

بسمه تعالی



دانشگاه صنعتی امیرکبیر
(پلی‌تکنیک تهران)

دانشکده مهندسی کامپیوتر

مهندسی نرم افزار ۱

Software Engineering 1

دوره کارشناسی

نیمسال اول ۱۴۰۰-۱۴۰۱

ساعت درس: روزهای شنبه و دوشنبه ساعت ۱۵:۰۰ تا ۱۶:۱۵

پیش نیاز: برنامه‌نویسی پیشرفته

سایت درس: «مهندسی نرم افزار ۱ - گروه ۱ - کلباسی» در سامانه مودل (courses.aut.ac.ir)

استاد درس: امیر کلباسی

دفتر کار: ساختمان دانشکده مهندسی کامپیوتر، طبقه ۳ (راست، راست، چپ، رو به رو)

ساعت مراجعه دانشجویان: روزهای دوشنبه ساعت ۱۶:۳۰ تا ۱۸:۰۰ یا با قرار قبلی

شماره تلفن دفتر کار: ۰۹۱۱-۵۱۱۴-۶۴۵۴

آدرس پست الکترونیکی: kalbasi@aut.ac.ir

دستیاران تدریس: آقایان مهدی خزاعی نژاد و مهدی جعفری

اهداف درس:

هدف کلی این درس آشنایی با نکات مهندسی و فرآیندهای توسعه نرم افزار است که کلیه مراحل تولید و توسعه نرم افزار را در بر می‌گیرد. در این درس فرآیندهای برنامه ریزی شده و چابک آموزش داده می‌شود و فعالیت‌های مهم در توسعه نرم افزار همچون مشخص کردن نیازمندی‌ها، تحلیل و طراحی، پیاده‌سازی، ارزیابی و تکامل سیستم مورد مطالعه قرار می‌گیرد.

برنامه درس:

موضوع درس	هفته
آشنایی با مهندسی نرم افزار، طبیعت نرم افزار	اول
ساختار فرآیند تولید نرم افزار	دوم
فرآیندهای ساخت یافته	سوم
فرآیند چابک	چهارم
ادامه فرآیند چابک	پنجم
مهندسی نیازمندیها	ششم
مدل سازی سناریو	هفتم
مدل سازی شیء گرا	هشتم
مدل سازی رفتاری	نهم
معماری نرم افزار	دهم
ادامه معماری نرم افزار	یازدهم
طراحی نرم افزار	دوازدهم
ادامه طراحی نرم افزار	سیزدهم
ارزیابی نرم افزار	چهاردهم
تکامل نرم افزار	پانزدهم
مرور مطالب و جمع بندی	شانزدهم

ارزیابی درس:

نمره	بخش
۳	آزمون میان ترم
۷	آزمون پایان ترم
۷	تمرین‌ها و پروژه
۱	حضور و مشارکت در مباحث کلاس
۲	آزمون کلاسی (معمولاً به صورت اتفاقی)
۲۰	مجموع

منابع درس:

- Software Engineering, 10th Edition, by Ian Sommerville, Pearson Publication, 2015.
- Software Engineering: A Practitioner's Approach, 8th Edition, by Roger S. Pressman, Bruce Maxim, McGraw-Hill Education, 2014.
 - نسخه ۹ این کتاب به تازگی توسط انتشارات آتی نگر ترجمه شده است.
- Engineering Software Products: An Introduction to Modern Software Engineering, 1st edition, by Ian Sommerville, Pearson Publication, 2019.
- UML Distilled: A Brief Guide to the Standard Object Modeling Language 3rd Edition, by Martin Fowler, Addison-Wesley Professional, 2003.

قوانين و مقررات درس:

حضور و غیاب

حضور به موقع دانشجویان در تمام جلسات درس الزامی است. تعداد غیبت‌های دانشجو نباید از ۳/۱۶ مجموع جلسات درس تجاوز کند و غیبت بیش از ۳/۱۶ منجر به گرفتن نمره صفر برای درس خواهد شد. مقدار حضور و مشارکت شما در مباحث درس تعیین کننده نمره شما برای قسمت «حضور و مشارکت در مباحث کلاس» از جدول بالا خواهد بود.

مشارکت پیوسته در کلاس

با توجه به مجازی بودن کلاس‌ها، در طول زمان کلاس باید به صورت پیوسته در کلاس حضور داشته باشید (و نه فقط وارد شده باشید).

کار شرافتمندانه

لازم است دانشجویان اصول کار شرافتمندانه را همیشه رعایت کنند. کار تحويل داده شده حتماً و فقط باید کار انجام داده شده برای این درس در این ترم و توسط خود دانشجو باشد. در غیر این صورت با دانشجو برخورد انضباطی خواهد شد. در صورت هرگونه تقلب هم با تقلب کننده و هم با تقلب شونده برخورد خواهد شد.



Chapter 1- Introduction

Software engineering



- ✧ The **economies of ALL developed and many developing nations** are dependent on software.
- ✧ More and more systems are software controlled
- ✧ Software engineering is concerned with **theories, methods and tools** for professional software development.
- ✧ Expenditure on software represents a **significant fraction of GNP** in all developed countries.

Largest Public Companies in the World (2000)



This *Financial Times*-based list is up to date as of 31 March 2001.^[79]

Rank	Name	Headquarters	Primary industry	Market value (USD million)
1	General Electric	United States	Conglomerate	477,406
2	Cisco Systems	United States	Networking hardware	304,699
3	Exxon Mobil	United States	Oil and gas	286,367
4	Pfizer	United States	Health care	263,996
5	Microsoft	United States	Software industry	258,436
6	Wal-Mart	United States	Retail	250,955
7	Citigroup	United States	Banking	250,143
8	Vodafone	United Kingdom	Telecommunications	227,175
9	Intel Corporation	United States	Computer hardware	227,048
10	Royal Dutch Shell	The Netherlands	Oil and gas	206,340

Source: Wikipedia

Largest Public Companies in the World (2010)



This *Financial Times* Global 500-based list is up to date as of December 31, 2009. Indicated changes in market value are relative to the previous quarter.

Rank	First quarter ^[59]	Second quarter ^[60]	Third quarter ^[61]	Fourth quarter ^[62]
1	Exxon Mobil ▼336,527	PetroChina ▲366,662.9	Exxon Mobil ▼329,725	PetroChina ▲353,140.1
2	PetroChina ▲287,185	Exxon Mobil ▲341,140.3	PetroChina ▼325,097.5	Exxon Mobil ▼323,717.1
3	Wal-Mart ▼204,365	ICBC ▲257,004.4	ICBC ▼237,951.5	Microsoft ▲270,635.4
4	ICBC ▲187,885	Microsoft ▲211,546.2	Microsoft ▲229,630.7	ICBC ▲268,956.2
5	China Mobile ▼174,673	China Mobile ▲200,832.4	HSBC ▲198,561.1	Wal-Mart ▲203,653.6
6	Microsoft ▼163,320	Wal-Mart ▼188,752.0	China Mobile ▼195,680.4	China Construction Bank ▲201,436.1
7	AT&T ▼148,511	China Construction Bank ▲182,186.7	Wal-Mart ▲189,331.6	BHP Billiton ▲201,248
8	Johnson & Johnson ▼145,481	Petrobras ▲165,056.9	Petrobras ▲189,027.7	HSBC ▲199,254.9
9	Royal Dutch Shell ▼138,999	Johnson & Johnson ▲156,515.9	China Construction Bank ▲186,816.7	Petrobras ▲199,107.9
10	Procter & Gamble ▼138,013	Royal Dutch Shell ▲156,386.7	Royal Dutch Shell ▲175,986.1	Apple Inc. ▲189,801.7

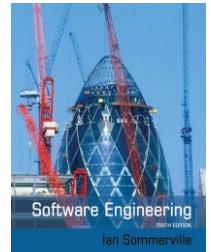
Source: Wikipedia

30/10/2014

Chapter 1 Introduction

4

Largest Public Companies in the World (2021)



This list is up to date as of June 30, 2021. Indicated changes in market value are relative to the previous quarter.

Rank		First quarter		Second quarter
1	🇺🇸	Apple ▼2,050,000 ^[16]	🇺🇸	Apple ▲2,286,000 ^[16]
2	🇺🇸	Microsoft ▲1,778,000 ^[17]	🇺🇸	Microsoft ▲2,040,000 ^[17]
3	🇺🇸	Amazon ▼1,558,000 ^[18]	🇺🇸	Amazon ▲1,735,000 ^[18]
4	🇺🇸	Alphabet ▲1,395,000 ^[19]	🇺🇸	Alphabet ▲1,680,000 ^[19]
5	🇺🇸	Facebook ▲838,720 ^[20]	🇺🇸	Facebook ▲985,920 ^[20]
6	🇨🇳	Tencent ▲766,970 ^[21]	🇨🇳	Tencent ▼721,460 ^[21]
7	🇺🇸	Tesla ▼641,110 ^[22]	🇺🇸	Tesla ▲654,780 ^[22]
8	🇨🇳	Alibaba Group ▼615,010 ^[23]	🇺🇸	Berkshire Hathaway ▲637,280 ^[24]
9	🇹🇼	TSMC ▲613,410 ^[25]	🇹🇼	TSMC ▲623,160 ^[25]
10	🇺🇸	Berkshire Hathaway ▲590,050 ^[24]	🇨🇳	Alibaba Group ▲615,140 ^[23]

Which industry is growing the fastest?

Source: Wikipedia

Software costs



- ✧ Software costs often dominate computer system costs.
The costs of software on a PC are often greater than the hardware cost.
- ✧ Software costs more to maintain than it does to develop.
For systems with a long life, maintenance costs may be several times development costs.
- ✧ Software engineering is concerned with cost-effective software development.

Software project failure



✧ *Increasing system complexity*

- As new software engineering techniques help us to build larger, more complex systems, the demands change. **Systems have to be built and delivered more quickly; larger, even more complex systems are required; systems have to have new capabilities that were previously thought to be impossible.**

✧ *Failure to use software engineering methods*

- It is fairly easy to write computer programs without using software engineering methods and techniques. **Many companies have drifted into software development as their products and services have evolved. They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be.**



Professional software development

Frequently asked questions about software engineering



Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.

Frequently asked questions about software engineering



Question	Answer
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.
What differences has the web made to software engineering?	The web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse.

Software products



✧ Generic products

- Stand-alone systems that are **marketed and sold to any customer** who wishes to buy them.
- Examples – PC software such as graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.

✧ Customized products

- Software that is **commissioned by a specific customer** to meet their own needs.
- Examples – embedded control systems, air traffic control software, traffic monitoring systems.

Product specification



✧ Generic products

- The specification of what the software should do is **owned by the software developer** and **decisions on software change are made by the developer**.

✧ Customized products

- The specification of what the software should do is **owned by the customer** for the software and they **make decisions on software changes** that are required.

Essential attributes of good software



Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use.

Software engineering



✧ Software engineering

- Engineering discipline
 - **Using appropriate theories and methods** to solve problems bearing in mind organizational and financial constraints.
- Concerned with all aspects of software production
 - from the early stages of system specification through to maintaining the system after it has gone into use.
 - **Not just technical process of development.** Also project management and the development of tools, methods etc. to support software production.

Importance of software engineering



- ✧ Individuals and society **rely** on advanced software systems.
- ✧ Need to produce reliable and trustworthy systems **economically and quickly**.
- ✧ It is usually cheaper
 - To use software engineering methods and techniques for software systems
 - Rather than just write the programs
 - For most types of system, the majority of costs are the costs of changing the software after it has gone into use.

Software process activities



✧ Software specification

- Customers and engineers define the software that is to be produced and the constraints on its operation.

✧ Software development

- The software is designed and programmed.

✧ Software validation

- The software is checked to ensure that it is what the customer requires.

✧ Software evolution

- The software is modified to reflect changing customer and market requirements.

General issues that affect software



✧ Heterogeneity

- Increasingly, systems are required to **operate as distributed systems** across networks that include **different types of computer and mobile devices**.

✧ Business and social change

- Business and society are **changing incredibly quickly** as emerging economies develop and **new technologies** become available. They need to be able to change their existing software and to rapidly develop new software.

General issues that affect software



✧ Security and trust

- As software is intertwined with **all aspects of our lives**, it is essential that we can trust that software.

✧ Scale

- Software has to be developed across a very wide range of scales, from **very small embedded systems** in portable or wearable devices through to **Internet-scale, cloud-based systems** that serve a global community.

Software engineering diversity



- ✧ Many different types of software
 - No universal set of software techniques

- ✧ The software engineering methods and tools
 - Depend on the type of application being developed
 - Requirements of the customer
 - The background of the development team.



Application types

✧ Stand-alone applications

- These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network.

✧ Interactive transaction-based applications

- Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.

✧ Embedded control systems

- These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system.



Application types

✧ Batch processing systems

- These are business systems that are designed to **process data in large batches**. They process large numbers of individual inputs to create corresponding outputs.

✧ Entertainment systems

- These are systems that are primarily for personal use and which are **intended to entertain the user**.

✧ Systems for modelling and simulation

- These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.

Application types



✧ Data collection systems

- These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing.

✧ Systems of systems

- These are systems that are composed of a number of other software systems.

Software engineering fundamentals



- ✧ Some **fundamental principles** apply to **all types** of software system, irrespective of the development techniques used:
 - Systems should be **developed using a managed and understood development process**. Of course, different processes are used for different types of software.
 - **Dependability and performance** are important for all types of system.
 - Understanding and managing the **software specification and requirements** (what the software should do) are important.
 - Where appropriate, you **should reuse** software that has already been developed rather than write new software.

Internet software engineering



- ✧ The Web is now a **platform for running application** and organizations are **increasingly developing web-based systems** rather than local systems.
- ✧ Web services allow **application functionality to be accessed over the web**.
- ✧ Cloud computing is an approach to the provision of computer services where applications run remotely on the ‘cloud’.
 - Users do not buy software but **pay according to use**.

Web-based software engineering



- ✧ Web-based systems
 - Complex distributed systems
 - The fundamental **principles of software engineering are as applicable to them** as they are to any other types of system.
- ✧ **Software reuse** is the dominant approach for constructing web-based systems.
 - When building these systems, you think about how you can **assemble** them from pre-existing software **components** and systems

Key points



- ✧ Software engineering is an engineering discipline that is concerned with all aspects of software production.
- ✧ Essential software product attributes are maintainability, dependability and security, efficiency and acceptability.
- ✧ The high-level activities of specification, development, validation and evolution are part of all software processes.

Key points



- ✧ There are **many different types of system** and each requires appropriate software engineering **tools and techniques** for their development.

- ✧ The **fundamental ideas of software engineering** are applicable to all types of software system.

Room/Group leader



- ✧ In each room leader will be chosen based on the following:

Assume:

$$rem_i = id_i \% 7 \quad i = 1 \dots n$$

The student with smallest rem_i is the leader. If two or more students have similar rem_i , the student with smaller id will be the leader.

Group discussion



- ✧ How hardware and software are different?
- ✧ Hardware can be any tangible item not just computers.
- ✧ 12 min to discuss in your group
- ✧ Leaders will let us know what the group thinks

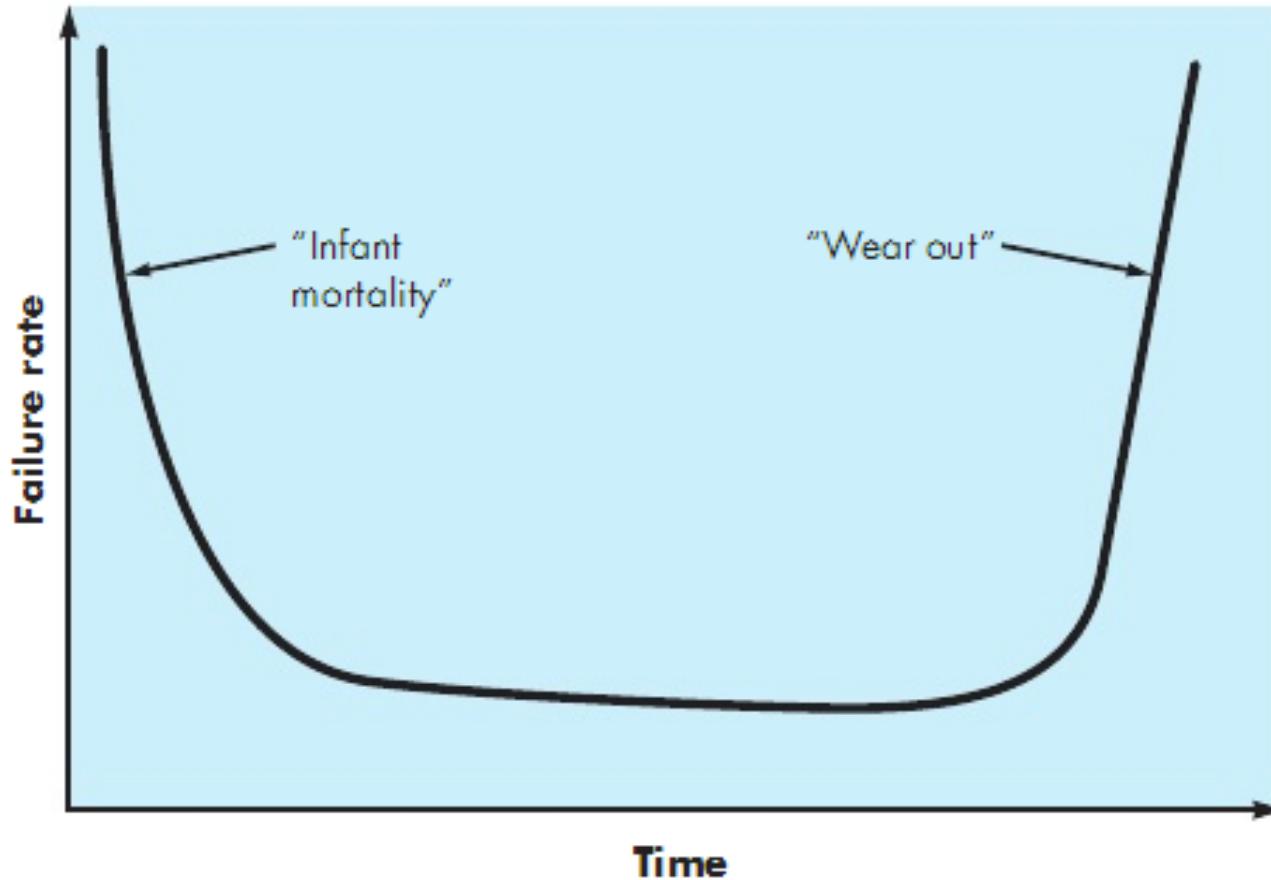
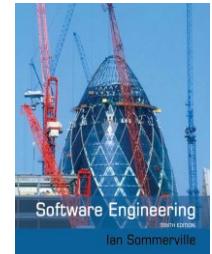
Hardware vs. software



✧ Hardware systems

- Physical
- Cost: mainly material cost and manufacturing cost
- Wear out
 - Dust, vibrations, extreme temperatures, abuse, etc.

Hardware



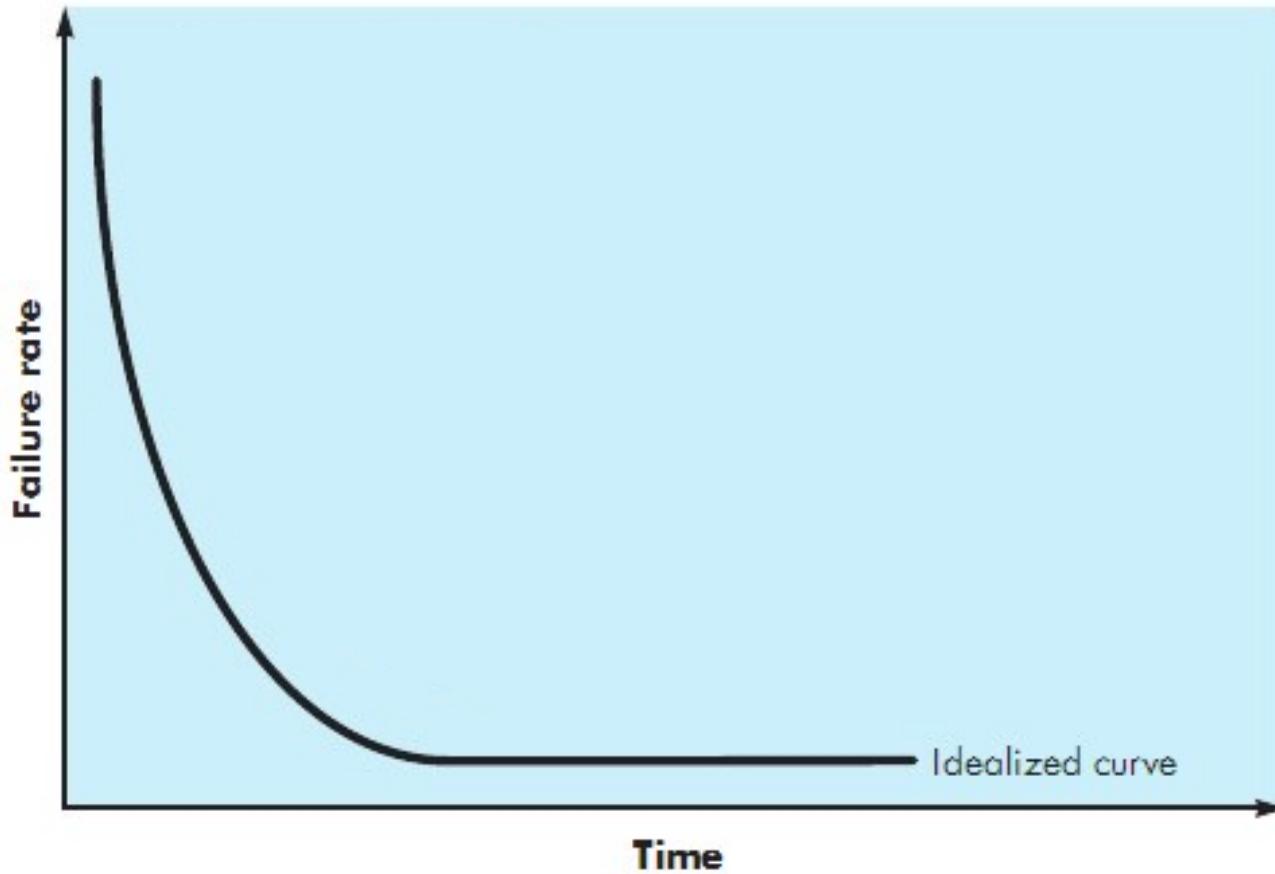
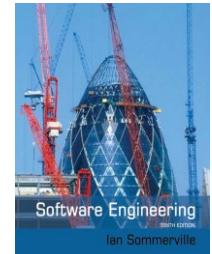
Hardware vs. software



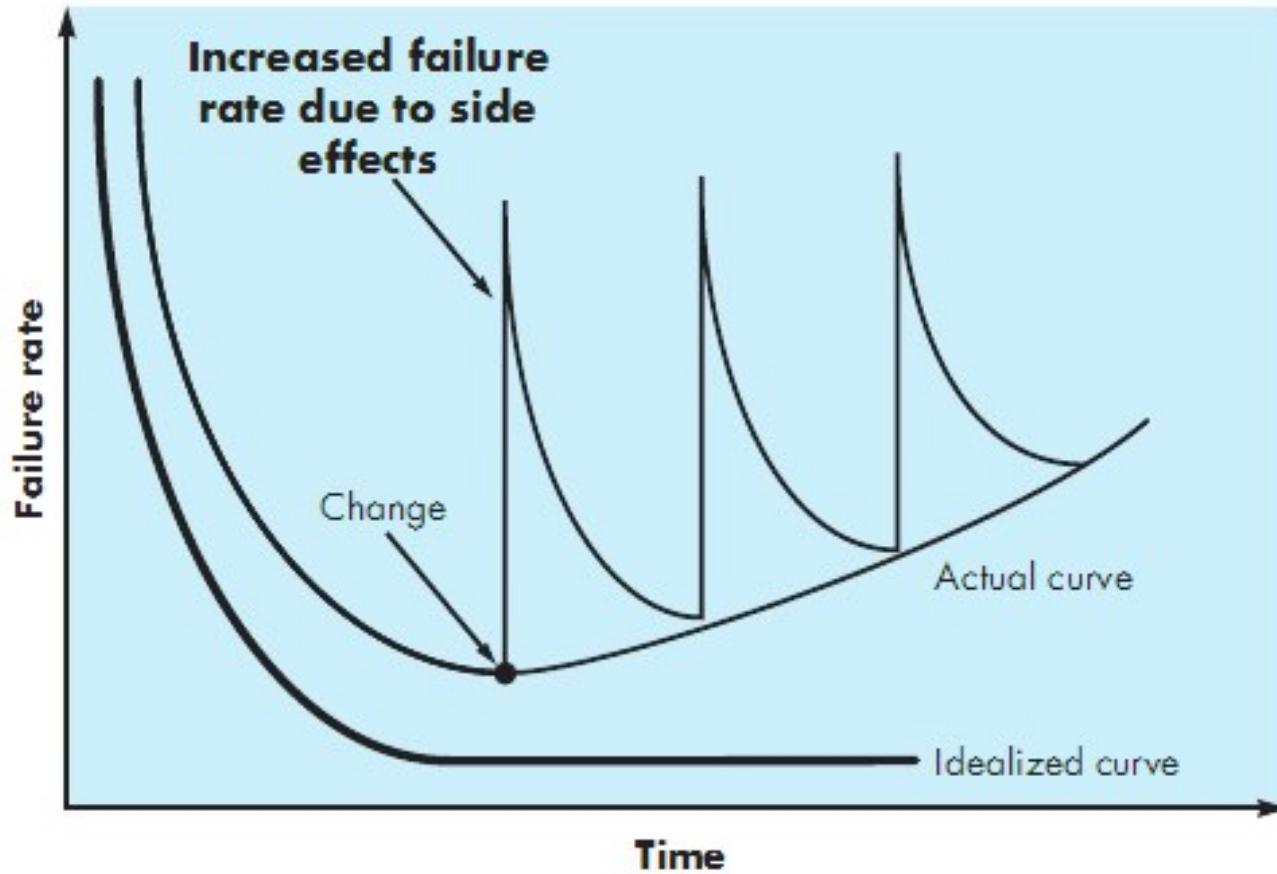
✧ Software

- Logical product
 - Itself is a product
 - Vehicle for other software products
- Not affected by environmental maladies
- Doesn't wear out
 - But will deteriorate

Software



Software





Software engineering ethics

Software engineering ethics



- ✧ Software engineering involves **wider responsibilities** than simply the **application of technical skills**.
- ✧ Software engineers must behave in an **honest and ethically responsible way** if they are to **be respected as professionals**.
- ✧ Ethical behaviour is **more than simply upholding the law** but involves following a set of **principles that are morally correct**.

Issues of professional responsibility



✧ Confidentiality

- Engineers should normally **respect the confidentiality** of their employers or clients irrespective of **whether or not a formal confidentiality agreement** has been signed.

