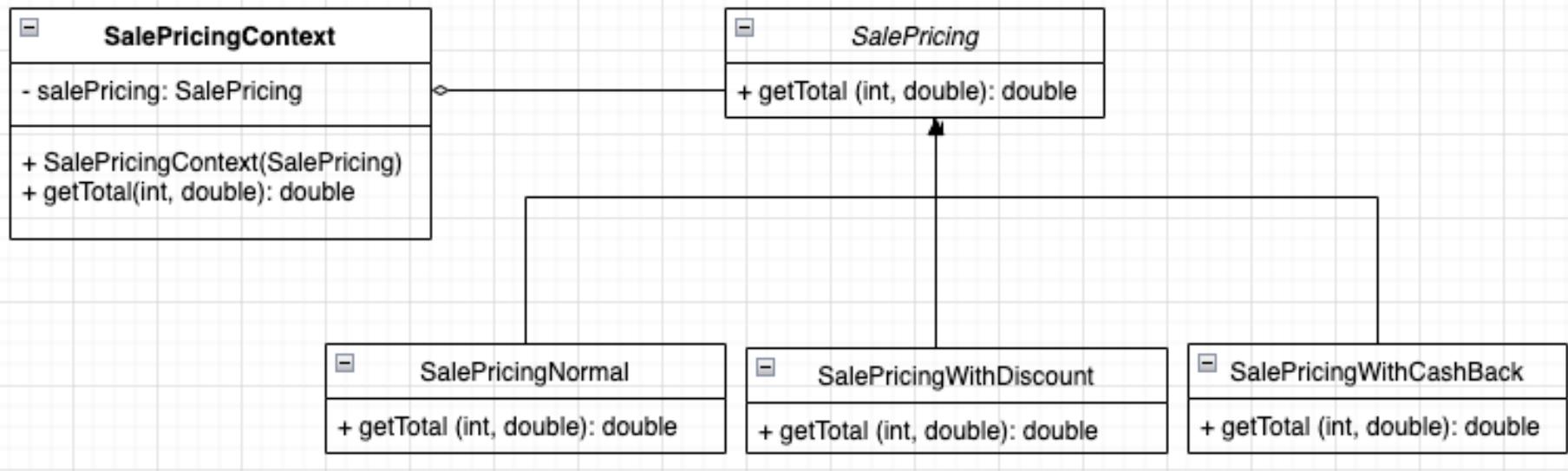
Strategy – Structure



Strategy – Participants

- **Strategy**
 - Declares an interface common to all supported algorithms
 - Used by context to call the algorithm defined by **ConcereteStrategy**
- **ConcereteStrategy**
 - Implements the algorithm using the **Strategy** interface
- **Context**
 - Is configured with a **ConcereteStrategy** object
 - Maintains a reference to a **Strategy** object
 - May define an interface that lets **Strategy** access its data

Revisit the Motivated Example



Client's perspective:

```
SalePricing salePricing = new SalePricingNormal();
SalePricingContext context = new SalePricingContext(salePricing);
double total = context.getTotal( quantity: 6, price: 4);
System.out.println("The total money you need to pay is: " + total);
```

Revisit the Motivated Example

```
public interface SalePricing {  
    1 usage 3 implementations  
    public double getTotal (int quantity, double price);  
}
```

```
public class SalePricingNormal implements SalePricing{  
    1 usage  
    public double getTotal (int quantity, double price) {  
        return price * quantity;  
    }  
}
```

```
public class SalePricingContext {  
    2 usages  
    private SalePricing salePricing;  
    public SalePricingContext (SalePricing salePricing){  
        this.salePricing = salePricing;  
    }  
    public double getTotal(int quantity, double price){  
        return salePricing.getTotal(quantity,price);  
    }  
}
```

```
public class SalePricingWithDiscount implements SalePricing{  
    1 usage  
    private double discount = 0.0;  
  
    1 usage  
    public double getTotal (int quantity, double price) {  
        return price * quantity * discount;  
    }  
}
```

```
public class SalePricingWithCashBack implements SalePricing {  
    1 usage  
    private double discount = 0.0;  
    1 usage  
    private double threshold = 0.0;  
  
    1 usage  
    public double getTotal (int quantity, double price) {  
        double total = price * quantity;  
        if (total >= threshold) return total-discount;  
        else return total;  
    }  
}
```

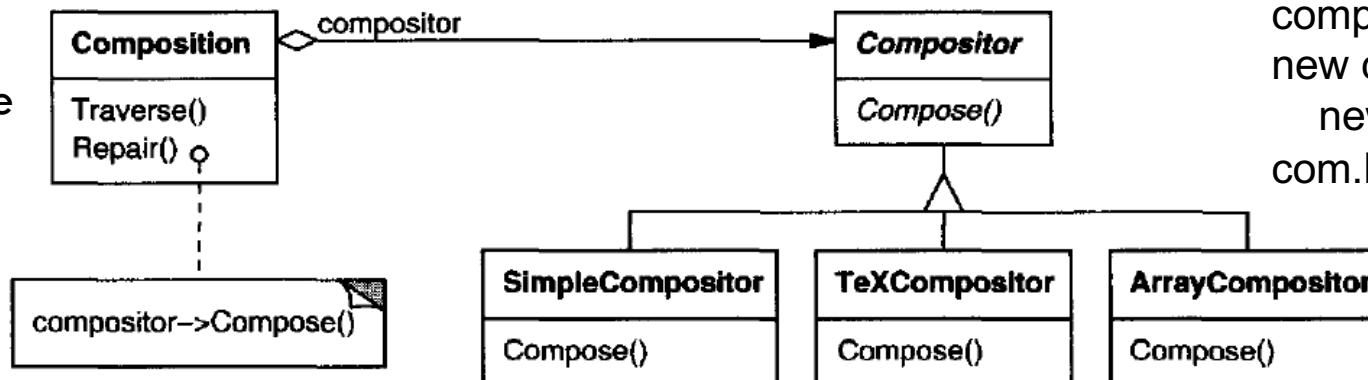
Strategy – Applicability

- Many related classes differ only in their behavior
- You need different variant of an algorithm
- An algorithm uses data that should be hidden from its clients
- A class defines many behaviors that appear as multiple statements in its operations

One more Example (Text Viewer)

- Many algorithms for breaking a stream of text into lines

Maintain &
update the line
breaks of text



Composition perspective:

```
private Compositor com;
public composition (Compositor c) {com = c;}
public void Repair () { com.compose();}
```

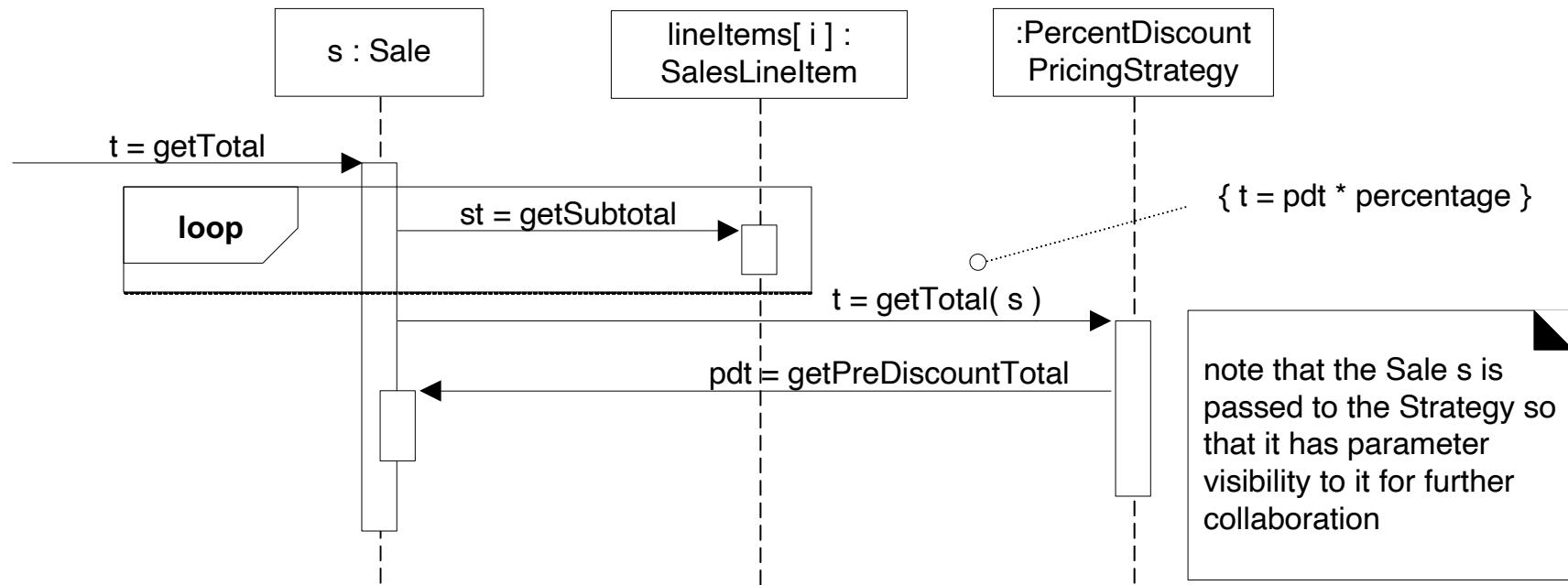
Client perspective:
composition com =
new composition(
 new SimpleCompositor());
com.Repair();

Different line breaking algorithms (strategies)

Strategy – Collaborations

- Strategy and Context interact to implement the chosen algorithm
 - A context may pass all data required by the algorithm to the Strategy
 - The context can pass itself as an argument to Strategy operations
- A context forwards requests from its clients to its strategy
 - Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively

Strategy Collaboration – POS Example



Strategy – Consequences

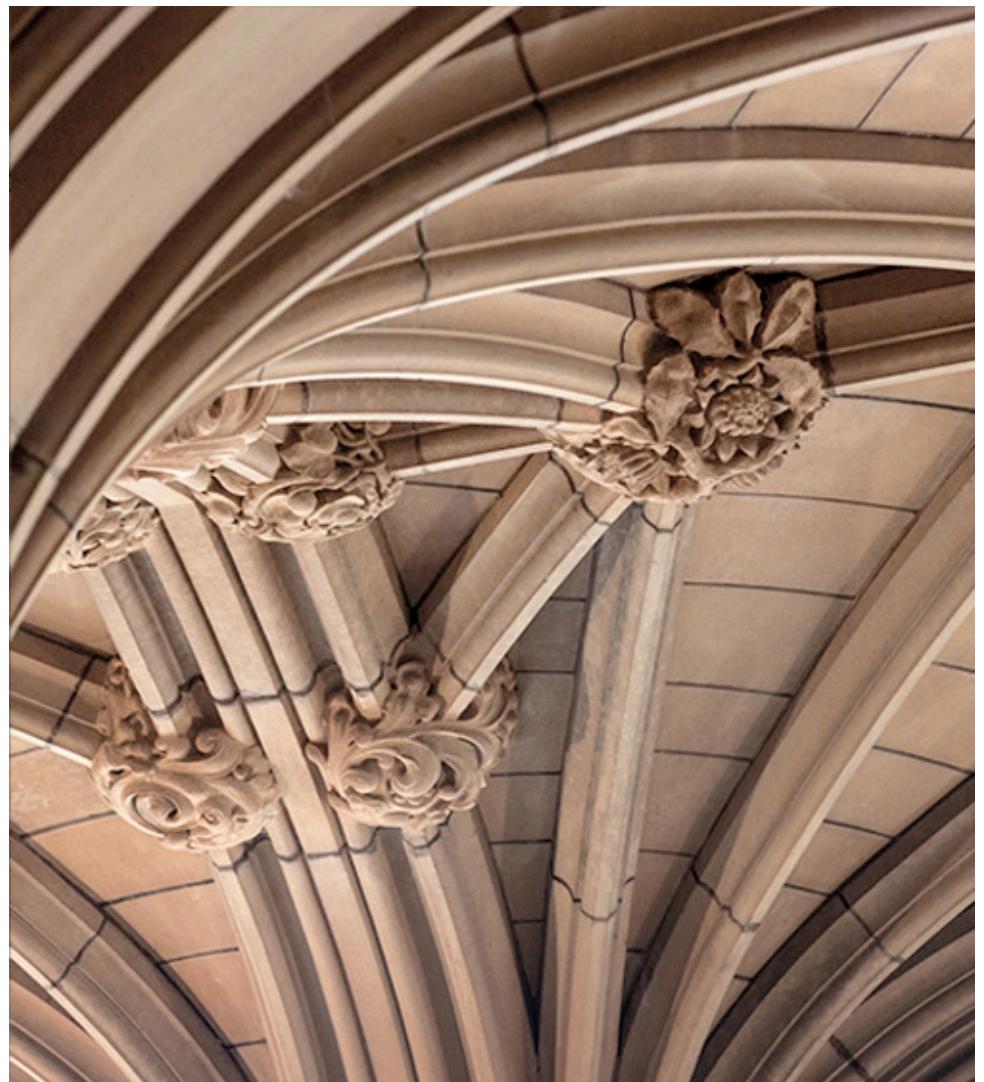
- Benefits
 - Family of related algorithms (behaviors) for context to reuse
 - Alternative to sub-classing
 - Strategies eliminate conditional statements
 - Provide choice of different implementation of the same behavior
- Drawbacks
 - Clients must be aware of different strategies
 - Communicate overhead between Strategy and Context
 - Increased number of objects in an application

State Design Pattern

Object Behavioural Pattern

Taking control of objects from the inside

A structured way to control the internal behaviour
of an object



Motivated Scenario

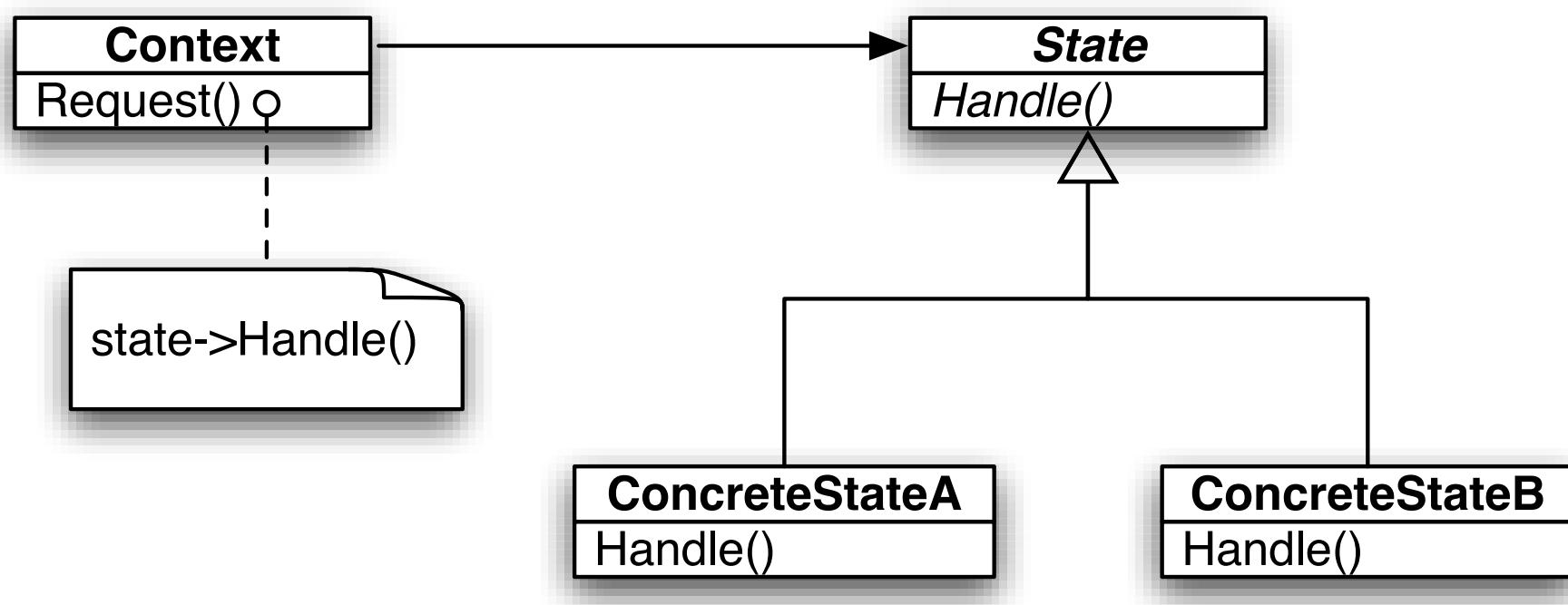
- Suppose you would like to connect to a TCP network and the TCP connection would respond based on its current state
 - Established
 - Listening
 - Closed



State Design Pattern

- **Purpose/Intent**
 - Allow an object to change its behaviour when its internal state changes
 - The object will appear to change its class when the state changes
 - We can use subtypes of classes with different functionality to represent different states, such as for a TCP connection with Established, Listening, Closed as states
- **Known as**
 - Objects for States

State Pattern – Structure

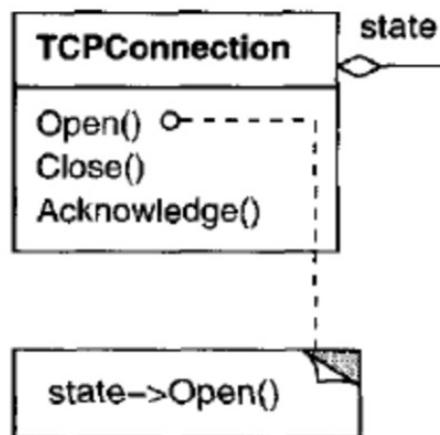


State Pattern – Participants

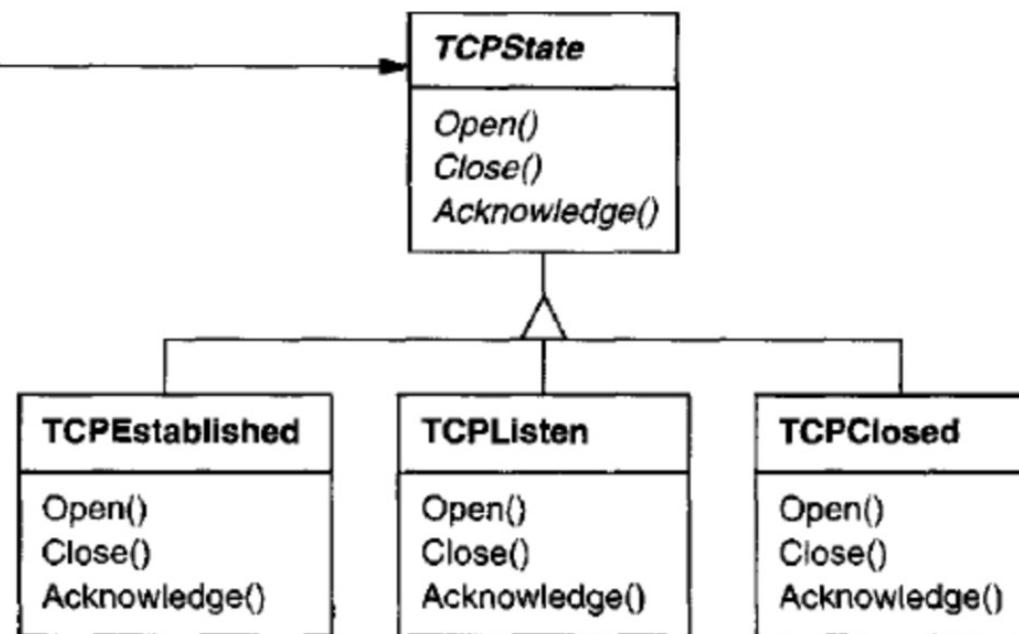
- **Context**
 - Defines the interface of interest to clients
 - Maintains an instance of a **ConcreteState** subclass that defines the current state
- **State**
 - Defines an interface for encapsulating the behaviour associated with a certain state of the Context
- **ConcreteState subclasses**
 - Each subclass implements a behaviour associated with a state of the Context

Revisited the Motivating Scenario

Context



State



ConcreteState

In Class Activity – Code Example

```
1 interface State {  
2     void writeName(StateContext context, String name);  
3 }  
4  
5 class LowerCaseState implements State {  
6     @Override  
7     public void writeName(StateContext context, String name) {  
8         System.out.println(name.toLowerCase());  
9         context.setState(new MultipleUpperCaseState());  
10    }  
11 }  
12  
13 class MultipleUpperCaseState implements State {  
14     /* Counter local to this state */  
15     private int count = 0;  
16  
17     @Override  
18     public void writeName(StateContext context, String name) {  
19         System.out.println(name.toUpperCase());  
20         /* Change state after StateMultipleUpperCase's writeName() gets invoked twice */  
21         if(++count > 1) {  
22             context.setState(new LowerCaseState());  
23         }  
24     }  
25 }
```

In Class Activity – Code Example

```
27 class StateContext {
28     private State state;
29
30     public StateContext() {
31         state = new LowerCaseState();
32     }
33
34     /**
35      * Set the current state.
36      * Normally only called by classes implementing the State interface.
37      * @param newState the new state of this context
38      */
39     void setState(State newState) {
40         state = newState;
41     }
42
43     public void writeName(String name) {
44         state.writeName(this, name);
45     }
46 }
```

In Class Activity – Code Example

```
48 public class StateDemo {  
49     public static void main(String[] args) {  
50         var context = new StateContext();  
51  
52         context.writeName("Monday");  
53         context.writeName("Tuesday");  
54         context.writeName("Wednesday");  
55         context.writeName("Thursday");  
56         context.writeName("Friday");  
57         context.writeName("Saturday");  
58         context.writeName("Sunday");  
59     }  
60 }
```

Question: What is the output of this application?

State Design Pattern

- **Applicability**
 - Any time you need to change behaviours dynamically, i.e., the state of an object drives its behavior and change its behavior dynamically at run-time
 - There are multi-part checks of an object's state to determine its behaviour, i.e., operations have large, multipart conditional statements that depend on the object's state
- **Benefits**
 - Removes case or if/else statements depending on state, and replaces them with function calls; makes the state transitions explicit; permits states to be shared
- **Limitations**
 - Does require that all the states have to have their own objects

State Pattern – Collaborations

- Context delegates state – specific requests to the current ConcreteState object
- A context may pass itself as an argument to the State object handling the request, so the State object access the context if necessary
- Context is the primary interface for clients
 - Clients can configure a context with State objects, so its clients don't have to deal with the State objects directly
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances

State Pattern – Consequences

- Localizes state-specific behaviour for different states**
 - Using data values and context operations make code maintenance difficult
 - State distribution across different subclasses useful when there are many states
 - Better code structure for state-specific code (better than monolithic)
- It makes state transition explicit**
 - State transitions as variable assignments
 - State objects can protect the context from inconsistent state
- State objects can be shared**
 - When the state they represent is encoded entirely in their type

State Pattern – Implementation (1)

- Defining the state transitions
 - Let the state subclasses specify their successor state to make the transition (decentralized)
 - Achieves flexibility – easy to modify and extend the logic
 - Introduces implementation dependencies between subclasses
- Table-based state transitions
 - Look-up table that maps every possible input to a succeeding state
 - Easy to modify (transition data not the program code) but:
 - Less efficient than a functional call
 - Harder to understand the logic (transition criteria is less explicit)
 - Difficult to add actions to accompany the state transitions

State Pattern – Implementation (2)

- When to create and destroy state objects?
 - Pre-create them and never destroy them
 - Useful for frequent state changes (save costs of re-creating states)
 - Context must keep reference to all states
 - Only when they are needed and destroyed them thereafter
 - States are not known at run-time and context change states frequently

References

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
1995. *Design Patterns: Elements of Reusable Object-Oriented Software*.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Task for Week 6

- Submit weekly exercise on canvas before 23.59pm Saturday
- Well organize time for assignment 2
 - JSON configuration text file (tutorial 3)
 - JavaFX and GUI (tutorial 4)

What are we going to learn next week?

- Structural Design Pattern
 - Adapter
- Behavioral Design Pattern
 - Observer

Software Design and Construction 1

SOFT2201 / COMP9201

Adapter and Observer

Dr. Xi Wu

School of Computer Science



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

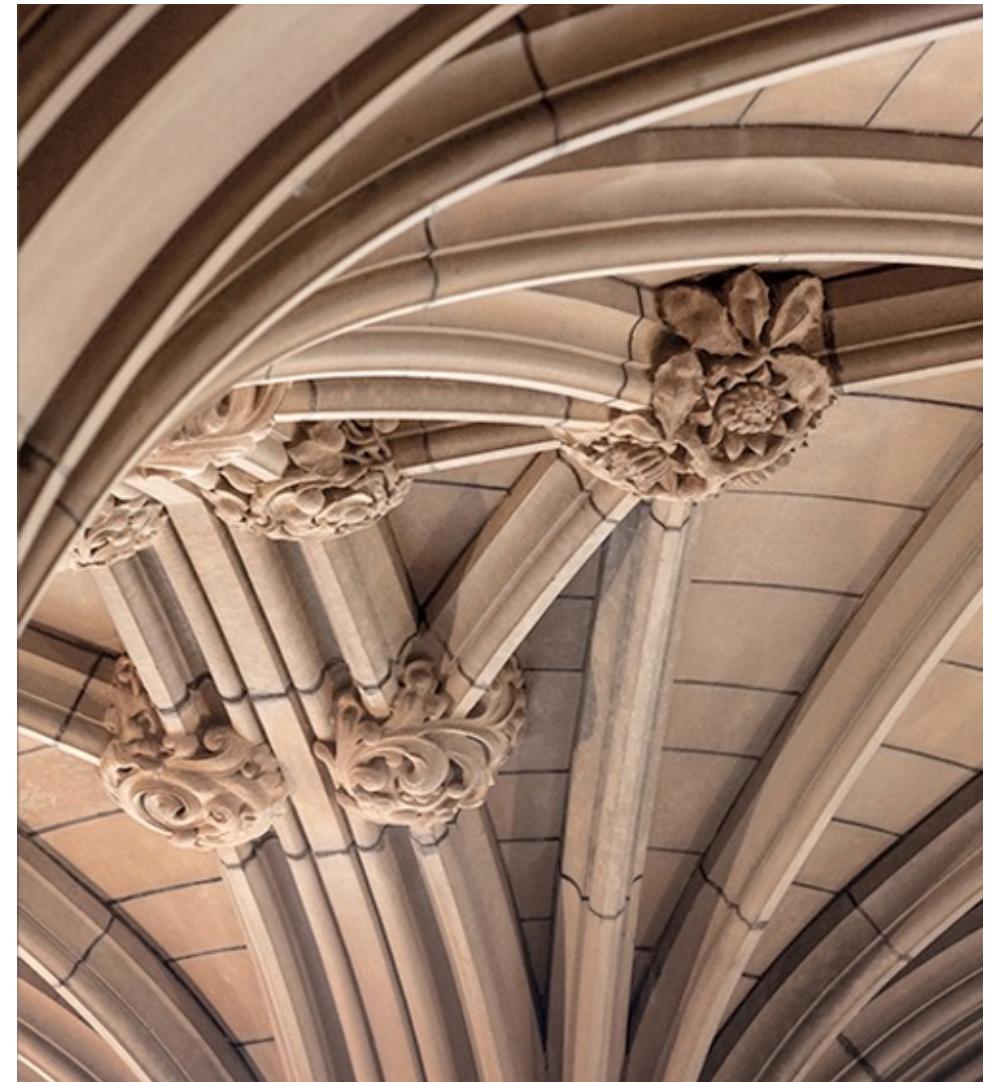
Agenda

- Structural Design Pattern
 - Adapter
- Behavioural Design Pattern
 - Observer

Structural Design Patterns



THE UNIVERSITY OF
SYDNEY



Structural Design Patterns

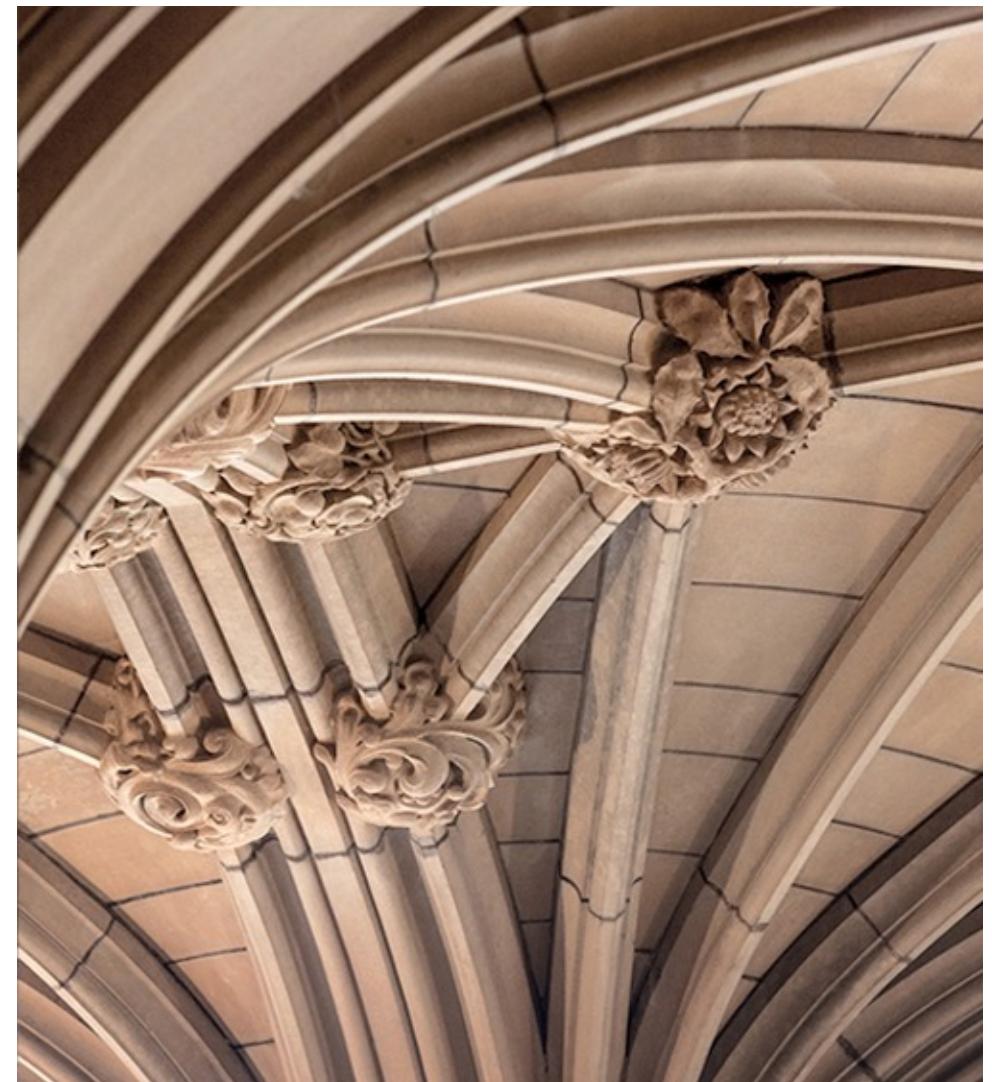
- How classes and objects are composed to form larger structures
- Structural *class* patterns use inheritance to compose interfaces or implementations
- Structural *object* patterns describe ways to compose objects to realise new functionality
 - The flexibility of object composition comes from the ability to change the composition at run-time

Structural Patterns (GoF)

Pattern Name	Description
Adapter	Allow classes of incompatible interfaces to work together. Convert the interface of a class into another interface that clients expect.
Façade	Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that makes the subsystem easier to use.
Decorator	Attach additional responsibilities to an object dynamically (flexible alternative to subclassing for extending functionality)
Composite	Compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently
Proxy	Provide a placeholder for another object to control access to it

Adapter Pattern

Class, Object Structural



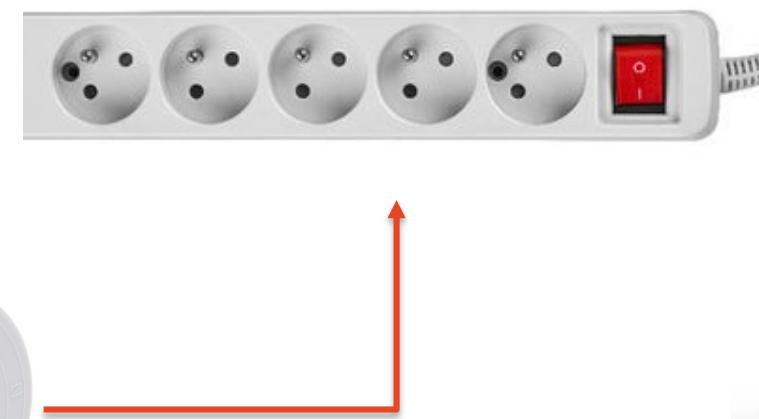
Motivated Scenario

- Suppose you travel to Europe countries with your laptop you have bought in Australia.

Power Charger for Computer



European Power Strip



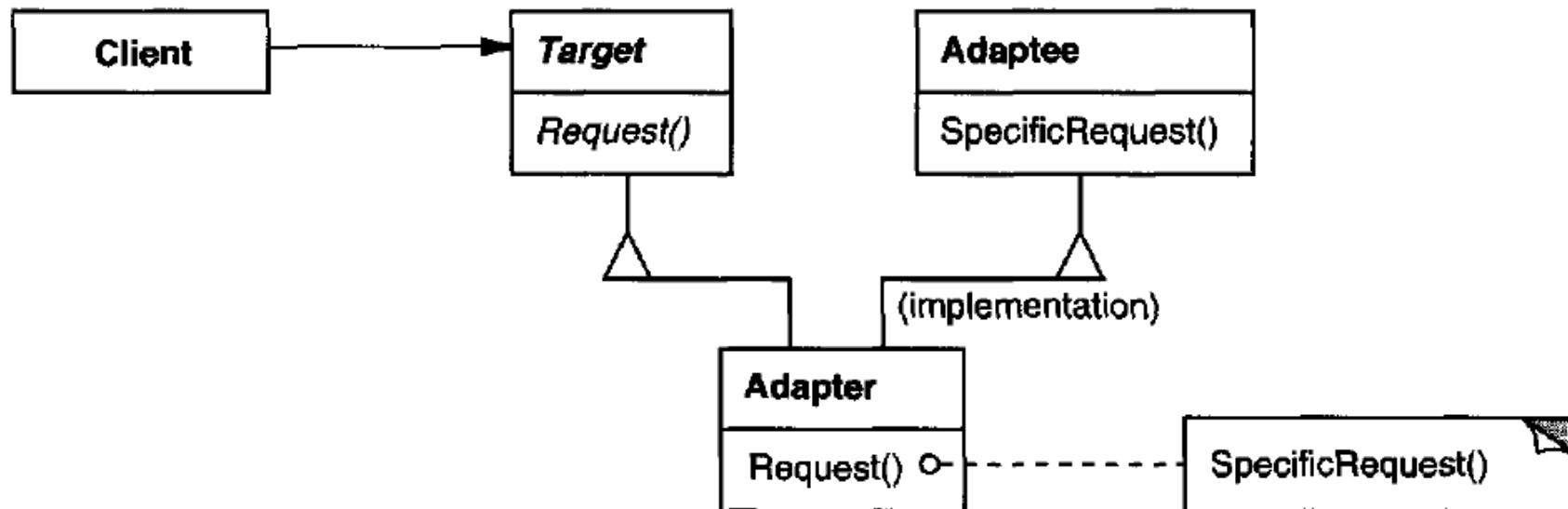
Adapter



Adapter

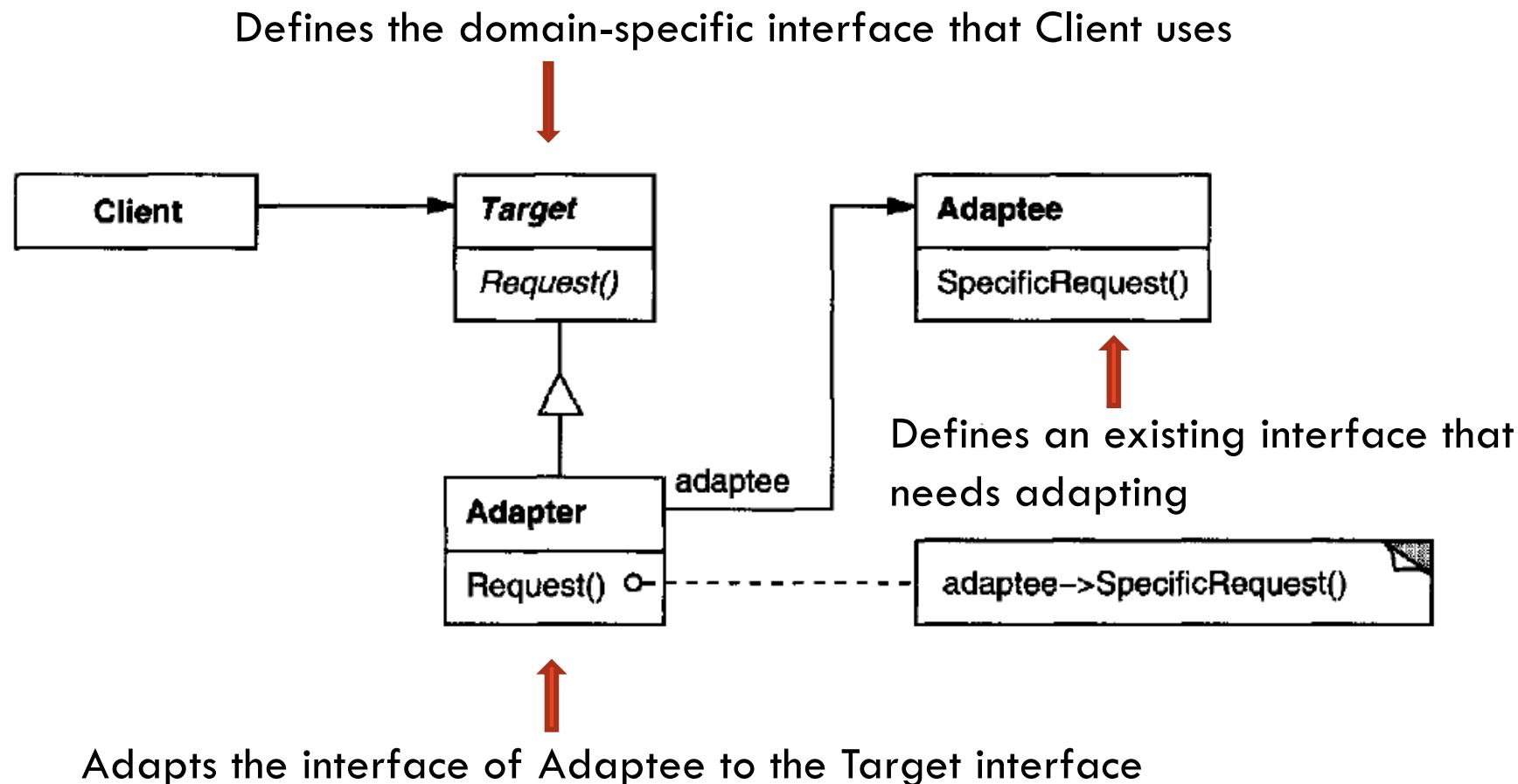
- Intent
 - Convert the interface of a class into another interface that clients expect.
 - Lets classes work together that couldn't otherwise because of incompatible interfaces.
 - Sometimes existing code that has the functionality we want doesn't have the right interface we want to use
- Known as
 - Wrapper

Class Adapter – Structure



- Request multiple inheritance to adapt the Adaptee to Target, **supported by C++**

Object Adapter – Structure



Adapter – Participants

- **Target**
 - Defines the domain-specific interface that Client uses
- **Client**
 - Collaborates with objects conforming to the Target interface.
- **Adaptee**
 - Defines an existing interface that needs adapting.
- **Adapter**
 - Adapts the interface of Adaptee to the Target interface

Adapter

- Applicability
 - To use an existing class with an interface does not match the one you need
 - You want to create a reusable class that cooperates with unrelated or unforeseen classes, i.e., classes that don't necessarily have compatible interfaces
 - (**Object adapter only**) Adapt an existing interface, which has several existing implementations.
- Benefits
 - Code reuse
- Collaborations
 - Clients call operations on an Adapter instance. In turn, the Adapter calls Adaptee operations that carry out the request

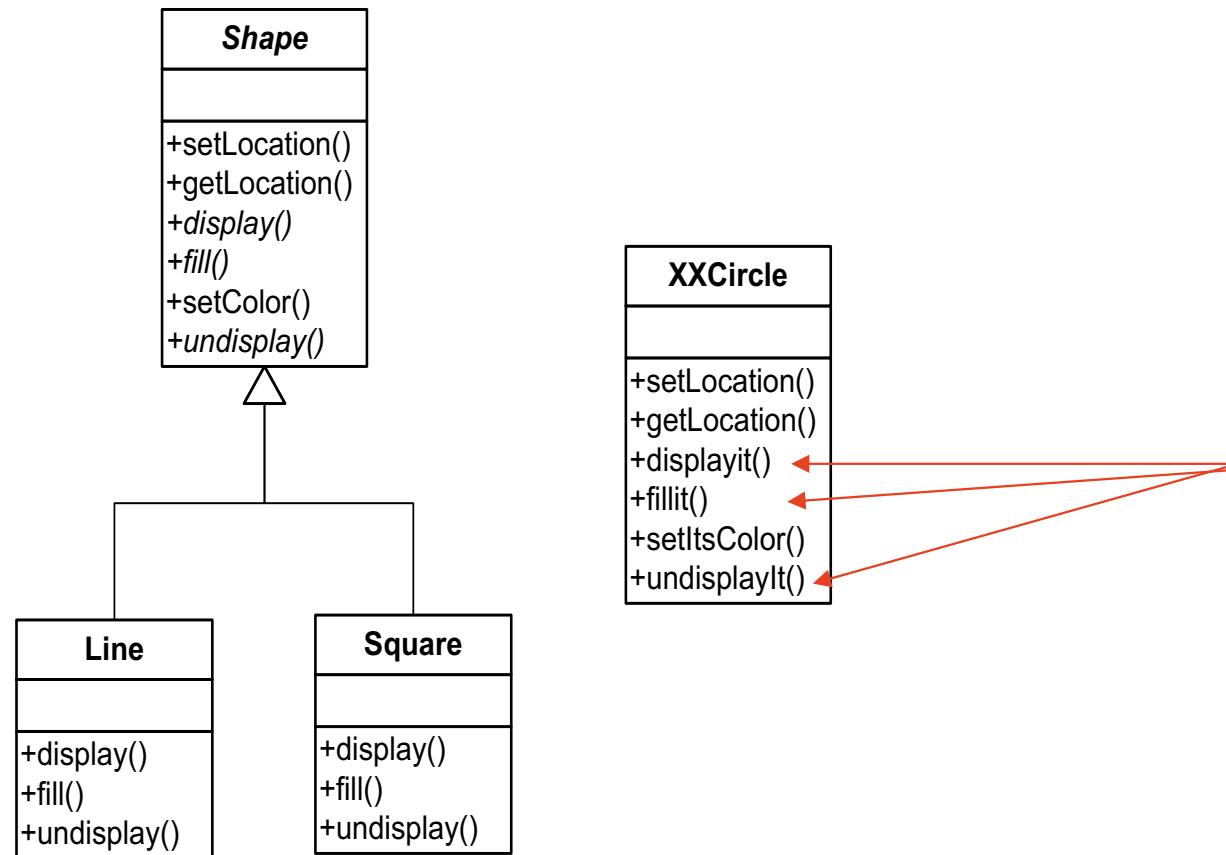
Class Adapter – Consequences

- When we want to adapt a class and all its subclasses, a class adapter won't work
 - It adapts Adaptee to Target by committing to a concrete Adaptee class
- Lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee

Object Adapter – Consequences

- Lets a single Adapter work with many Adaptees – i.e., the Adaptee itself and all of its subclasses (if any).
- Makes it harder to override Adaptee behavior. It will require sub-classing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself

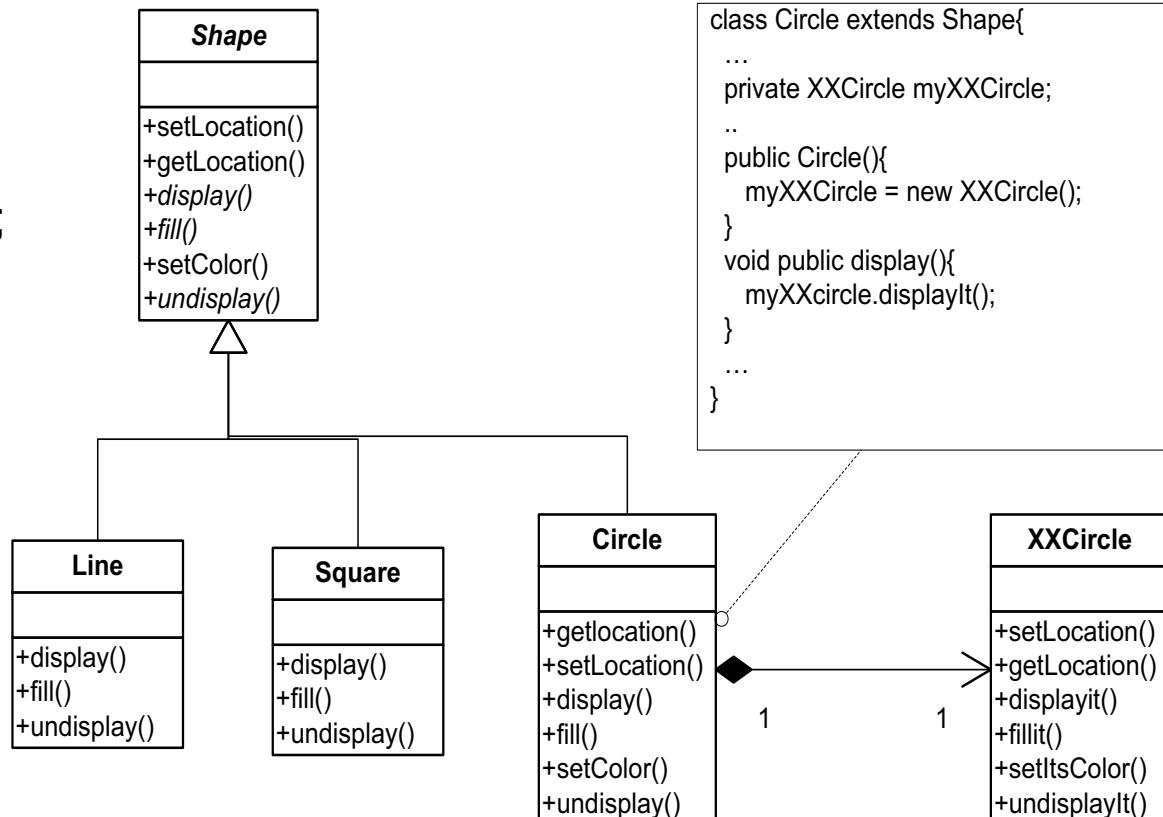
Adapter Example -- Problem



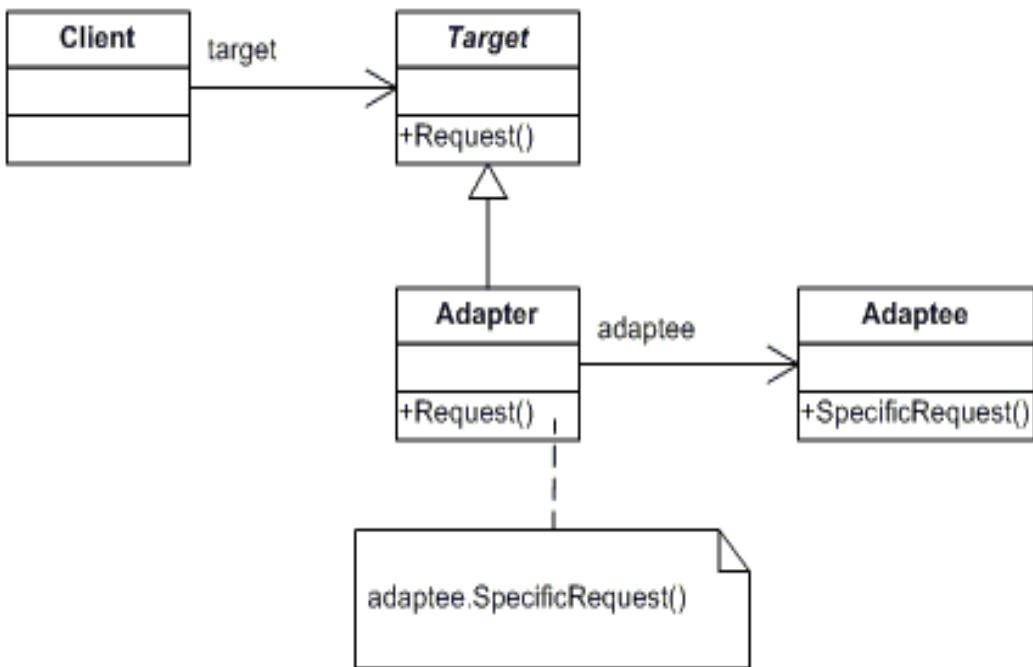
Adapter Example -- Solution

What should a client do?

```
Shape shape = new Circle();
shape.display();
```

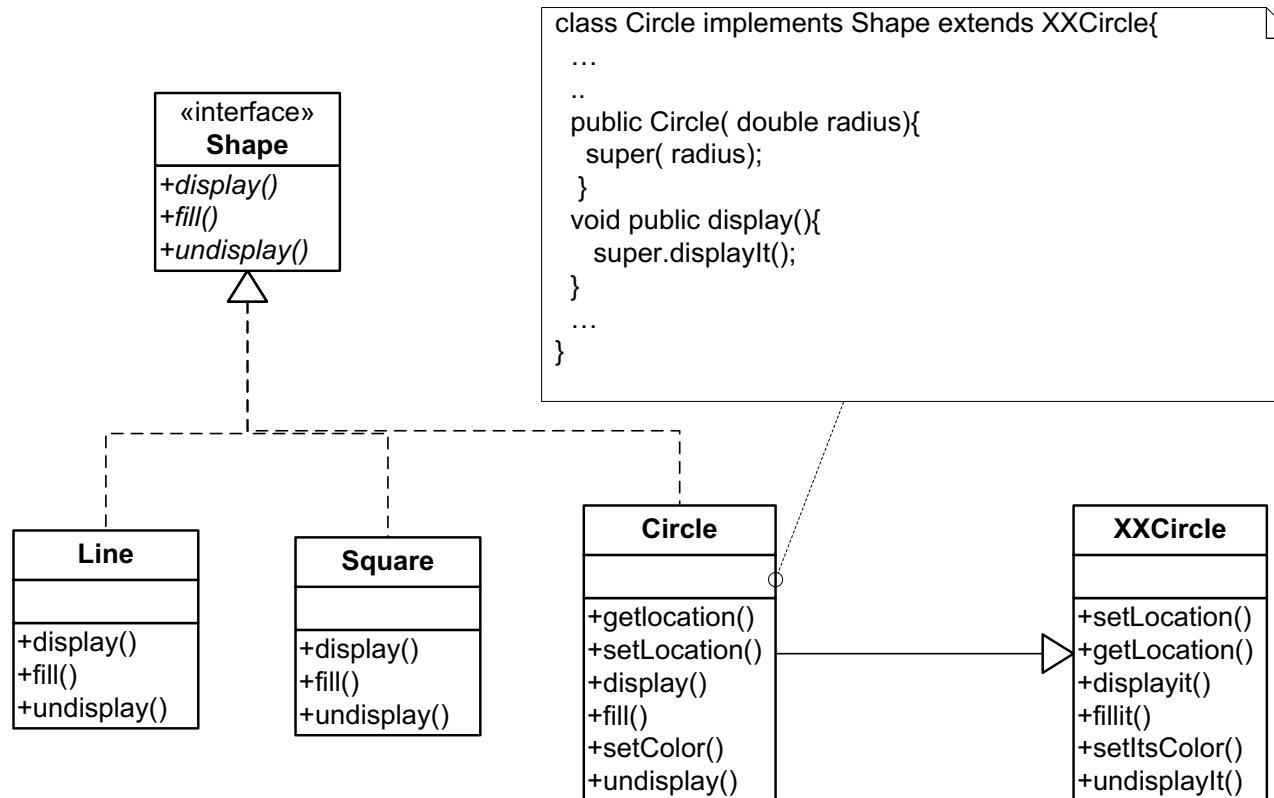


Adapter Example: Identify Participants



- **Target (Shape)**
 - defines the domain-specific interface that Client uses.
- **Adapter (Circle)**
 - adapts the interface **Adaptee** to the **Target** interface.
- **Adaptee (XXCircle)**
 - defines an existing interface that needs adapting.
- **Client (ShapeApp)**
 - collaborates with objects conforming to the Target interface.