✧ Competence

- Engineers **should not misrepresent their level of competence**. They should not knowingly accept work which is above their competence.

Issues of professional responsibility



✧ Intellectual property rights

- Engineers **should be aware of local laws governing the use of intellectual property such as patents, copyright, etc.** They should be careful to ensure that the intellectual property of employers and clients is protected.

✧ Computer misuse

- Software engineers **should not use their technical skills to misuse other people's computers.** Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

ACM/IEEE Code of Ethics



- ✧ The **professional societies** in the US have cooperated to produce a **code of ethical practice**.
- ✧ Members of these organisations **sign up to the code of practice** when they join.
- ✧ The Code contains **eight Principles** related to the **behaviour of and decisions** made by **professional software engineers**, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession.

Rationale for the code of ethics



- Computers have a *central and growing role* in commerce, industry, government, medicine, education, entertainment and society at large. *Software engineers* are those who *contribute by direct participation* or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems.
- Because of *their roles in developing software systems*, software engineers have *significant opportunities to do good or cause harm*, to *enable others to do good or cause harm*, or *to influence others to do good or cause harm*. To ensure, as much as possible, that *their efforts will be used for good*, software engineers must commit themselves to *making software engineering a beneficial and respected profession*.

The ACM/IEEE Code of Ethics



Software Engineering Code of Ethics and Professional Practice

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. **In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:**

Ethical principles



1. PUBLIC - Software engineers shall act **consistently with the public interest**.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is **in the best interests of their client and employer consistent with the public interest**.
3. PRODUCT - Software engineers shall ensure that **their products** and related modifications **meet the highest professional standards possible**.
4. JUDGMENT - Software engineers shall **maintain integrity and independence** in their professional judgment.
5. MANAGEMENT - Software engineering **managers and leaders** shall subscribe to and **promote an ethical approach** to the management of software development and maintenance.
6. PROFESSION - Software engineers shall advance **the integrity and reputation of the profession** consistent with the public interest.
7. COLLEAGUES - Software engineers **shall be fair to and supportive** of their colleagues.
8. SELF - Software engineers shall participate in **lifelong learning regarding the practice of their profession** and shall promote an ethical approach to the practice of the profession.



Chapter 2 – Software Processes

Topics covered



- ✧ Software process models
- ✧ Process activities
- ✧ Coping with change
- ✧ Process improvement

فرایند



❖ فرهنگ فارسی معین:

(فَئَ) مجموعه عملیات و مراحل لازم برای رسیدن به یک هدف مشخص

مثال: فرایند تعویض لاستیک پنچر شده، فرایند پخت دلمه، فرایند ثبت نام دانشجو، فرایند تولید نرم افزار

The software process



- ✧ A **structured set of activities** required to develop a software system.

- ✧ Many different software processes but all involve:
 - Specification – defining what the system should do;
 - Design and implementation – defining the organization of the system and implementing the system;
 - Validation – checking that it does what the customer wants;
 - Evolution – changing the system in response to changing customer needs.

Software process descriptions



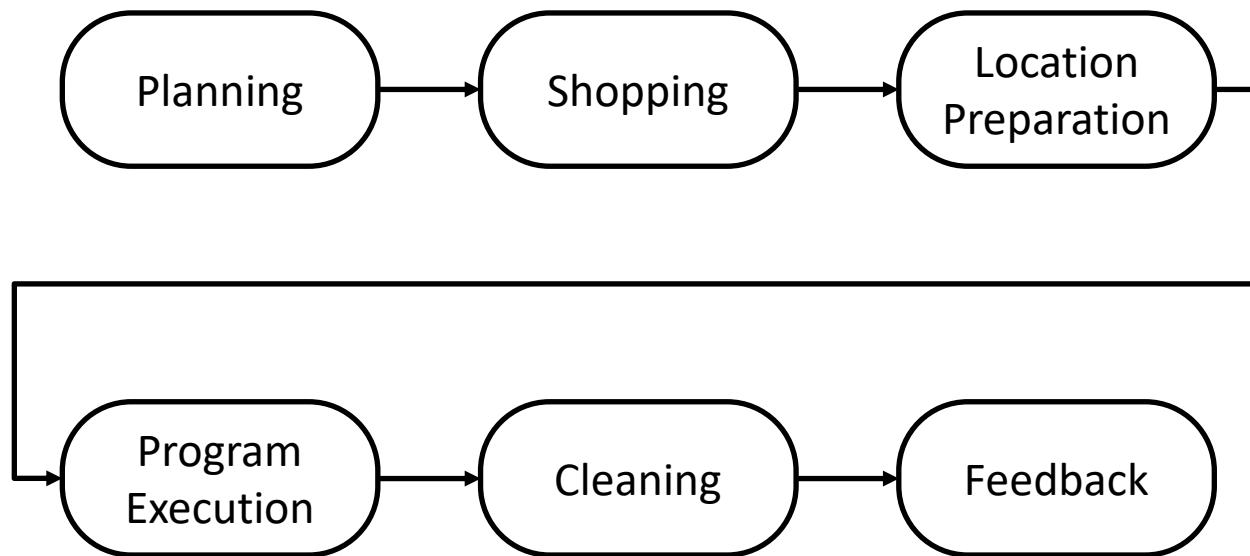
- ✧ When we describe and discuss processes, we usually **talk about the activities** in these processes such as specifying a data model, designing a user interface, etc. **and the ordering of these activities.**
- ✧ Process descriptions may also include:
 - **Products**, which are the outcomes of a process activity;
 - **Roles**, which reflect the responsibilities of the people involved in the process;
 - **Pre- and post-conditions**, which are statements that are true before and after a process activity has been enacted or a product produced.

Example of a process



- ✧ Let's practice!
- ✧ Preparing for a birthday party

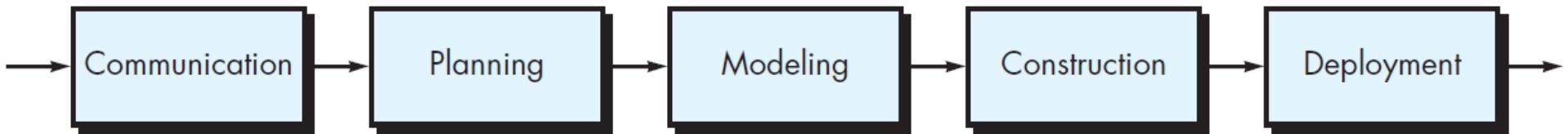
Birthday party



Process flow – linear process flow



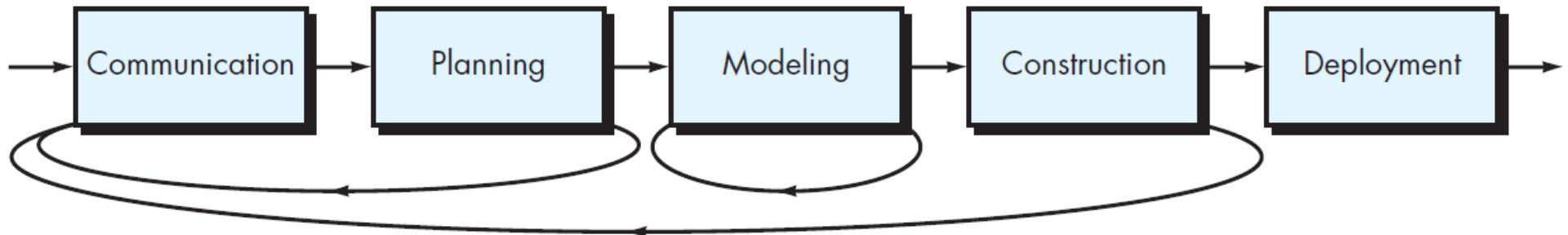
- ✧ A *linear process flow* executes each of the activities in sequence, beginning with communication and culminating with deployment



Process flow – iterative process flow



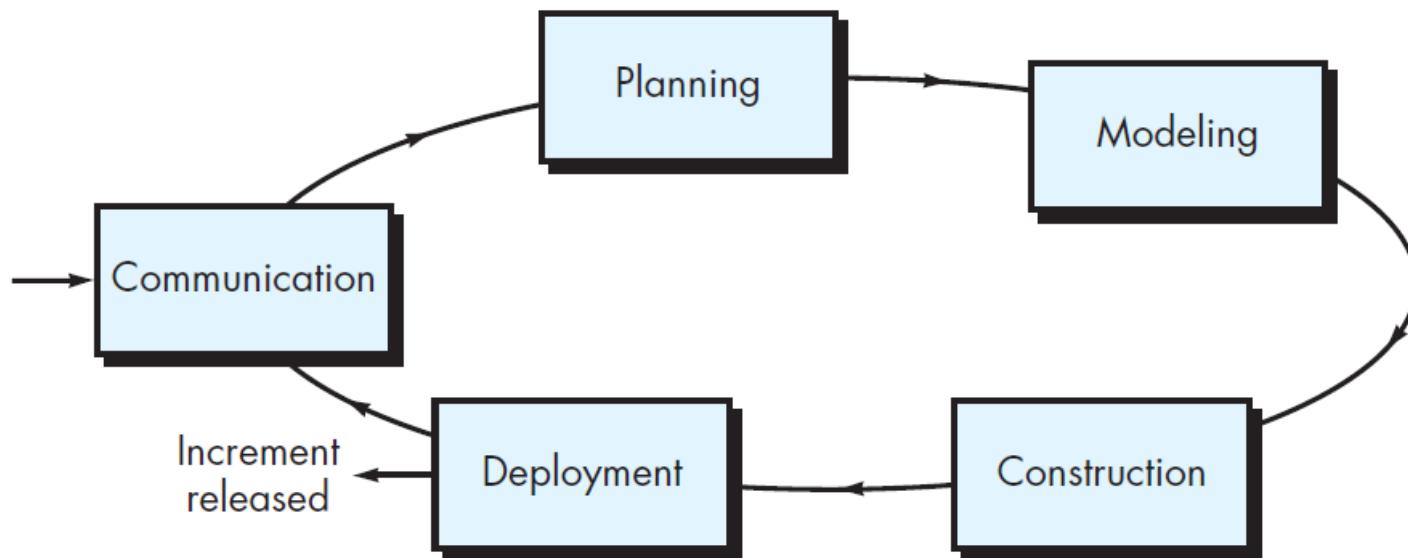
- ✧ An *iterative process flow* repeats one or more of the activities before proceeding to the next



Process flow – evolutionary process flow



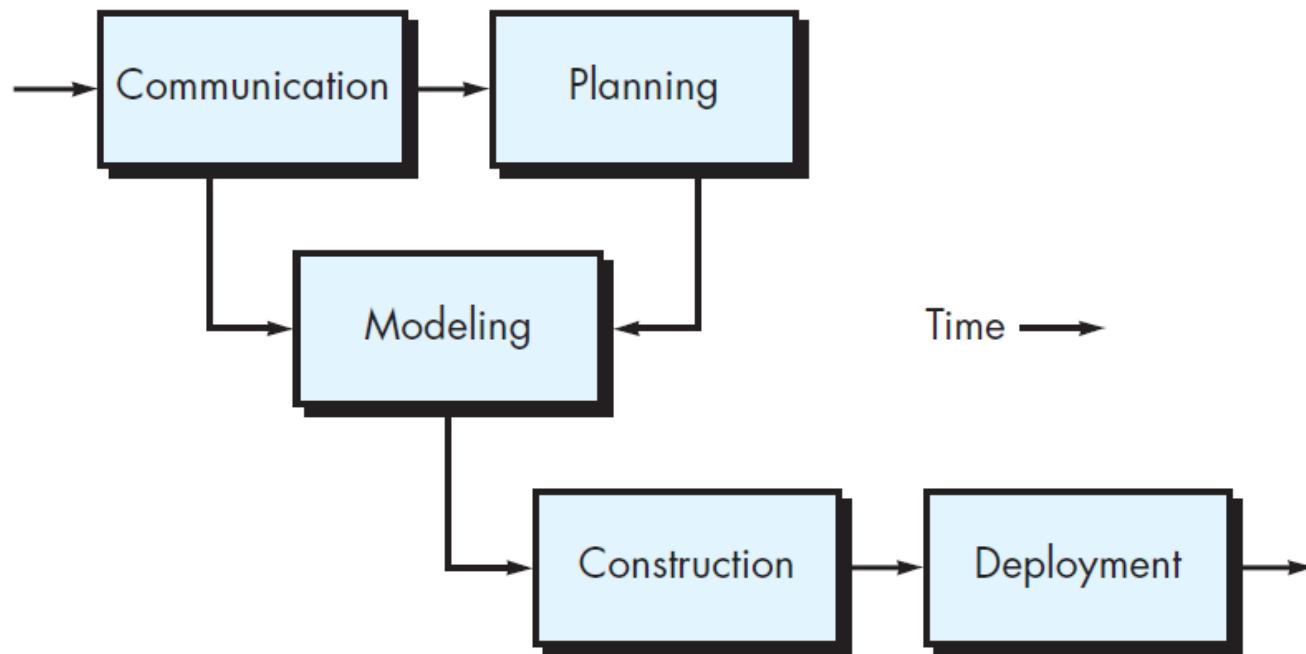
- ✧ An *evolutionary process flow* executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software.





Process flow – parallel process flow

- ✧ A *parallel process flow* executes one or more activities in parallel with other activities



Plan-driven and agile processes



- ✧ Plan-driven processes
 - All activities are **planned in advance** and **progress is measured against this plan**.
- ✧ Agile processes
 - **planning is incremental**
 - **Easier to change** the process to reflect changing customer requirements.
- ✧ In practice, most practical processes include elements of both plan-driven and agile approaches.
- ✧ There are no right or wrong software processes.



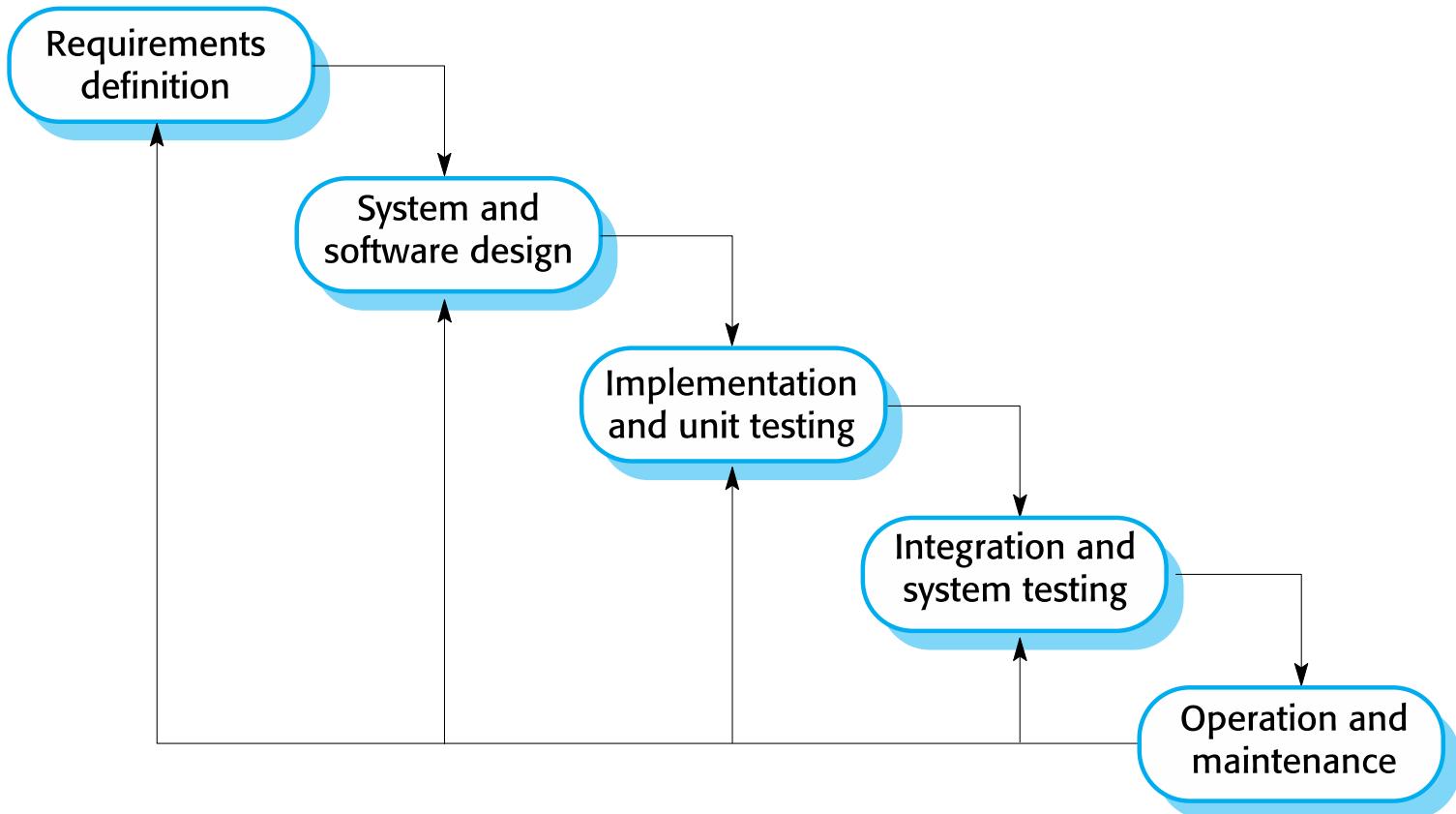
Software process models

Software process models

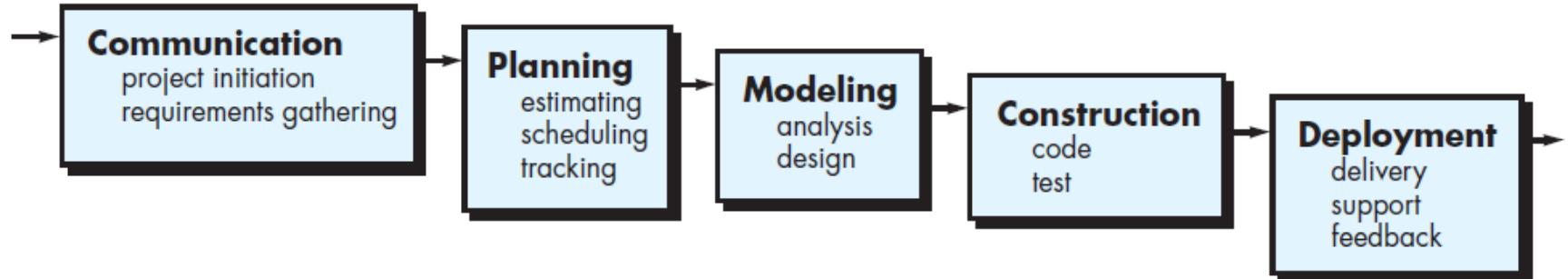


- ✧ The waterfall model
 - Plan-driven model. Separate and distinct phases of specification and development.
- ✧ Incremental development
 - Specification, development and validation are interleaved. May be plan-driven or agile.
- ✧ Integration and configuration
 - The system is assembled from existing configurable components. May be plan-driven or agile.
- ✧ In practice, most large systems are developed using a process that incorporates elements from all of these models.

The waterfall model (classic life cycle)



The waterfall model (classic life cycle)



Waterfall model phases



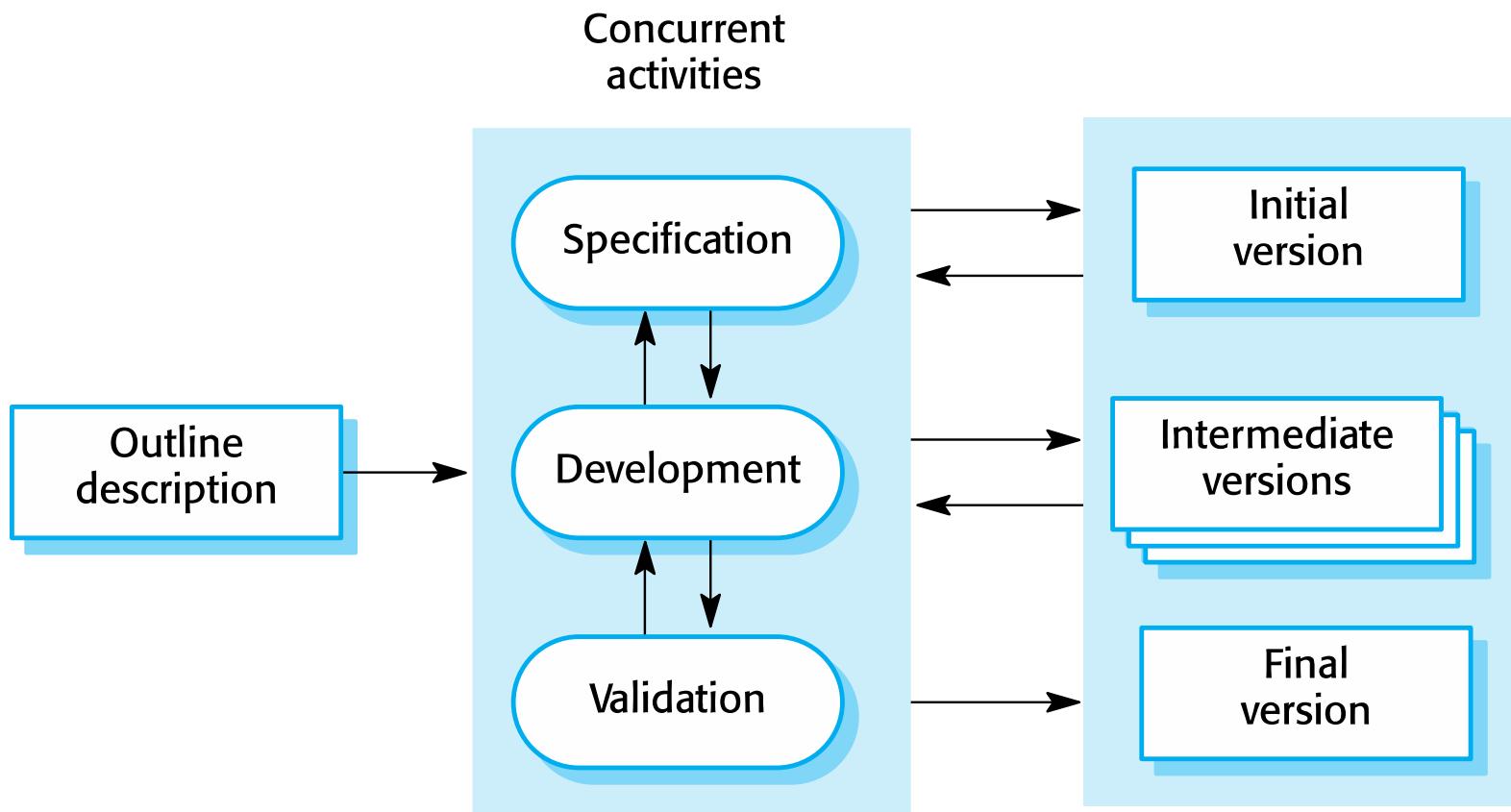
- ✧ There are **separate identified phases** in the waterfall model:
 - Requirements analysis and definition
 - System and software design
 - Implementation and unit testing
 - Integration and system testing
 - Operation and maintenance
- ✧ Main drawback
 - **Difficulty of accommodating change** after the process is underway.
 - A phase has to be complete before moving onto the next phase.

Waterfall model problems



- ✧ Inflexible **partitioning of the project into distinct stages** makes it difficult to respond to changing customer requirements.
 - Appropriate when
 - Requirements are well-understood
 - Changes will be fairly limited during the design process
 - Few business systems have stable requirements.
- ✧ Mostly used for
 - Large systems engineering projects
 - System is developed at several sites
 - The plan-driven nature of the waterfall model helps coordinate the work

Incremental development



Incremental development benefits



- ✧ Lower cost to accommodate changes.
 - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- ✧ Easier to get customer feedback on the development work that has been done.
 - Customers can comment on demonstrations of the software and see how much has been implemented.
- ✧ More rapid delivery and deployment of useful software to the customer is possible.
 - Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Incremental development problems



- ✧ The process is not visible.
 - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- ✧ System structure tends to degrade as new increments are added.
 - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

Incremental development problems



- ✧ Less suitable for systems that are
 - Large
 - Complex
 - Long-lifetime
 - Different teams develop different parts

- ✧ Large complex systems need
 - Stable framework
 - Clear division of responsibilities

Integration and configuration



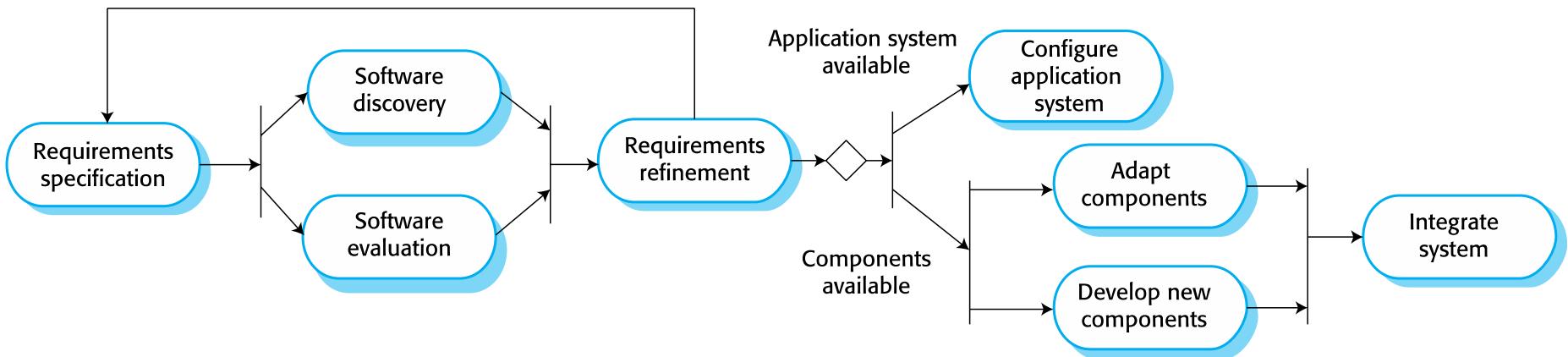
- ✧ Based on **software reuse** where systems are integrated from **existing components or application systems** (sometimes called COTS -Commercial-off-the-shelf) systems).
- ✧ Reused **elements may be configured** to adapt their behaviour and functionality to a user's requirements
- ✧ Reuse is now the standard approach for building many types of business system.

Types of reusable software



- ✧ Stand-alone application systems (COTS) that are configured for use in a particular environment.
- ✧ Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
- ✧ Web services that are developed according to service standards and which are available for remote invocation.