Different Implementations of Adapter



Object Adapter and Class Adapter

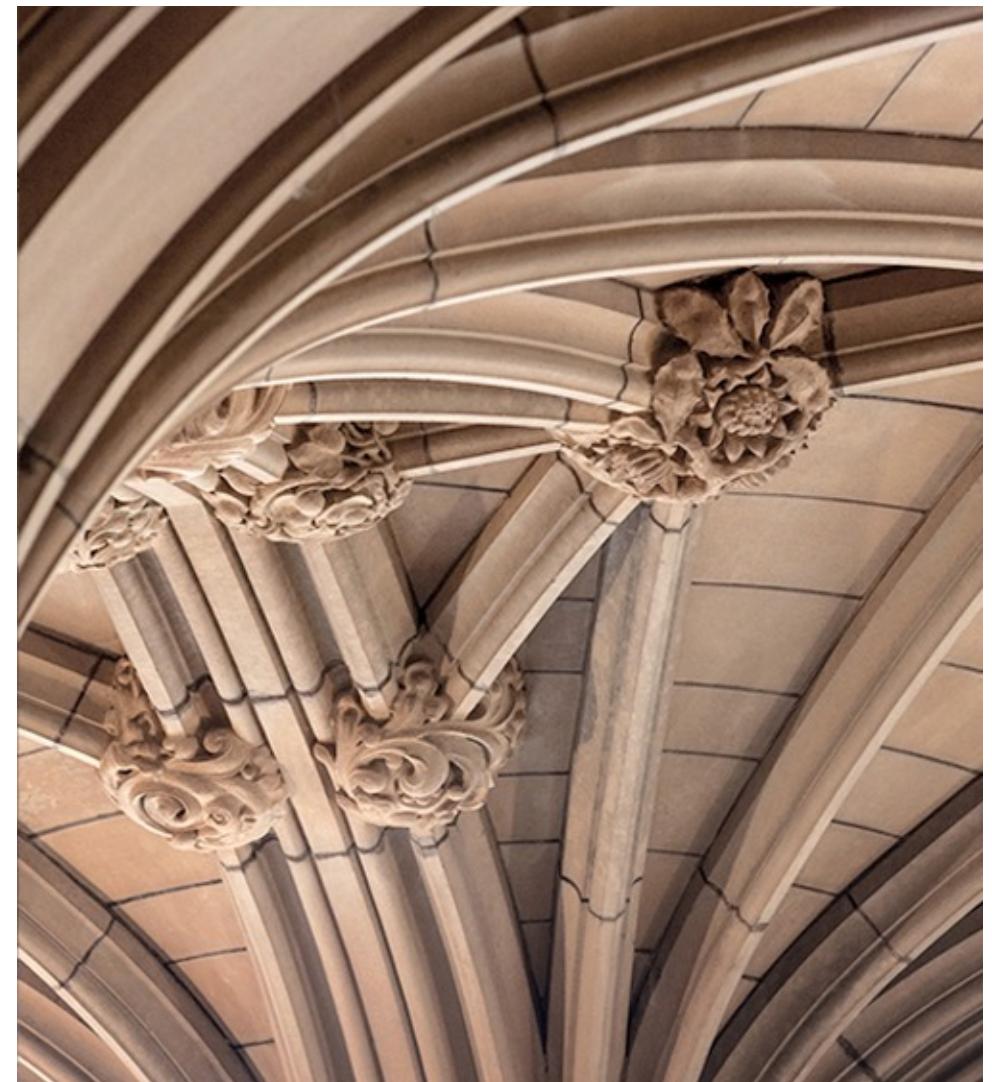
- Object Adapter
 - Relies on object composition to achieve adapter
- Class Adapter
 - Relies on class inheritance to achieve adapter

Two Reuse Mechanisms

- Inheritance and Delegation
 - Inheritance: reuse by subclassing;
 - “is-a” relationship (**white-box reuse**)
 - Delegation: reuse by composition;
 - “has-a” relationship (**black-box reuse**)
 - A class is said to delegate another class if it implements an operation by resending a message to another class
- Rule of thumb – design principles #1
 - **Favour object composition over class inheritance**

Observer Pattern

Object Behavioural



Motivated Scenario

- Anytime the SOFT2201/COMP9201 unit coordinator Xi sent an announcement on Ed, all students could be notified by receiving an email.



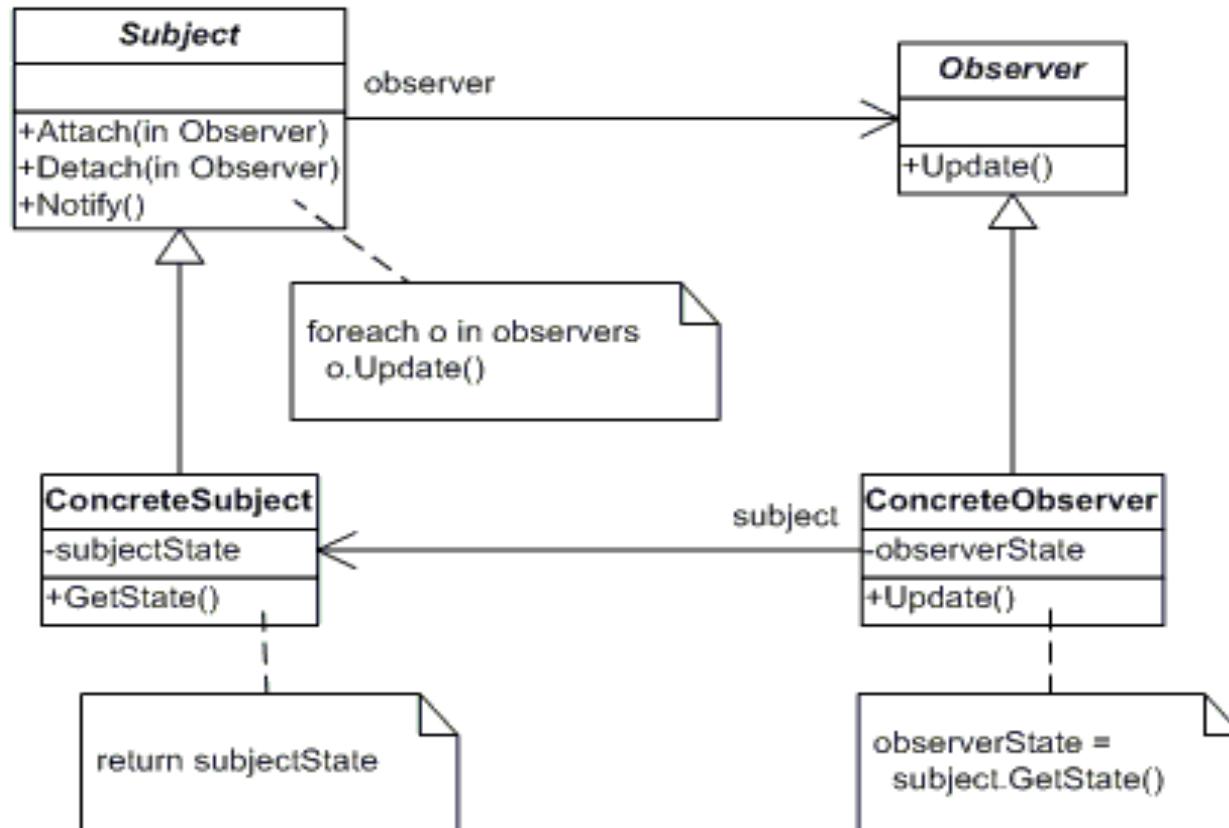
A1 marks has been released



Observer

- **Intent**
 - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
 - A collection of cooperating classes (consistency between related objects)
- **Known as**
 - Dependents, Publish-Subscribe

Observer – Structure



Observer – Participants

Participant	Goals
Subject	Knows its observers. Any number of observer objects may observe a subject. Provides an interface for attaching and detaching observer objects
Observer	Defines an updating interface for objects that should be notified of changes in a subject
ConcreteSubject	Stores state of interest to ConcreteObserver objects Sends notifications to its observers when its state changes
ConcreteObserver	Maintains a reference to a ConcreteSubject object Stores state that should stay consistent with the subject's. Implements the observer's updating interface to keep its state consistent

Revisit the Motivated Example

Subject Perspective

```
public abstract class Subject {  
    3 usages  
    private ArrayList<Observer> observers = new ArrayList<Observer>();  
  
    3 usages  
    public void Attach(Observer observer){  
        observers.add(observer);  
    }  
  
    public void Detach(Observer observer){  
        observers.remove(observer);  
    }  
  
    1 usage  
    public void Notify() {  
        for (Observer o: observers) {  
            o.Update();  
        }  
    }  
}
```

```
public class concreteSubject extends Subject{  
    2 usages  
    private String subjectState;  
  
    1 usage  
    public String getSubjectState() {  
        return subjectState;  
    }  
  
    1 usage  
    public void setSubjectState (String newState) {  
        subjectState = newState;  
    }  
}
```

Revisit the Motivated Example

Observer Perspective

```
public interface Observer {  
    1 usage 1 implementation  
    public void Update();  
}
```

Client Perspective

```
concreteSubject s = new concreteSubject();  
s.Attach(new concreteObserver(s, name: "Tim"));  
s.Attach(new concreteObserver(s, name: "Daniel"));  
s.Attach(new concreteObserver(s, name: "Abbey"));  
  
s.setSubjectState("A1 mark has been released");  
s.Notify();
```

```
public class concreteObserver implements Observer{  
    2 usages  
    private String name;  
    2 usages  
    private String observerState;  
    2 usages  
    private concreteSubject subject;  
  
    3 usages  
    public concreteObserver(concreteSubject subject, String name){  
        this.name = name;  
        this.subject = subject;  
    }  
  
    1 usage  
    @Override  
    public void Update() {  
        observerState = subject.getSubjectState();  
        System.out.println("The latest announcement for " + name + " is: " + observerState);  
    }  
}
```

The latest announcement for Tim is: A1 mark has been released

The latest announcement for Daniel is: A1 mark has been released

The latest announcement for Abbey is: A1 mark has been released

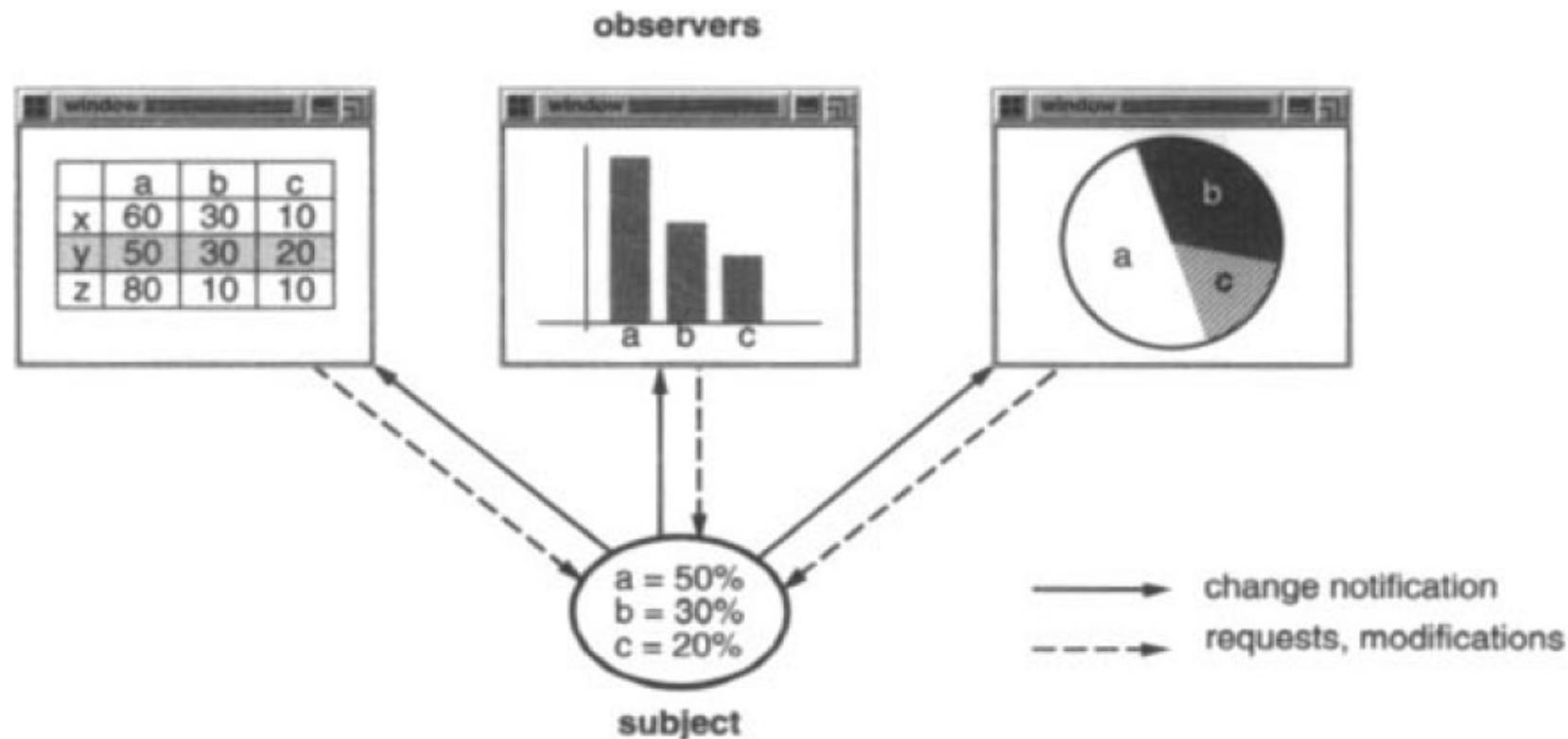
Observer – Applicability

- An abstraction has two aspects, one dependent on the other
- A change to one object requires changing others, and it's not clear how many objects need to be changed
- An object should be able to notify other objects without making assumptions about who these objects are

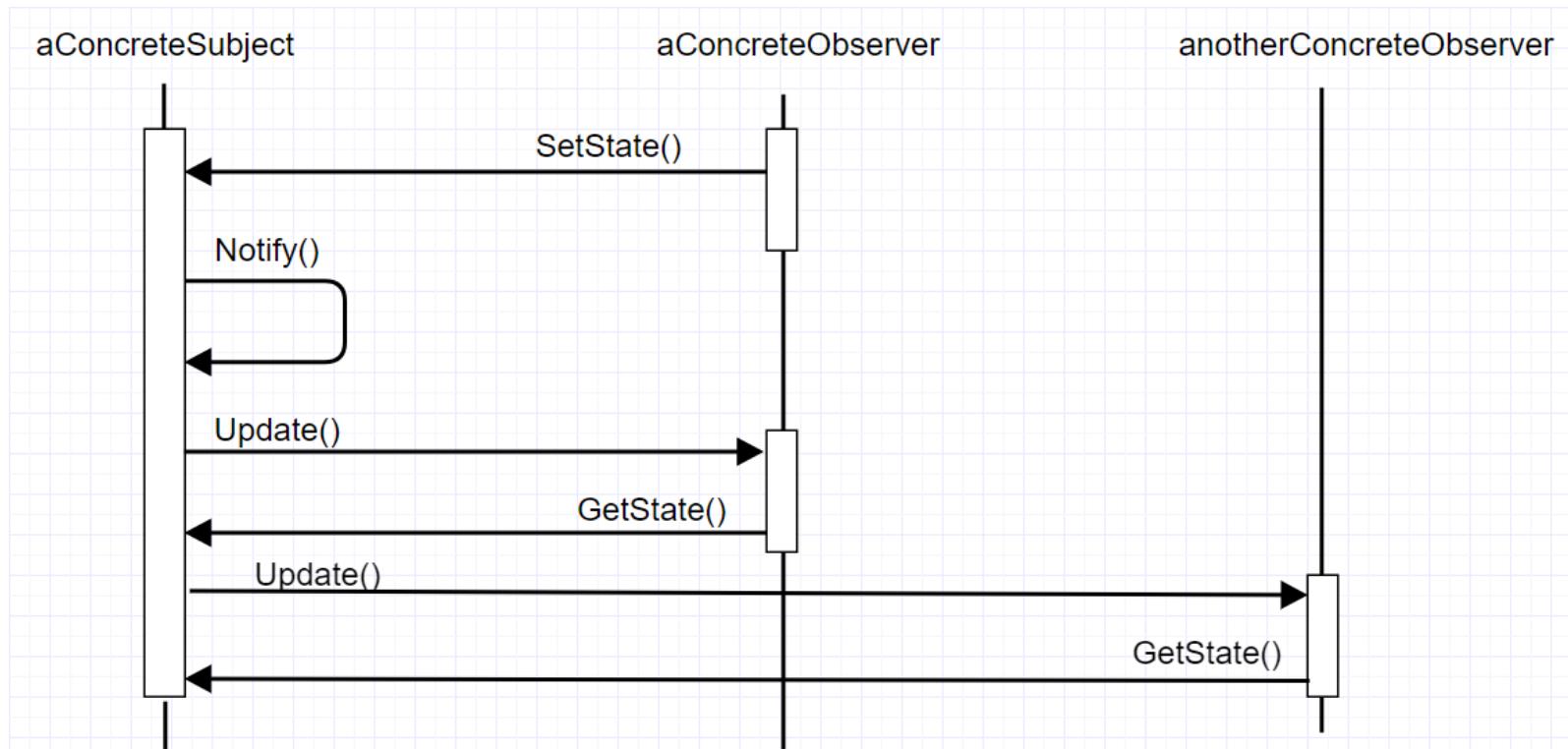
Observer – Publish-Subscribe

- Problem
 - You need to notify a varying list of objects that an event has occurred
- Solution
 - Subscriber/listener interface
 - Publisher: dynamically register interested subscribers and notify them when an event occurs

Observer – Example (Data Representation)



Observer – Collaborations



Observer – Consequences

- Abstract coupling between Subject and Observer
 - *Subject* only knows its Observers through the abstract *Observer* class (it doesn't know the concrete class of any observer)
- Support for broadcast communication
 - Notifications are broadcast automatically to all interested objects that subscribe to the *Subject*
 - Add/remove Observers anytime
- Unexpected updates
 - Observers have no knowledge of each other's presence, so they can be blind to the cost of changing the subject
 - An innocent operation on the subject may cause a cascade of updates to Observers and their dependents

Observer In GUI Class Library

- Observer pattern is the basic structure of event handling system in Java's GUI library
- Each GUI widget is a publisher of GUI related events
 - ActionEvent, MouseEvent, ListSelectionEvent,...
- Observer can be any other objects (GUI or non-GUI objects)

Observer in Java GUI

- Example
 - A simple GUI with a **JTextField** and a **JList** component
 - Each time the List is selected, the selected text is displayed in the **JTextField**.
 - **JList** supports **ListSelectionListener**
 - A subclass of **JTextField** should implements **ListSelectionListener**



Observer in Java GUI

```
1 import javax.swing.*;
2 import javax.swing.event.*;
3 import java.awt.*;
4
5 class MyTextField extends JTextField implements ListSelectionListener{
6     public void valueChanged(ListSelectionEvent lse){
7         JList myList = (JList)lse.getSource();
8         setText((String)(myList.getSelectedValue()));
9     }
10 }
11
```

Method Summary

void	<u>valueChanged(ListSelectionEvent e)</u>
------	-----------------------------------------------------------

Called whenever the value of the selection changes.

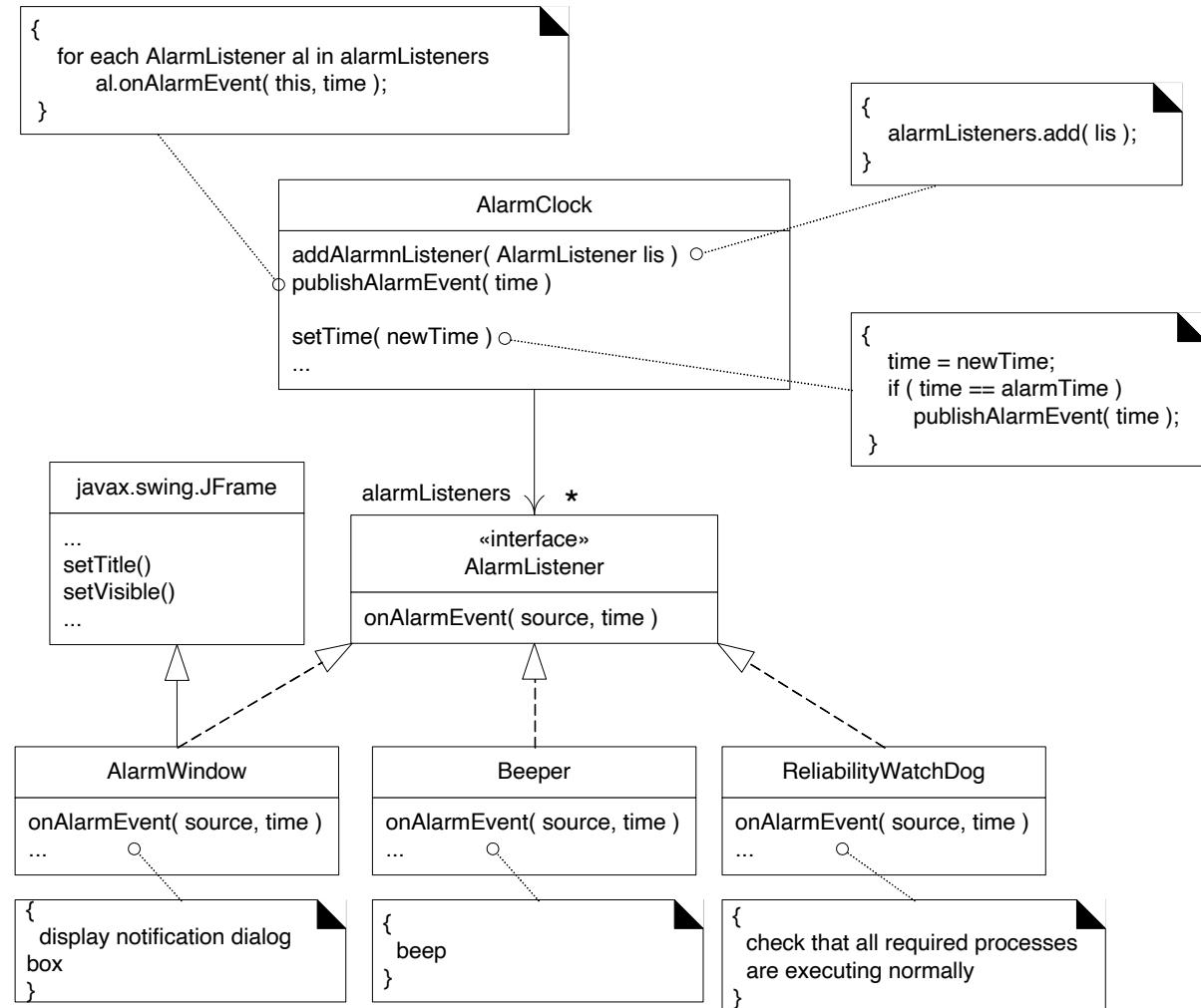
QuickEntryApplication

```
1 public class QuickEntryApplication{
2     public static void main(String argv[]){
3         MyTextField myTextField = new MyTextField();
4         String[]listItems = {"NSW","VIC","SW"};
5         JList myList = new JList(listItems);
6         myList.addListSelectionListener(myTextField); //add listener
7         JFrame myFrame = new JFrame();
8         Container contentpane = myFrame.getContentPane();
9         contentpane.setLayout(new GridLayout(0,1));
10        contentpane.add(new Label("State:"));
11        contentpane.add(myTextField);
12        contentpane.add(myList);
13        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14        myFrame.pack();
15        myFrame.show();
16    }
17 }
18 }
```

Publisher-Many-Subscribers

- One publisher instance could have zero to many registered subscribers
 - E.g., one instance of an *AlarmClock* could have three registered alarm windows, four Beepers, and one *ReliabilityWatchDog*
 - When an alarm even happens, all eight of these *AlarmListeners* are notified via an *onAlarmEvent*
- Observer Implementation (Java)
 - Events are communicated via a regular message, such as on *onPropertyEvent*
 - Event is more formally defined as a class, and filled with appropriate event data
 - The event is then passed as a parameter in the event message

Publisher-Many-Subscribers



Task for Week 7

- Submit weekly exercise on canvas before 23.59pm Saturday
- Well organize your time for assignment 2
- Prepare questions and ask during tutorials
- Self learning on Next Gen POS system

What are we going to learn in week 8?

- Code Review

References

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

SOFT2201/COMP9201: **Software Design and** **Construction 1**

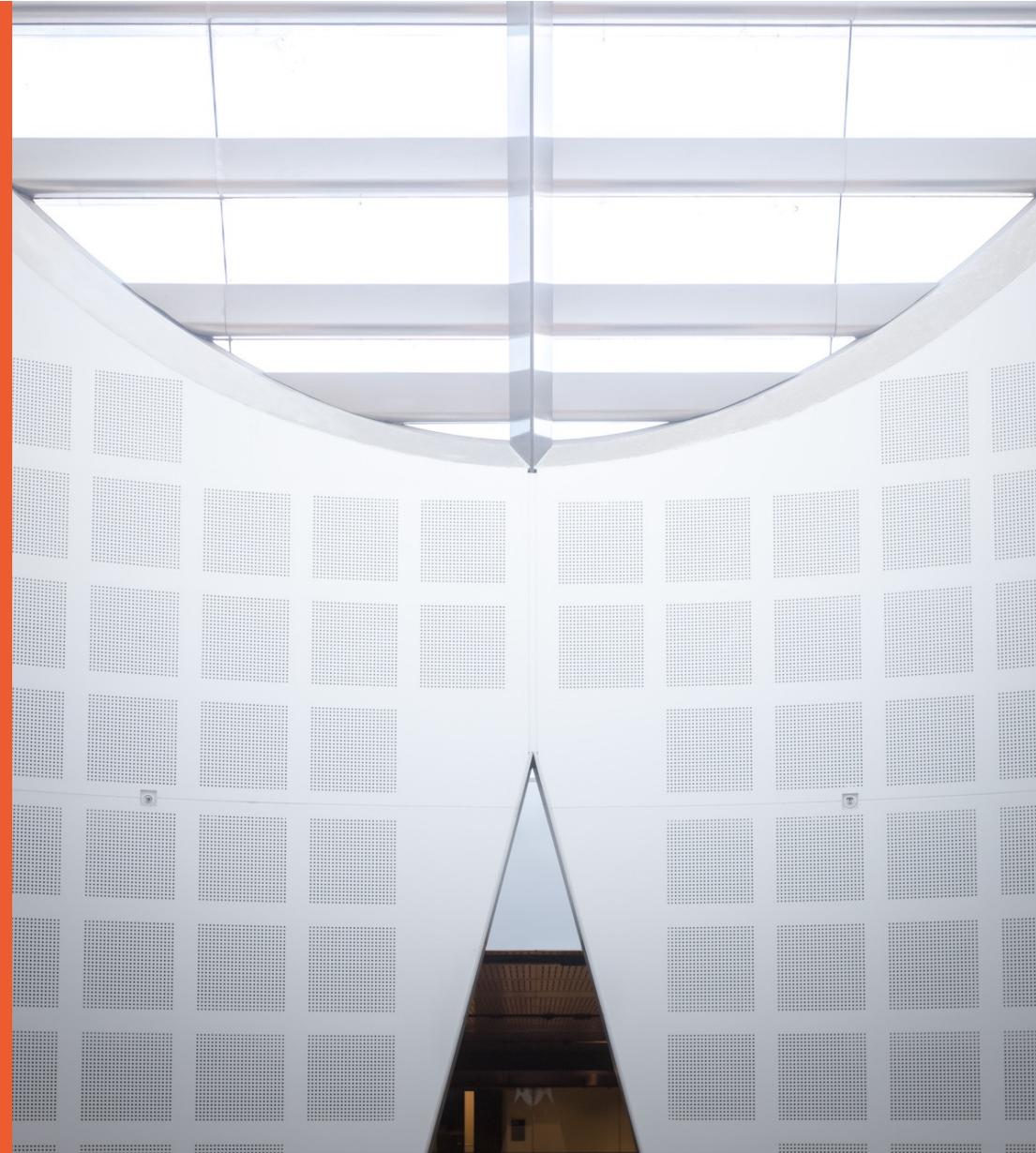
Code Reviews

Dr Xi Wu

School of Computer Science



THE UNIVERSITY OF
SYDNEY



Announcement

- We will do an interactive code review session guided by your tutor during this week's tutorial
 - Pair programming will be used during the tutorial. Please have a look at the preview-recommendation page on canvas (under module 8) before you go to your tutorial
- An announcement could be found on Ed regarding the reschedule of tutorials on Thursday (which will be a public holiday)

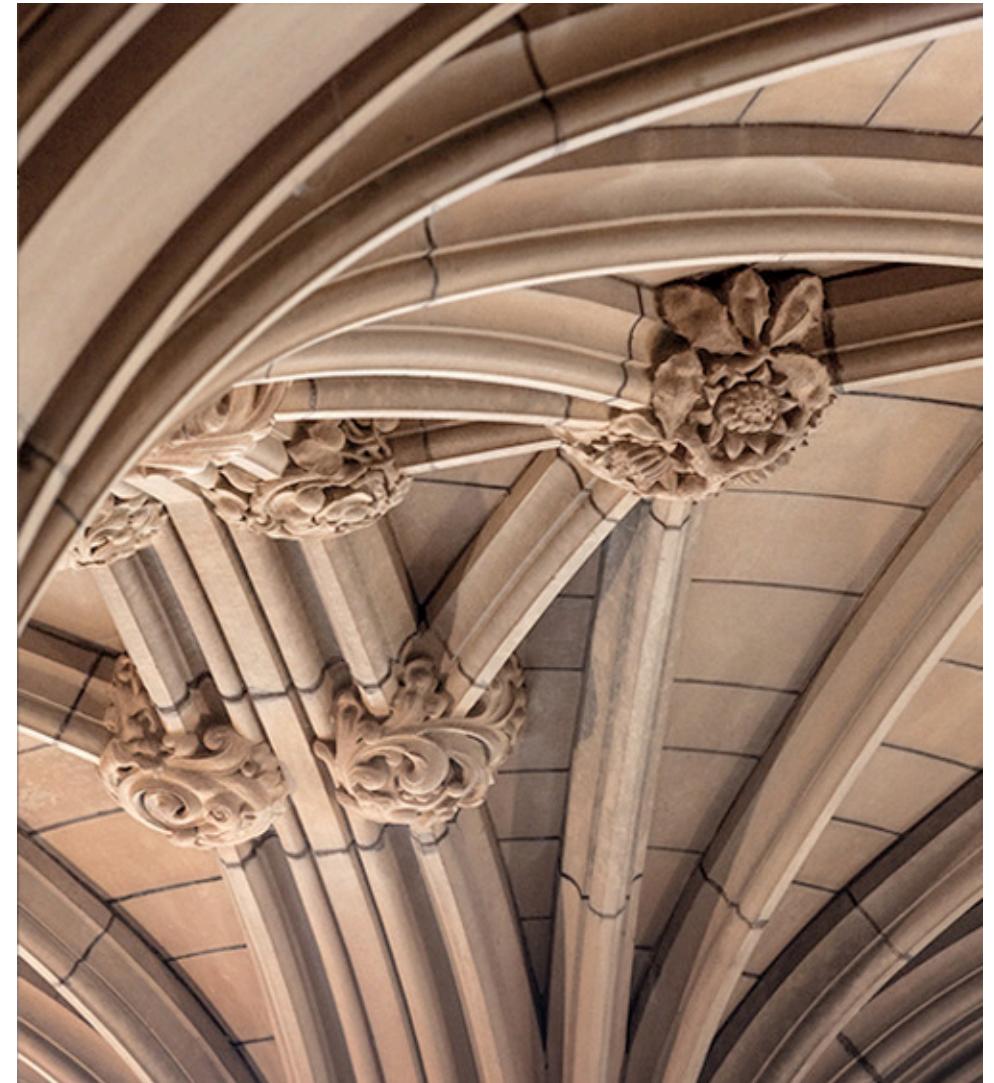
Outline

- Why review
- What to review
- What is a review
- What is a formal review
- What is an informal review

Why review?



THE UNIVERSITY OF
SYDNEY



Why review?

- What is the software, or aspect of the software, to be reviewed for?
 - What are the specifications that need to be met?
- Does the software, or aspect thereof, being reviewed, meet those specifications?

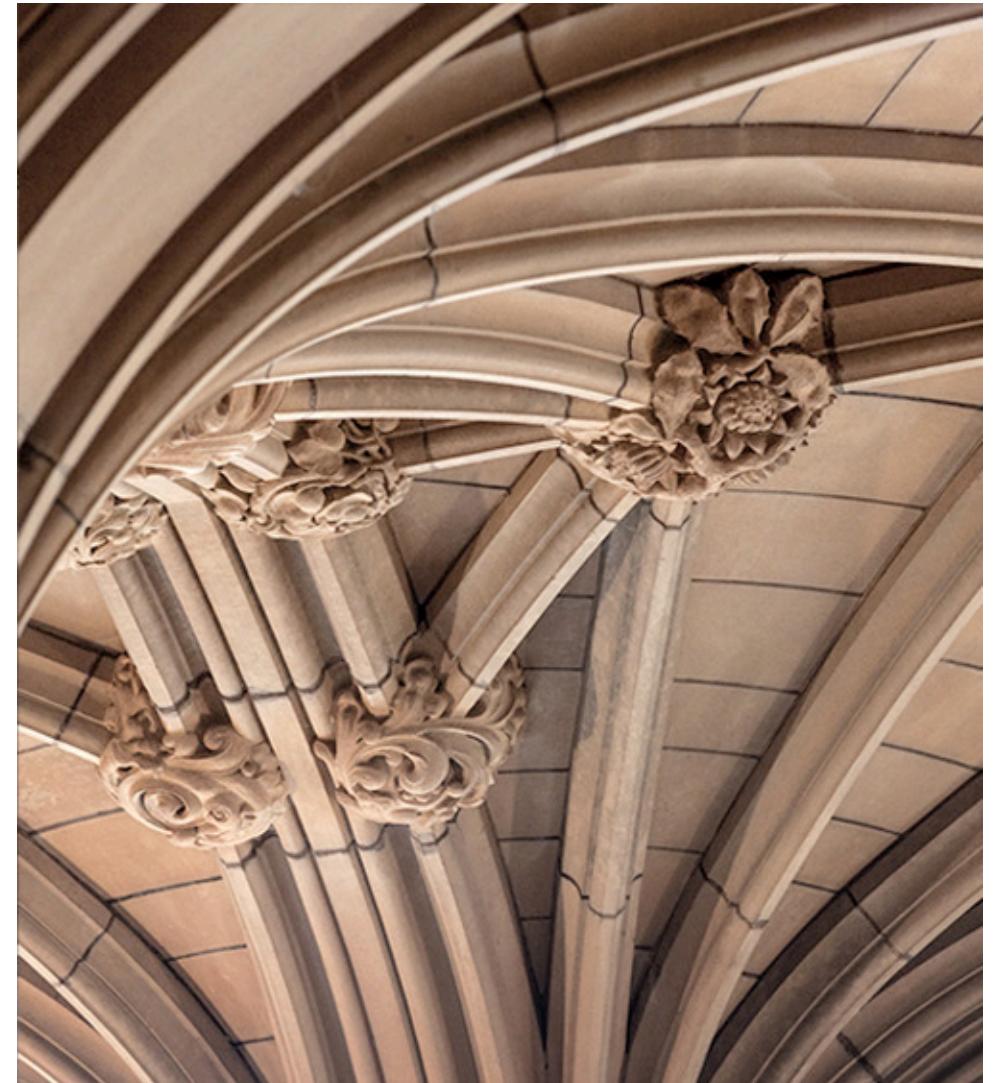
What is the software for and does it work?

- What is the software, or aspect of the software, to be reviewed for?
 - What are the specifications that need to be met?
- Does the software, or aspect thereof, being reviewed, meet those specifications?
- We need a way to check that the software is appropriately designed to meet the expected criteria

What to review?



THE UNIVERSITY OF
SYDNEY



What to review?

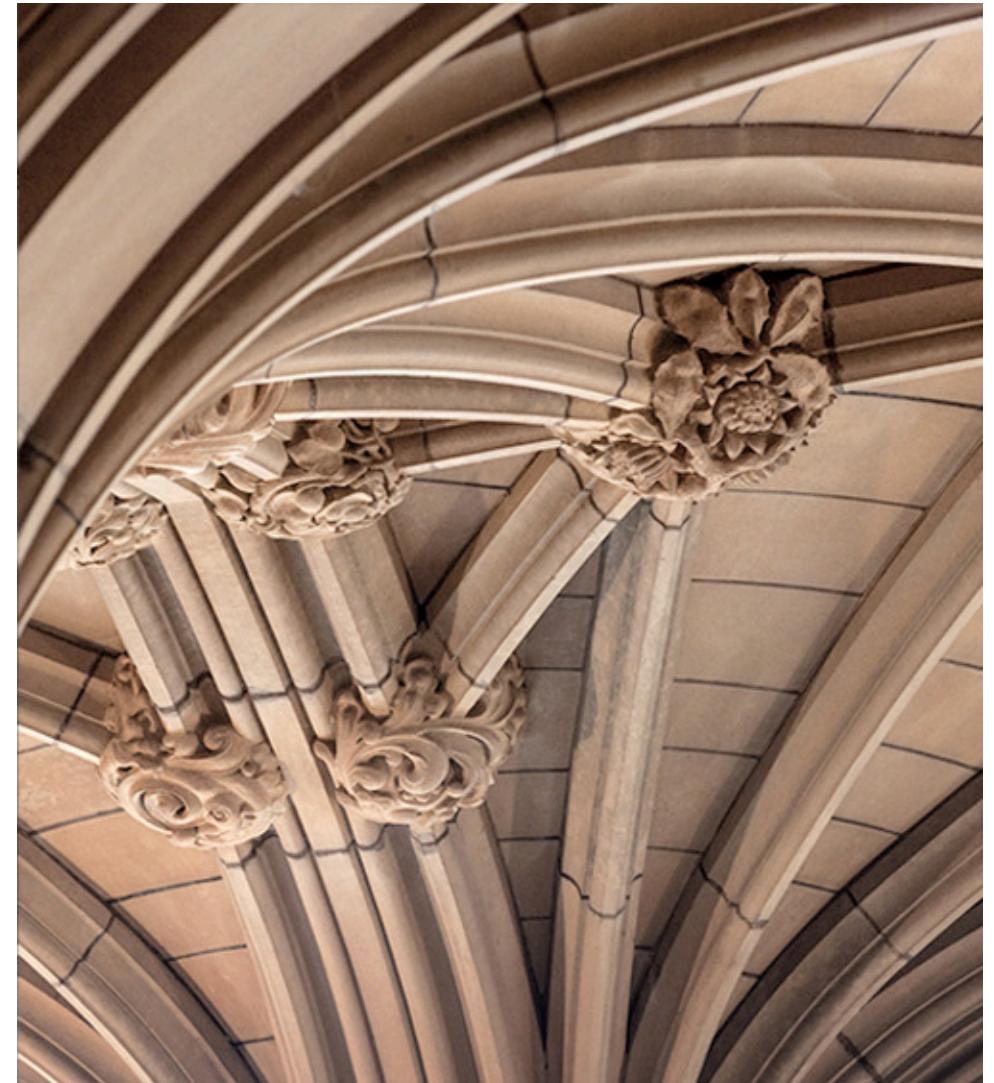
- Any part of a software project can be reviewed
 - Code
 - Documentation
 - Processes
 - Management
 - Specifications
 - Planning

Status Item	Status up to this week	Planned for next week
Major deliverables		
Major issues		
Major risks		
Estimated effort (hr)		
Status (R, Y, G)		(Not Applicable)

What is a review?



THE UNIVERSITY OF
SYDNEY



What is a review?

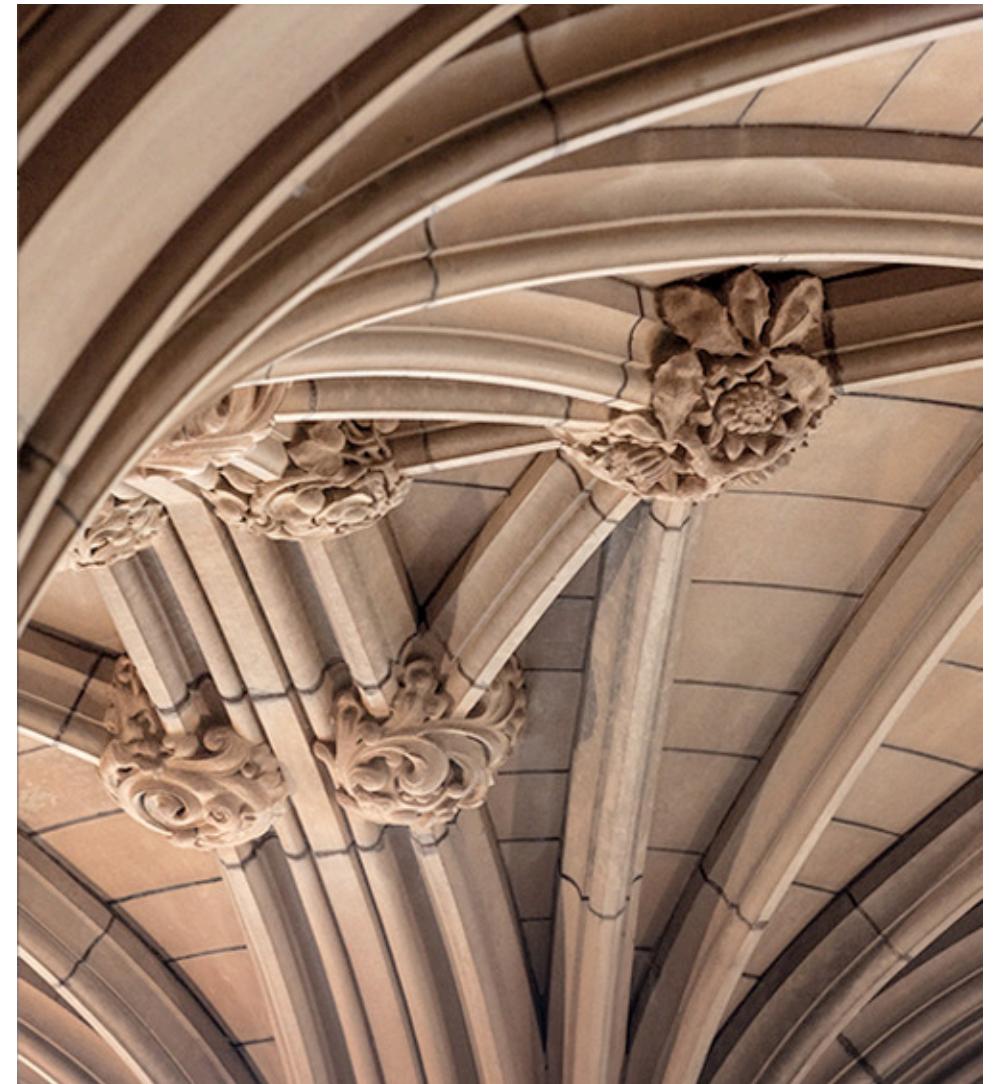
- A software review is
 - "A process or meeting during which a software product, set of software products, or a software process is presented to project personnel, managers, users, customers, user representatives, auditors or other interested parties for examination, comment or approval."

IEEE Standard 1028-2008, "IEEE Standard for Software Reviews", clause 3.5

What is a formal review?



THE UNIVERSITY OF
SYDNEY



Fagan inspection

- An early effort to formalise the process of reviews
- The basis, or at least similar, to subsequent formal approaches
- Describe program development process in terms of operations
- Define ‘entry’ and ‘exit’ criteria for all operation
- Specify objectives for the inspection process to keep the team focused on one objective at a time

Fagan inspection

- Classify errors by type, rank by frequency, identify which types to spend most time looking for
- Analyse results and use for constant process improvement

Fagan inspection operation

- Specify objectives for the inspection process to keep the team focused on one objective at a time
 - Planning
 - Overview
 - Preparation
 - Inspection
 - Rework
 - Follow-up

Fagan inspection operations

- Planning
 - Preparation of materials
 - Arranging of participants
 - Arranging of meeting place
- Overview
 - Group education of participants on review materials
 - Assignment of roles

Fagan inspection operations

- Preparation
 - Participants review item to be inspected and supporting material to prepare for the meeting, noting any questions or possible defects
 - Participants prepare their roles
- Inspection meeting
 - Actual finding of defect

Fagan inspection operations

- Rework
 - Rework is the step in software inspection in which the defects found during the inspection meeting are resolved by the author, designer or programmer. On the basis of the list of defects the low-level document is corrected until the requirements in the high-level document are met.
- Follow-up
 - In the follow-up phase of software inspections all defects found in the inspection meeting should be corrected. The moderator is responsible for verifying that this is indeed the case. They should verify that all defects are fixed and no new defects are inserted while trying to fix the initial defects.

Formal inspection

- Management review
 - Monitor progress
 - Determine status of plans
 - Evaluate management effectiveness
 - Supports decisions about changes in direction, resource allocation, and scope

Formal inspection

- Management review
 - Maintenance plans
 - Disaster recovery
 - Migration strategies
 - Customer complaints
 - Risk management plans
 - ...

Formal inspection

- Management review roles
 - Decision maker
 - Review leader
 - Recorder
 - Management staff
 - Technical staff

Formal inspection

- Management review processes
 - Preparation
 - Plan time and resources
 - Provide funding
 - Provide training
 - Ensure reviews are conducted
 - Act on reviews

Formal inspection

- Technical review
 - Software requirements
 - Software design
 - Software test documentation
 - Specifications
 - ... procedures

Formal inspection

- Technical review roles
 - Decision maker
 - Review leader
 - Recorder
 - Technical reviewer

Formal inspection

- Inspections
 - The purpose of an inspection is to detect and identify software product anomalies. An inspection is a systematic peer examination that does one or more of the following:
 - Verifies that the software product satisfies its specifications
 - Verifies that the software product exhibits specified quality attributes
 - Verifies that the software product conforms to applicable regulations, standards, guidelines, plans, specifications, and procedures

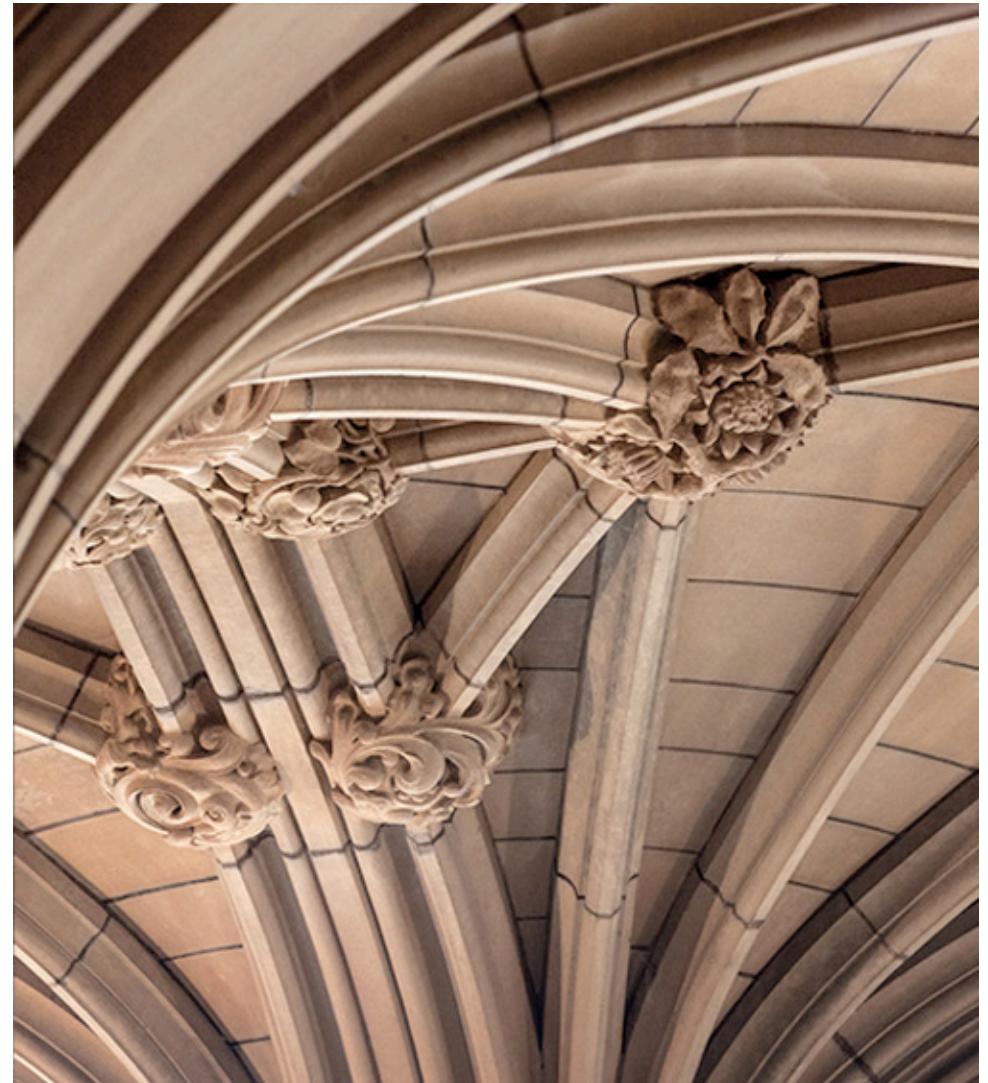
External Audits

- Reviews may also be performed by external groups
 - A formal process is generally highly desirable when dealing with external groups and may extend any of the above approaches

What is an informal review?



THE UNIVERSITY OF
SYDNEY



Informal reviews

- Reviews may be far less formal
 - Pair-programming
 - “Over the shoulder”
 - Walkthroughs
 - Presentations
 - Self-guided reviews
 - Checklists

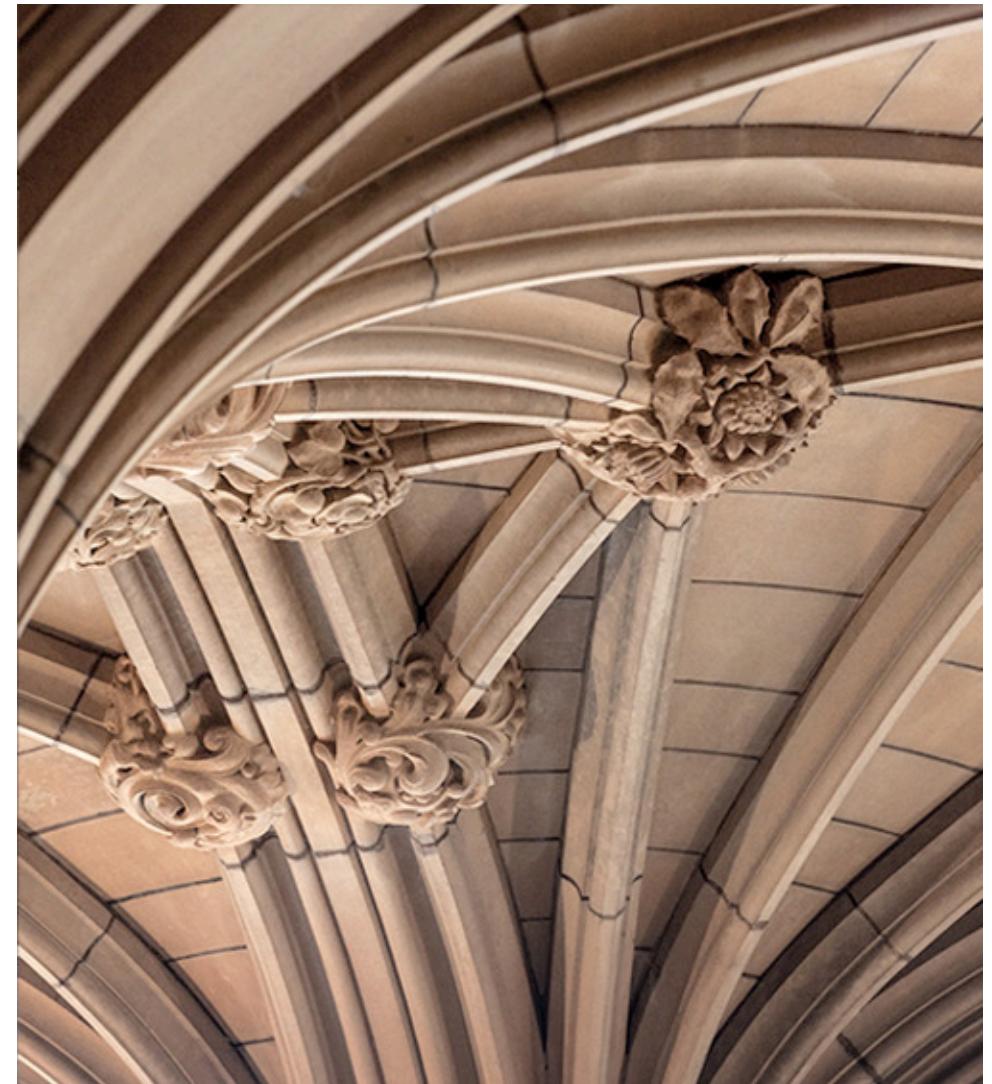
Informal reviews

- Formal reviews are far more effective than informal reviews
- Informal reviews are far more convenient than formal reviews

How about a semi-formal code review?



THE UNIVERSITY OF
SYDNEY



Purpose

- Have a clear purpose for the review
- Identify all relevant project specifications
- Identify how the specifications can be verified
- Identify who the relevant stakeholders are

Change List/Pull Request reviews

- A complete, ‘working’ project exists
- A set of changes are to be applied
- The changes must be reviewed before they are accepted

Change List/Pull Request reviews

- Owner
 - Is this change wanted?
- Technical reviewer
 - Does this change work?
- Technical reviewer
 - Is this change maintainable?
 - (Can I understand it?)