Reuse-oriented software engineering



Key process stages



- ✧ Requirements specification
- ✧ Software discovery and evaluation
- ✧ Requirements refinement
- ✧ Application system configuration
- ✧ Component adaptation and integration

Reuse-oriented software engineering



✧ Benefits:

- Reduces amount of software to be developed
 - Reduced cost
 - Reduced risk
 - Usually results in faster delivery of software

✧ Problems:

- Requirements compromises
 - Software may not meet user needs
 - No control over component evolution



Process activities

Process activities



✧ Four basic process activities:

- Specification, development, validation and evolution are organized differently in different development processes.
- In the waterfall model, **they are organized in sequence**
- In incremental development, **they are inter-leaved**

✧ Real software processes are inter-leaved sequences

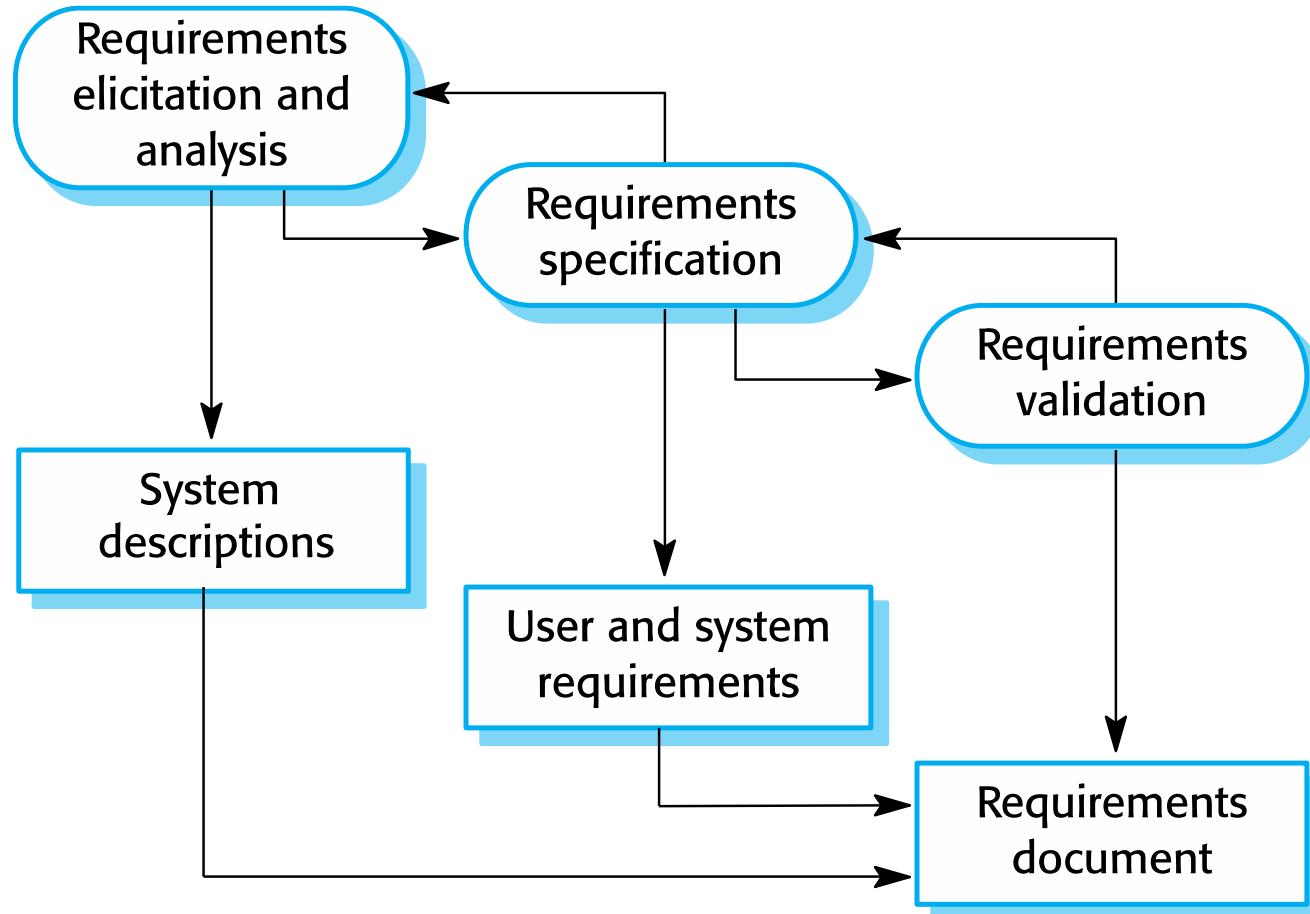
- Technical, collaborative, managerial activities
- To develop a software

Software specification



- ✧ The process of establishing **what services are required and the constraints** on the system's operation and development.
- ✧ Requirements engineering process
 - Requirements elicitation and analysis
 - What do the system stakeholders require or expect from the system?
 - Requirements specification
 - Defining the requirements in detail
 - Requirements validation
 - Checking the validity of the requirements

The requirements engineering process

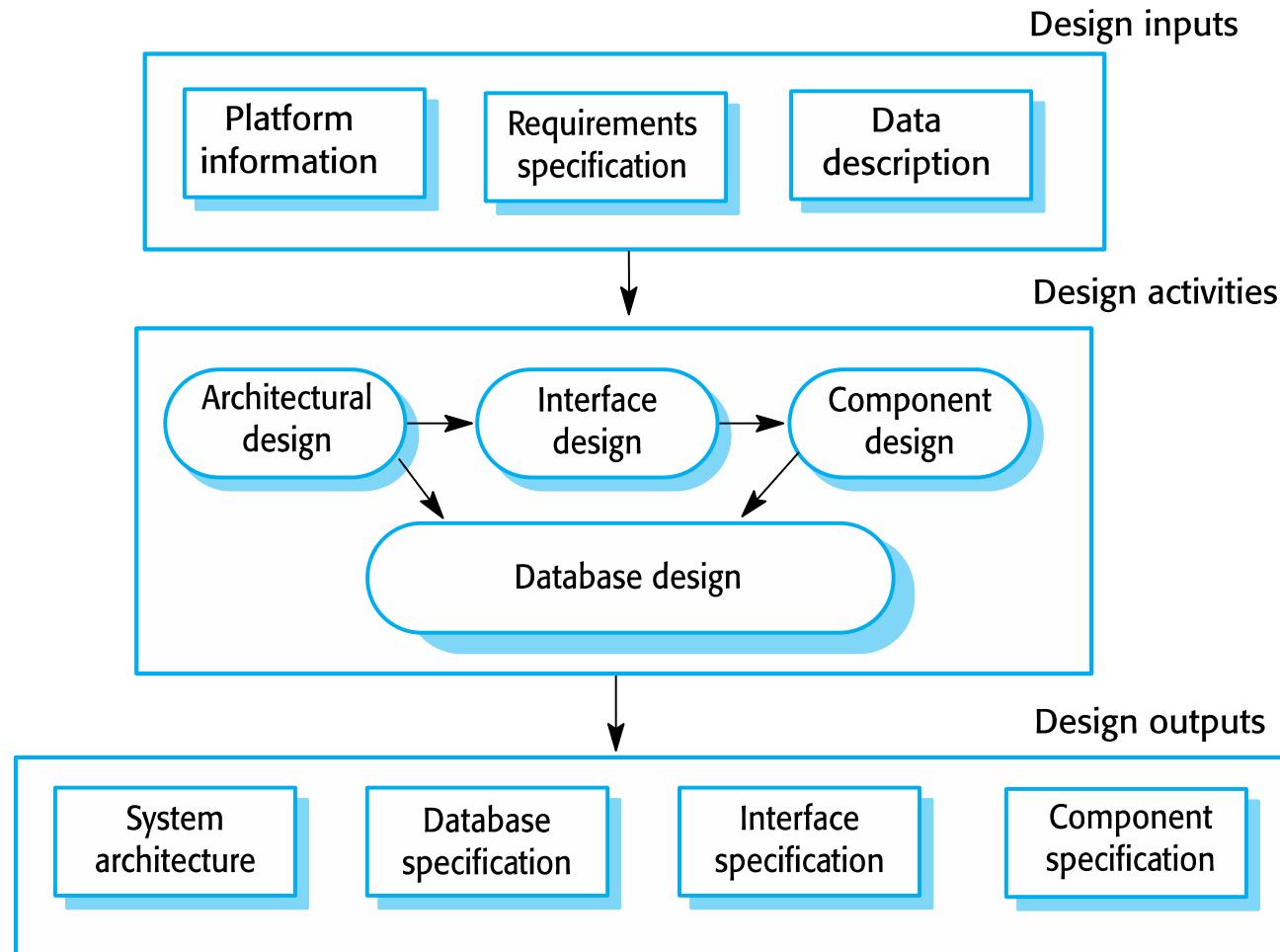


Software design and implementation



- ✧ The process of **converting the system specification into an executable system**.
- ✧ Software design
 - Design a software structure that realises the specification;
- ✧ Implementation
 - Translate this structure into an executable program;
- ✧ The activities of design and implementation are **closely related** and may be inter-leaved.

A general model of the design process



Design activities



- ✧ *Architectural design*, where you identify the **overall structure of the system**, the **principal components** (or subsystems), **their relationships** and **how they are distributed**.
- ✧ *Interface design*, where you define the **interfaces between system components**.
- ✧ *Database design*, where you **design the system data structures** and how these are to be represented in a database.
- ✧ *Component selection and design*, where you **search for reusable components**. If unavailable, you **design how it will operate**.

System implementation



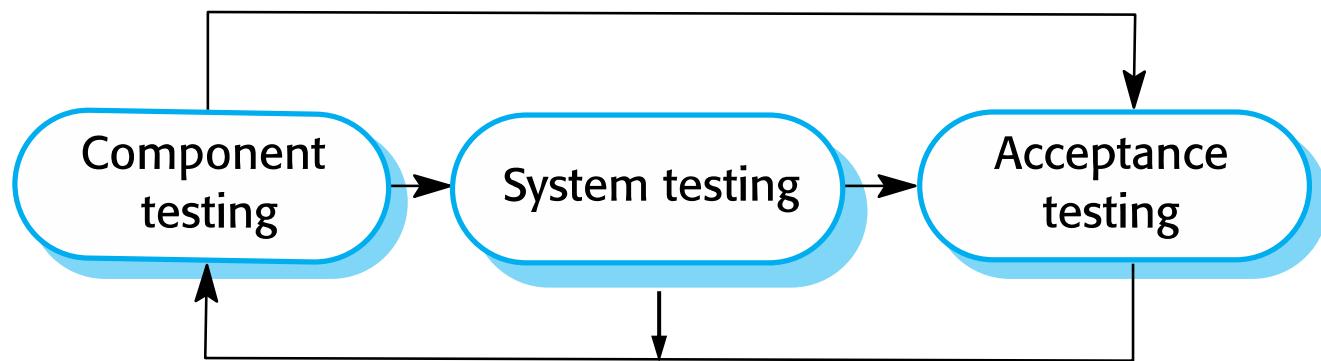
- ✧ The software is implemented either by **developing a program** or programs or by **configuring an application system**.
- ✧ Design and implementation are **interleaved activities** for most types of software system.
- ✧ Programming is an activity with no standard process.
- ✧ Debugging is the activity of **finding program faults** and **correcting these faults**.

Software validation



- ✧ Verification and validation (V & V) is intended to show that a system **conforms to its specification** and **meets the requirements of the system customer**.
- ✧ Involves checking and review processes and system testing.
- ✧ System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.
- ✧ Testing is the most commonly used V & V activity.

Stages of testing





Testing stages

✧ Component testing

- Individual components are tested independently;
- Components may be functions or objects or coherent groupings of these entities.

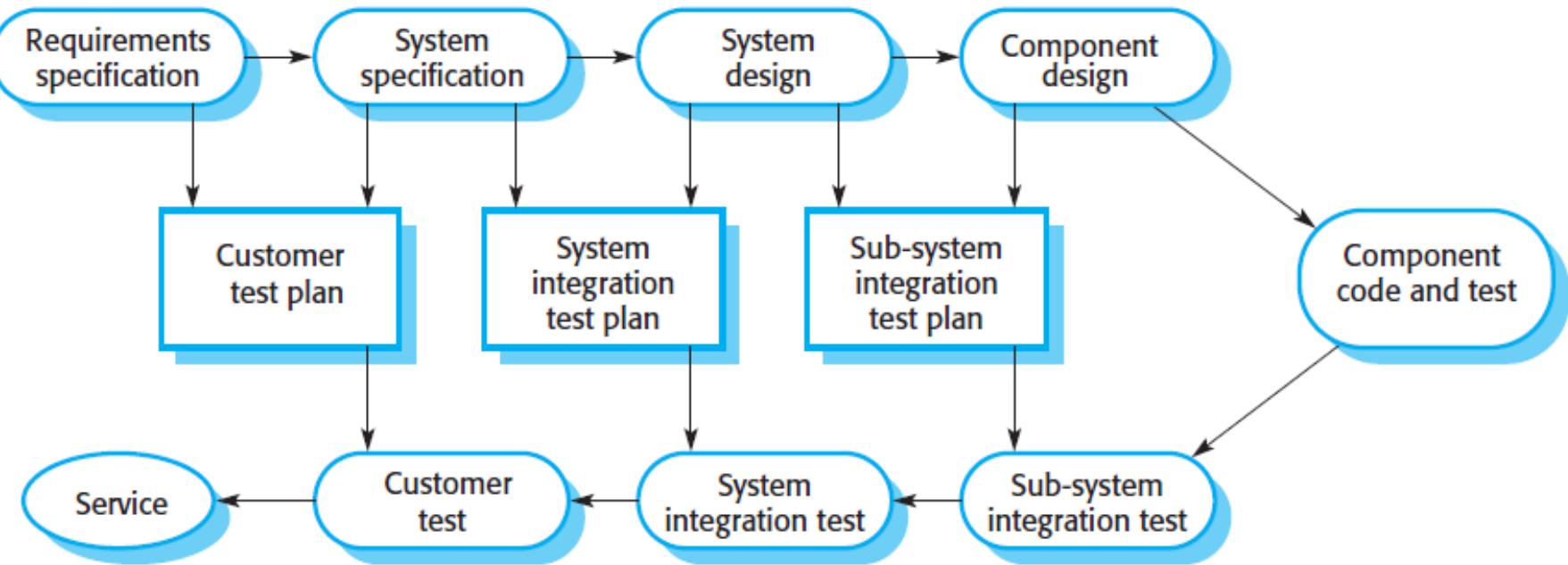
✧ System testing

- Testing of the **system as a whole**. Testing of emergent properties is particularly important.

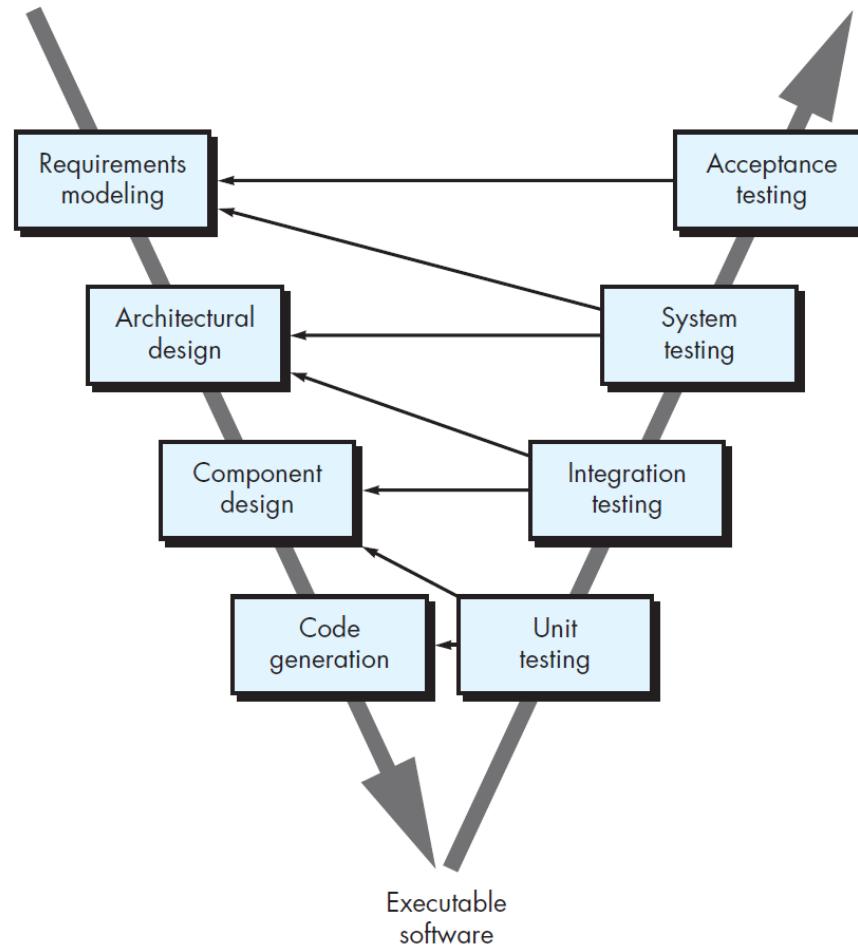
✧ Customer testing

- Testing with **customer data** to check that the system meets the customer's needs.

Testing phases in a plan-driven software process (V-model)



Testing phases in a plan-driven software process (V-model)

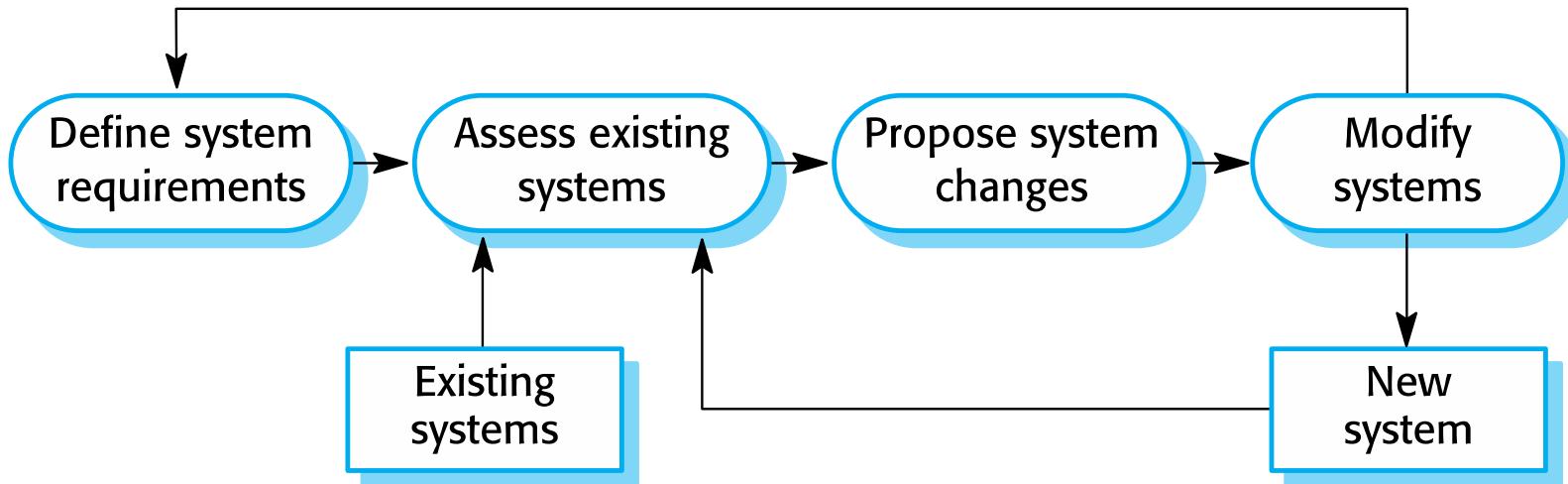


Software evolution



- ✧ Software is **inherently flexible and can change**.
- ✧ As requirements change through **changing business circumstances**, the software that supports the business must also **evolve and change**.
- ✧ Although there has been a **demarcation between development and evolution** (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.

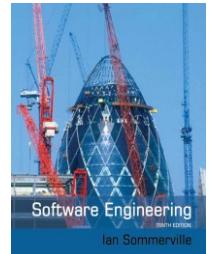
System evolution





Coping with change

Coping with change



- ✧ Change is **inevitable** in all large software projects.
 - **Business changes** lead to new and changed system requirements
 - **New technologies** open up new possibilities for improving implementations
 - **Changing platforms** require application changes
- ✧ **Change** leads to **rework** so the costs of change include both **rework** (e.g. re-analysing requirements) as well as the costs of **implementing new functionality**

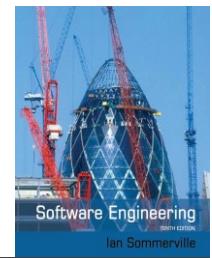
Reducing the costs of rework



- 1) **Change anticipation**, where the software process includes activities that can anticipate possible changes before significant rework is required.
 - For example, a **prototype** system may be developed to show some key features of the system to customers.







Software prototyping



- ✧ A prototype is an **initial version** of a system used to demonstrate **concepts** and try out **design options**.

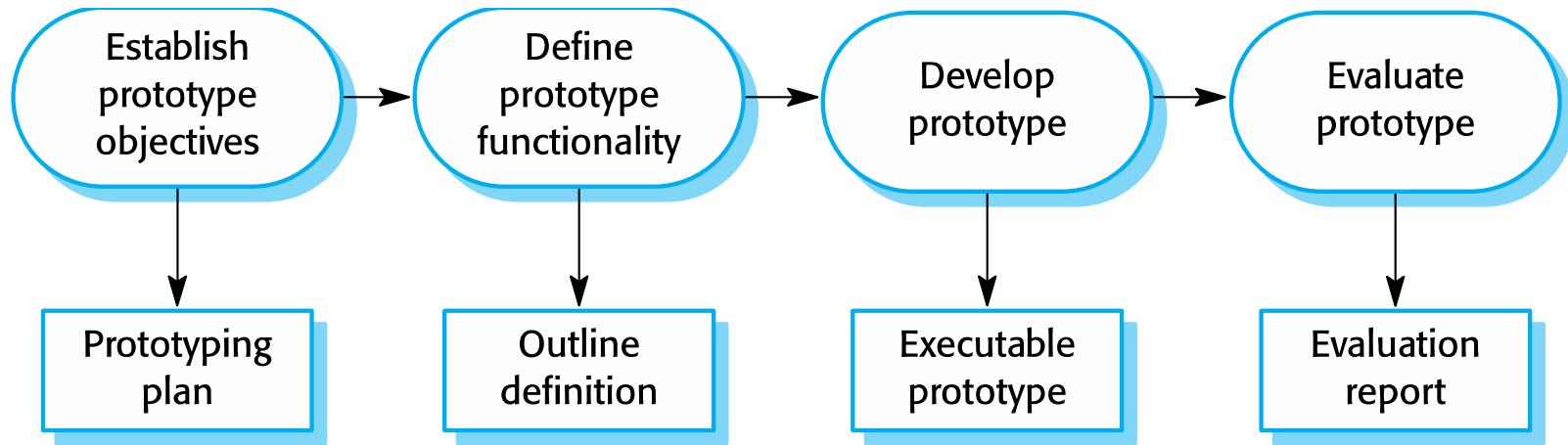
- ✧ A prototype can be used in:
 - The requirements engineering process to help with **requirements elicitation and validation**;
 - In design processes to explore options and **develop a UI design**;

Benefits of prototyping



- ✧ Improved system usability.
- ✧ A closer match to users' real needs.
- ✧ Improved design quality.
- ✧ Improved maintainability.
- ✧ Reduced development effort.

The process of prototype development



Prototype development



- ✧ May be based on rapid prototyping languages or tools
- ✧ May involve leaving out functionality
 - Prototype should focus on areas of the product that are **not well-understood**;
 - Error checking and recovery may not be included in the prototype;
 - Focus on **functional rather than non-functional** requirements such as reliability and security

Throw-away prototypes



- ✧ Prototypes should be **discarded** after development as they are not a good basis for a production system:
 - It may be **impossible to tune** the system to meet non-functional requirements;
 - Prototypes are normally **undocumented**;
 - The prototype structure is usually degraded through rapid change;
 - The prototype probably will not meet normal organizational quality standards.

Reducing the costs of rework



- 2) **Change tolerance**, where the process is designed so that changes can be accommodated at relatively low cost.
 - This normally involves some form of **incremental development**.
 - Proposed **changes may be implemented in increments** that have not yet been developed.
 - If this is impossible, then only a single or few increments (a small part of the system) may have to be altered to incorporate the change.

Incremental delivery



- ✧ Rather than deliver the system as a single delivery
 - Development and delivery is **broken down into increment**
 - Each increment **delivering part of the required functionality**
- ✧ User requirements are **prioritised**
 - Highest priority requirements are included in early increments
- ✧ Once the development of an increment is started
 - Requirements are frozen
 - Requirements for later increments can continue to evolve

Incremental development and delivery



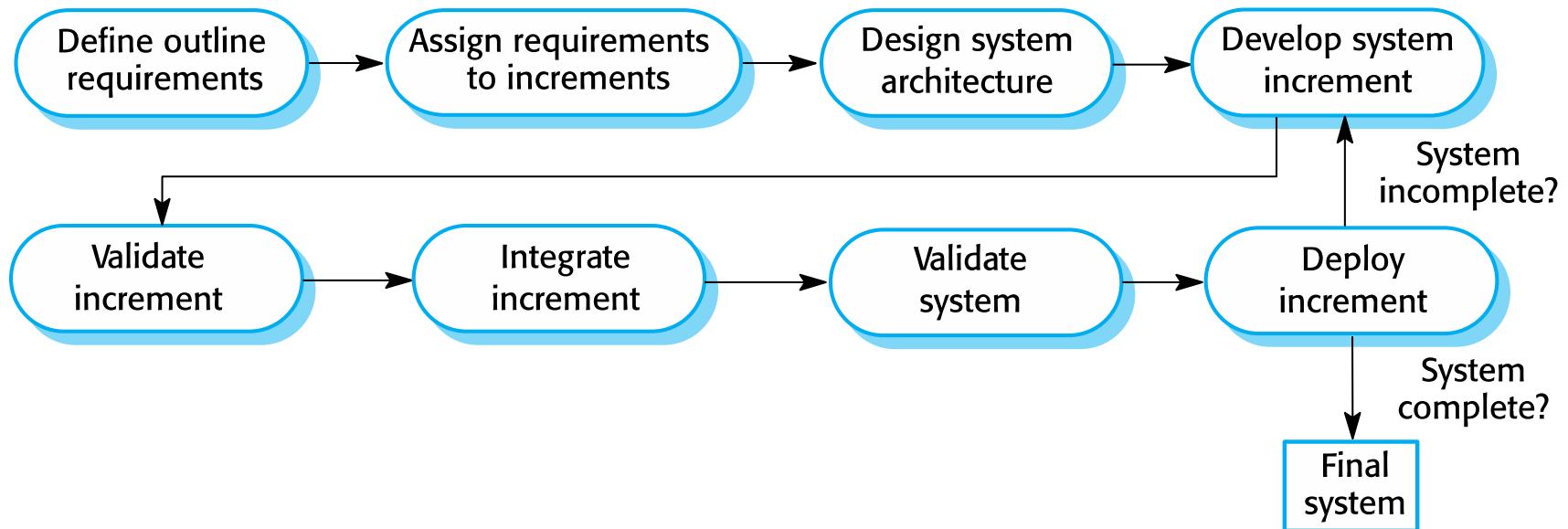
✧ Incremental development

- Develop the system in increments
 - Evaluate each increment before proceeding to the development of the next increment.
- Normal approach used in agile methods
- Evaluation done by user/customer proxy.