A brief visit to Design by Contract (DbC)

- A software design approach for program correctness
- Also known as contract programming, programming by contract, design-by-contract programming
- Definition of formal, precise and verifiable interface specification for software components
 - Pre-conditions, postconditions and invariants (contract)

Reviewing DbC change lists

- Definition of formal, precise and verifiable interface specification for software components
 - Pre-conditions, postconditions and invariants (contract)
- Are the pre-conditions met?
- Are the post-conditions met?
- Are the invariants invariant?
- Are these likely to stay so?
- How hard is it to verify?

Change List/Pull Request reviews

- Owner
 - Does the documentation agree with the specifications?
- Technical reviewer
 - Are there tests?
 - Do the tests verify the pre/post conditions are met?
- Technical reviewer
 - Is the code written according to the style guide?
 - Does the code use appropriate design patterns?

Change List review challenges

- Size
 - The more there is to read, the more scope there is to miss problems
 - Many small code reviews are more manageable (somewhat like unit tests vs blackbox system tests)
 - A large code review can take so much time that special planning is needed
 - May need to place limits on acceptable sizes for review

Change List review challenges

- Scope
 - Changes that affect many processes may need more reviewers for the required technical knowledge
 - May sometimes be helped by multiple, smaller changelists
 - May bring specifications from different processes into conflict, requiring management review
 - May also be triggered by small changes with big impacts, such as changing JDK version

Change List review challenges

- Complexity
 - Fast, efficient, code may be hard to read and hard to understand
 - Is the code complexity, thus review complexity and maintenance complexity, worth the improvement?
 - Does the complexity affect maintainability and testability?

Change List review challenges

- Confusion
 - What is the change meant to be for?
 - May be doing too many unrelated things
 - May be making unnecessary changes
 - May even be about a disagreement between colleagues (my approach is better!)

Change List review techniques

- Formalise the review process
 - All changes must be reviewed
 - We already have review roles – Owner, Readability, Technical
 - The planning and preparation are only needed once per project (plus for each major change in direction)
 - The project needs clear specifications, requirements, and sufficient documentation
 - The CL must meet the requirements
 - Changes are recommended and either acted on or disagreed with and the result re-evaluated

Change List review comments

- Be clear
- Be objective
- State the issue
- State what is needed to fix it

- E.g., “Document this method”
- E.g., “These braces aren’t needed, please remove”
- E.g., “Split this file into one per class, for readability”
- E.g., “Use informative variable names”

Change List review examples

- This needs tests
- This needs unit tests
- This needs integration tests
- This needs documentation
- What is this for?
- This doesn't follow the style guide
- Resubmit without the temp files

Change List review techniques

- Automation is great!
 - When it works
- Code style
- Coverage
- Test suites
- Performance tests
- Spelling (A little more challenging in code)
- Common code issues

Reviewing larger projects

- You may need to review larger projects, rather than changes.
- A formal review process will ensure all parties understand what is expected
- The code inspection will be similar to a changelist review, but with no existing base to compare to
 - More work is needed to verify requirements

Change List review comments

- E.g., “Document this method”
- E.g., “These braces aren’t needed, please remove”
- E.g., “Split this file into one per class, for readability”
- E.g., “Use informative variable names”
- A **report** on a set of changes would discuss why the comments were made, the benefits, the problems, and discuss other approaches
- A **code review** is far more concise and directed

Task for Week 8

- Additional presentation video about code review on project can be found on canvas (Recorded Lecture section)
- Submit weekly exercise on canvas before 23.59pm Saturday
- Attend tutorial on time for an interactive code review session guided by your tutor during tutorial

What are we going to learn on week 10?

- Creational Design Pattern
 - Prototype
- Behavioral Design pattern
 - Memento

Attention Please: due to Monday of week 9 (3 Oct) is a public holiday, we don't have lecture on week 9. BUT we still have tutorials on week 9.

References

- Fagan, M. E. (1976). "Design and code inspections to reduce errors in program development". IBM Systems Journal. 15 (3): 182–211. doi:[10.1147/sj.153.0182](https://doi.org/10.1147/sj.153.0182)
- IEEE Std . 1028-1997, "IEEE Standard for Software Reviews", doi:[10.1109/IEEESTD.2008.4601584](https://doi.org/10.1109/IEEESTD.2008.4601584)

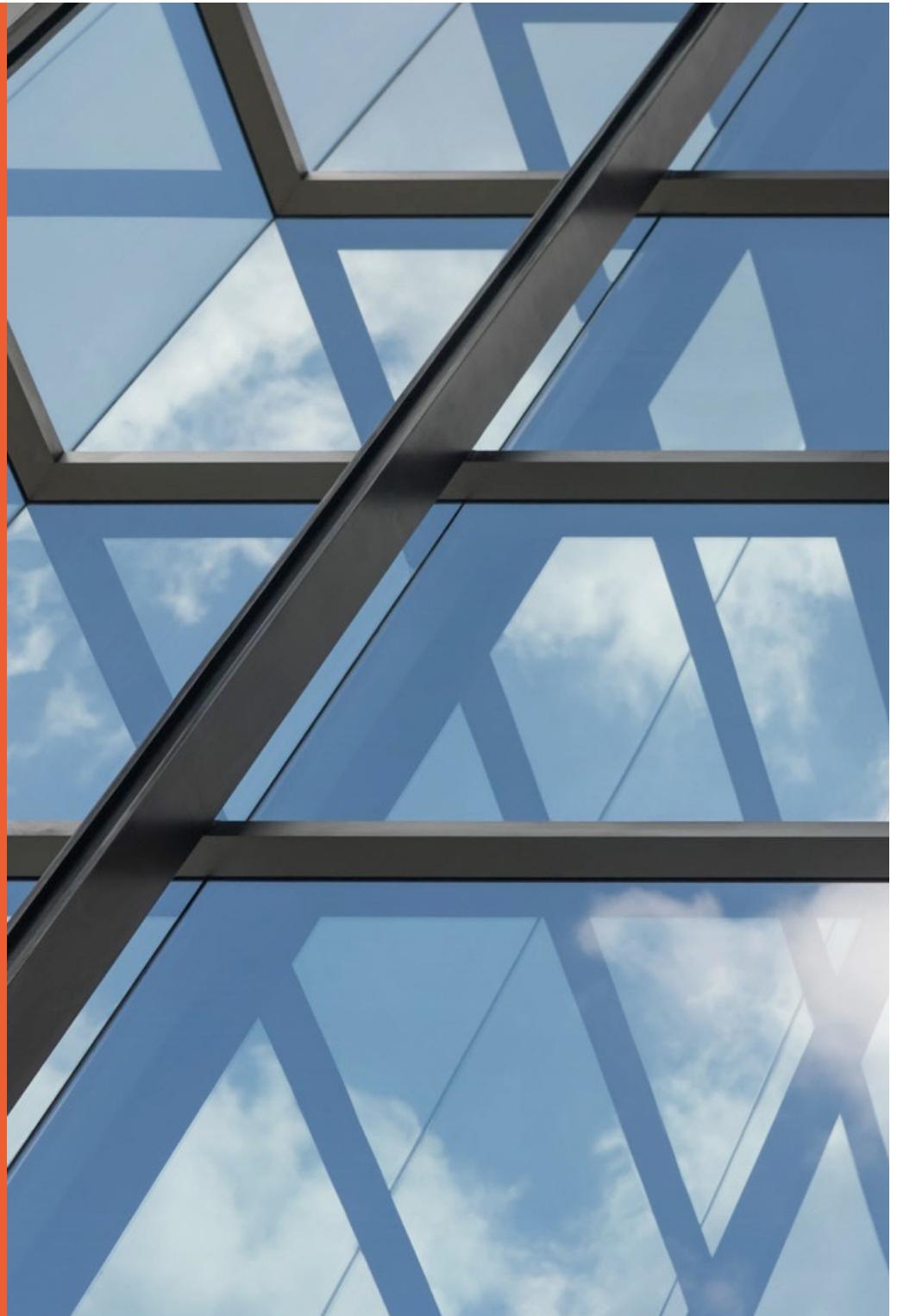
SOFT2201/COMP9201: Software Construction and Design 1

Testing

Dr. Xi Wu
School of Computer Science



THE UNIVERSITY OF
SYDNEY



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Agenda

- Software Testing
- Unit Testing

Software Engineering Body of Knowledge

- Software Requirements
- **Software Design / Modelling**
- **Software Construction**
- **Software Testing**
- Software Maintenance
- Software Configuration Management
- Software Engineering Process
- Software Engineering Tools and Methods
- Software Quality

IEEE® computer society

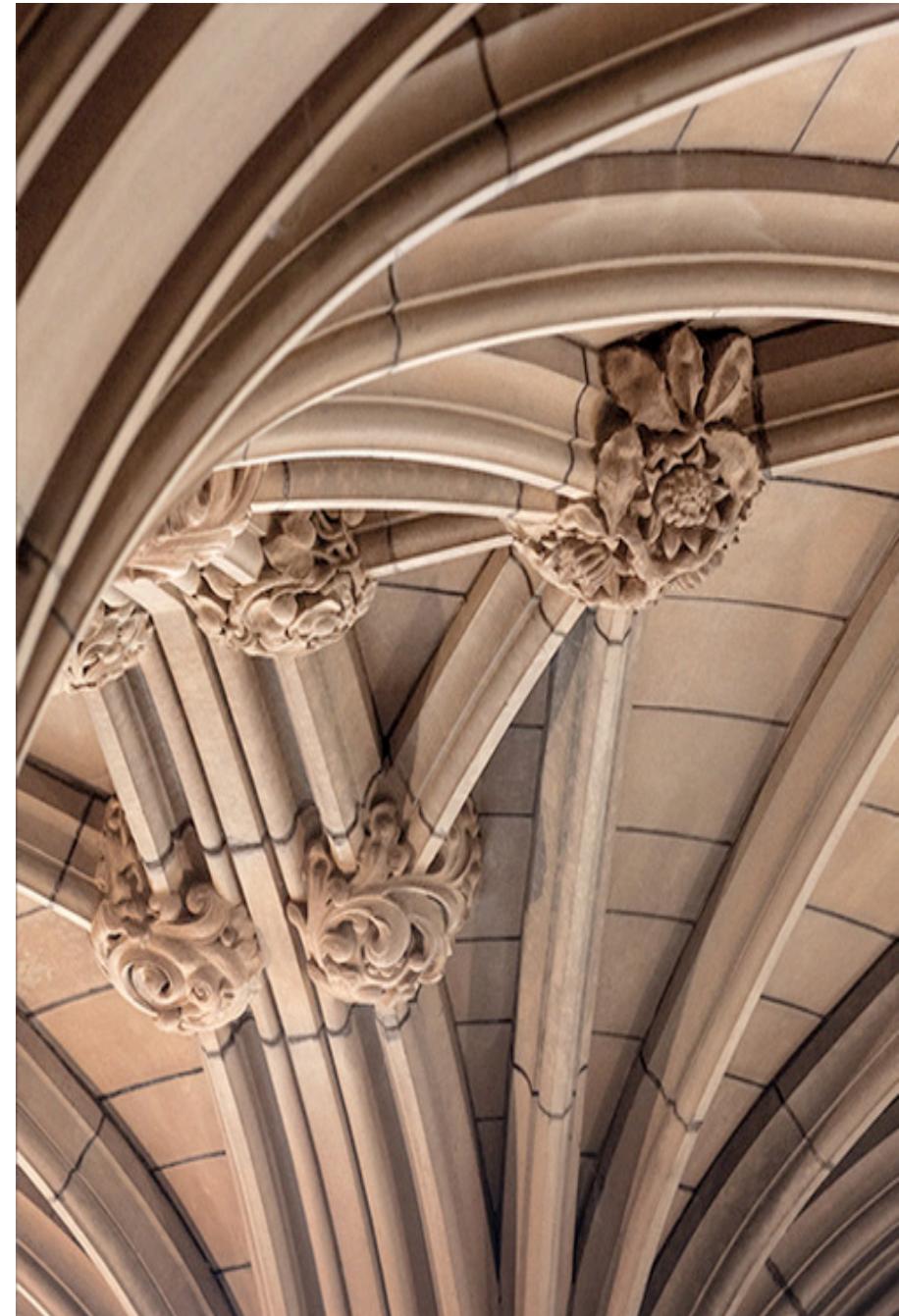


Software Engineering Body of Knowledge (SWEBOK) <https://www.computer.org/web/swebok/>

Why Software Testing?



THE UNIVERSITY OF
SYDNEY



Software is Everywhere!

- Societies, businesses and governments depend on SW
 - Power, Telecommunication, Education, Government, Transport, Finance, Health
 - Work automation, communication, control of complex systems
- Large software economies in developed countries
 - IT application development expenditure in the US more than \$250bn/year¹
 - Total value added GDP in the US²: \$1.07 trillion
- Emerging challenges
 - Security, robustness, human user-interface, and new computational platforms

¹ Chaos Report, Standish group Report, 2014

² softwareimpact.bsa.org

Software Failure - Ariane 5 Disaster⁵

What happened?

- European large rocket - 10 years development, ~\$7 billion
- Unmanaged software exception resulted from a data conversion from 64-bit floating point to a 16-bit signed integer
- Backup processor failed straight after using the same software
- Exploded 37 seconds after lift-off



Why did it happen?

- Design error, incorrect analysis of changing requirements, inadequate validation and verification, testing and reviews, ineffective development processes and management

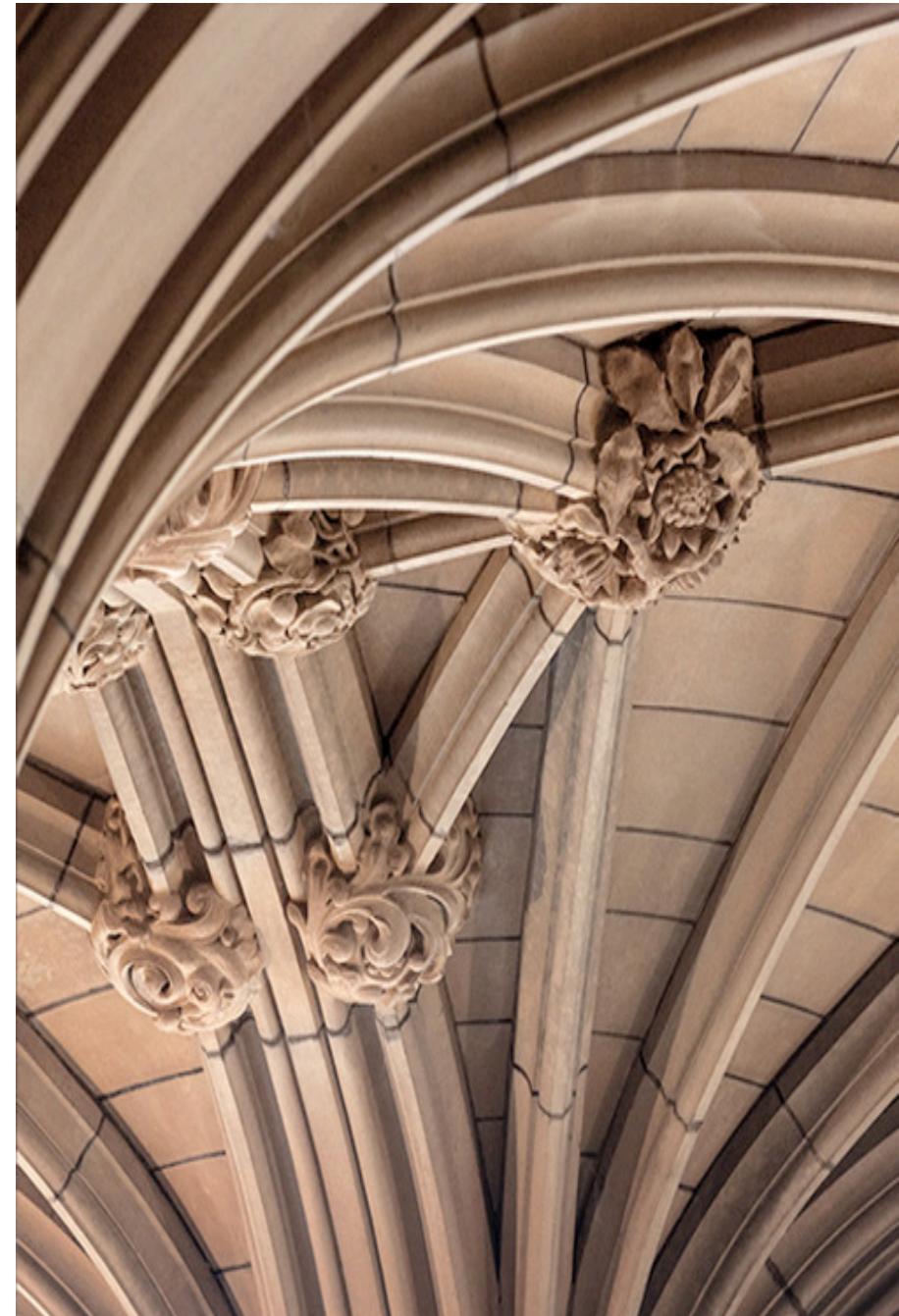
⁵ <http://iansommerville.com/software-engineering-book/files/2014/07/Bashar-Ariane5.pdf>

Why Software Testing?

- Software development and maintenance costs
 - Financial burden of failure
- Total costs of imperfect software testing for the US in 2002 was AUD86 billion*
 - One third of the cost could be eliminated by ‘easily’ improved software testing
- Need to develop functional, robust and reliable software
 - Human/social factor
 - Dependence on software in many aspects of their lives
 - Small software errors can lead to disasters

* NIST study 2002

What is Software Testing?



Software testing

- Software process to
 - demonstrate that software meets its requirements (validation testing)
 - Find incorrect or undesired behaviour caused by defects (defect testing)
 - e.g. crashes, incorrect results, data corruption
- Part of the software Verification and Validation (V&V) process

Types of testing

- **Unit testing**
 - Verify functionality of software components independent of the whole system
- **Integration testing**
 - Verify interactions between software components
- **System Testing**
 - Verify functionality and behaviour of the entire software system
 - Includes security, performance, reliability, and external interfaces
- **Acceptance testing**
 - Verify desired acceptance criteria are met from the users point of view

Software Verification and Validation

- Software testing is part of software V&V
- The goal of V&V is to establish confidence that the software is “fit for purpose”
- Software Validation
 - Are we building the right product?
 - Ensures that the software meets customer expectations
- Software Verification
 - Are we building the product correctly
 - Ensure the software meets the stated functional and non-functional requirements

Black box or White box

- **Black box testing**
 - The internals of the software system is unknown
 - Only inputs to the system are controlled, and outputs from the system are measured
 - Specification-based testing
 - May be the only choice to test libraries without access to internal
- **White box testing**
 - The internals of the software system are known
 - The internal structure is tested directly
 - Unit, integration, system testing

Types of testing

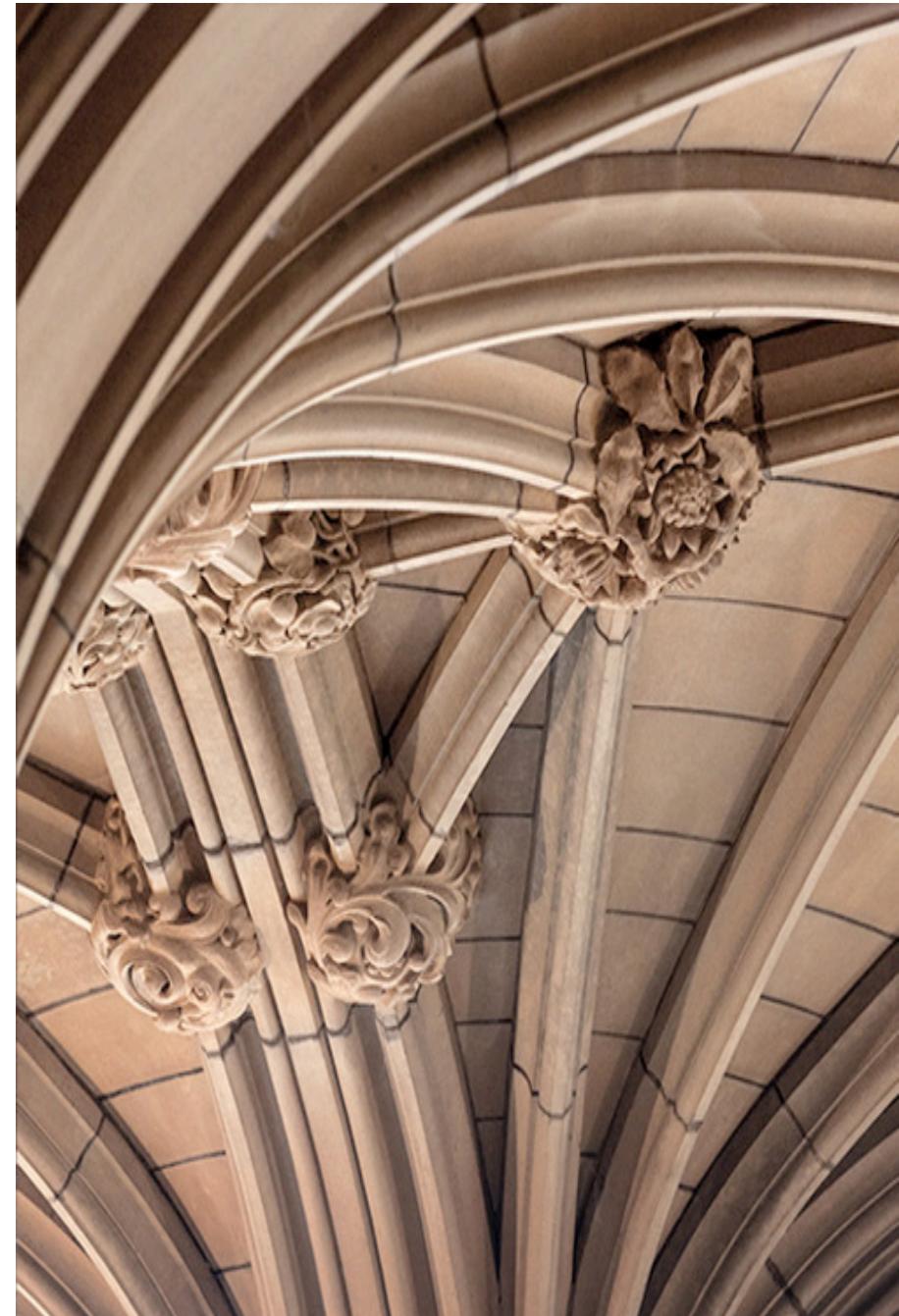
Functional testing

- Unit
- Integration
- System
- Regression
- Interface
- User Acceptance
- Configuration
- Sanity

Non-functional testing

- Performance
- Stress
- Reliability
- Usability
- Load
- Security

Who should design and run tests?



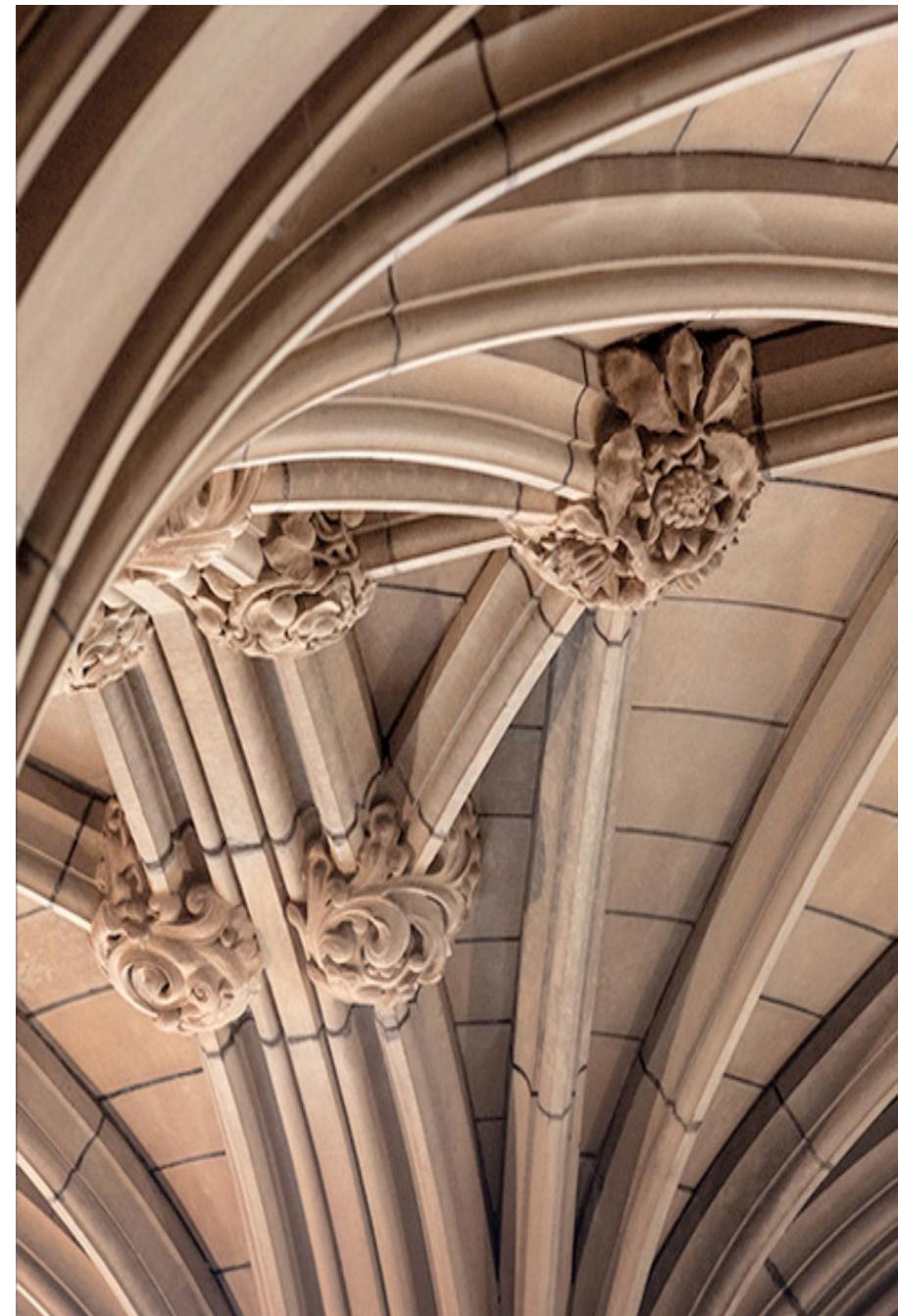
Test engineer

- **Independent testers**
 - Independent testers do not have the same biases as the developer
 - Different assumptions
 - Domain specific knowledge of testing
- **Developer**
 - Understands the system being developed
 - Domain specific knowledge of the system
 - Cheaper
 - Can finish writing the system faster without tests since they won't make mistakes

Unit Testing



THE UNIVERSITY OF
SYDNEY



Unit testing

- The process of verifying functionality of software components independently
 - Unit can mean methods, functions, or object classes
 - Verify that each unit behaves as expected
 - Carried out by developers and software testers
 - First level of testing

Why unit testing

- Maintain and change code at a smaller scale
- Discover defects early and fix it when its cheaper
- Simplify integration testing
- Simplify debugging
- Code reusability

How to do the unit test

- Identify the unit that you want to test
- Design test case
- Prepare test data (input and expected output)
- Run test case using test data
- Compare result to expected output
- Prepare test reports

Designing test cases

- Effective test cases show:
 - The unit does what it is supposed to do
 - Reveal defects, if they exist (does not do what it is not supposed to do)
- Design two types of test case
 - Test normal operation of the unit
 - Test abnormal operation (common problems)

Designing test cases - techniques

- Partition testing (equivalence partitioning)
 - Identify groups of tests that have common characteristics
 - From each group, choose specific tests
 - Use program specifications, documentation, and experience
- Guideline-based testing
 - Use testing guidelines based on previous experience of the kinds of errors made
 - Depends on the existence of previous experience (developer/product)

Equivalence partitioning

- Groups of test input that have common characteristics
 - Positive numbers
 - Negative numbers
 - Boundaries
- Program is expected to behave in a comparable way for all members of a group
 - Control flow should be similar for all members
- Choose test cases from each partition

Test case selection

- Understanding developers thinking
 - Easy to focus on typical values of input
 - Common case, and what was asked for
 - Easy to overlook a typical value of input
 - Users, other developers, new features, all have different expectations
- Choose test cases that are
 - On boundaries of partitions
 - In ‘midpoint’ of partitions
 - NB: Boundaries may be unclear (-1, 0, 1, 0.5)

Test cases – identifying partitions

- Consider this specification:
 - The program accepts 4 to 8 inputs that are five digit integers greater than 10,000
- Identify the input partitions and possible test inputs

?

Test cases – identifying partitions

- Consider this specification:
 - The program accepts 4 to 8 inputs that are five digit integers greater than 10,000
- Identify the input partitions and possible test inputs
- How many values
 - <4, 4-8, >8
- How many digits
 - < 5, 5, > 5, non-digits
- How big
 - > 10000
 - etc.

Test case selection guidelines

- Knowledge of types of test case effective for finding errors
- If testing sequences, arrays, lists:
 - Single value
 - Different sequences of different sizes
 - Test partition boundaries (first, middle, last)
 - Consider order of values

Test case selection guidelines

- Choose inputs that force the system to generate all expected error messages
- Design inputs that cause buffer overflows
- Repeat input
- Force invalid outputs to be generated
- Force computations results that are too large or too small
- Domain specific knowledge!

Acquiring domain specific knowledge

- Be an expert on the system, or type of system
- or,
- Make many mistakes
- Identify mistakes
- Write tests to identify mistakes
- Fix mistakes
- Be an expert on the system, or type of system
- Regression testing!

Regression testing

- If a defect is identified in software it can be fixed
 - How did it get there?
 - How do you stop it happening again?

Regression testing

- Regression: a defect that has been fixed before, happens again
 - Human error
 - Version control problems
 - Specific case is fixed, but the general case remains
 - Convergent evolution

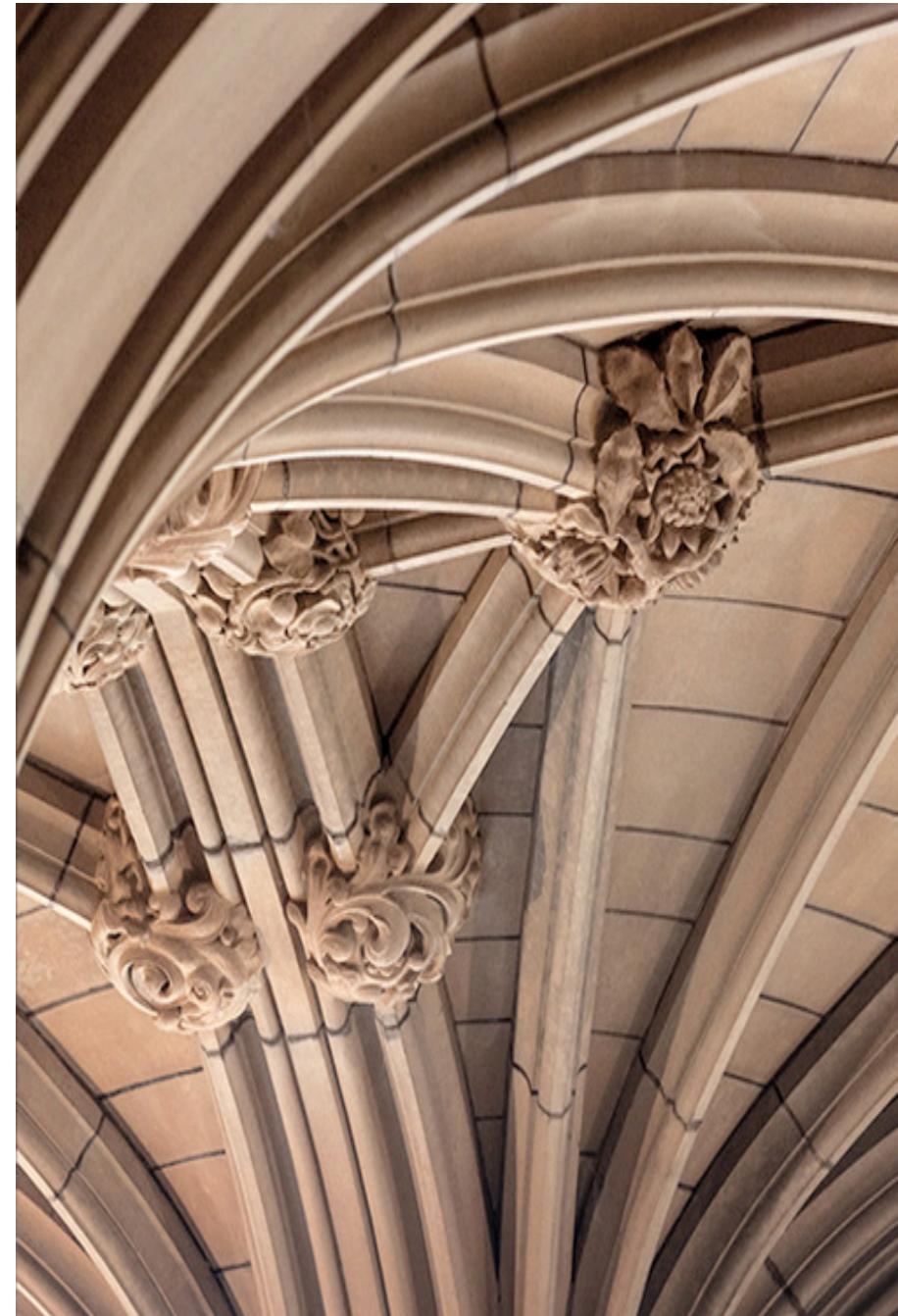
Regression testing

- As defects in software are fixed, tests are written that demonstrate that the software is fixed (at least in regard to that particular defect)
 - Tests can be re-run with each change in the software system
 - Regression testing
 - Frequently automated

When to test



THE UNIVERSITY OF
SYDNEY



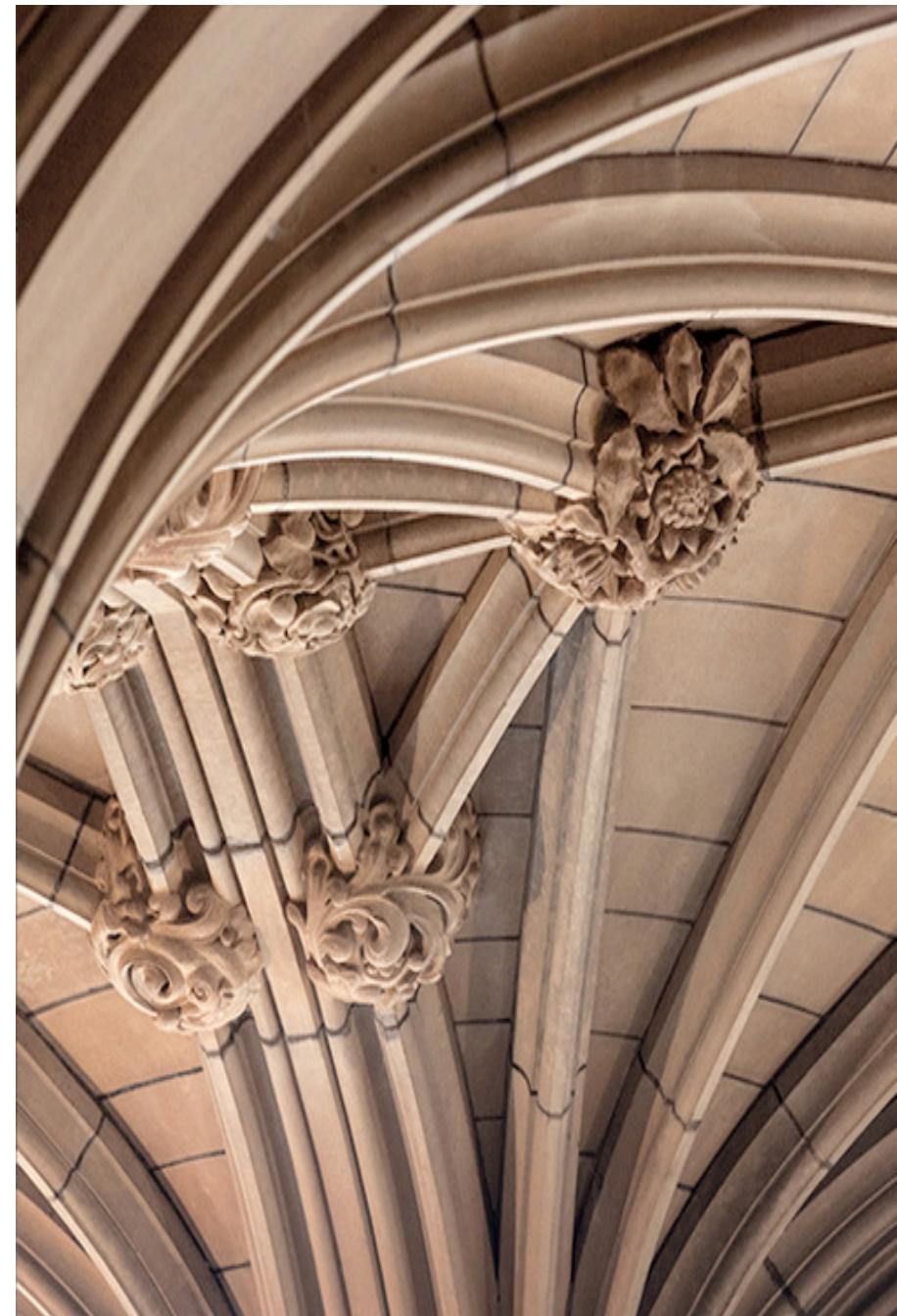
When to test

- Continuously
- When the software system changes
 - Code changes
 - Design changes
 - Infrastructure changes
 - At regular intervals in case the above missed a change

How to test



THE UNIVERSITY OF
SYDNEY



How to test

- Write testable code

```
public static void main(String[] args) {  
    // All the code  
    // All  
}
```

How to test

- Write testable code

```
public static void main(String[] args) {  
    Application app = new Application();  
}
```

```
Public class Application {  
    Application() {  
        // All the code  
    }  
}
```

How to test

- Write testable code

```
public static void main(String[] args) {
    Application app = new Application();
    app.doEverything();
}

public class Application {
    Application() {
        // Construct the application
    }
    public void doEverything() {
        // All the code
    }
}
```

How to test

- Write testable code

```
public class Application {  
    Application() {  
        // Construct the application  
    }  
    public void doEverything() {  
        // Most of the code  
        doSomeOfTheThings();  
    }  
    public void doSomeOfTheThings() {  
        // Some of the code  
    }  
}
```

How to test

- Write testable code

```
public class Application {  
    Application() {  
        // Construct the application  
    }  
    public void doEverything() {  
        // Some code  
        Thing = doSomeOfTheThings(thing);  
        // More code  
    }  
    public BigThing doSomeOfTheThings(LittleThing littleThing) {  
        // Some of the code that deals with LittleThings  
    }  
}
```

How to test

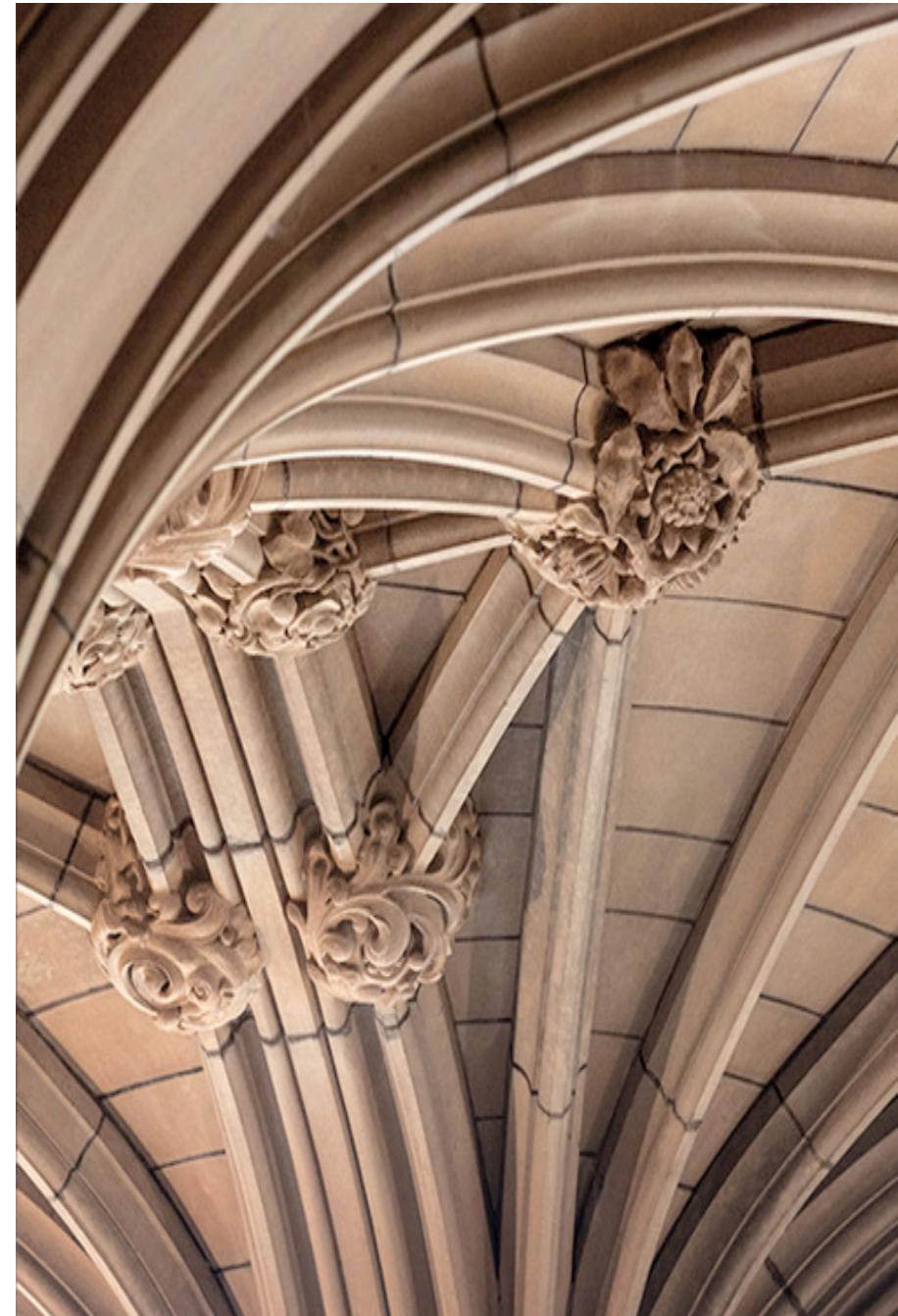
- Write testable code

```
public class Application {  
    // ...  
    public void doEverything(LittleThingFactory littleThingFactory) {  
        LittleThing firstThing= littleThingFactory.makeThing();  
        LittleThing secondThing = doStuff(firstThing);  
        doStuff(secondThing);  
        doStuffWithTwoThings(firstThing, secondThing);  
        doSomeOfTheThings(thing);  
        // ...  
    }  
    protected BigThing doSomeOfTheThings(LittleThing littleThing) {  
        // Some of the code that deals with LittleThings  
    }  
    // ...  
}
```

Unit Testing in Java



THE UNIVERSITY OF
SYDNEY



Unit testing terminology

- **Unit test**
 - A piece of code written by a developer that executes a specific functionality in the code under test and asserts a certain behaviour or state as correct
 - Small unit of code (method/class)
 - External dependencies are removed
 - (Mocking)
- **Test fixture**
 - Testing context
 - Shared test data
 - Methods for setting up test data

Unit testing frameworks for Java

- JUnit
- TestNG
- Jtest
- Many others
- Custom, developer-written, tests

JUnit

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        assertEquals(2, calculator.add(1, 1));
    }
}
```

JUnit

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        assertEquals(2, calculator.add(1, 1));
    }
}
```



JUnit

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;  
  
import mypackage.Calculator;  
  
class CalculatorTest {  
    @Test  
    void addition() {  
        Calculator calculator = new Calculator();  
        assertEquals(2, calculator.add(1, 1));  
    }  
}
```

JUnit

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;  
  
import mypackage.Calculator;  
  
class CalculatorTest {  
    @Test  
    void addition() {  
        Calculator calculator = new Calculator();  
        assertEquals(2, calculator.add(1, 1));  
    }  
}
```



JUnit

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;  
  
import mypackage.Calculator;  
  
class CalculatorTest {  
    @Test  
    void addition() {  
        Calculator calculator = new Calculator();  
        int expected = 2;  
        int actual = calculator.add(1, 1);  
        assertEquals(expected, actual);  
    }  
}
```



JUnit

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;  
  
import mypackage.Calculator;  
  
class CalculatorTest {  
    @Test  
    void addition() {  
        Calculator calculator = new Calculator();  
        int expected = 2;  
        int actual = calculator.add(1, 1);  
        assertEquals(expected, actual);  
    }  
}
```



JUnit

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        int expected = 2;
        int actual = calculator.add(1, 1);
        assertEquals(expected, actual);
    }
}
```



JUnit constructs

- **JUnit test**
 - A method only used for testing
- **Test suite**
 - A set of test classes to be executed together
- **Test annotations**
 - Define test methods (e.g., @Test, @Before)
 - JUnit uses the annotations to build the tests
- **Assertion methods**
 - Check expected result is the actual result
 - e.g., assertEquals, assertTrue, assertSame

JUnit annotations

- **@Test**
 - Identifies a test method
- **@Before**
 - Execute before each test
- **@After**
 - Execute after each test
- **@BeforeClass**
 - Execute once, before all tests in this class
- **@AfterClass**
 - Execute once, after all tests in this class

JUnit assertions

- `assertEquals(expected, actual)`

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;  
import mypackage.Calculator;
```

```
class CalculatorTest {  
    @Test  
    void addition() {  
        Calculator calculator = new Calculator();  
        int expected = 2;  
        int actual = calculator.add(1, 1);  
        assertEquals(expected, actual);  
    }  
}
```

JUnit assertions

- `assertEquals(message, expected, actual)`

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;  
import mypackage.Calculator;
```

```
class CalculatorTest {  
    @Test  
    void addition() {  
        Calculator calculator = new Calculator();  
        int expected = 2;  
        int actual = calculator.add(1, 1);  
        assertEquals("Expected value != actual", expected, actual);  
    }  
}
```

JUnit assertions

- `assertTrue`

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;  
import mypackage.Calculator;
```

```
class CalculatorTest {  
    @Test  
    void addition() {  
        Calculator calculator = new Calculator();  
        assertTrue(2 == calculator.add(1, 1));  
    }  
}
```

JUnit assertions

- `assertTrue`

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;  
import mypackage.Calculator;
```

```
class CalculatorTest {  
    @Test  
    void addition() {  
        Calculator calculator = new Calculator();  
        assertTrue("Can't do 1 + 1 :(", 2 == calculator.add(1, 1));  
    }  
}
```

JUnit assertions

```
import ...
```

```
class CalculatorTest {  
    Calculator calculator  
  
    @Before  
    void setup() {  
        calculator = new Calculator();  
    }  
  
    @Test  
    void additionBothPositive() {  
        assertEquals(2, calculator.add(1, 1));  
        assertEquals(5, calculator.add(4, 1));  
        assertEquals(5, calculator.add(2, 3));  
    }  
  