✧ Incremental delivery

- Deploy an increment for **use** by end-users.
- More **realistic evaluation** about practical use of software.
- Difficult to implement for replacement systems
 - Increments have less functionality than the system being replaced.

Incremental delivery



Incremental delivery advantages



- ✧ Customer **value** can be delivered with each increment so system functionality is **available earlier**.
- ✧ Early increments **act as a prototype** to help elicit requirements for later increments.
- ✧ The **highest priority** system services tend to receive the **most testing**.
- ✧ **Lower risk** of overall project failure.

Incremental delivery problems



- ✧ Most systems require a set of **basic facilities** that are **used by different parts** of the system.
 - As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
- ✧ The essence of iterative processes is that the **specification is developed in conjunction** with the software.
 - However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract.

Coping with changing requirements



- ✧ System prototyping, where a version of the system or part of the system is developed quickly **to check the customer's requirements and the feasibility of design decisions**. This approach supports change anticipation.
- ✧ Incremental delivery, where system **increments are delivered to the customer for comment and experimentation**. This supports both **change avoidance and change tolerance**.



Process improvement

Process improvement



- ✧ Many software companies have turned to software process improvement as a way of **enhancing the quality of their software, reducing costs or accelerating their development processes.**
- ✧ Process improvement means understanding existing processes and changing these processes **to increase product quality and/or reduce costs and development time.**



Approaches to improvement

- ✧ The **process maturity approach**, which focuses on improving process and project management and introducing good software engineering practice.
 - The level of process maturity reflects the extent to which **good technical and management practice** has been adopted in organizational software development processes.
- ✧ The **agile approach**, which focuses on iterative development and the **reduction of overheads** in the software process.
 - The primary characteristics of agile methods are **rapid delivery of functionality and responsiveness to changing customer requirements**.

Process improvement activities



✧ *Process measurement*

- You measure one or more **attributes of the software process or product**. These measurements forms a baseline that helps you decide if process improvements have been effective.

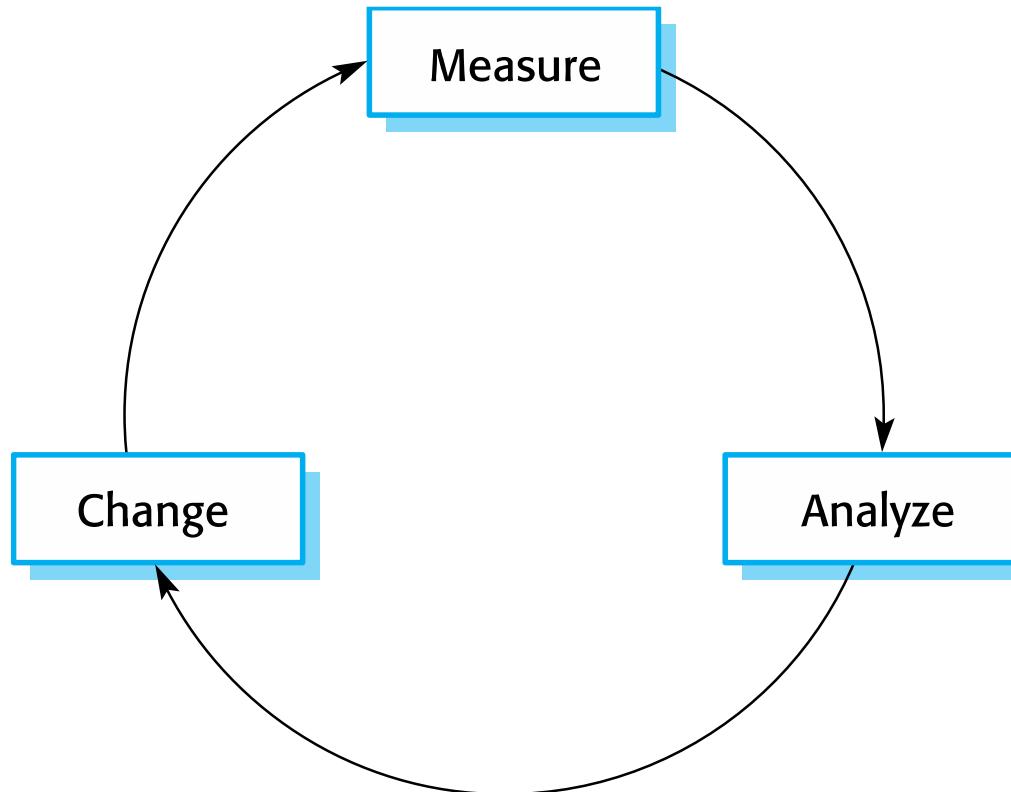
✧ *Process analysis*

- The current process is **assessed**, and process **weaknesses and bottlenecks** are identified. Process models (sometimes called process maps) that describe the process may be developed.

✧ *Process change*

- Process changes are proposed to address some of the identified process weaknesses. These are introduced and the cycle resumes to collect data about the effectiveness of the changes.

The process improvement cycle



Process measurement



- ✧ Wherever possible, **quantitative process data** should be collected
 - However, where organisations do not have clearly defined process standards this is very difficult as you don't know what to measure. A process may have to be defined before any measurement is possible.
- ✧ Process measurements should be used to assess process improvements
 - But this does not mean that measurements should drive the improvements. The improvement driver should be the organizational objectives.

Process metrics



- ✧ Time taken for process activities to be completed
 - E.g. Calendar time or effort to complete an activity or process.
- ✧ Resources required for processes or activities
 - E.g. Total effort in person-days.
- ✧ Number of occurrences of a particular event
 - E.g. Number of defects discovered.

Key points



- ✧ Processes should include activities to **cope with change**. This may involve a **prototyping** phase that helps avoid poor decisions on requirements and design.
- ✧ Processes may be structured for **incremental development and delivery** so that changes may be made without disrupting the system as a whole.
- ✧ The principal approaches to process improvement are **agile approaches**, **geared to reducing process overheads**, and **maturity-based approaches** based on better process management and the **use of good software engineering practice**.



Chapter 3 – Agile Software Development

Topics covered



- ✧ Agile methods
- ✧ Agile development techniques
- ✧ Agile project management
- ✧ Scaling agile methods

Rapid software development



- ✧ Rapid development and delivery is now often the most important requirement for software systems
 - Businesses operate in a fast-changing requirement and it is practically impossible to produce a set of stable software requirements
 - Software must evolve quickly to reflect changing business needs.
- ✧ Plan-driven development is essential for some types of system but does not meet these business needs.
- ✧ Agile development methods emerged in the late 1990s whose aim was to radically reduce the delivery time for working software systems

Agile development

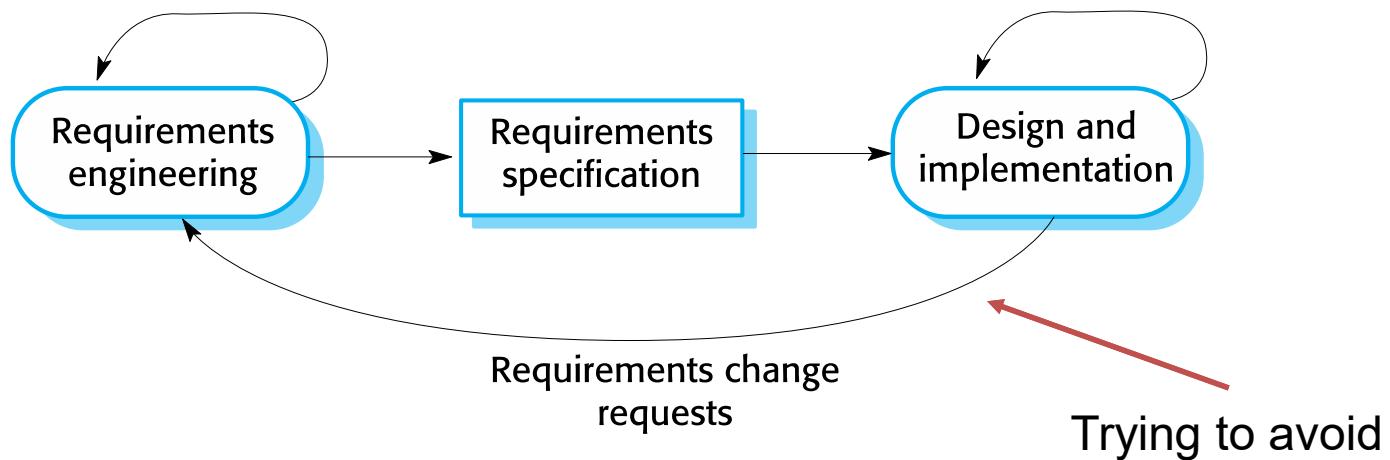


- ✧ Program specification, design and implementation are **inter-leaved**
- ✧ The system is developed as a **series of versions** or increments with **stakeholders involved in version specification and evaluation**
- ✧ **Frequent delivery** of new versions for evaluation
- ✧ Extensive tool support (e.g. automated testing tools) used to support development.
- ✧ Minimal documentation – **focus on working code**

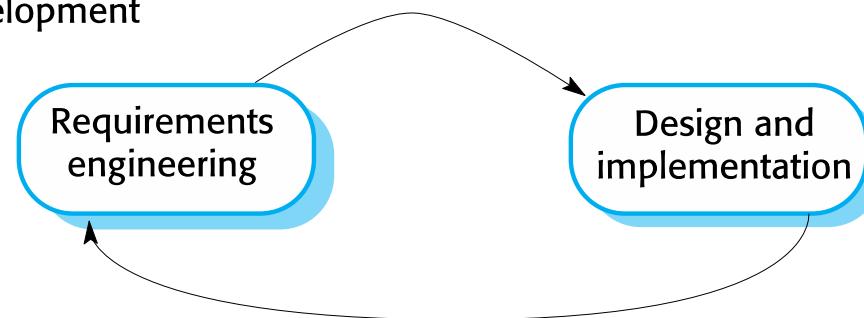


Plan-driven and agile development

Plan-based development



Agile development



Plan-driven and agile development



✧ Plan-driven development

- A plan-driven approach to software engineering is based around **separate development stages** with the **outputs to be produced at each of these stages planned in advance**.
- Not necessarily waterfall model – **plan-driven, incremental development is possible**
- **Iteration occurs within activities.**

✧ Agile development

- **Specification, design, implementation and testing are interleaved** and the outputs from the development process are decided through a process of negotiation **during the software development process**.



Agile methods

Agile methods



- ✧ Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
 - Focus on the **code rather than the design**
 - Are based on an **iterative** approach to software development
 - Are intended to **deliver working software quickly** and **evolve this quickly** to meet changing requirements.
- ✧ The aim of agile methods is to **reduce overheads** in the software process (e.g. by limiting documentation) and to **be able to respond quickly to changing requirements** without excessive rework.

Agile manifesto



- ✧ We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
 - *Individuals and interactions* over *processes and tools*
 - *Working software* over *comprehensive documentation*
 - *Customer collaboration* over *contract negotiation*
 - *Responding to change* over *following a plan*
- ✧ That is, while there is value in the items on the right, we value the items on the left more.

The principles of agile methods



Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is to provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process . Wherever possible, actively work to eliminate complexity from the system.

Agile method applicability



- ✧ Product development where a software company is developing a **small or medium-sized** product for sale.
 - Virtually all software products and apps are now developed using an agile approach
- ✧ Custom system development within an organization, where there is a **clear commitment from the customer to become involved** in the development process and where there are **few external rules and regulations** that affect the software.



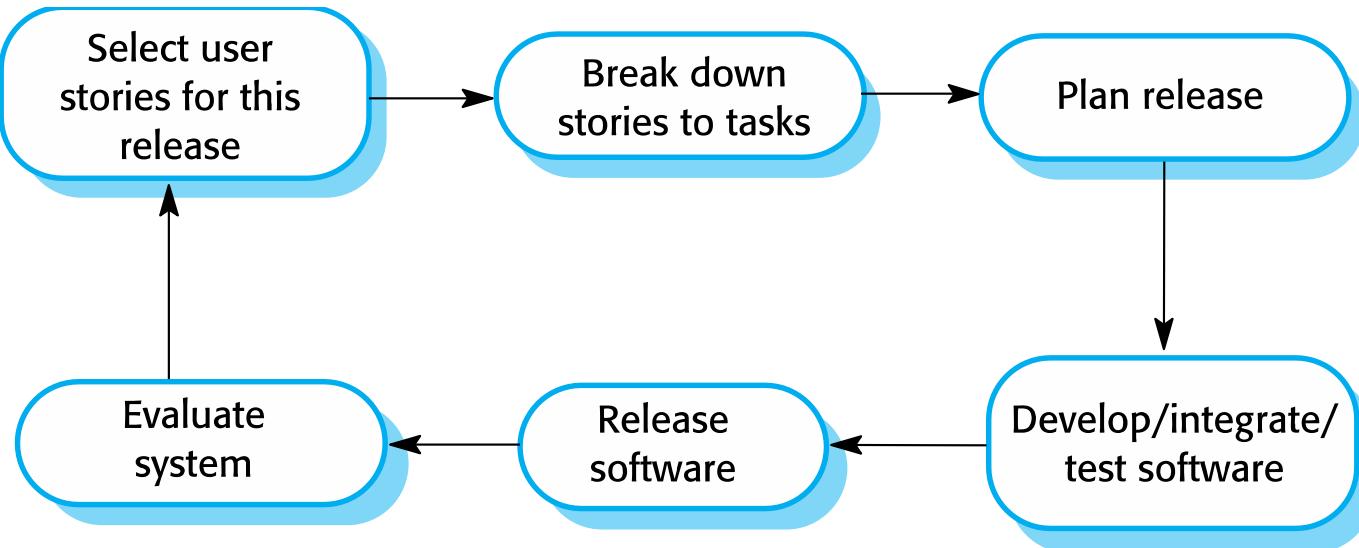
Agile development techniques

Extreme programming



- ✧ A very influential agile method, developed in the late 1990s, that introduced a range of agile development techniques.
- ✧ Extreme Programming (XP) takes an '**extreme**' approach to iterative development.
 - New versions may be built **several times per day**;
 - Increments are **delivered to customers every 2 weeks**;
 - All **tests** must be run for **every build** and the build is only accepted if tests run successfully.

The extreme programming release cycle



Extreme programming practices (a)



Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority . The developers break these stories into development 'Tasks'.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found . This keeps the code simple and maintainable.

Extreme programming practices (b)



Pair programming	Developers work in pairs , checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system , so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete , it is integrated into the whole system . After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

XP and agile principles



- ✧ Incremental development is supported through **small, frequent** system releases.
- ✧ Customer involvement means full-time **customer engagement** with the team.
- ✧ People not process through **pair programming, collective ownership** and a process that **avoids long working hours**.
- ✧ Change supported through regular system releases.
- ✧ Maintaining simplicity through **constant refactoring** of code.

Influential XP practices



- ✧ Extreme programming **has a technical focus** and is not easy to integrate with management practice in most organizations.
- ✧ Key practices
 - User stories for specification
 - Refactoring
 - Test-first development
 - Pair programming

User stories for requirements



- ✧ In XP, a **customer** or user is part of the XP team and is responsible for making decisions on requirements.
- ✧ User **requirements** are expressed as **user stories** or scenarios.
- ✧ These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- ✧ The **customer chooses the stories** for inclusion in the next release based on their **priorities** and the **schedule estimates**.



A ‘prescribing medication’ story

Software Engineering
Ian Sommerville

Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either ‘current medication’, ‘new medication’ or ‘formulary’.

If you select ‘current medication’, you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose ‘formulary’, you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click ‘OK’ or ‘Change’. If you click ‘OK’, your prescription will be recorded on the audit database. If you click ‘Change’, you reenter the ‘Prescribing medication’ process.

Examples of task cards for prescribing medication



Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.

Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.

Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low. If within the range, enable the 'Confirm' button.

Refactoring



- ✧ Conventional wisdom in software engineering is to **design for change**. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
- ✧ XP, however, maintains that this is not worthwhile as **changes cannot be reliably anticipated**.
- ✧ Rather, it proposes **constant code improvement** (refactoring) to make changes easier when they have to be implemented.

Refactoring



- ✧ Programming team look for **possible software improvements** and make these improvements even where there is no immediate need for them.
- ✧ This **improves the understandability** of the software and so **reduces the need for documentation**.
- ✧ Changes are easier to make because the code is well-structured and clear.
- ✧ However, some **changes requires architecture refactoring** and this is **much more expensive**.

Examples of refactoring



- ✧ Re-organization of a **class hierarchy** to remove duplicate code.
- ✧ Tidying up **and renaming attributes and methods** to make them easier to understand.
- ✧ The replacement of inline code with calls to methods that have been included in a program library.

Testing in XP



- ✧ Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.
- ✧ XP testing features:
 - Test-first development.
 - Incremental test development from scenarios.
 - User involvement in test development and validation.
 - Automated test harnesses are used to run all component tests each time that a new release is built.

Test-first development



- ✧ Writing tests before code **clarifies the requirements** to be implemented.
- ✧ **Tests are written as programs** rather than data so that they can be **executed automatically**. The test includes a check that it has executed correctly.
 - Usually relies on a testing framework such as Junit.
- ✧ All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has **not introduced errors**.

Customer involvement in testing



- ✧ The role of the customer in the testing process is to help develop **acceptance tests** for the stories that are to be implemented in the next release of the system.
- ✧ The **customer** who is part of the team **writes tests** as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
- ✧ The customer may feel that providing the **requirements was enough of a contribution** and so may be reluctant to get involved in the testing process.

Test case description for dose checking



Test 4: Dose checking

Input:

1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.
3. A string representing medication name or formula.

Tests:

1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:

OK or error message indicating that the dose is outside the safe range.

Test automation



- ✧ Test automation means that tests are **written as executable components** before the task is implemented
 - These testing components should be
 - stand-alone
 - simulate the submission of input to be tested
 - should check that the result meets the output specification.
 - An automated test framework (e.g. Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.

Test automation



✧ As testing is automated

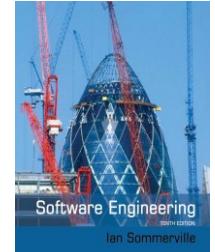
- There is always a set of tests that can be quickly and easily executed
- Whenever **any functionality is added**, the **tests can be run** and problems that the new code **has introduced (Software regression)** can be caught immediately.
 - Regression testing

XP testing difficulties



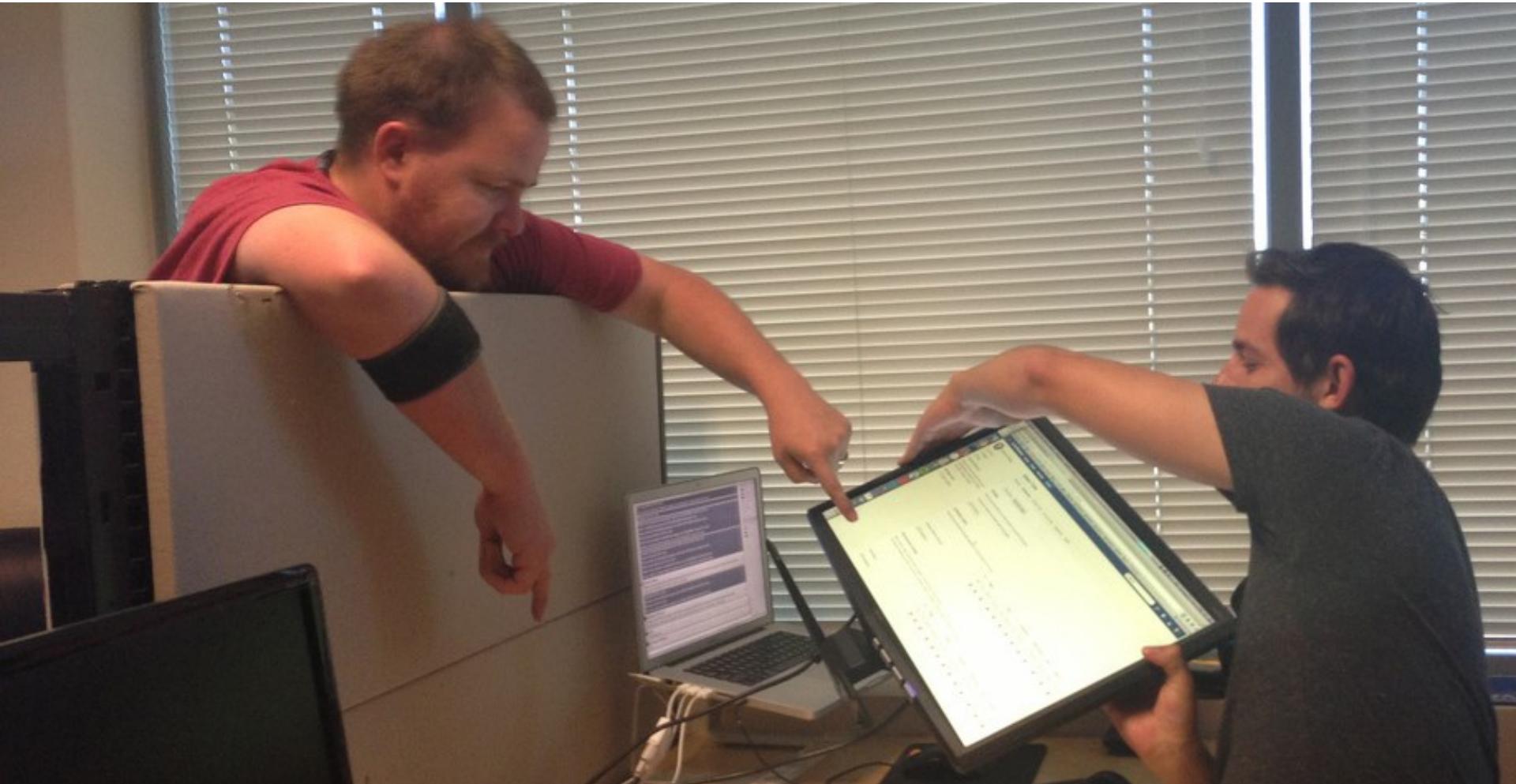
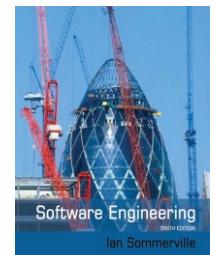
- ✧ Programmers prefer programming to testing and sometimes they take short cuts when writing tests.
 - For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
- ✧ Some tests can be very difficult to write incrementally
 - For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.
- ✧ It is difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage.

Pair programming

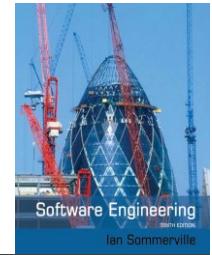


- ✧ Pair programming involves programmers **working in pairs**, developing code together.
- ✧ This helps develop **common ownership** of code and **spreads knowledge** across the team.
- ✧ It serves as an **informal review** process as each line of code is looked at by more than 1 person.
- ✧ It **encourages refactoring** as the whole team can benefit from improving the system code.

Pair programming



Pair programming



Pair programming



Pair programming



- ✧ In pair programming, programmers **sit together at the same computer** to develop the software.
- ✧ Pairs are **created dynamically** so that **all team members work with each other** during the development process.
- ✧ The sharing of knowledge that happens during pair programming is very important as it **reduces the overall risks** to a project when **team members leave**.
- ✧ Pair programming is **not necessarily inefficient** and there is some evidence that suggests that a pair working together is more efficient than 2 programmers working separately.

Impact of agile software development



- ✧ Waterfall originates in manufacturing and construction
 - Divides the process into sequential phases
 - Avoid iteration as much as possible
 - Leaves little room for changes
- ✧ Agile development process originates in software development
 - Encourages iterations
 - Results in multiple increments/versions
 - Process is open to changes
- ✧ Other industries are adapting Agile!