    ...  
}
```

Tasks for Week 11

- Submit weekly exercise on canvas before 23.59pm Saturday
- Continue assignment 3 and ask questions on Ed platform.
 - All assignments are individual assignments
 - Please note that: work must be done individually without consulting someone else's solutions in accordance with the University's "Academic Dishonesty and Plagiarism" policies

What are we going to learn next week?

- Creational Design Pattern
 - Singleton
- Structural Design pattern
 - Decorator and Façade

SOFT2201/COMP9201:

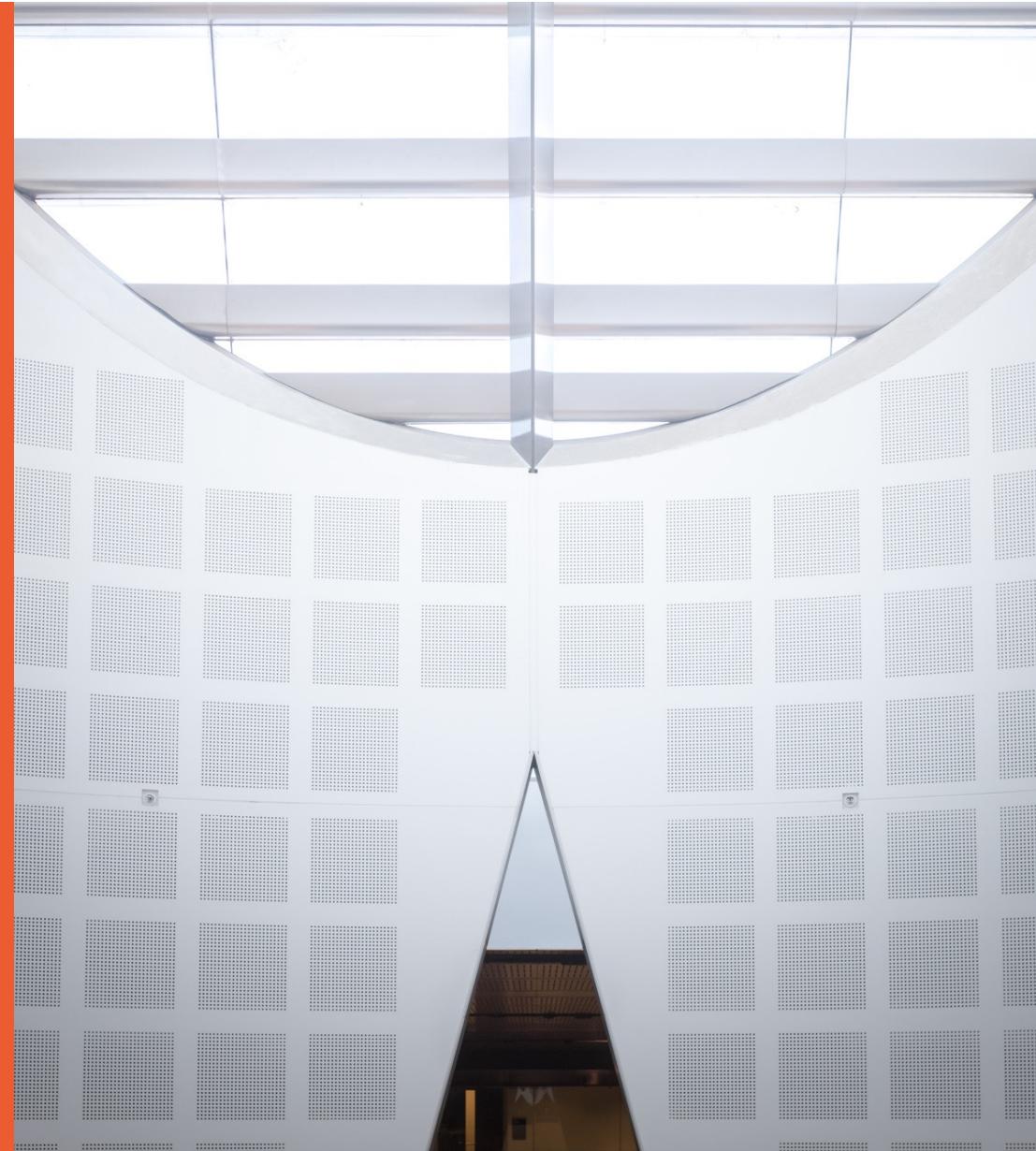
Software Design and

Construction 1

Singleton, Decorator, and

Façade

Dr Xi Wu
School of Computer Science



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

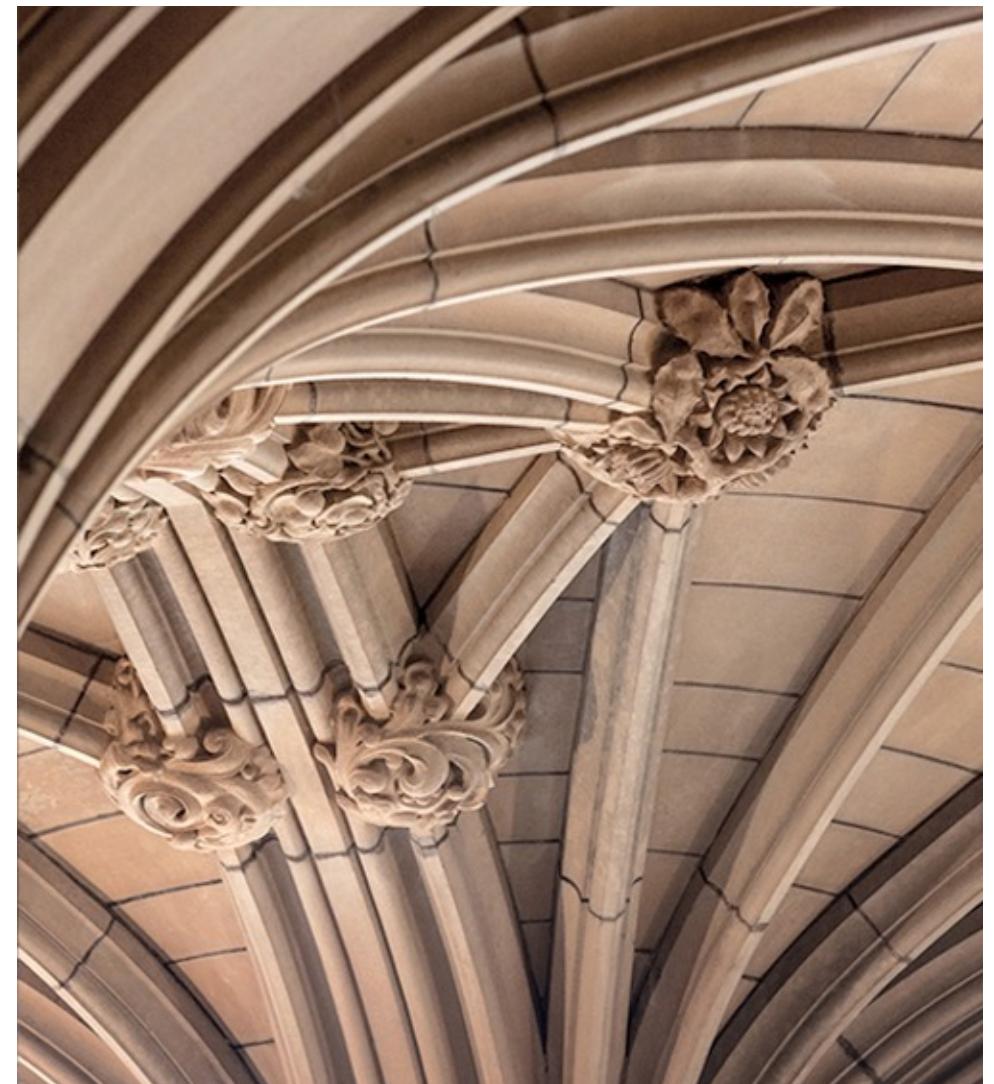
Do not remove this notice.

Agenda

- Creational Design Pattern
 - Singleton
- Structural Design Pattern
 - Decorator and Façade

Singleton Pattern

Object Creational

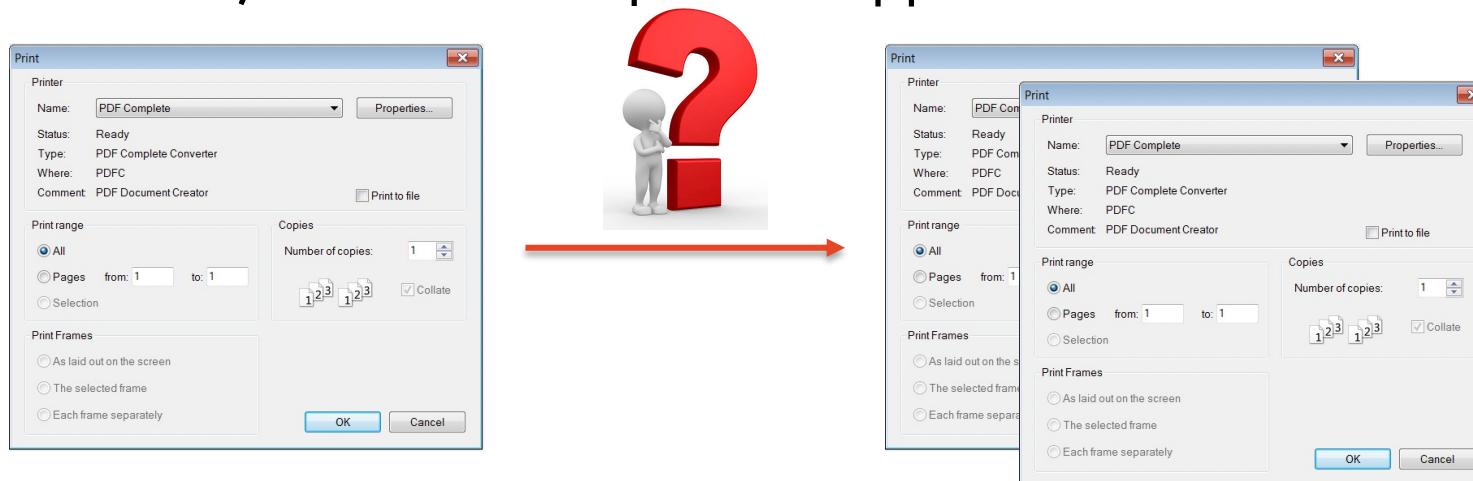


Creational Patterns (GoF)

Pattern Name	Description
Factory Method	Define an interface for creating an object, but let sub-class decide which class to instantiate (class instantiation deferred to subclasses)
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations
Prototype	Specify the kinds of objects to create using a prototype instance, and create new objects by copying this prototype
Singleton	Ensure a class only has one instance, and provide global point of access to it

Motivated Scenario

- Suppose you have finished your assignment report in your computer and would like to print it out.
 - The first time you click on “print”, it pops up a printing set up window
 - The second time you click on “print” without closing the previous printing window, what do we expect to happen?

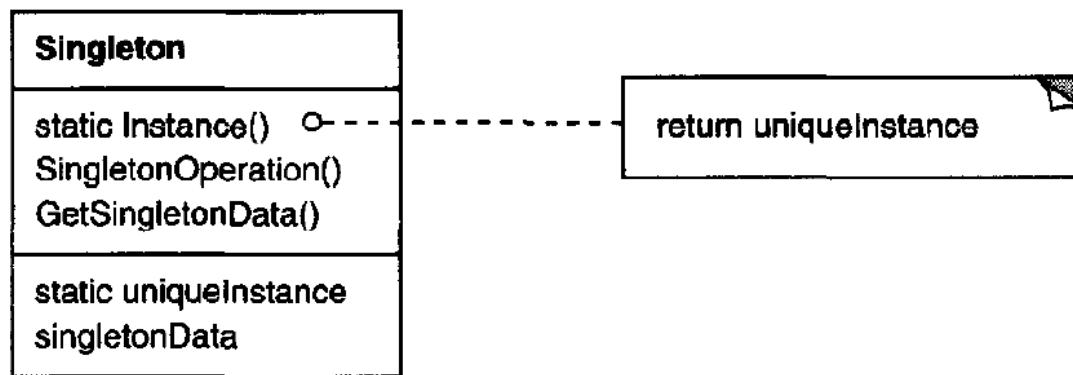


Singleton

- Intent
 - Ensure a class only has one instance, and provide a global point of access to it
- Motivation
 - Make the class itself responsible for keeping track of its sole instance (intercept requests to create new objects and provide a way to access its instance)

Singleton

- Structure



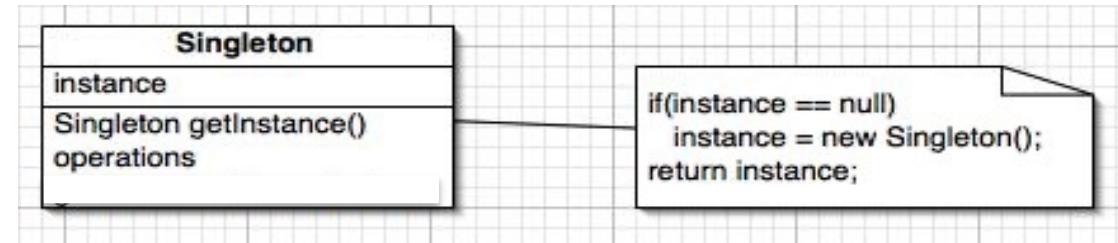
- Participants

- Defines an `instance()` operation that lets clients access its unique instance.
`instance()` is a class operation
 - May be responsible for creating its own unique instance

Singleton

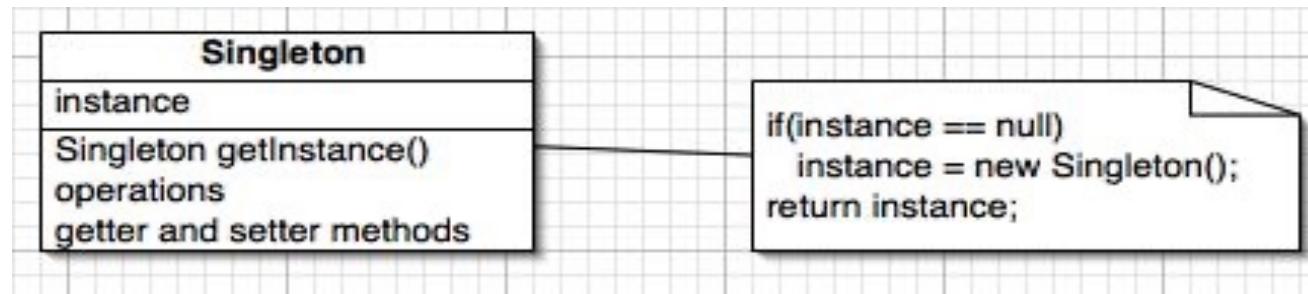
- **Collaboration**
 - Clients access a Singleton instance solely through Singleton's instance() operation.
- **Applicability**
 - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
 - The sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Singleton Implementation



```
public class Singleton {
    private static Singleton instance = null;
        // Private constructor to prevent direct
    initialisation.
    private Singleton() {
    }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
public class Client {
    ...
    Singleton single = Singleton.getInstance();
}
```

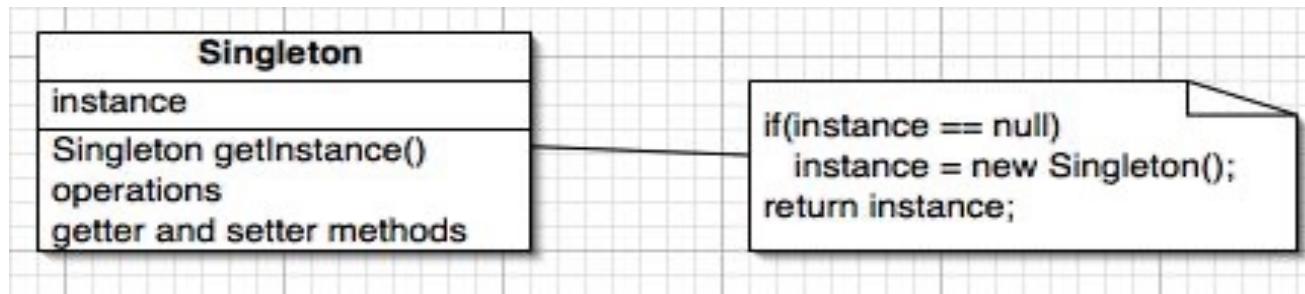
Singleton Example



```
public class ConfigurationReader {
    private static ConfigurationReader instance = null;

    private ConfigurationReader() { }
    public static ConfigurationReader getInstance() {
        if (instance == null) {
            instance = new ConfigurationReader();
        }
        return instance;
    }
}
public class Client {
    public boolean doStuff() {
        ...
        ConfigurationReader theOneAndOnlyConfigurationFileReader = ConfigurationReader.getInstance();
    }
}
```

Singleton Example

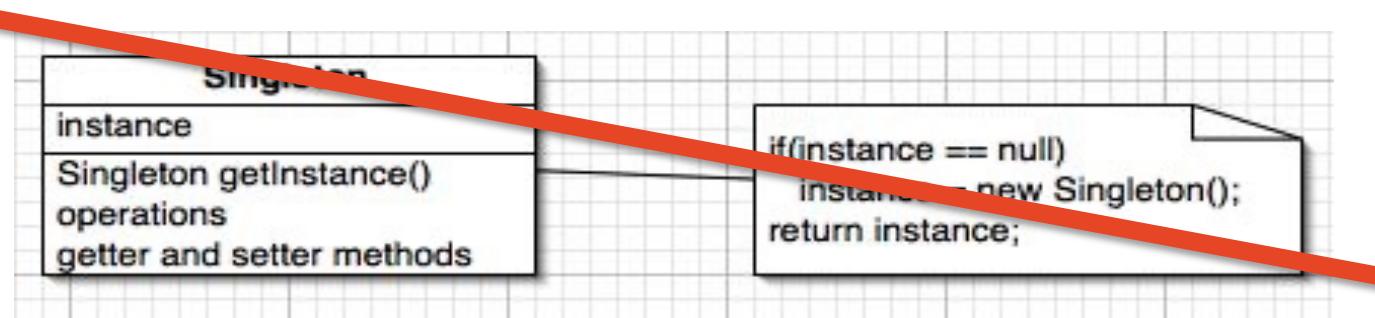


```
public class ConfigurationReader2 {
    private static ConfigurationReader2 instance = null;

    private ConfigurationReader2() { }
    public static ConfigurationReader2 getInstance() {
        if (instance == null) {
            instance = new ConfigurationReader2();
        }
        return instance;
    }
}
public class Client {
    public boolean doStuff() {
        ...
        ConfigurationReader theOneAndOnlyConfigurationFileReader = ConfigurationReader.getInstance();
        ConfigurationReader2 theOtherConfigurationFileReader = ConfigurationReader2.getInstance();
    }
}
```

Do you really always want one specific instance of one specific class?

Singleton Alternative Example

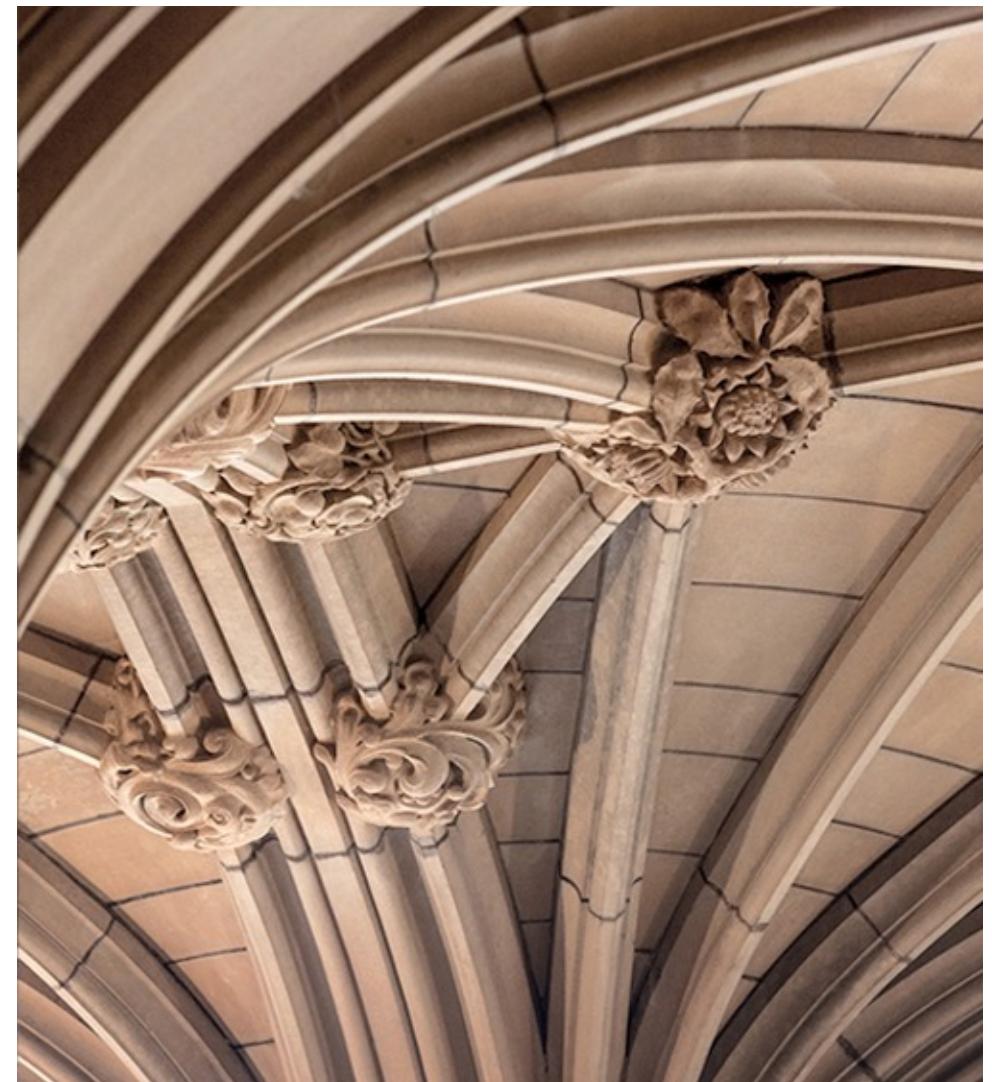


```
public class ConfigurationReader {
    public ConfigurationReader() { }
    // ...
}

public class Client {
    // Explicit dependency in interface.
    // The 'single' instance is passed around by reference
    public boolean doStuff(ConfigurationReader configurationReader) {
        // ...
    }
}
```

Decorator Pattern

Object Structural



Structural Patterns (GoF)

Pattern Name	Description
Adapter	Allow classes of incompatible interfaces to work together. Convert the interface of a class into another interface clients expect.
Decorator	Attach additional responsibilities to an object dynamically (flexible alternative to subclassing for extending functionality)
Façade	Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that simplifies subsystem use.

Motivated Scenario

- Suppose you are shopping in a famous store, which has a virtual clothing try on application.
 - T-Shirts, Trousers and Sneakers
 - Suit and Leather Shoes



Motivated Scenario



```
public class People {  
    2 usages  
    private String name;  
    1 usage  
    public People (String name){this.name = name;}  
    1 usage  
    public void wearTShirt() {System.out.print("T-Shirt ");}  
    1 usage  
    public void wearTrousers() {System.out.print("Trousers ");}  
    1 usage  
    public void wearSneakers() {System.out.print("Sneakers ");}  
    public void wearSuit() {System.out.print("Suit ");}  
    public void wearLeatherShoes() {System.out.print("Leather Shoes ");}  
    1 usage  
    public void show() {System.out.println(name + " is trying on:");}  
}
```

Client Perspective:

```
People people = new People( name: "Happy");  
people.show();  
people.wearTShirt();  
people.wearTrousers();  
people.wearSneakers();
```

Output:

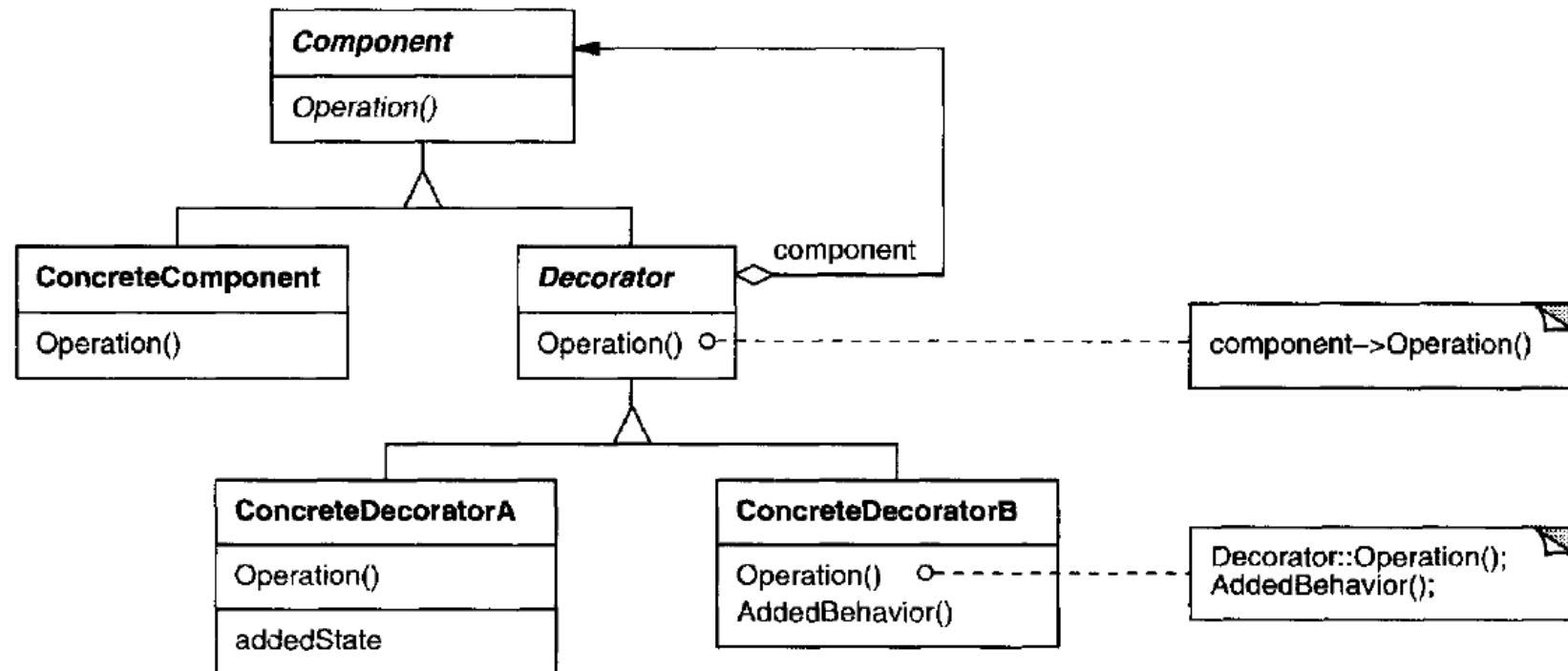
```
Happy is trying on:  
T-Shirt Trousers Sneakers
```

Question: I also want to try on Dress and Boots. What will happen here?

Decorator Pattern

- **Intent**
 - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality

Decorator – Structure



Decorator – Participants

- **Component**
 - Defines the interface for objects that can have responsibilities added to them dynamically
- **ConcreteComponent**
 - Defines an object to which additional responsibilities can be attached
- **Decorator**
 - Maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator**
 - Adds responsibilities to the component

Revisit the Motivated Example

```
public interface People {  
    7 usages 7 implementations  
    public void show();  
}
```

```
public class Women implements People{  
    2 usages  
    private String name;  
    1 usage  
    public Women(String name){this.name = name;}  
    7 usages  
    public void show() {System.out.println(name + " is trying on:");}  
}
```

```
public abstract class Cloth implements People {  
    2 usages  
    protected People people;  
    5 usages  
    public Cloth (People people){this.people = people;}  
    7 usages 5 overrides  
    public void show(){people.show();}  
}
```

Revisit the Motivated Example

```
public abstract class Cloth implements People {  
    2 usages  
    protected People people;  
    5 usages  
    public Cloth (People people){this.people = people;}  
    7 usages 5 overrides  
    public void show(){people.show();}  
}
```

```
public class TShirt extends Cloth{  
    1 usage  
    public TShirt(People people) {super(people);}  
    7 usages  
    public void show() {  
        super.show();  
        System.out.print("T-Shirt ");  
    }  
}
```

```
public class Trousers extends Cloth{  
    1 usage  
    public Trousers (People people) {super(people);}  
    7 usages  
    public void show() {  
        super.show();  
        System.out.print("Trousers ");  
    }  
}
```

```
public class Sneakers extends Cloth{  
    1 usage  
    public Sneakers(People people){super(people);}  
    7 usages  
    public void show(){  
        super.show();  
        System.out.print("Sneakers ");  
    }  
}
```

Revisit the Motivated Example

Client Perspective

```
People people = new Women( name: "Happy");  
  
TShirt tShirt = new TShirt(people);  
Trousers trousers = new Trousers(tShirt);  
Sneakers sneakers = new Sneakers(trousers);  
  
sneakers.show();
```

Question 1:

I also want to try on Dress and Boots.
What will happen here?

Question 2:

This system opens to Men and Baby.
What will happen here?

Output