Impact of agile software development

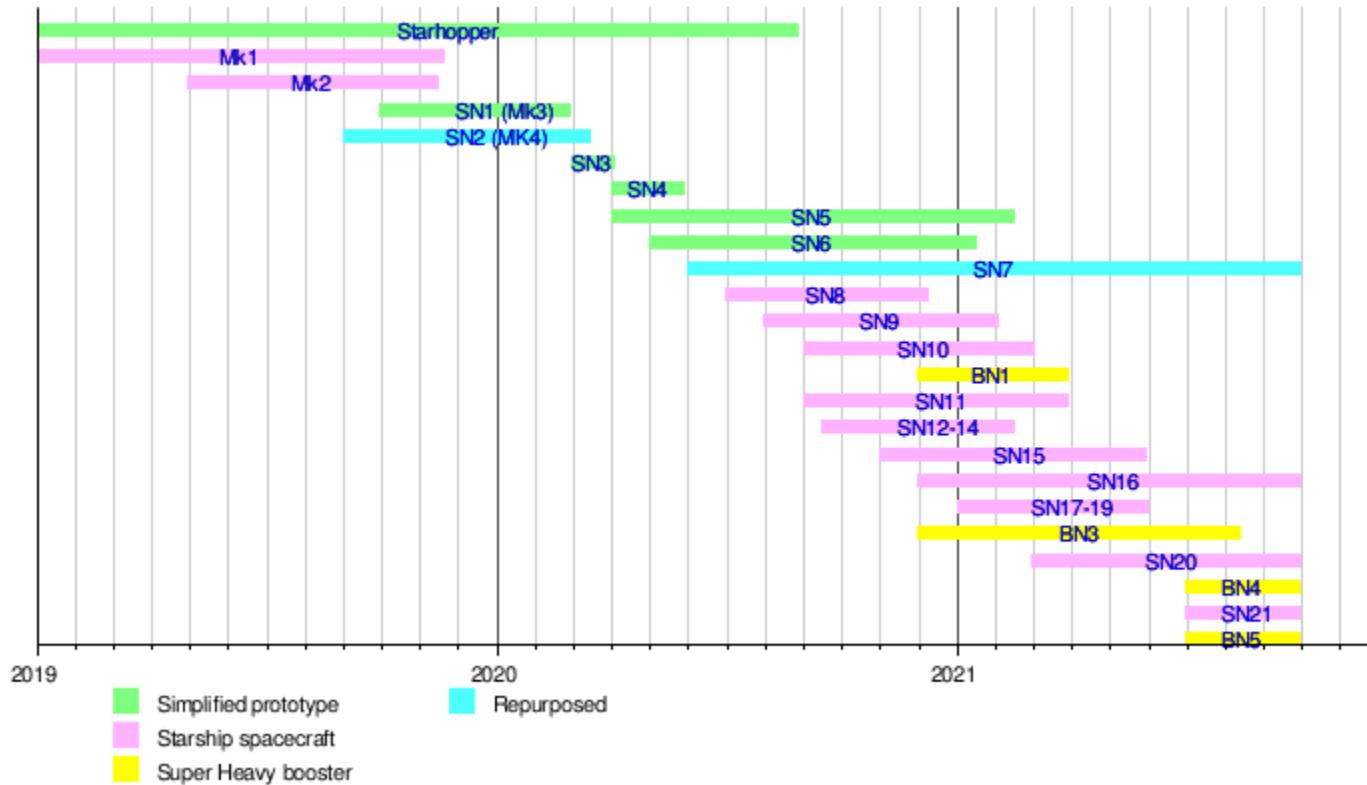


- ✧ SpaceX has adapted some principles from Agile
- ✧ Extensive use of tools for
 - Modeling
 - Fabrication/Prototyping
 - Simulating
- ✧ Generates lots of versions
 - Many challenges -> many versions (prototypes)
 - Some versions for very specific goal
 - Some for integration testing

SpaceX Starship development



Software Engineering
Ian Sommerville



SpaceX Starship development



SpaceX Starship development

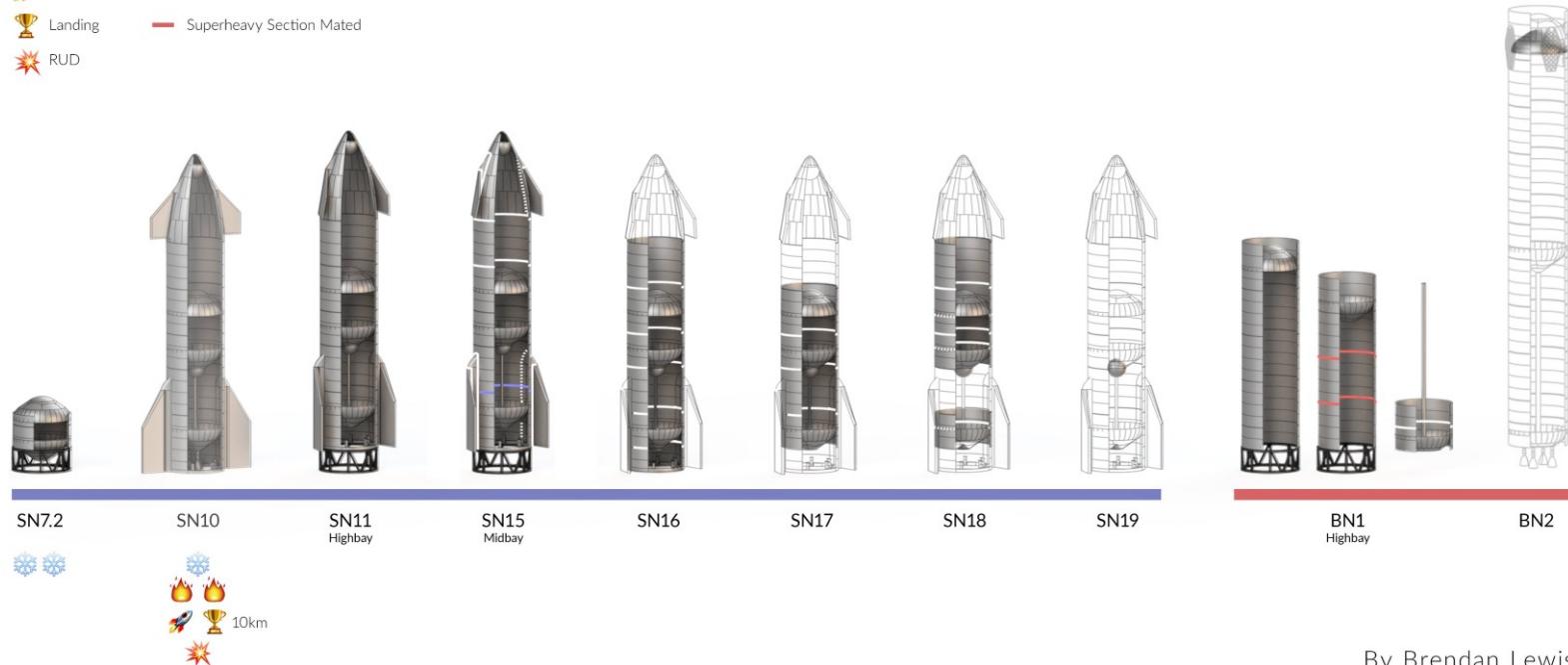


Software Engineering
Ian Sommerville

STATUS OF STARSHIP & SUPERHEAVY PROTOTYPES

Key

	Cryo Proof
	Static Fire
	Flight
	Landing
	RUD
	New Starship Section Spotted
	New Superheavy Section Spotted
	Starship Section Mated
	Superheavy Section Mated



5 March 2021

Proudly supported by NASASpaceFlight.com, Marcus House, & Patreon members

By Brendan Lewis
✉ @brendan_lewis

SpaceX Starship vs SLS



Software Engineering
Ian Sommerville

- ✧ SLS started in 2011 vs Starship 2018 (had small team before that)
- ✧ SLS supposed to have first launch in 2016
 - first launch is scheduled for no earlier than January 2022!
- ✧ Starship is may have its first orbital launch in few month



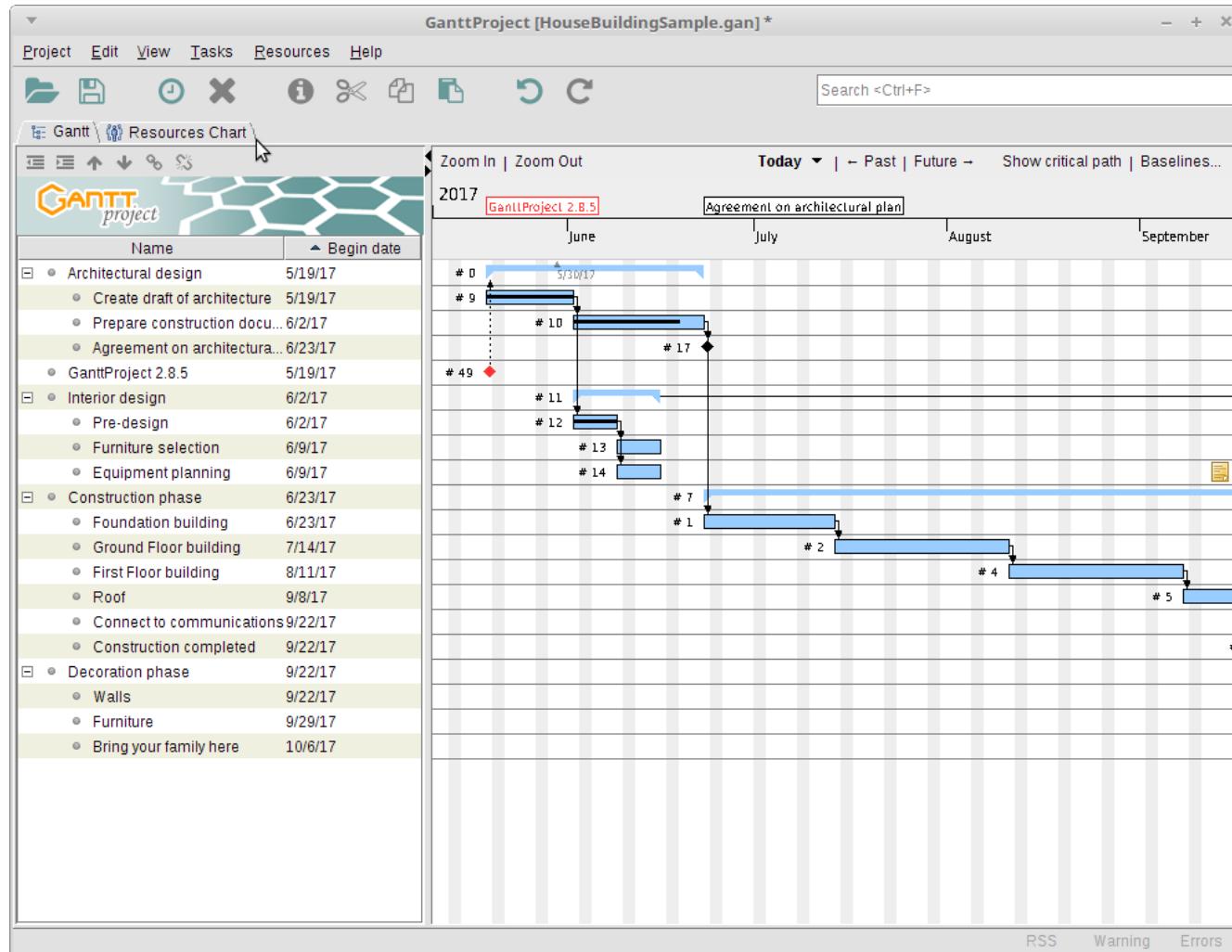
Agile project management

Agile project management



- ✧ The **principal responsibility** of software **project managers** is to manage the project so that the software is **delivered on time and within the planned budget** for the project.
- ✧ The **standard approach** to project management is **plan-driven**. Managers draw up a plan for the project showing **what** should be delivered, **when** it should be delivered and **who** will work on the development of the project deliverables.
- ✧ **Agile project management** requires a different approach, which is adapted to **incremental development** and the practices used in agile methods.

Standard approach to project management



Scrum



- ✧ Scrum is an **agile method** that focuses on **managing iterative development** rather than specific agile practices.
- ✧ There are three phases in Scrum.
 - The **initial phase** is an **outline planning phase** where you establish the **general objectives** for the project and **design the software architecture**.
 - This is followed by a **series of sprint cycles**, where each cycle develops an increment of the system.
 - The project **closure phase** wraps up the project, **completes required documentation** such as system help frames and user manuals and assesses the **lessons learned** from the project.

Scrum terminology (a)



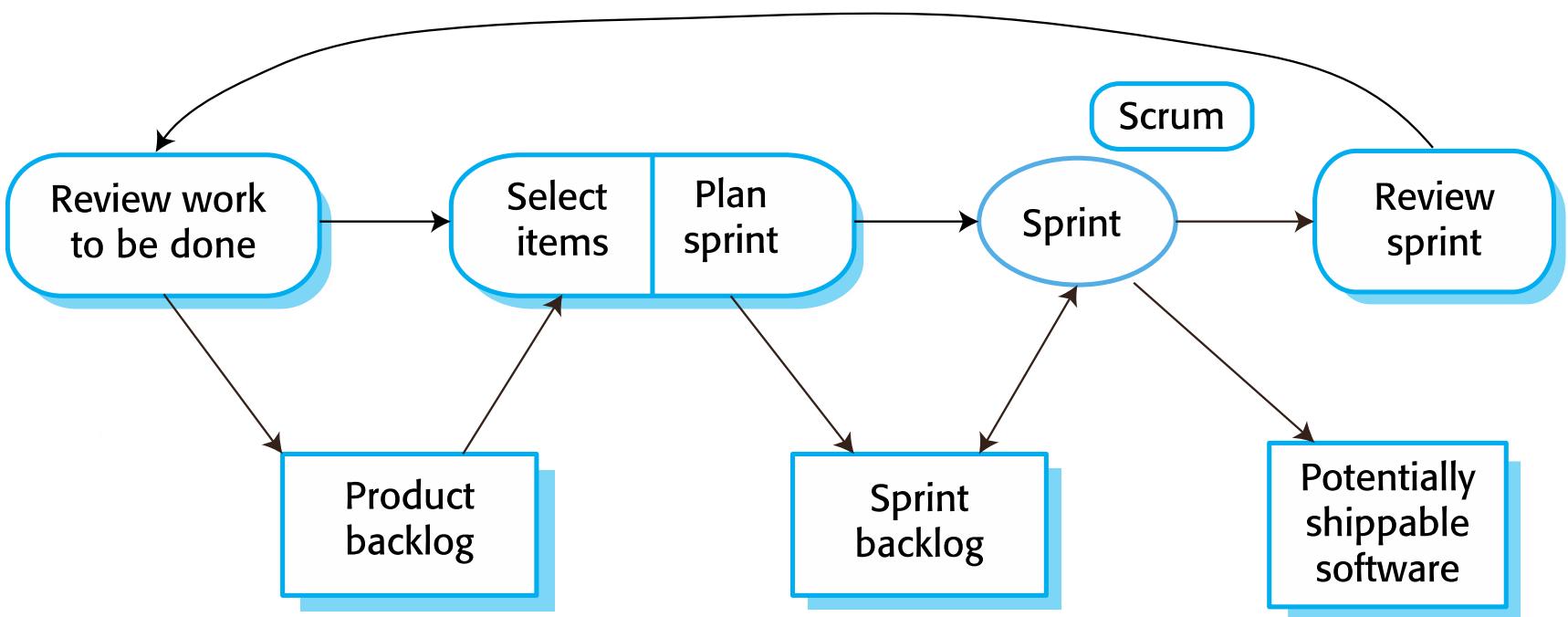
Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents .
Potentially shippable product increment	The software increment that is delivered from a sprint . The idea is that this should be 'potentially shippable' which means that it is in a finished state and no further work, such as testing , is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of ' to do ' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements , user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements , prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.



Scrum terminology (b)

Scrum term	Definition
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day . Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum . He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration . Sprints are usually 2-4 weeks long.
Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance .

Scrum sprint cycle



The Scrum sprint cycle



- ✧ Sprints are **fixed length**, normally 2 (common) to 4 weeks.
- ✧ The **starting point for planning** is the **product backlog**, which is the list of work to be done on the project.
- ✧ The **selection phase involves all of the project team** who work with the **customer** to **select the features and functionality** from the product backlog to be developed during the sprint.

The Sprint cycle



- ✧ Once these are agreed, the team organize themselves to develop the software.
- ✧ During this stage the team is isolated from the customer and the organization, with **all communications channelled through the so-called 'Scrum master'**.
- ✧ The role of the Scrum master is to **protect the development team from external distractions**.
- ✧ **Sprint review:** At the end of the sprint, the work done is **reviewed and presented to stakeholders**. The next sprint cycle then begins.

Teamwork in Scrum



- ✧ The ‘Scrum master’ is a facilitator who arranges **daily meetings**, **tracks the backlog** of work to be done, **records decisions**, **measures progress** against the backlog and **communicates with customers** and management outside of the team.
- ✧ The whole team attends **short daily meetings (Scrums)** where all team members **share information**, describe **their progress** since the last meeting, **problems** that have arisen and **what is planned** for the following day.
 - This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.

9 Best Practices for your Daily Scrum Meeting

1 FOCUS ON THE 3 QUESTIONS

- What did you do yesterday?
- What are you working on today?
- Do you have any roadblocks keeping you from doing your job?



2 KEEP THE DAILY SCRUM MEETING SHORT

The daily scrum meeting shouldn't last longer than 15 minutes. As your team gets more skilled, you may finish in less time. Don't take longer than you need.



3 HOLD A POST-MEETING DISCUSSION

When the team needs to discuss something more in-depth, save it for after the scrum meeting. Those who need to participate in the discussion can stay.



4 START THE SCRUM MEETING ON TIME

Starting late sets the wrong tone for the meeting. It also makes the meeting last longer. Respect one another's time by being punctual.



5

INCLUDE REMOTE TEAM MEMBERS

Remote team members working on the same goals should be a part of the daily scrum meeting. Meet in a room with a conference phone. Take time zones into account.



6

MEET AT THE SAME TIME EACH DAY

Get your team into a regular cadence. If you choose to have the meeting early in the day, team members can determine what they'll focus on for that day.



7

NO MULTITASKING

The daily scrum meeting is held for team members to communicate with one another. Multitasking can cause the meeting to last longer if someone must repeat themselves, and it's disrespectful to your peers.



8

REMAIN STANDING

Standing keeps everyone focused, and the meeting runs more quickly. People are aware the meeting should be short if they don't get comfortably seated.



9

STRIVE FOR CONTINUOUS IMPROVEMENT

Conducting a daily scrum meeting takes discipline. Don't get careless and adopt bad habits. Even if you're doing well, there are always opportunities to improve. Regularly assess how to do better.



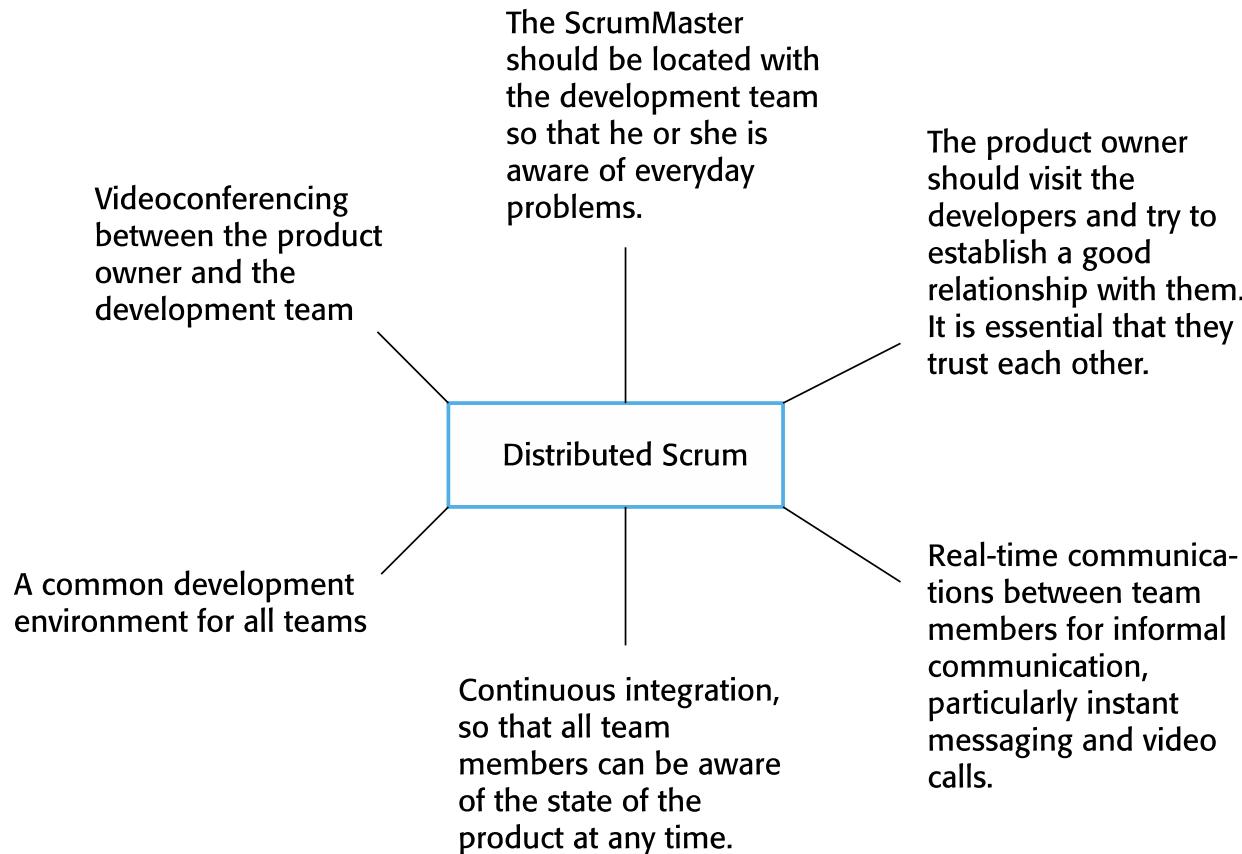
For more tips on project management, leadership & performance, visit ProjectBliss.net

Scrum benefits



- ✧ The **product is broken down** into a set of manageable and understandable chunks.
- ✧ **Unstable requirements** do not hold up progress.
- ✧ The **whole team have visibility** of everything and consequently team communication is improved.
- ✧ Customers see **on-time delivery** of increments and **gain feedback** on how the product works.
- ✧ **Trust between customers and developers** is established and a positive culture is created in which everyone expects the project to succeed.

Distributed Scrum





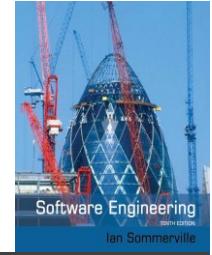
Scaling agile methods

Scaling agile methods



- ✧ Agile methods have proved to be **successful for small and medium sized projects** that can be developed by a small co-located team.
- ✧ It is sometimes argued that the success of these methods comes because of **improved communications** which is possible when everyone is working together.
- ✧ Scaling up agile methods involves changing these **to cope with larger, longer projects** where there are **multiple development teams**, perhaps working in **different locations**.

Scaling out and scaling up



- ✧ ‘Scaling up’ is concerned with using agile methods for **developing large software systems** that cannot be developed by a small team.
- ✧ ‘Scaling out’ is concerned with how agile methods can be **introduced across a large organization** with many years of software development experience.
- ✧ When scaling agile methods it is important to maintain agile fundamentals:
 - Flexible planning, frequent system releases, continuous integration, test-driven development and good team communications.

Practical problems with agile methods



- ✧ The **informality of agile** development is **incompatible with the legal approach to contract definition** that is commonly used in large companies.
- ✧ Agile methods are most appropriate for **new software development rather than software maintenance**. Yet the majority of software costs in large companies come from maintaining their **existing software** systems.
- ✧ Agile methods are designed for **small co-located teams** yet much software development now involves **worldwide distributed teams**.

Contractual issues



- ✧ Most **software contracts** for custom systems are **based around a specification**, which sets out what has to be implemented by the system developer for the system customer.
- ✧ However, this **precludes interleaving specification and development** as is the norm in agile development.
- ✧ A contract that pays for **developer time rather than functionality** is required.
 - However, this is seen as a high risk by many legal departments because what has to be delivered cannot be guaranteed.

Agile maintenance



- ✧ Key problems are:
 - Lack of product documentation
 - Keeping customers involved in the development process
 - Maintaining the continuity of the development team
- ✧ Agile development relies on the **development team knowing and understanding what has to be done.**
- ✧ For long-lifetime systems, this is a real problem as the **original developers will not always work on the system.**

Agile principles and organizational practice



Principle	Practice
Customer involvement	<p>This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development.</p> <p>Where there are external stakeholders, such as regulators, it is difficult to represent their views to the agile team.</p>
Embrace change	<p>Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.</p>
Incremental delivery	<p>Rapid iterations and short-term planning for development does not always fit in with the longer-term planning cycles of business planning and marketing. Marketing managers may need to know what product features several months in advance to prepare an effective marketing campaign.</p>

Agile principles and organizational practice



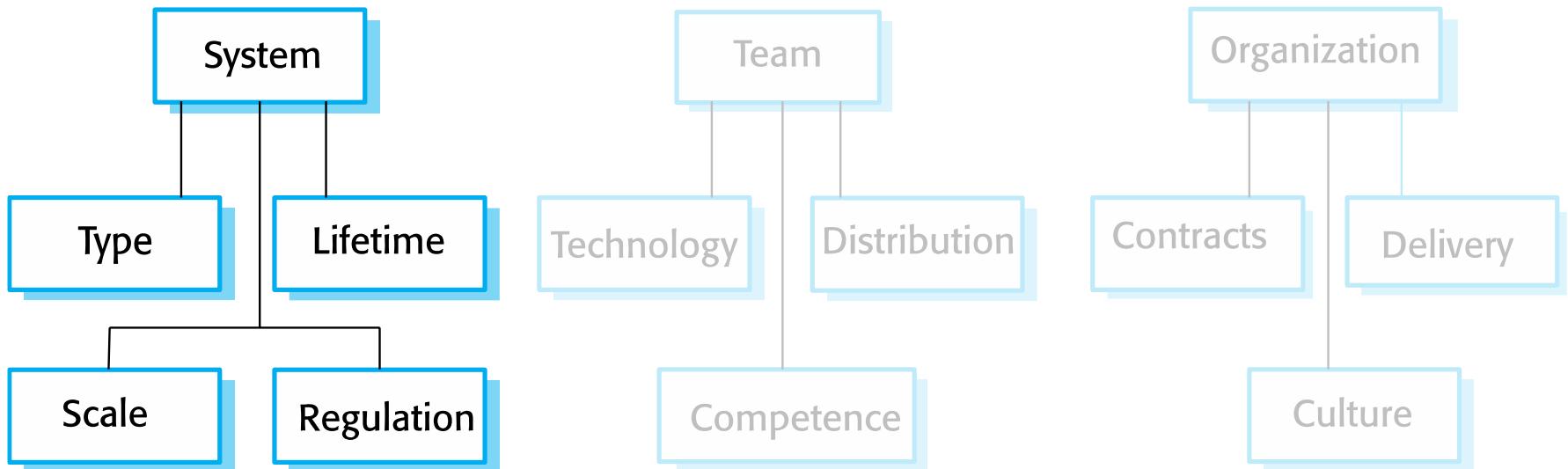
Principle	Practice
Maintain simplicity	Under pressure from delivery schedules, team members may not have time to carry out desirable system simplifications.
People not process	Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore may not interact well with other team members.

Agile and plan-driven methods



- ✧ Most projects include elements of **plan-driven and agile processes**. Deciding on the balance depends on:
 - Is it important to have a **very detailed specification and design** before moving to implementation? If so, you probably need to use a plan-driven approach.
 - Is an **incremental delivery** strategy, where you deliver the software to customers and get rapid feedback from them, **realistic**? If so, consider using agile methods.
 - **How large is the system** that is being developed? Agile methods are most effective when the system can be developed with a **small co-located team who can communicate informally**. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.

Agile and plan-based factors

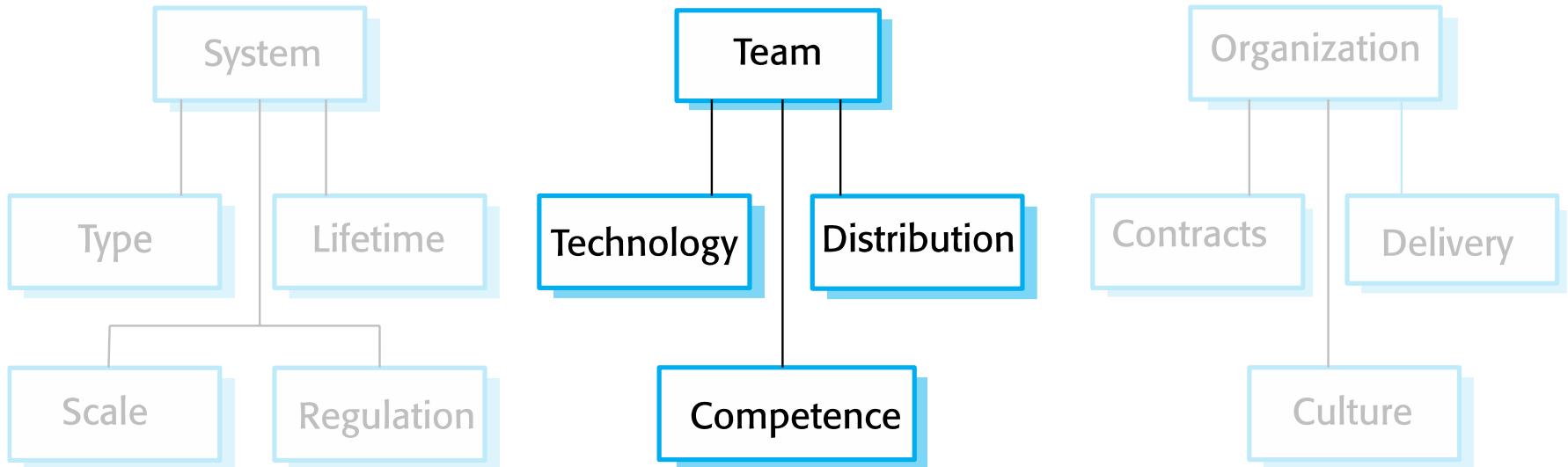




System issues

- ✧ How large is the system being developed?
 - Agile methods **are most effective a relatively small co-located team** who can communicate informally.
- ✧ What type of system is being developed?
 - Systems that require a **lot of analysis before implementation** need a fairly detailed design to carry out this analysis.
- ✧ What is the expected system lifetime?
 - Long-lifetime systems **require documentation** to communicate the intentions of the **system developers to the support team**.
- ✧ Is the system subject to external regulation?
 - If a system is regulated you will probably be required to produce **detailed documentation** as part of the system safety case.