```
Happy is trying on:  
T-Shirt Trousers Sneakers
```

Decorator Pattern – Why Not Inheritance?

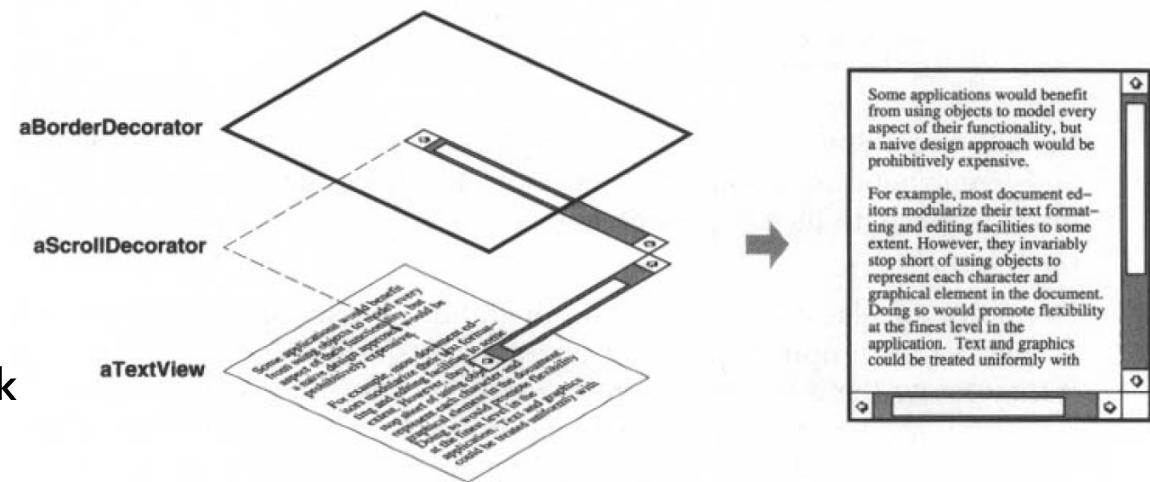
- We want to add responsibilities to individual objects, not an entire class
 - E.g., A GUI toolkit should let you add properties like borders or behaviors like scrolling to any user interface component
- Is adding responsibilities using inheritance a good design? For example, inheriting a border class puts a border every subclass instance
 - Why, why not?

Decorator Pattern – Why Not Inheritance?

- Adding responsibilities using inheritance restricts runtime change, and requires an implementation for every decoration.
 - This design is inflexible
 - The choice of border is made statically; a client cannot control how and when to decorate the component with a border
 - More flexible design is to enclose the component in another object that adds the border

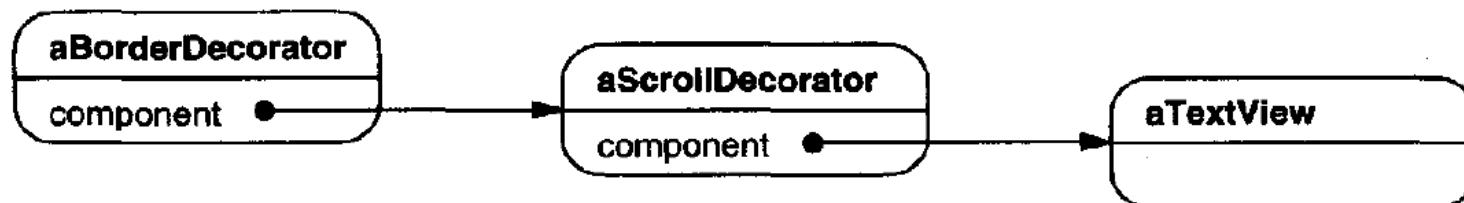
Decorator Pattern – Text Viewer Example

- TextView object has no scroll bars and border by default (not always needed)
- ScrollDecorator to add them
- BorderDecorator to add a thick black border around the TextView

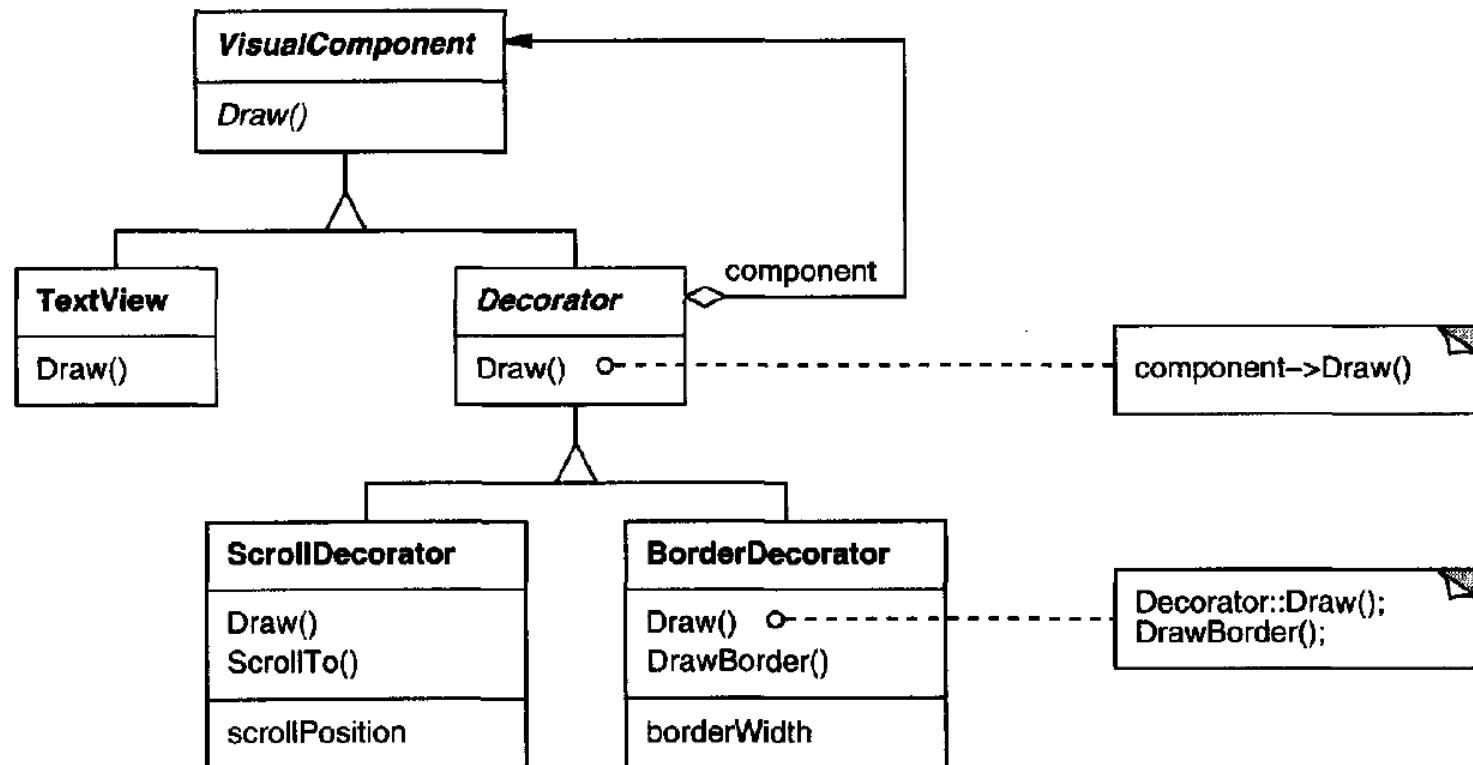


Decorator Pattern – Text Viewer Example

- Compose the decorators with the TextView to produce both the border and the scroll behaviours for the TextView



Decorator Pattern – Text Viewer Example



Decorator – Participants

- **Component** (*VisualComponent*)
 - Defines the interface for objects that can have responsibilities added to them dynamically
- **ConcreteComponent** (*TextView*)
 - Defines an object to which additional responsibilities can be attached
- **Decorator**
 - Maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator** (*BorderDecorator, ScrollDecorator*)
 - Adds responsibilities to the component

Decorator – Text Viewer Example

- VisualComponent is the abstract class for visual objects
 - It defines their drawing and event handling interface
- Decorator is an abstract class for visual components that decorate the other visual components
 - It simply forwards draw requests to its component; Decorator subclasses can extend this operation
- The ScrollDecorator and BorderDecorator classes are subclasses of Decorator
 - Can add operations for specific functionality (e.g., ScrollTo)

Decorator

- Collaborations
 - Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.
- Applicability
 - to add responsibilities to individual objects dynamically and transparently, without affecting other objects
 - For responsibilities that can be withdrawn
 - When extension by sub-classing is impractical
 - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.

Consequences (1)

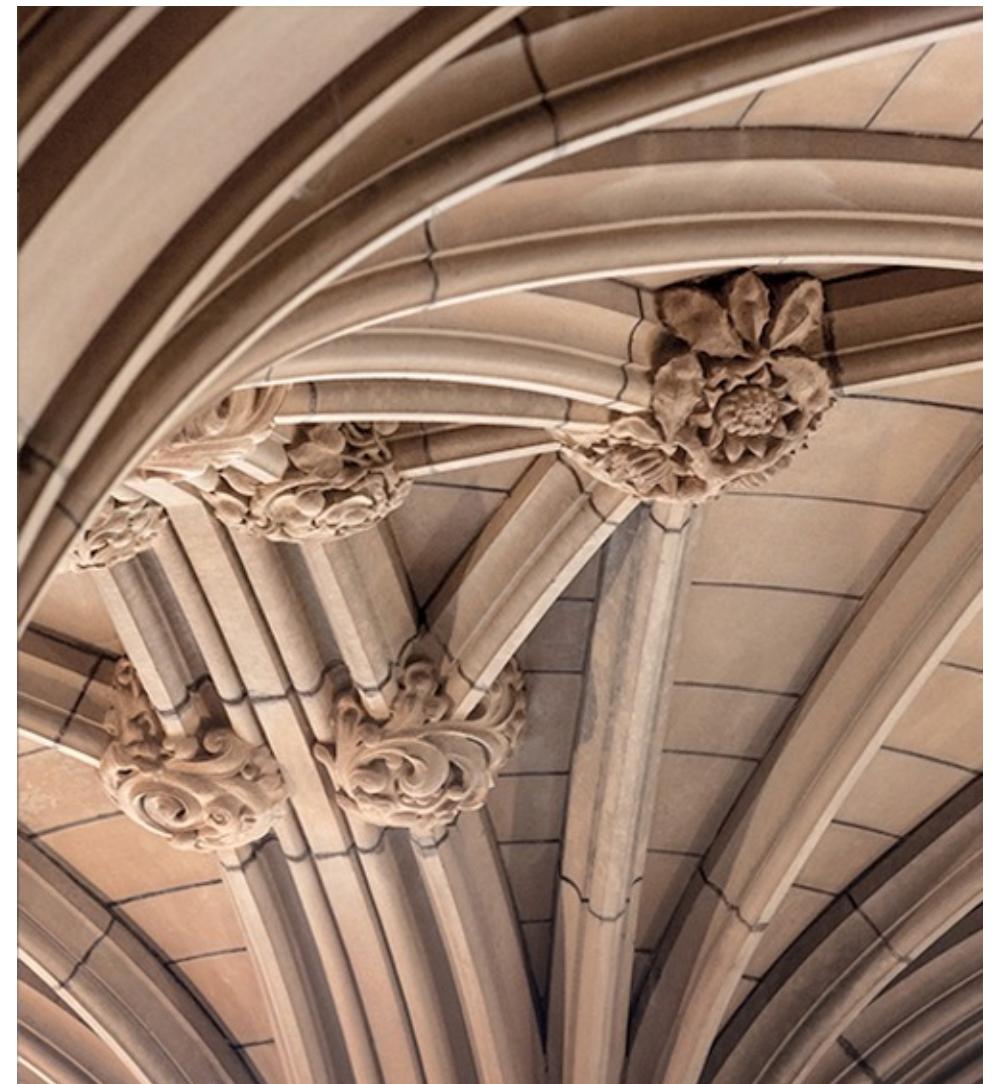
- More flexibility and less complexity than static inheritance
 - Can add and remove responsibilities to objects at run-time
 - Inheritance requires adding new class for each responsibility (increase complexity)
- Avoids feature-laden (heavily loaded) classes high up in the hierarchy
 - Defines a simple class and add functionality incrementally with Decorator objects – applications do not need to have un-needed features
 - You can define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions

Consequences (2)

- Decorator and its component are not identical
 - Decorated component is not identical to the component itself - you shouldn't rely on object identity when you use decorator
- Many little objects
 - Can become hard to learn and debug when lots of little objects that look alike
 - Still not difficult to customize by those who understand them

Façade Pattern

Object Structural



Structural Patterns (GoF)

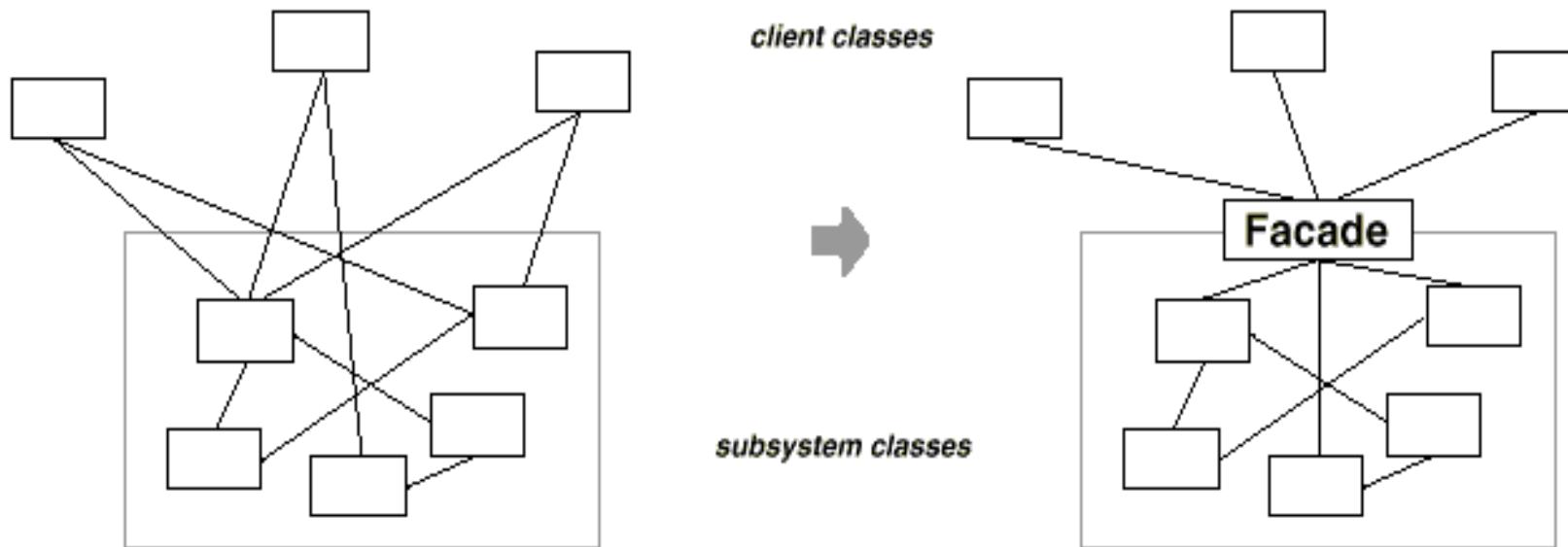
Pattern Name	Description
Adapter	Allow classes of incompatible interfaces to work together. Convert the interface of a class into another interface clients expect.
Decorator	Attach additional responsibilities to an object dynamically (flexible alternative to subclassing for extending functionality)
Façade	Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that simplifies subsystem use.

Motivated Scenario

- Suppose you are going to seek some advices from your UG/PG academic advisors. However, there are many academics and you might know who takes care of your major.



Façade Motivation

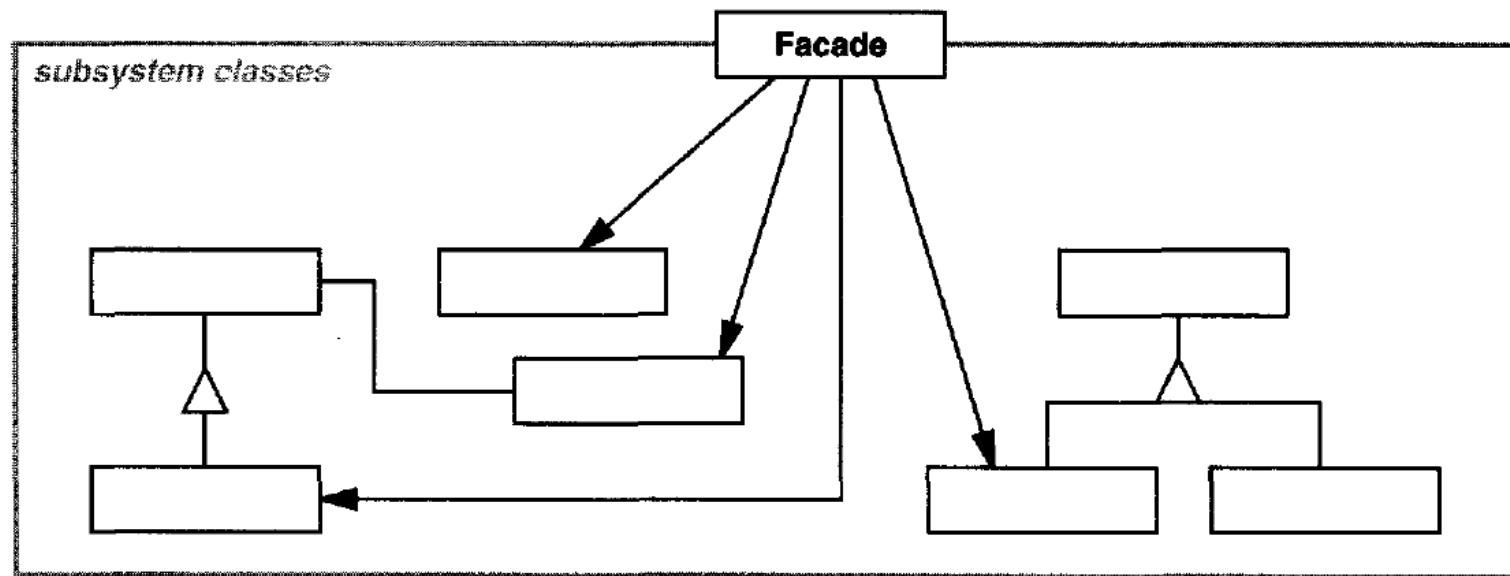


A **facade** object provides a single, simplified interface to the more general facilities of a subsystem

Façade Pattern

- Intent
 - Provide a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use
- Applicability
 - You want to provide a simple interface to a complex subsystem
 - There are many dependencies between clients and the implementation classes of an abstraction
 - You want to layer your subsystem. Façade would define an entry point to each subsystem level

Façade – Structure



Façade – Participants

- **Facade**
 - Knows which subsystem classes are responsible for a request.
 - Delegates client requests to appropriate subsystem objects.
- **Subsystem classes**
 - Implement subsystem functionality.
 - Handle work assigned by the Façade object
 - Have no knowledge of the facade; they keep no references to it.
- **Collaborations**
 - Clients communicate with the subsystem by sending requests to Façade, which forwards them to the appropriate subsystem object(s).
 - Although the subsystem objects perform the actual work, the façade may have to do work of its own to translate its interface to subsystem interfaces
 - Clients that use the facade don't have to access its subsystem objects directly

Façade

– Collaborations

- Clients communicate with the subsystem by sending requests to Façade, which forwards them to the appropriate subsystem object(s).
 - Although the subsystem objects perform the actual work, the façade may have to do work of its own to translate its interface to subsystem interfaces
- Clients that use the facade don't have to access its subsystem objects directly

Façade Pattern - Example

```
class subClass1 {  
    public void method1() {  
        // method body }  
}
```

```
class subClass2 {  
    public void method2() {  
        // method body }  
}
```

How about Client?

```
Façade façade = new Façade();  
façade.methodA();
```

```
class Façade {  
    subClass1 s1;  
    subClass2 s2;  
    public Façade() {  
        s1 = new subClass1();  
        s2 = new subClass2();  
    }  
    public void methodA() {  
        s1.method1();  
        s2.method2();  
    }  
}
```

Consequences

- Simplify the usage of an existing subsystem by defining your own interface
- Shields clients from subsystem components, reduce the number of objects that clients deal with and make the subsystem easier to use.
- Promote weak coupling between the subsystem and the clients
 - Vary the components of the subsystem without affecting its clients
 - Reduce compilation dependencies (esp. large systems) – when subsystem classes change
- Does not prevent applications from using subsystem classes if they need to. Choice between ease of use and flexibility.

Façade – A Brief Summary

- Name: **Façade**
- Problem: A common, unified interface to a disparate set of implementations or interfaces is required. There may be undesirable coupling to many things in the subsystem, or the implementation of the subsystem may change
- Solution: Define a single point of contact to the subsystem – a façade object that wraps the subsystem. It represents a single unified interface and is responsible for collaborating with the subsystem components.

Task for Week 12

- Submit weekly exercise on canvas before 23.59pm Saturday
- Well organize time for assignment 3

What are we going to learn on week 13?

- Unit Review

References

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Software Design and Construction 1

(SOFT2201/COMP9201)

Unit/Exam Review

Dr. Xi Wu

School of Computer Science



Agenda

- Assessment Review
- Exam Preparation & Info
- Exam Sample Questions
- Advice

Unit of Study Survey – Reminder

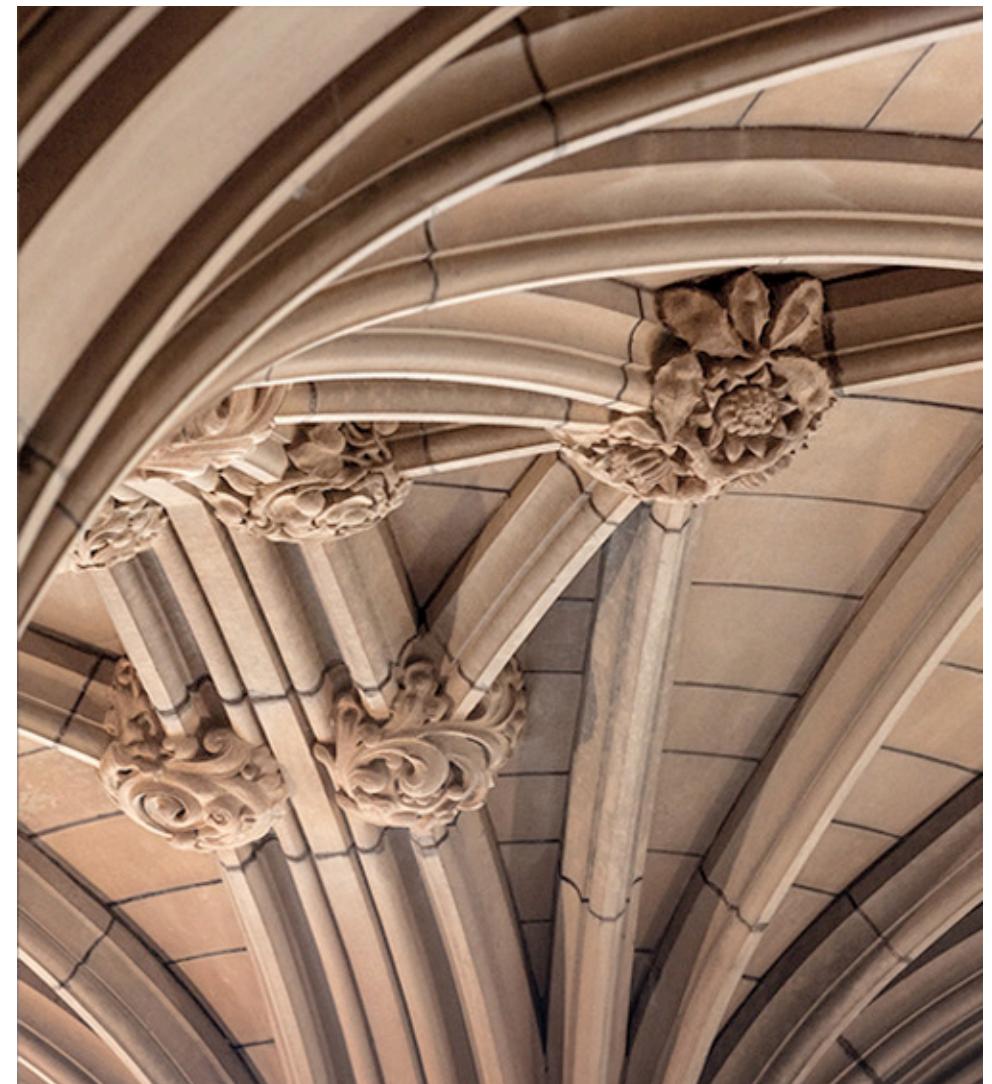
- ONLINE survey
 - <https://student-surveys.sydney.edu.au/students/>



Assessment Review



THE UNIVERSITY OF
SYDNEY



SOFT2201/COMP9201 Assessment

What (Assessment)*	When (due)	How	Value
Weekly Exercises	Weekly (2-12)	Online Exercises (Canvas)	10%
Assignment 1	Week 4	Submission on Canvas	5%
Assignment 2	Week 8	Submission on Canvas	15%
Assignment 3	Week 12	Submission on Canvas	20%
Online Final Exam	(see exam timetable)	Individual exam	50%

Passing this unit

- To pass this unit you need both two:
 - Get at least 50% overall¹
 - Get at least 40% for your final exam mark²

¹ A requirement of all University of Sydney units of study

² A requirement of all School of CS units of study

Exam Preparation & Info



THE UNIVERSITY OF
SYDNEY



Exam Preparation

- All materials covered (week 1 to week 13)**
 - Lecture materials**
 - Tutorial materials**
 - Weekly exercises**
 - Three Individual assignments (i.e., A1, A2 and A3)**
 - Supporting resources (adopted books, other references)**
- Note that, Libraries such as JavaFX and SimpleJson, are not examinable**

Exam Preparation Tools

- Questions can be asked to your tutors during w13 tutorial**
 - You are allowed on week 13 to attend as many tutorials as you want**
- Questions can be asked on Ed discussion forum**
 - Answering and discussing questions about the subject are very helpful to your own understanding**

Exam Information

- Check your personal exam timetable
- The exam is held on an Exam-specific Canvas site, to which you will be added by exam office.
- Type D -- short release take-home exam
- Exam is not supervised but is timed.

Please note that, the exam time is scheduled for AEDT

Exam Information

- Duration: 2 hours and 25 minutes (145minutes) including:
 - 2 hours writing time + 10 minutes reading time + 15 minutes uploading time
 - 10 minutes of reading time, but **you can start writing** whenever you are ready – you are strongly encouraged to use this time to carefully plan and structure your response before you start writing.
 - 15 minutes of upload time to allow you to upload your files as per your exam instructions. **Do NOT** treat this as extra writing time. The upload time must be used solely to save and upload your files correctly as per the exam instructions. Manage your time carefully. Check that you have saved and named your file correctly **and uploaded the correct file**. If your time runs out while you are uploading this is not considered a technical issue.

Exam Information

- The total marks for this exam is 100
- Two types of questions
 - Short Answer Questions (7)
 - Scenario-based Questions (2)
- Write your answers clearly and concisely
- Answer all questions
 - Read the question and answer what it asks you to answer

Exam Questions

- Short answer questions
 - Answers should be a short paragraph, or code snippets
 - Must use the proper or most accurate terminology

Exam Questions

- Scenario-based questions
 - Contain several sub-questions
 - May include name, briefly explain, design/model, implement
 - Based on contextual information
 - Answer as instructed
 - In the provided space
 - Clearly and concisely
 - If you **make any assumptions** about the problem beyond what is specified, make sure to **note these** in your answer

Exam Sample Questions



THE UNIVERSITY OF
SYDNEY



Exam Questions

- E.g., Describe pattern/principle X and compare it to p/p Y
 - X is a design pattern that ...
 - X is generally useful because ..., although it has the drawbacks ...
 - Y is a design pattern that ...
 - Y is generally useful because ..., although it has the drawbacks ...
 - X differs from Y because ...

Exam Questions

- E.g., Describe the key classes you would like to use in the design pattern and map the participants of the design pattern to the classes
 - The classes I would like to use is X, Y, Z, ...
 - Class X is ... (refer to participant name) of pattern ...
 - Class X is used for/to ... (specific roles) in pattern ...
 - Class Y is ...
 - Class Z is ...
 - ...

Exam Questions

- E.g., Apply a design pattern to solve problem Y
 - I will use X
 - X is a design pattern that ...
 - X is generally useful because ..., although it has the drawbacks ...
 - X is specifically applicable to Y because ...
 - ... implementation ... (if asked for)

Exam Questions

- E.g., Given the following code, state the pattern has been used on it. Map the participants to the classes in the code and explain the roles of participants in the pattern.
 - The pattern has been used in the code is ...
 - Class X is ... (refer to participant name) of pattern ...
 - Class X is used for/to ... (specific roles) in pattern ... in the given code

Exam Questions

- E.g., List all classifier names together with attributes and operations of the UML class diagram of system S. Identify the relationships among these classifiers.
 - Class X
 - Attributes: + variableName : String
 - Operations: + methodName (par: Z): double
 - Interface Y
 - Operations: + methodName (par: Z): double
 - Relationship:
 - Class X implements interface Y
 - Y is dependent on Z

Advice



THE UNIVERSITY OF
SYDNEY



Exam Technique

- **Plan how you will allocate time (wisely)**
 - Use “reading time” to read and understand the question
 - Plan time allocation to questions
 - Plan the order of answering questions
 - E.g., answer easier ones first
- **Answer everything (get the “easy marks”)**
 - Plan your answer, write answers relevant to the questions
 - Quality not quantity
- **Write clearly and efficiently**
 - If you are asked to list something, using bullet points is more efficient and easier to read

Exam Technique

- **Upload time**
 - Use “uploading time” to upload your answer
 - Check you are uploading the correct file
 - Check the file is uploaded
 - Check that you submitted the uploaded file

Exam Technique

- If you are uncertain about a question during the exam, answer to the best of your ability. Do not contact the unit coordinator or other teaching staff during the exam.
 - Any assumptions or interpretations must be noted together with your answers
- If you encounter technical difficulties during the exam that you cannot fix, email canvas.tests@sydney.edu.au immediately to allow them enough time to respond and assist before the submission closes. Please include your unit of study code, SID and contact number.
- Further information could be found on the home page of Canvas Exam site (i.e., the one you are going to be added by the exam office)

Good Luck!



THE UNIVERSITY OF
SYDNEY