Agile and plan-based factors

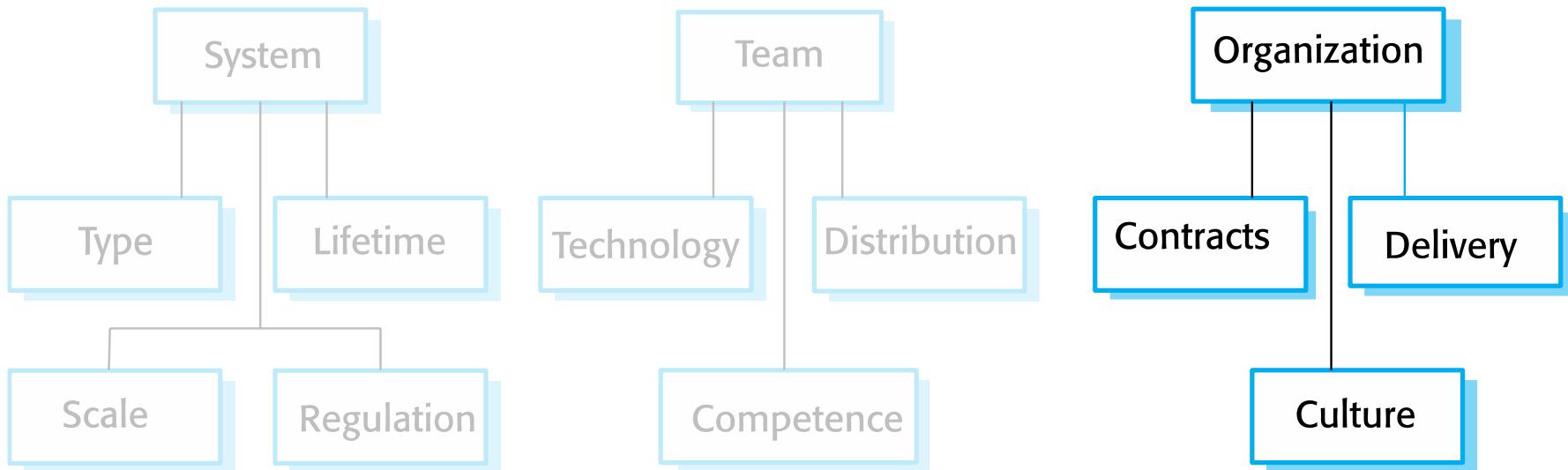


People and teams



- ✧ How good are the designers and programmers in the development team?
 - It is sometimes argued that **agile methods require higher skill** levels than plan-based approaches in which programmers simply translate a detailed design into code.
- ✧ How is the development team organized?
 - **Design documents** may be required if the **team is distributed**.
- ✧ What support **technologies are available**?
 - IDE support for visualisation and program analysis is essential if design documentation is not available.

Agile and plan-based factors

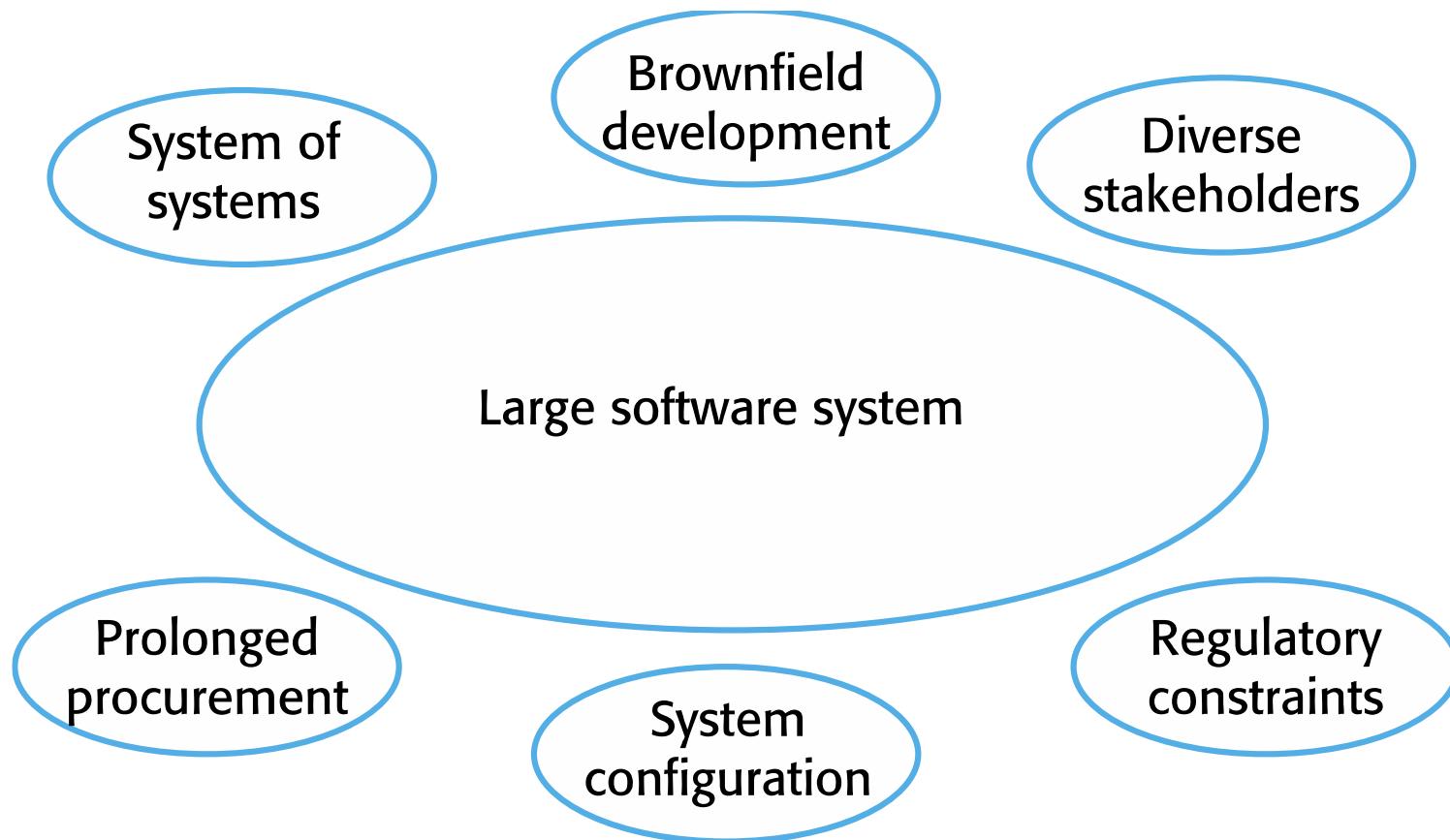


Organizational issues



- ✧ Traditional engineering organizations have a **culture of plan-based development**, as this is the norm in engineering.
- ✧ Is it standard organizational practice to develop a **detailed system specification**? (e.g. for contracts)
- ✧ Can informal agile development fit into the organizational culture of detailed documentation?
- ✧ Will **customer representatives be available** to provide feedback of system increments?

Factors in large systems



Agile methods for large systems



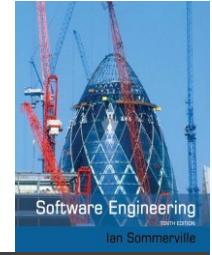
- ✧ Large systems are usually **collections of separate, communicating systems**, where **separate teams develop each system**. Frequently, these teams are working in different places, sometimes in different time zones.
- ✧ Large systems are 'brownfield systems', that is **they include and interact with a number of existing systems**. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development.
- ✧ Where several systems are integrated to create a system, a significant fraction of the **development is concerned with system configuration** rather than original code development.

Large system development



- ✧ Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
- ✧ Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
- ✧ Large systems usually have a diverse set of stakeholders. It is practically impossible to involve all of these different stakeholders in the development process.

Scaling up to large systems



- ✧ A **completely incremental approach** to requirements engineering **is impossible**.
- ✧ There **cannot be a single product owner** or customer representative.
- ✧ For large systems development, it is **not possible to focus only on the code** of the system.
- ✧ **Cross-team communication mechanisms** have to be designed and used.
- ✧ **Continuous integration is practically impossible**. However, it is essential to maintain frequent system builds and regular releases of the system.

Multi-team Scrum



✧ *Role replication*

- Each team has a **Product Owner** for their work component and **ScrumMaster**.

✧ *Product architects*

- Each team chooses a **product architect** and these architects collaborate to design and evolve the overall system architecture.

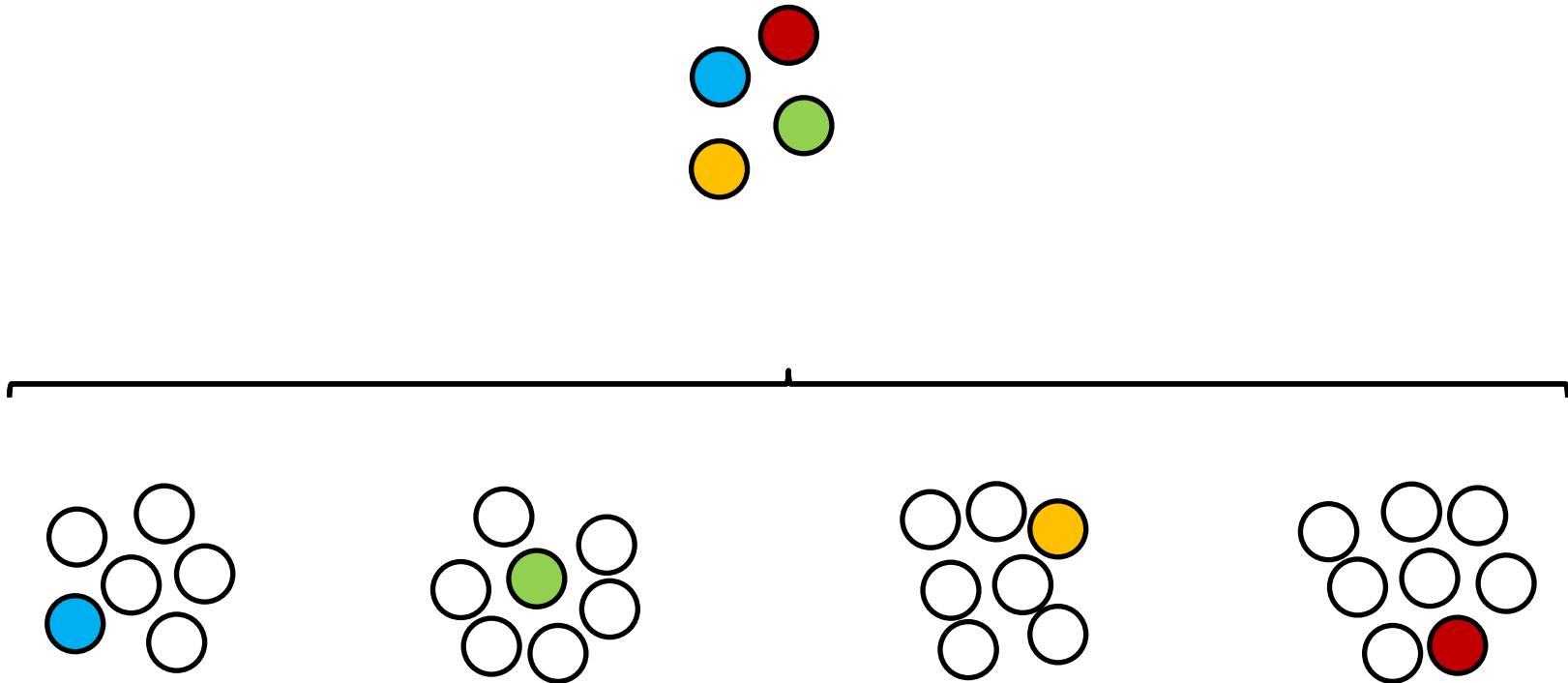
✧ *Release alignment*

- The **dates of product releases** from each team are aligned so that a demonstrable and complete system is produced.

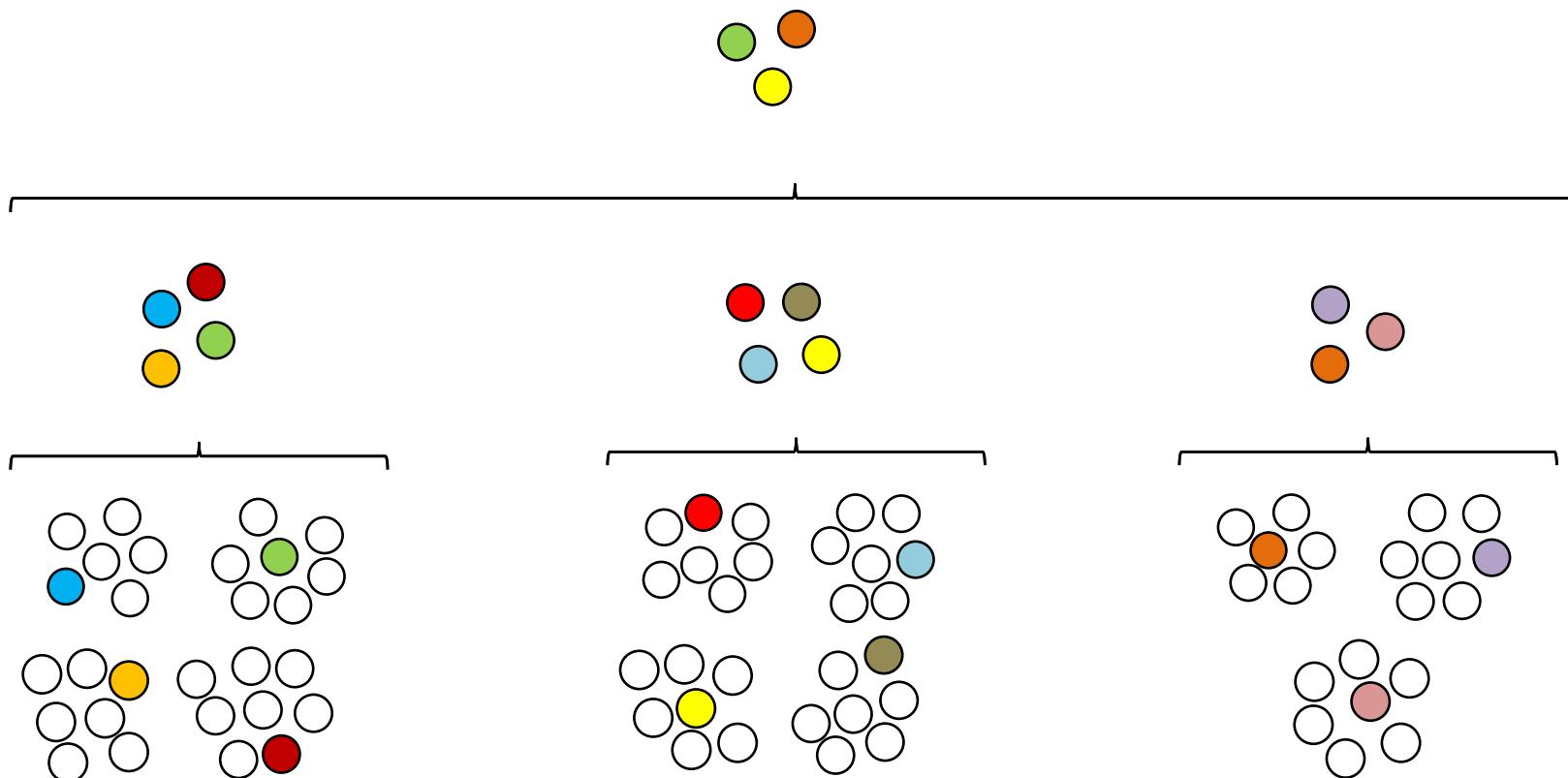
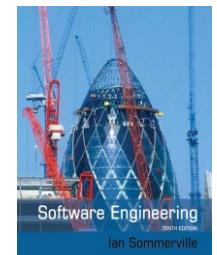
✧ *Scrum of Scrums*

- There is a daily Scrum of Scrums where representatives from each team meet to discuss progress and plan work to be done.

Scrum of Scrums



Scrum of Scrums of Scrums



Agile methods across organizations

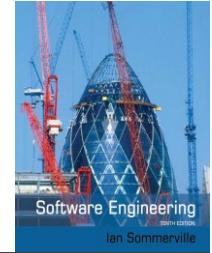


- ✧ Project managers who do not have experience of agile methods may be **reluctant to accept the risk of a new approach**.
- ✧ Large organizations often have **quality procedures and standards that all projects are expected to follow** and, because of their bureaucratic nature, these are likely to be incompatible with agile methods.
- ✧ Agile methods seem to work best when **team members have a relatively high skill level**. However, within large organizations, there are likely to be a **wide range** of skills and abilities.
- ✧ There may be **cultural resistance to agile methods**, especially in those organizations that have a long history of using conventional systems engineering processes.

Kanban (Scrum vs. Kanban)



- ✧ Check Scrum vs Kanban cheat sheet



Key points

- ✧ Agile methods are incremental development methods that focus on rapid software development, frequent releases of the software, reducing process overheads by minimizing documentation and producing high-quality code.
- ✧ Agile development practices include
 - User stories for system specification
 - Frequent releases of the software
 - Continuous software improvement
 - Test-first development
 - Customer participation in the development team.

Key points



- ✧ Scrum is an agile method that provides a project management framework.
 - It is centred round a set of sprints, which are fixed time periods when a system increment is developed.
- ✧ Many practical development methods are a mixture of plan-based and agile development.
- ✧ Scaling agile methods for large systems is difficult.
 - Large systems need up-front design and some documentation and organizational practice may conflict with the informality of agile approaches.



Chapter 4 – Requirements Engineering

Topics covered



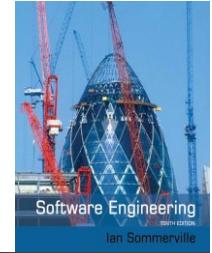
- ✧ Functional and non-functional requirements
- ✧ Requirements engineering processes
- ✧ Requirements elicitation
- ✧ Requirements specification
- ✧ Requirements validation
- ✧ Requirements change

Requirements engineering



- ✧ The process of establishing the services that a customer requires from a system and the constraints under which it operates and is developed.

- ✧ The system requirements are the descriptions of the system services and constraints that are generated during the requirements engineering process.



What is a requirement?

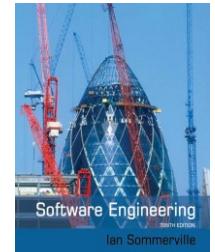
- ✧ It may range from a **high-level** abstract statement of a service or of a system constraint to a **detailed mathematical functional** specification.
- ✧ This is **inevitable** as requirements may serve a dual function
 - May be the basis for a **bid for a contract** - therefore must be open to interpretation;
 - May be the basis for the **contract itself** - therefore must be defined in detail;
 - Both these statements may be called requirements.

Requirements abstraction (Davis)



“If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization’s needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system.”

Types of requirement



✧ User requirements

- Statements in natural language plus diagrams of the services the system provides and its operational constraints. **Written for customers.**

✧ System requirements

- A **structured document** setting out **detailed descriptions of the system's functions, services** and **operational constraints**. Defines what should be implemented so may be part of a contract between client and contractor.

User and system requirements



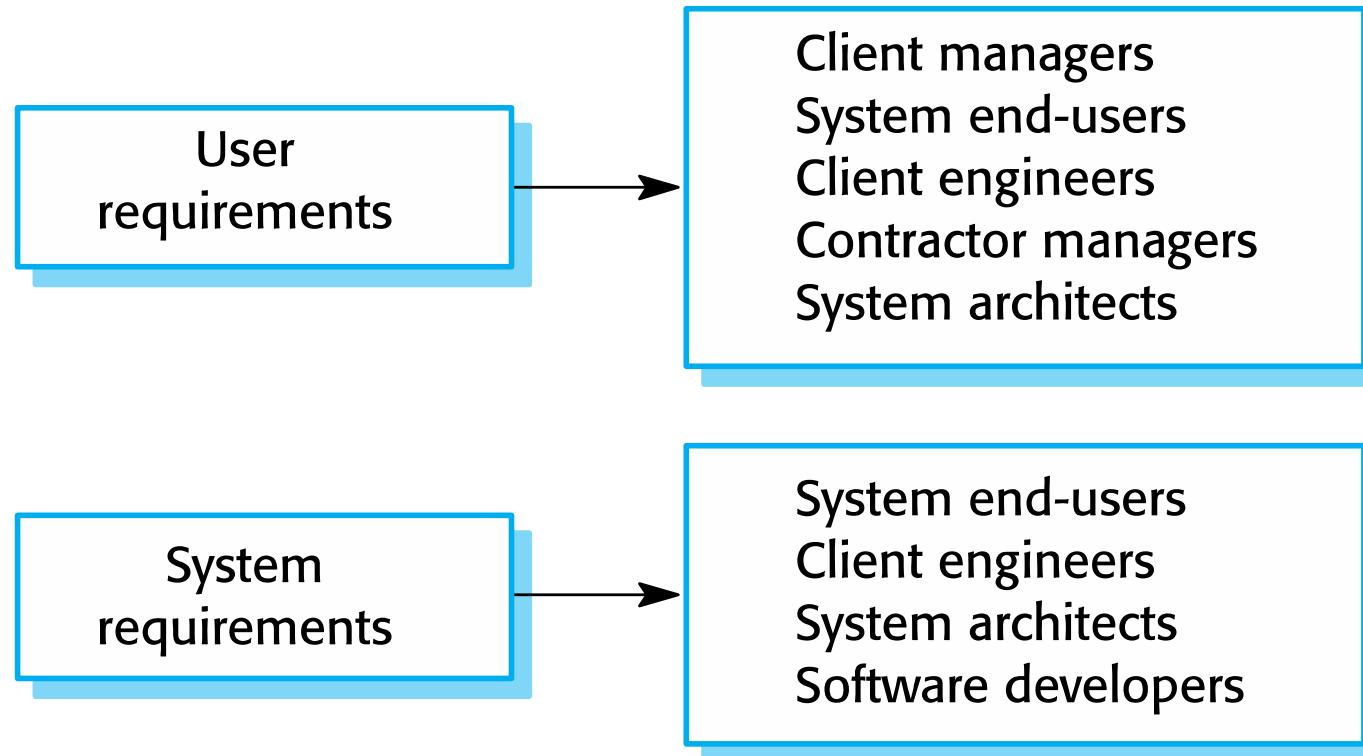
User requirements definition

- 1.** The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

- 1.1** On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
- 1.2** The system shall generate the report for printing after 17.30 on the last working day of the month.
- 1.3** A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
- 1.4** If drugs are available in different dose units (e.g. 10mg, 20mg, etc) separate reports shall be created for each dose unit.
- 1.5** Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

Readers of different types of requirements specification



System stakeholders



- ✧ Any person or organization **who is affected** by the system in some way and so who has a legitimate interest

- ✧ Stakeholder types
 - End users
 - System managers
 - System owners
 - External stakeholders

Stakeholders in the Mentcare system



- ✧ **Patients** whose information is recorded in the system.
- ✧ **Doctors** who are responsible for assessing and treating patients.
- ✧ **Nurses** who coordinate the consultations with doctors and administer some treatments.
- ✧ **Medical receptionists** who manage patients' appointments.
- ✧ **IT staff** who are responsible for installing and maintaining the system.

Stakeholders in the Mentcare system



- ✧ A **medical ethics manager** who must ensure that the system meets current ethical guidelines for patient care.
- ✧ **Health care managers** who obtain management information from the system.
- ✧ **Medical records staff** who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

Agile methods and requirements



- ✧ Many agile methods argue that producing **detailed system requirements** is a **waste of time** as requirements change so quickly.
- ✧ The requirements document is therefore **always out of date**.
- ✧ Agile methods usually use incremental requirements engineering and may express requirements as '**user stories**' (discussed in Chapter 3).
- ✧ This is **practical for business systems** but problematic for systems that require **pre-delivery analysis** (e.g. critical systems) or systems **developed by several teams**.

User story format



As a <type of user>, I want <some goal> so that <some reason>

As an instructor, I want to sort students based on their average marks in a course so that I can supervise the students that may need more help.

As a seller, I want to add items to a shopper's cart manually so I can add shoppers requested items after a shopper's cart has been finalized by the shopper.



Functional and non-functional requirements

Functional and non-functional requirements



✧ Functional requirements

- Statements of **services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.**
- May state what the system should not do.

✧ Non-functional requirements

- **Constraints on the services or functions offered by the system** such as timing constraints, constraints on the development process, standards, etc.
- **Often apply to the system as a whole** rather than individual features or services.

✧ Domain requirements

- Constraints on the system from the domain of operation

Functional requirements



- ✧ Describe **functionality or system services**.
- ✧ Depend on the type of software, expected users and the type of system where the software is used.
- ✧ **Functional user requirements** may be high-level statements of what the system should do.
- ✧ **Functional system requirements** should describe the system services in detail.

Functional requirements for the Mentcare system



- ✧ A user shall be able to search the appointments lists for all clinics.
 - یک کاربر بتواند لیست ملاقات‌های تمامی کیلینک‌ها را جستجو کند.
- ✧ The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
- ✧ Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.

Requirements imprecision



- ✧ Problems arise when requirements are **not precisely stated**.
- ✧ Ambiguous requirements may be **interpreted in different ways by developers and users**.
- ✧ Consider the term 'search' in requirement 1
 - User intention – search for a patient name across all appointments in all clinics;
 - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

Requirements completeness and consistency



- ✧ In principle, requirements should be both **complete** and **consistent**.
- ✧ Complete
 - They should include descriptions of all **facilities required**.
- ✧ Consistent
 - There should be **no conflicts or contradictions** in the descriptions of the system facilities.
- ✧ In practice, because of system and environmental complexity, it is impossible to produce a complete and consistent requirements document.

Non-functional requirements



- ✧ These define **system properties and constraints** e.g. reliability, response time and storage requirements. Constraints on I/O device capability, data representations, etc.
- ✧ Process requirements may also be specified **mandating a particular IDE, programming language or development method**.
- ✧ Non-functional requirements **may be more critical** than functional requirements. If these are not met, the system may be useless.

Non-functional requirements implementation



- ✧ Non-functional requirements may affect the **overall architecture of a system** rather than the individual components.
 - For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- ✧ A single **non-functional requirement**, such as a security requirement, **may generate a number of related functional requirements** that define system services that are required.
 - It may also generate requirements that restrict existing requirements.



Non-functional classifications

✧ Product requirements

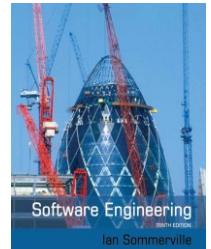
- Requirements which specify that the delivered **product must behave in a particular way** e.g. execution speed, reliability, etc.

✧ Organisational requirements

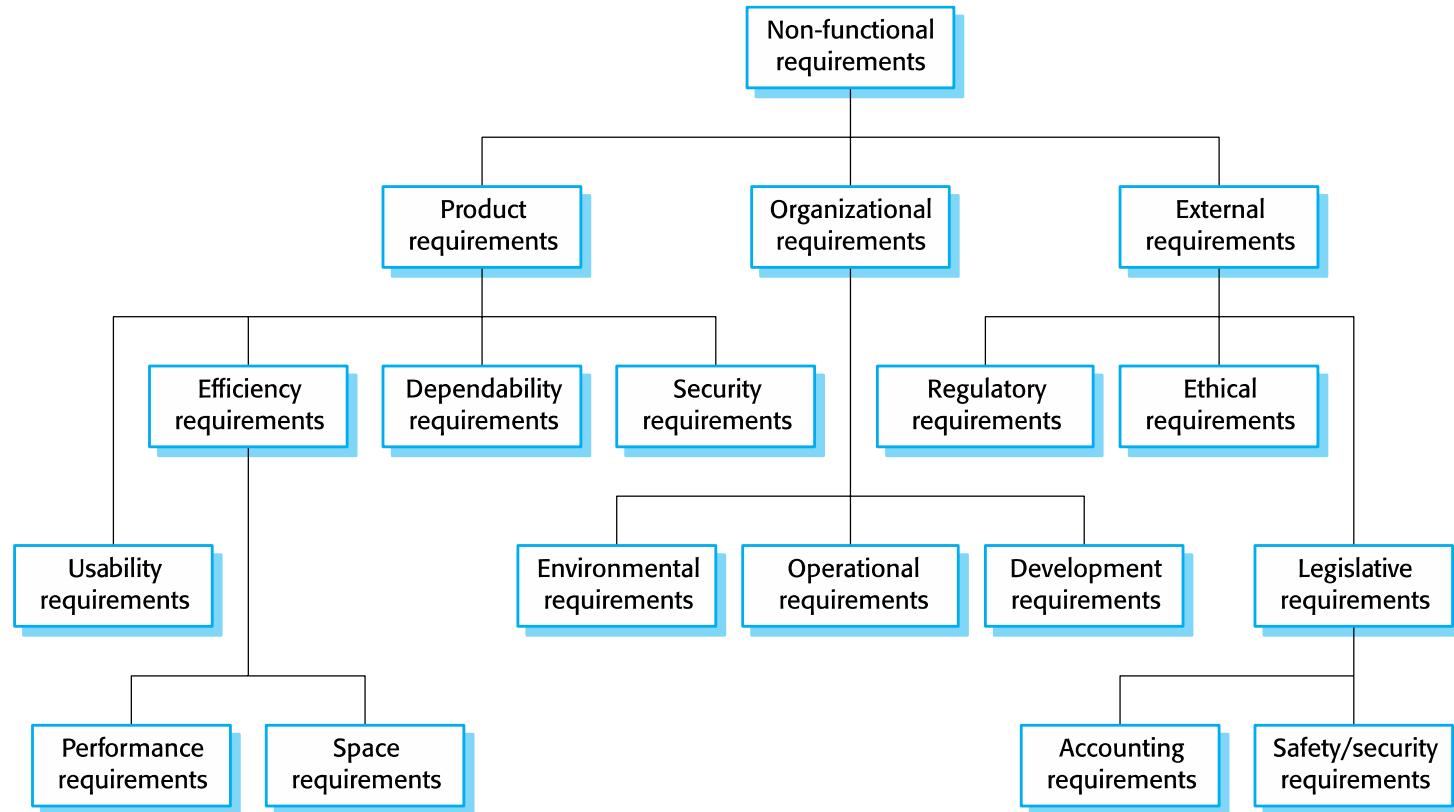
- Requirements which are a **consequence of organisational policies and procedures** e.g. process standards used, implementation requirements, etc.

✧ External requirements

- Requirements which arise from factors which are **external to the system and its development process** e.g. interoperability requirements, legislative requirements, etc.



Types of nonfunctional requirement



Examples of nonfunctional requirements in the Mentcare system



Product requirement

The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 0830–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational requirement

Users of the Mentcare system shall authenticate themselves using their health authority identity card.

External requirement

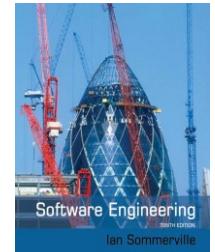
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Goals and requirements



- ✧ Non-functional requirements **may be very difficult to state precisely** and imprecise requirements may be difficult to verify.
 - Example: user specified goal
 - A general intention of the user such as ease of use.
- ✧ Verifiable non-functional requirement
 - A statement using some measure that can be objectively tested.
- ✧ Goals are helpful to developers as they convey the intentions of the system users. However, can cause issues as they may be ambiguous.

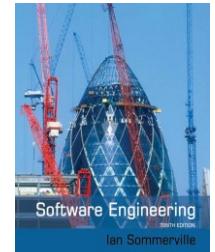
Usability requirements



- ✧ The system **should be easy to use by medical staff** and should be organized in such a way that **user errors are minimized**. (Goal)

- ✧ Medical staff shall be able to use all the system functions **after four hours of training**. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use. (Testable non-functional requirement)

Metrics for specifying nonfunctional requirements



Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems



Availability Examples

- ✧ Carrier airlines (2002 FAA fact book)
 - 41 accidents, 6.7M departures
 - 99.9993% (five nines) availability
- ✧ 911 Phone service (1993 NRIC report)
 - 29 minutes per line per year
 - 99.994% (four nines) availability
- ✧ Standard phone service (various sources)
 - 53+ minutes per line per year
 - 99.99+% (> four nines) availability
- ✧ End-to-end Internet Availability
 - 95% - 99.6% (one to two nines) availability

Domain requirements



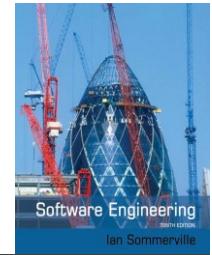
- ✧ The **system's operational domain** imposes requirements on the system.
 - For example, a train control system has to take into account the braking characteristics in different weather conditions.
- ✧ Domain requirements may be **new functional requirements, constraints on existing requirements or define specific computations.**
- ✧ If domain requirements are not satisfied, the system may be unworkable.

Train protection system

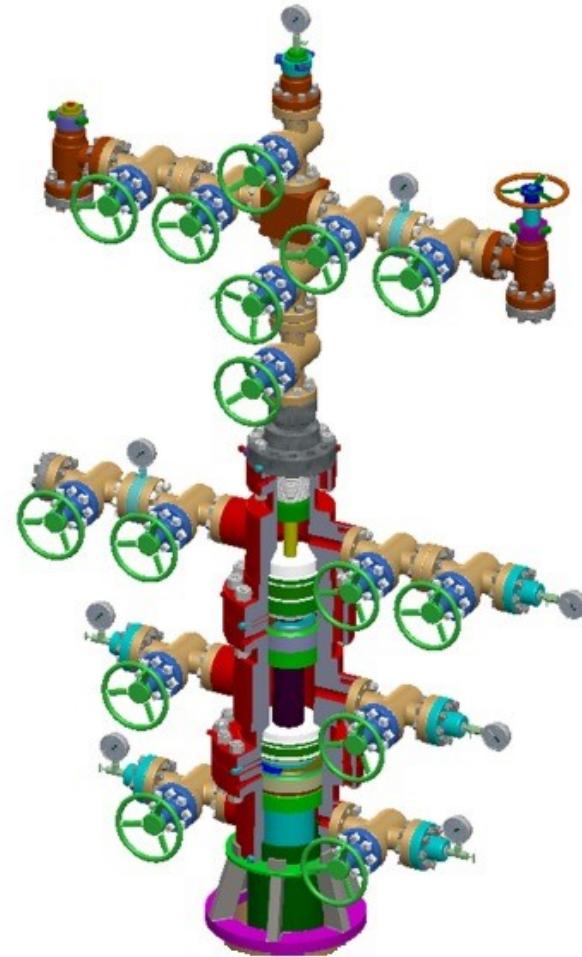


- ✧ This is a domain requirement for a train protection system:
- ✧ The deceleration of the train shall be computed as:
 - $D_{train} = D_{control} + D_{gradient}$
 - where $D_{gradient}$ is $9.81\text{ms}^2 * \text{compensated gradient}/\alpha$ and where the values of $9.81\text{ms}^2 / \alpha$ are known for different types of train.
- ✧ It is **difficult for a non-specialist** to understand the **implications** of this and how it interacts with other requirements.

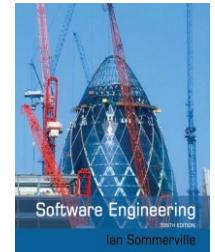
Example



Software Engineering
Ian Sommerville



Domain requirements problems



✧ Understandability

- Requirements are **expressed in the language of the application domain**;
- This is **often not understood** by software engineers developing the system.

✧ Implicitness

- Domain specialists understand the area so well that they **do not think of making the domain requirements explicit**.



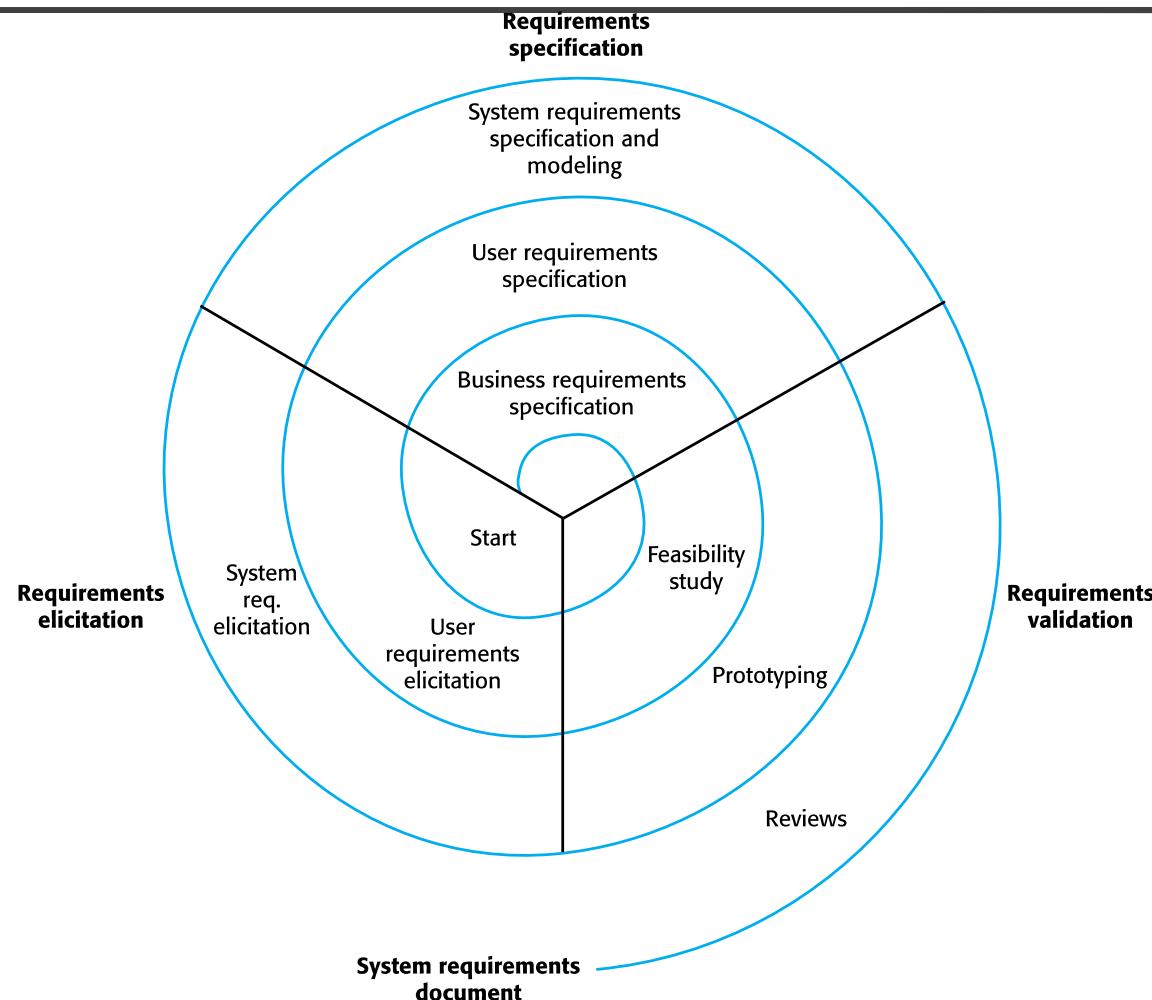
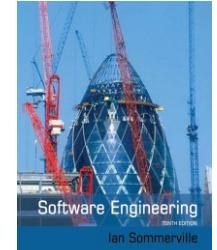
Requirements engineering processes

Requirements engineering processes



- ✧ The processes used for RE vary widely depending on
 - the application domain
 - the people involved
 - the organisation developing the requirements
- ✧ However, there are a number of generic activities common to all processes
 - Requirements elicitation;
 - Requirements analysis;
 - Requirements validation;
 - Requirements management.
- ✧ In practice, RE is an iterative process in which these activities are interleaved.

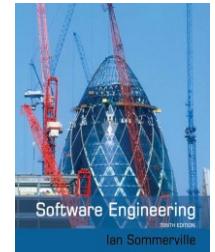
A spiral view of the requirements engineering process





Requirements elicitation

Requirements elicitation and analysis



- ✧ Sometimes called requirements **elicitation** or requirements **discovery**.
- ✧ Involves **technical staff working with customers** to find out about the **system**.
- ✧ May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called **stakeholders**.

Problems of requirements elicitation



- ✧ Stakeholders **don't know what they really want**.
- ✧ Stakeholders express requirements in **their own terms**.
- ✧ Different stakeholders may have **conflicting requirements**.
- ✧ Organisational and political factors **may influence the system requirements**.
- ✧ The requirements change during the analysis process.
New stakeholders may emerge and the **business environment may change**.

Requirements elicitation



- ✧ Software engineers work with a **range of system stakeholders** to find out about the **application domain**, the **services that the system should provide**, the **required system performance**, **hardware constraints**, other **systems**, etc.
- ✧ Stages include:
 - Requirements discovery,
 - Requirements classification and organization,
 - Requirements prioritization and negotiation,
 - Requirements specification.



Process activities

✧ Requirements discovery

- **Interacting with stakeholders** to discover their requirements.
Domain requirements are also discovered at this stage.

✧ Requirements classification and organisation

- **Groups related requirements** and organises them into coherent clusters.

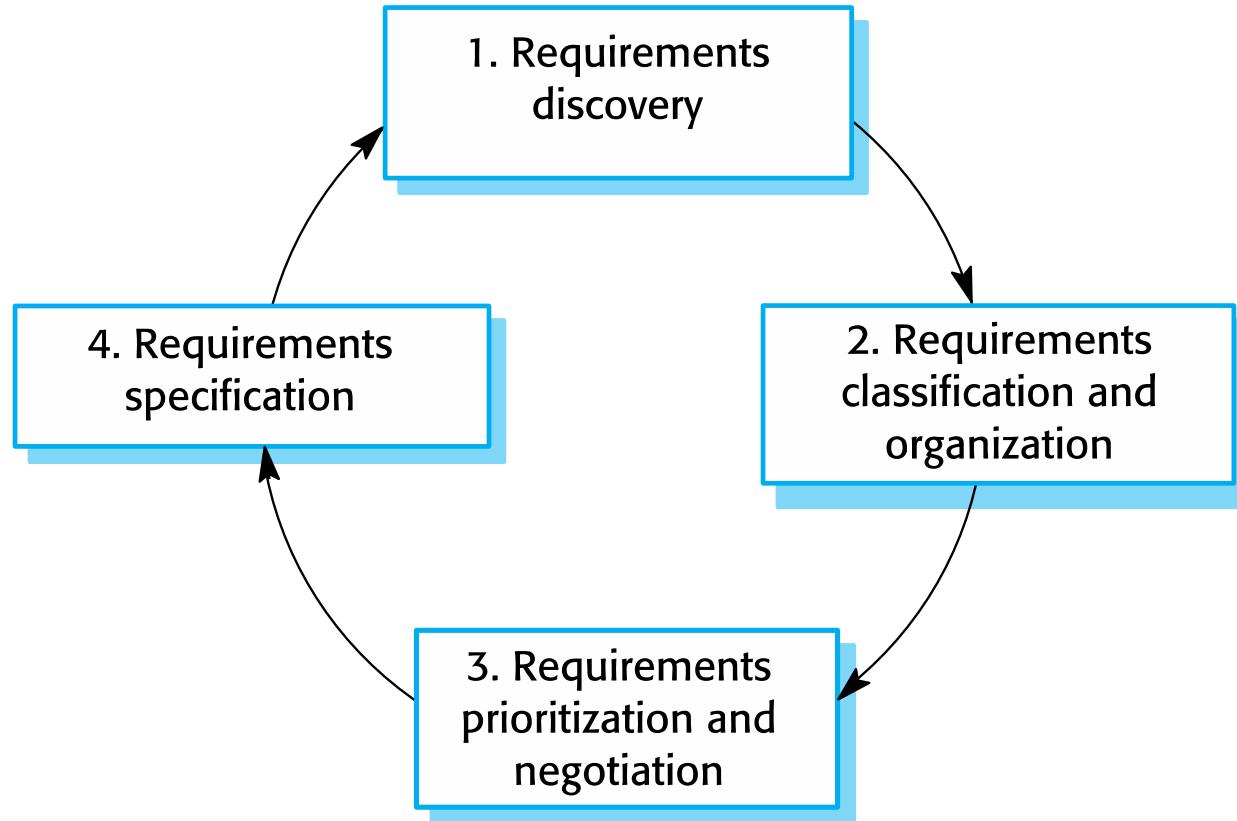
✧ Prioritisation and negotiation

- **Prioritising requirements** and resolving requirements **conflicts**.

✧ Requirements specification

- Requirements are **documented** and input into the next round of the spiral.

The requirements elicitation and analysis process



Requirements discovery



- ✧ The process of gathering information about
 - the required and existing systems
 - And distilling the user and system requirements from this information.
- ✧ Systems normally have a range of stakeholders.
- ✧ Interaction is with system stakeholders from managers to external regulators.

Interviewing

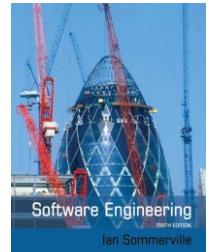


- ✧ Formal or informal interviews with stakeholders are part of most RE processes.
- ✧ Types of interview
 - **Closed** interviews based on pre-determined list of questions
 - **Open** interviews where various issues are explored with stakeholders.
- ✧ Effective interviewing
 - Be **open-minded, avoid pre-conceived ideas** about the requirements and be **willing to listen** to stakeholders.
 - Prompt the interviewee to get discussions going using a **springboard question**, a **requirements proposal**, or by working together on a **prototype system**.

Interviews in practice



- ✧ Normally a **mix of closed and open-ended interviewing**.
- ✧ Interviews are good for **getting an overall understanding of what stakeholders do and how they might interact with the system**.
- ✧ Interviewers need to be **open-minded without pre-conceived ideas** of what the system should do
- ✧ You need to prompt the user to talk about the system by **suggesting requirements** rather than simply asking them what they want.



Problems with interviews

- ✧ Application **specialists** may use **language** to describe their work that **isn't easy** for the requirements engineer to understand.
- ✧ Interviews are **not good** for understanding domain requirements
 - Requirements engineers cannot understand **specific domain terminology**;
 - Some domain knowledge is so familiar that people **find it hard to articulate** or think that **it isn't worth articulating**.

Ethnography



- ✧ A social scientist spends a considerable time observing and analysing how people actually work.
- ✧ People **do not have to explain** or **articulate their work**.
- ✧ Social and organisational factors of importance may be observed.
- ✧ Ethnographic studies have shown that **work is usually richer and more complex than suggested by simple system models**.



Scope of ethnography

- ✧ Requirements that are **derived from the way that people actually work rather than the way which process definitions suggest that they ought to work.**
- ✧ Requirements that are derived from cooperation and awareness of other people's activities.
 - Awareness of what other people are doing leads to changes in the ways in which we do things.
- ✧ Ethnography is **effective for understanding existing processes** but **cannot identify new features** that should be added to a system.

Stories and scenarios



- ✧ Scenarios and user stories are **real-life examples of how a system can be used.**
- ✧ Stories and scenarios are a description of how a system may be **used for a particular task.**
- ✧ Because they are based on a practical situation, **stakeholders can relate to them** and can comment on their situation with respect to the story.

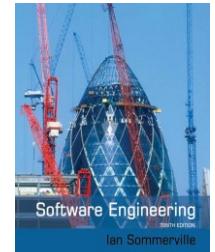
Photo sharing in the classroom (iLearn)



- ✧ Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused around the fishing industry in the area, looking at the history, development and economic impact of fishing. As part of this, pupils are asked to gather and share reminiscences from relatives, use newspaper archives and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAP (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo sharing site as he wants pupils to take and comment on each others' photos and to upload scans of old photographs that they may have in their families.

Jack sends an email to a primary school teachers group, which he is a member of to see if anyone can recommend an appropriate system. Two teachers reply and both suggest that he uses KidsTakePics, a photo sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.

Scenarios



- ✧ A structured form of user story with more details
- ✧ Scenarios should include
 - A description of the starting situation;
 - A description of the normal flow of events;
 - A description of what can go wrong;
 - Information about other concurrent activities;
 - A description of the state when the scenario finishes.

Uploading photos iLearn)



- ✧ **Initial assumption:** A user or a group of users have one or more digital photographs to be uploaded to the picture sharing site. These are saved on either a tablet or laptop computer. They have successfully logged on to KidsTakePics.
- ✧ **Normal:** The user chooses upload photos and they are prompted to select the photos to be uploaded on their computer and to select the project name under which the photos will be stored. They should also be given the option of inputting keywords that should be associated with each uploaded photo. Uploaded photos are named by creating a conjunction of the user name with the filename of the photo on the local computer.
- ✧ On completion of the upload, the system automatically sends an email to the project moderator asking them to check new content and generates an on-screen message to the user that this has been done.

Uploading photos



- ✧ **What can go wrong:**
- ✧ No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed that there could be a delay in making their photos visible.
- ✧ Photos with the same name have already been uploaded by the same user. The user should be asked if they wish to re-upload the photos with the same name, rename the photos or cancel the upload. If they chose to re-upload the photos, the originals are overwritten. If they chose to rename the photos, a new name is automatically generated by adding a number to the existing file name.
- ✧ **Other activities:** The moderator may be logged on to the system and may approve photos as they are uploaded.
- ✧ **System state on completion:** User is logged on. The selected photos have been uploaded and assigned a status 'awaiting moderation'. Photos are visible to the moderator and to the user who uploaded them.

Scenario for collecting medical history in MentCare



INITIAL ASSUMPTION:

The patient has seen a medical receptionist who has created a record in the system and collected the patient's personal information (name, address, age, etc.). A nurse is logged on to the system and is collecting medical history.

NORMAL:

The nurse searches for the patient by family name. If there is more than one patient with the same surname, the given name (first name in English) and date of birth are used to identify the patient.

The nurse chooses the menu option to add medical history.

The nurse then follows a series of prompts from the system to enter information about consultations elsewhere on mental health problems (free text input), existing medical conditions (nurse selects conditions from menu), medication currently taken (selected from menu), allergies (free text), and home life (form).

Scenario for collecting medical history in MentCare



WHAT CAN GO WRONG:

The patient's record does not exist or cannot be found. The nurse should create a new record and record personal information.

Patient conditions or medication are not entered in the menu. The nurse should choose the 'other' option and enter free text describing the condition/medication.

Patient cannot/will not provide information on medical history. The nurse should enter free text recording the patient's inability/unwillingness to provide information. The system should print the standard exclusion form stating that the lack of information may mean that treatment will be limited or delayed. This should be signed and handed to the patient.

OTHER ACTIVITIES:

Record may be consulted but not edited by other staff while information is being entered.

SYSTEM STATE ON COMPLETION:

User is logged on. The patient record including medical history is entered in the database, a record is added to the system log showing the start and end time of the session and the nurse involved.



Requirements specification

Requirements specification



- ✧ The process of writing the user and system requirements in a requirements document.
- ✧ **User requirements** have to be understandable by **end-users and customers** who do not have a technical background.
- ✧ **System requirements** are more **detailed requirements** and may include **more technical** information.
- ✧ The requirements may be **part of a contract** for the system development
 - It is therefore important that these are **as complete as possible**.

Requirements and design



- ✧ In principle, requirements should state what the system should do and the design should describe how it does this.
- ✧ In practice, requirements and design are inseparable
 - A system architecture may be designed to structure the requirements;
 - The system may inter-operate with other systems that generate design requirements;
 - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
 - This may be the consequence of a regulatory requirement.

Ways of writing a system requirements specification



Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

Natural language specification



- ✧ Requirements are written as natural language sentences supplemented by diagrams and tables.

- ✧ Used for writing requirements because it is **expressive, intuitive and universal**. This means that the requirements can be understood by users and customers.

Guidelines for writing requirements



- ✧ Invent a **standard format** and use it for all requirements.
- ✧ Use language in a **consistent way**. Use shall for mandatory requirements, should for desirable requirements.
- ✧ Use **text highlighting** to identify key parts of the requirement.
- ✧ Avoid the use of computer jargon.
- ✧ **Include an explanation** (rationale) of why a requirement is necessary.

Example requirements for the insulin pump software system



- 3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)
- 3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

Problems with natural language



- ✧ Lack of clarity
 - Precision is difficult without making the document difficult to read.
- ✧ Requirements confusion
 - Functional and non-functional requirements tend to be mixed-up.
- ✧ Requirements amalgamation
 - Several different requirements may be expressed together.

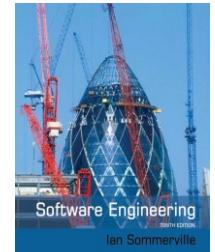
Structured specifications



- ✧ An approach to writing requirements where the **freedom of the requirements writer is limited** and requirements are **written in a standard way**.

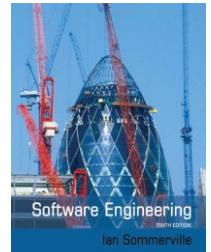
- ✧ This works well for some types of requirements e.g. requirements for **embedded control system** but is sometimes too rigid for writing business system requirements.

Form-based specifications



- ✧ Definition of the **function** or entity.
- ✧ Description of **inputs** and where they **come from**.
- ✧ Description of **outputs** and where they **go to**.
- ✧ Information about the **information needed** for the computation and other entities used.
- ✧ Description of the **action to be taken**.
- ✧ **Pre and post conditions** (if appropriate).
- ✧ The **side effects** (if any) of the function.

A structured specification of a requirement for an insulin pump



Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r_2); the previous two readings (r_0 and r_1).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

A structured specification of a requirement for an insulin pump



Action

CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

Requirements

Two previous readings so that the rate of change of sugar level can be computed.

Pre-condition

The insulin reservoir contains at least the maximum allowed single dose of insulin.

Post-condition r0 is replaced by r1 then r1 is replaced by r2.

Side effects None.



هزینه ورود به طرح ترافیک چقدر است؟

- ۱- اگر خودرویی ورودش بعد از ساعت ۱۰ صبح و خروجش قبل از ساعت ۱۶ ثبت شود، در صورتی که دارای معاینه فنی برتر باشد عوارضش برای آن روز ۱۳ هزار و ۴۰۰ تومان خواهد بود.
- ۲- اگر خودرویی ورودش بعد از ساعت ۱۰ و خروجش قبل از ساعت ۱۶ ثبت شده باشد اما دارای معاینه فنی عادی باشد، عوارض یک روز تردد برای این خودرو ۱۶ هزار و ۸۰۰ تومان خواهد بود.
- ۳- اگر ورود خودرویی به داخل محدوده طرح ترافیک قبل از ساعت ۱۰ و خروجش قبل از ساعت ۱۶ باشد اگر دارای معاینه فنی برتر باشد عوارضش ۱۹ هزار و ۲۰۰ تومان خواهد بود.
- ۴- اگر ورود خودرویی به داخل محدوده طرح ترافیک قبل از ساعت ۱۰ و خروجش قبل از ساعت ۱۶ باشد اگر دارای معاینه فنی عادی باشد نرخ عوارض برای این خودرو ۲۴ هزار تومان خواهد بود.
- ۵- اگر ورود خودرویی قبل از ساعت ۱۰ و خروج آن نیز بعد از ساعت ۱۶ باشد اگر این خودرو معاینه فنی برتر داشته باشد نرخ عوارض ۲۸ هزار و ۸۰۰ تومان خواهد بود.
- ۶- اگر ورود خودرویی قبل از ساعت ۱۰ و خروج آن نیز بعد از ساعت ۱۶ باشد اگر این خودرو با این شرایط دارای معاینه فنی عادی باشد ۳۶ هزار تومان عوارض برای آن محاسبه می‌شود.

عوارض طرح ترافیک



بعد از ۱۹ ۱۶-۱۹ ۱۰-۱۶ ۶:۳۰-۱۰ قبل از ۶:۳۰

۱۹
د

بعد از ۱۹	۱۶-۱۹	۱۰-۱۶	۶:۳۰-۱۰	قبل از ۶:۳۰	
.				.	قبل از ۶:۳۰
					۶:۳۰-۱۰
		۱۶۸۰۰ ۱۳۴۰۰	۲۴۰۰۰ ۱۹۲۰۰		۱۰-۱۶
			۳۶۰۰۰ ۲۸۸۰۰		۱۶-۱۹
.				.	بعد از ۱۹



Tabular specification



- ✧ Used to **supplement natural language**.
- ✧ Particularly useful when you have to **define a number of possible alternative** courses of action.
- ✧ For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.

Tabular specification of computation for an insulin pump

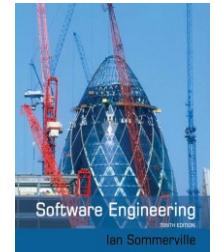


Condition	Action
Sugar level falling ($r_2 < r_1$)	$\text{CompDose} = 0$
Sugar level stable ($r_2 = r_1$)	$\text{CompDose} = 0$
Sugar level increasing and rate of increase decreasing $((r_2 - r_1) < (r_1 - r_0))$	$\text{CompDose} = 0$
Sugar level increasing and rate of increase stable or increasing $((r_2 - r_1) \geq (r_1 - r_0))$	$\text{CompDose} = \text{round}((r_2 - r_1)/4)$ If rounded result = 0 then $\text{CompDose} = \text{MinimumDose}$

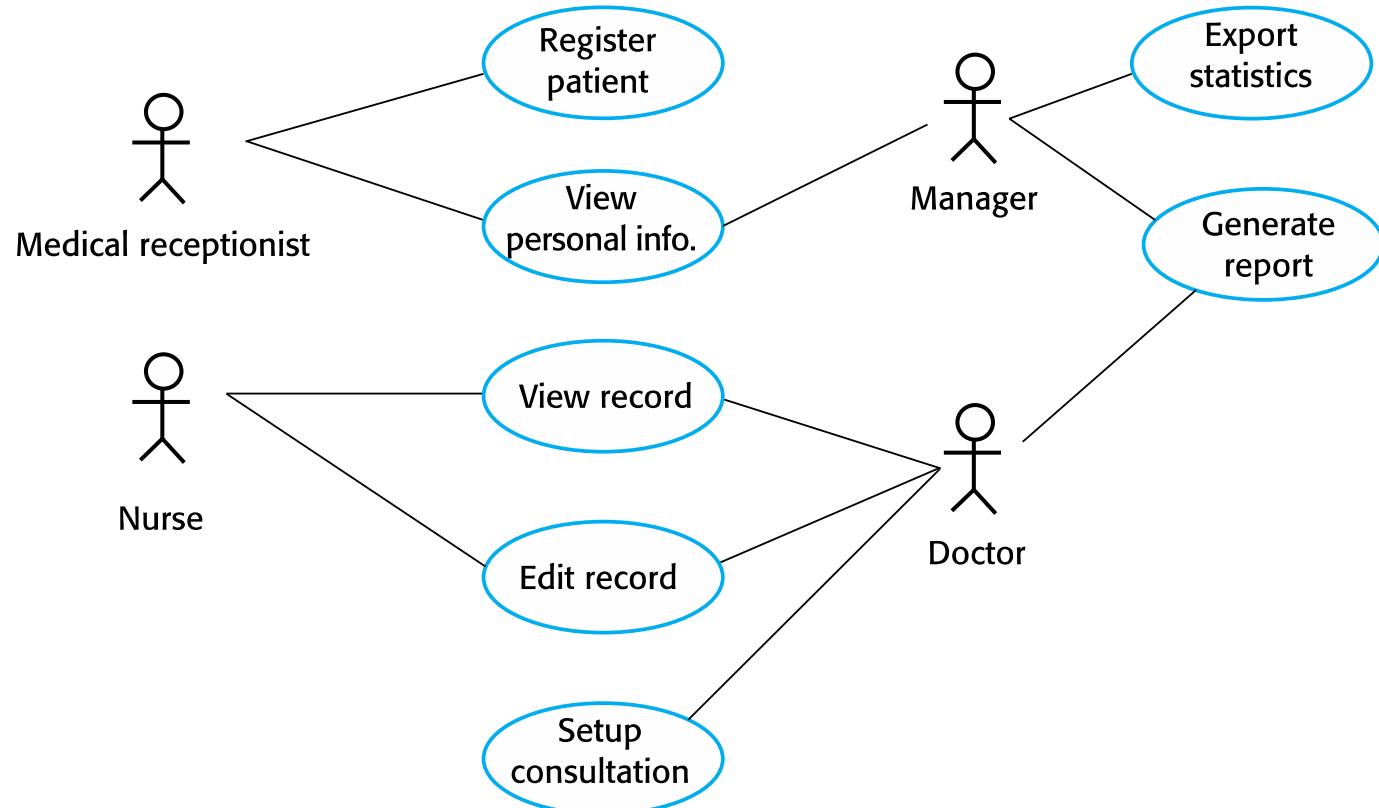
Use cases



- ✧ Use-cases are a **kind of scenario** that are included in the UML.
- ✧ Use cases **identify the actors** in an interaction and which describe the interaction itself.
- ✧ A set of use cases should describe all **possible interactions with the system**.
- ✧ High-level graphical model supplemented by more detailed tabular description (see Chapter 5).
- ✧ UML **sequence diagrams** may be used to add detail to use-cases by showing the sequence of event processing in the system.



Use cases for the Mentcare system



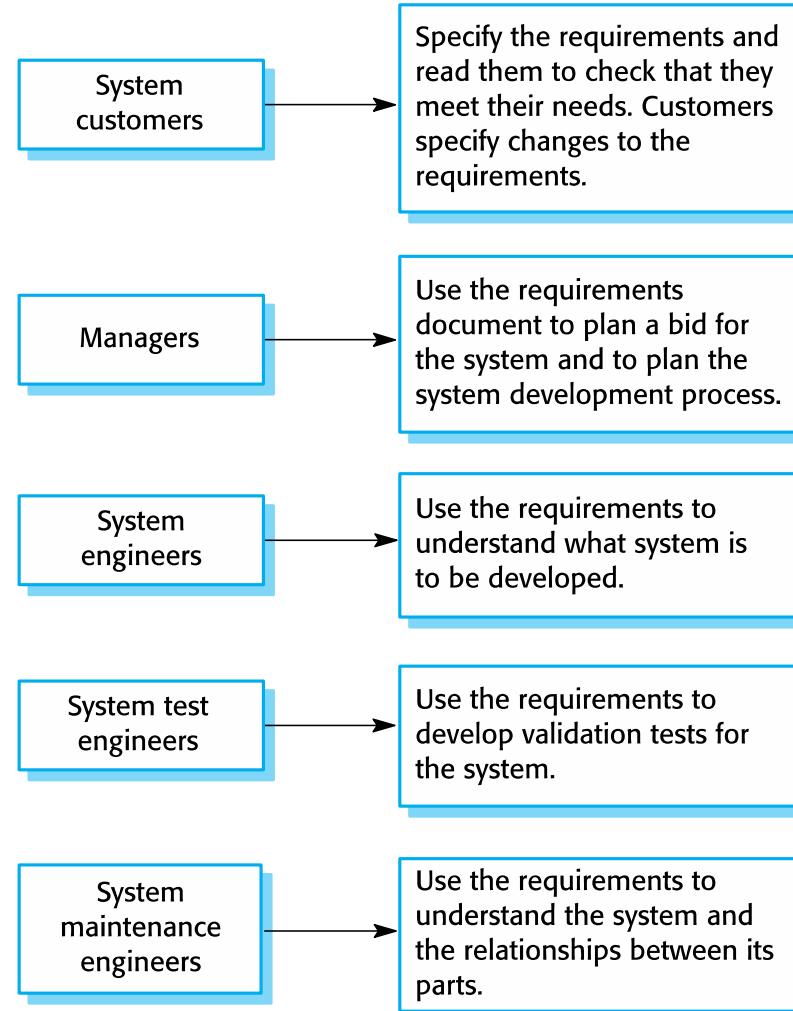
The software requirements document



- ✧ The software requirements document is the **official statement** of what is required of the system developers.
- ✧ Should include both a definition of **user requirements** and a specification of the **system requirements**.
- ✧ It is **NOT** a design document. As far as possible, it should set of **WHAT** the system should do rather than **HOW** it should do it.



Users of a requirements document



Requirements document variability



- ✧ Information in requirements document depends
 - type of system
 - the development approach used
- ✧ Systems developed incrementally will, typically, have less detail in the requirements document.
- ✧ Requirements documents standards have been designed e.g. IEEE standard.
 - These are mostly applicable to the requirements for large systems engineering projects.

The structure of a requirements document



Chapter	Description
Preface	This should define the expected readership of the document and describe its version history , including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system . It should briefly describe the system's functions and explain how it will work with other systems . It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers . Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture , showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.

The structure of a requirements document



Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail . If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined .
System models	This might include graphical system models showing the relationships between the system components and the system and its environment . Examples of possible models are object models, data-flow models, or semantic data models .
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs , and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.



Requirements validation

Requirements validation



- ✧ Concerned with demonstrating that the requirements **define the system that the customer really wants.**

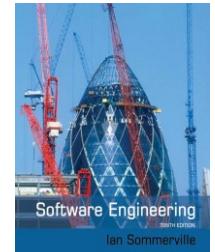
- ✧ Requirements **error costs are high** so validation is very important
 - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking



- ✧ **Validity.** Does the system provide the functions which best support the customer's needs?
- ✧ **Consistency.** Are there any requirements conflicts?
- ✧ **Completeness.** Are all functions required by the customer included?
- ✧ **Realism.** Can the requirements be implemented given available budget and technology
- ✧ **Verifiability**
 - Is the requirement realistically testable?

Requirements validation techniques



✧ Requirements reviews

- Systematic manual analysis of the requirements.

✧ Prototyping

- Using an executable model of the system to check requirements.
Covered in Chapter 2.

✧ Test-case generation

- Developing tests for requirements to check testability.

Requirements reviews



- ✧ Regular reviews should be held while the requirements definition is being formulated.
- ✧ Both client and contractor staff should be involved in reviews.
- ✧ Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.



Requirements change

Changing requirements



- ✧ The business and technical environment of the system always changes during development/after deployment.
 - New hardware may be introduced
 - It may be necessary to interface the system with other systems
 - Business priorities may change (with consequent changes in the system support required)
 - New legislation and regulations may be introduced that the system must necessarily abide by

for example using google systems for authentication

Changing requirements



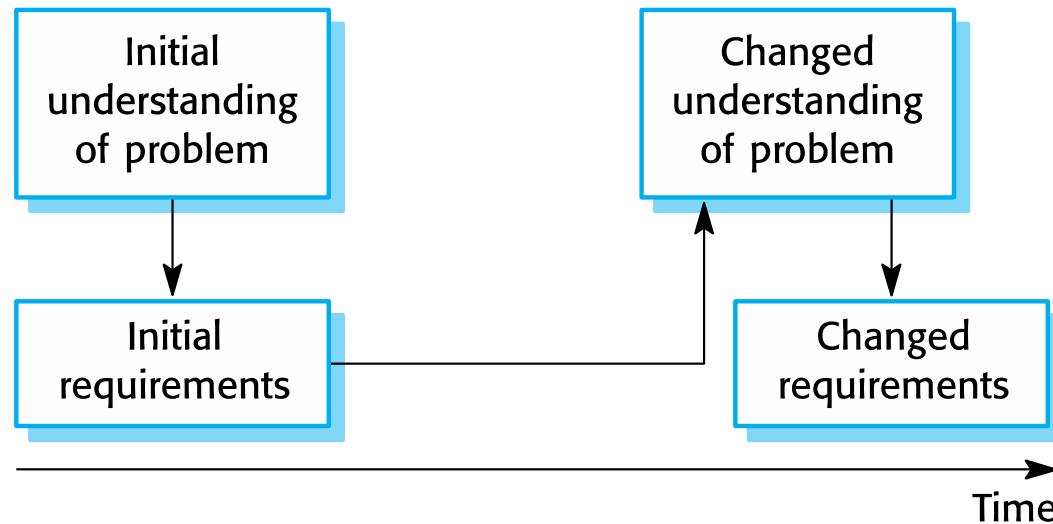
- ✧ The people who pay for a system and the users of that system are rarely the same people.
 - System customers impose requirements because of organizational and budgetary constraints.
 - These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.

Changing requirements



- ✧ Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.
 - The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements evolution



Requirements management



- ✧ Managing changing requirements during
 - The requirements engineering
 - The system development
- ✧ New requirements emerge as a system is **being developed and after** it has gone into use (deployed).
- ✧ You need to:
 - track individual requirements
 - maintain links between dependent requirements so that you can assess the impact of requirements changes.
 - establish a formal process for making change proposals and linking these to system requirements.



Requirements management planning

- ✧ Establishes the level of requirements management detail that is required.
- ✧ Requirements management decisions:
 - *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
 - *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
 - *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
 - *Tool support* Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements change management



✧ Deciding if a requirements change should be accepted

- *Problem analysis and change specification*

- Problem of the change proposal is analyzed for validity. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

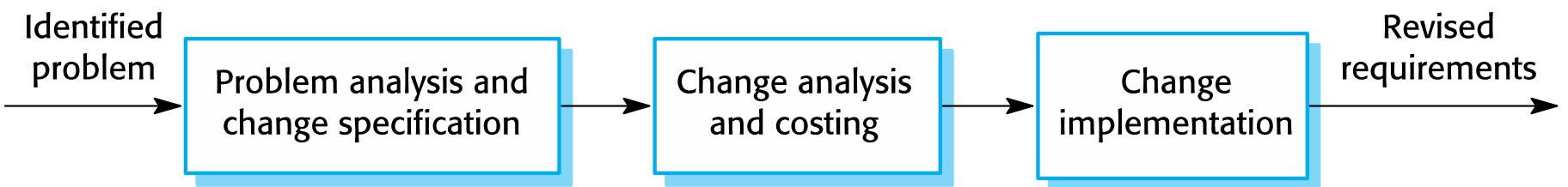
- *Change analysis and costing*

- The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.

- *Change implementation*

- The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

Requirements change management



Key points



- ✧ Requirements for a software system set out **what the system should do** and **define constraints** on its **operation and development**
- ✧ Functional requirements are statements of the **services that the system must provide** or are descriptions of how some computations must be carried out.
- ✧ Non-functional requirements **often constrain the system being developed** and the **development process** being used.
 - They often relate to the emergent properties of the system and therefore **apply to the system as a whole**.

Key points



- ✧ The **requirements engineering** process is an **iterative process** that includes requirements **elicitation**, **specification** and **validation**.
- ✧ Requirements elicitation is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.
- ✧ You can use a range of techniques for **requirements elicitation** including **interviews** and **ethnography**. User stories and scenarios may be used to facilitate discussions.

Key points



- ✧ Requirements specification is the process of **formally documenting the user and system requirements** and creating a software requirements document.
- ✧ The software requirements document is **an agreed statement of the system requirements**. It should be organized so that **both system customers and software developers** can use it.

Key points



- ✧ Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism and verifiability.
- ✧ Business, organizational and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.



Chapter 5 – System Modeling

Topics covered



- ✧ Context models
- ✧ Interaction models
- ✧ Structural models
- ✧ Behavioral models

System modeling



✧ System modeling

- Developing abstract models of a system
- Each model presenting a different view or perspective of that system.
- Almost always based on notations in the Unified Modeling Language (UML).

✧ Usage

- Helps analysts to understand the functionality of the system
- Are used to communicate with customers

Existing and planned system models



✧ Models of the existing system

- Used during requirements engineering
- They help clarify what the existing system does
- Can be used as a basis for discussing its strengths and weaknesses
- May lead to requirements for a new system

✧ Models of the new system

- Used during requirements engineering to help explain the proposed requirements to other system stakeholders
- Engineers use these models to discuss design proposals
- Document the system for implementation

System perspectives



- ✧ An external perspective
 - where you model the **context or environment** of the system.
- ✧ An interaction perspective
 - where you model the **interactions between a system and its environment**, or **between the components** of a system.
- ✧ A structural perspective
 - where you model the **organization of a system**
 - or the **structure of the data** that is processed by the system
- ✧ A behavioral perspective
 - where you model the **dynamic behavior** of the system
 - and how it responds to events.



UML diagram types

- ✧ Activity diagrams
 - show the **activities involved in a process or in data processing**
- ✧ Use case diagrams
 - show the interactions between a **system and its environment**
- ✧ Sequence diagrams
 - show interactions between **actors and the system and between system components.**

UML diagram types



✧ Class diagrams

- show the object **classes** in the system and the **associations** between these classes.

✧ State diagrams

- show how the **system reacts** to internal and external **events**

Use of graphical models



- ✧ As a means of **facilitating discussion** about an existing or proposed system
 - Incomplete and incorrect models are OK as their role is to support discussion.
- ✧ As a way of **documenting** an existing system
 - Models should be an accurate representation of the system but need not be complete.
- ✧ As a **detailed system description** that can be used to generate a system implementation
 - Models have to be both correct and complete.



Context models

Context models

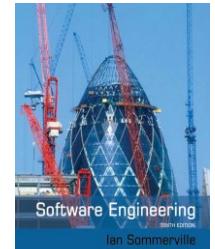


- ✧ Context models are used to **illustrate the operational context of a system** - they show **what lies outside** the system boundaries.
- ✧ Social and organisational concerns may affect the decision on where to position system boundaries.
- ✧ Architectural models show the system and its relationship with other systems.

System boundaries

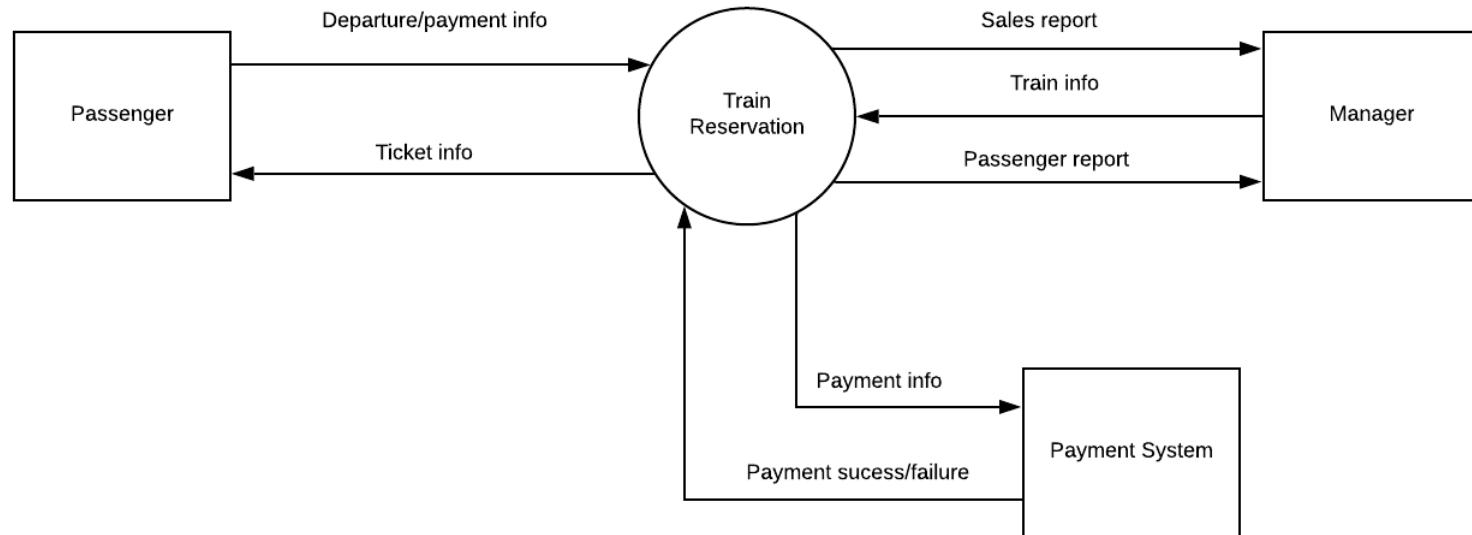


- ✧ System boundaries are established to define what is inside and what is outside the system.
 - They show other systems that are used or depend on the system being developed.
- ✧ The position of the system boundary has a profound effect on the system requirements.
- ✧ Defining a system boundary is a political judgment
 - There may be pressures to develop system boundaries that increase / decrease the influence or workload of different parts of an organization.



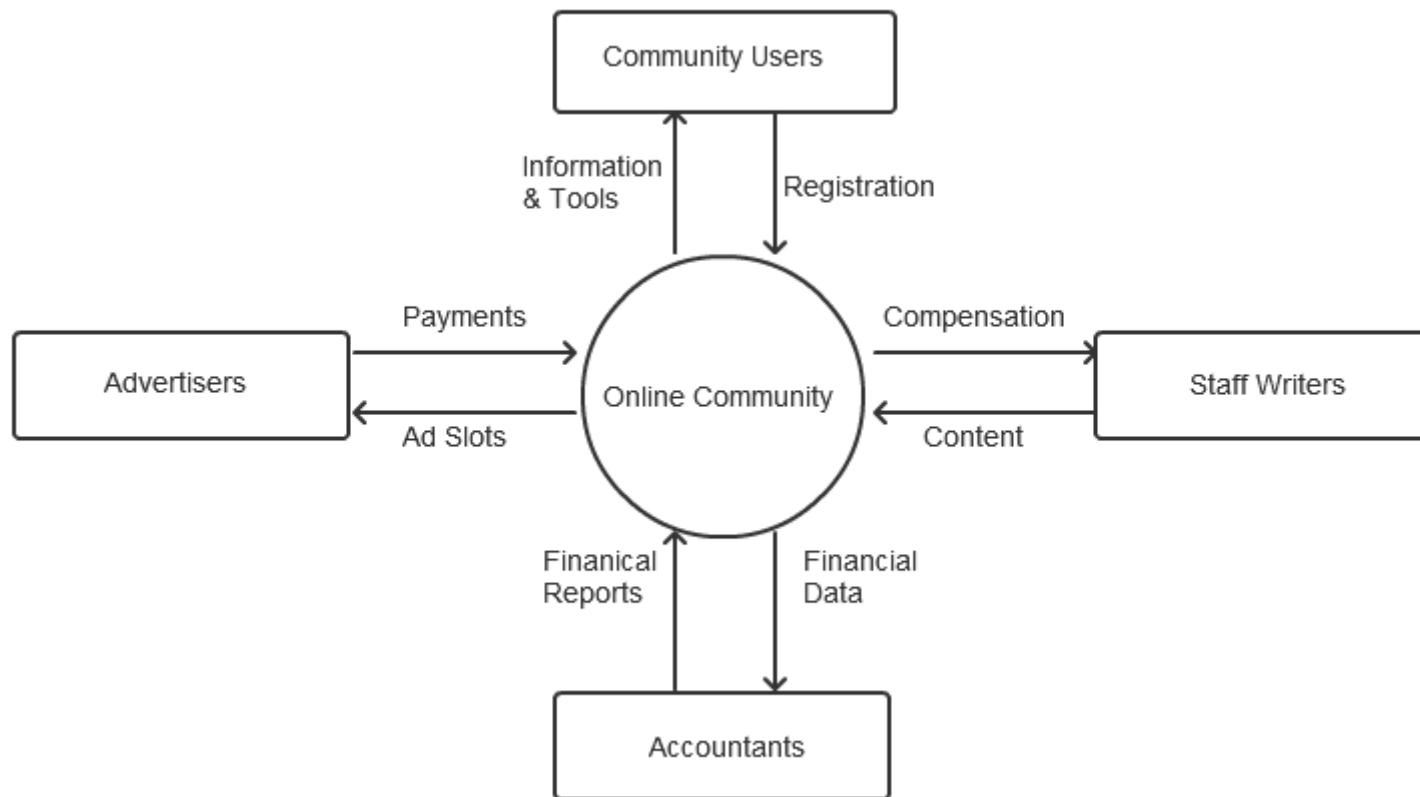
Data Flow Diagram (Level 0 – Context-level DFD)

the last zoom out perspective of our project

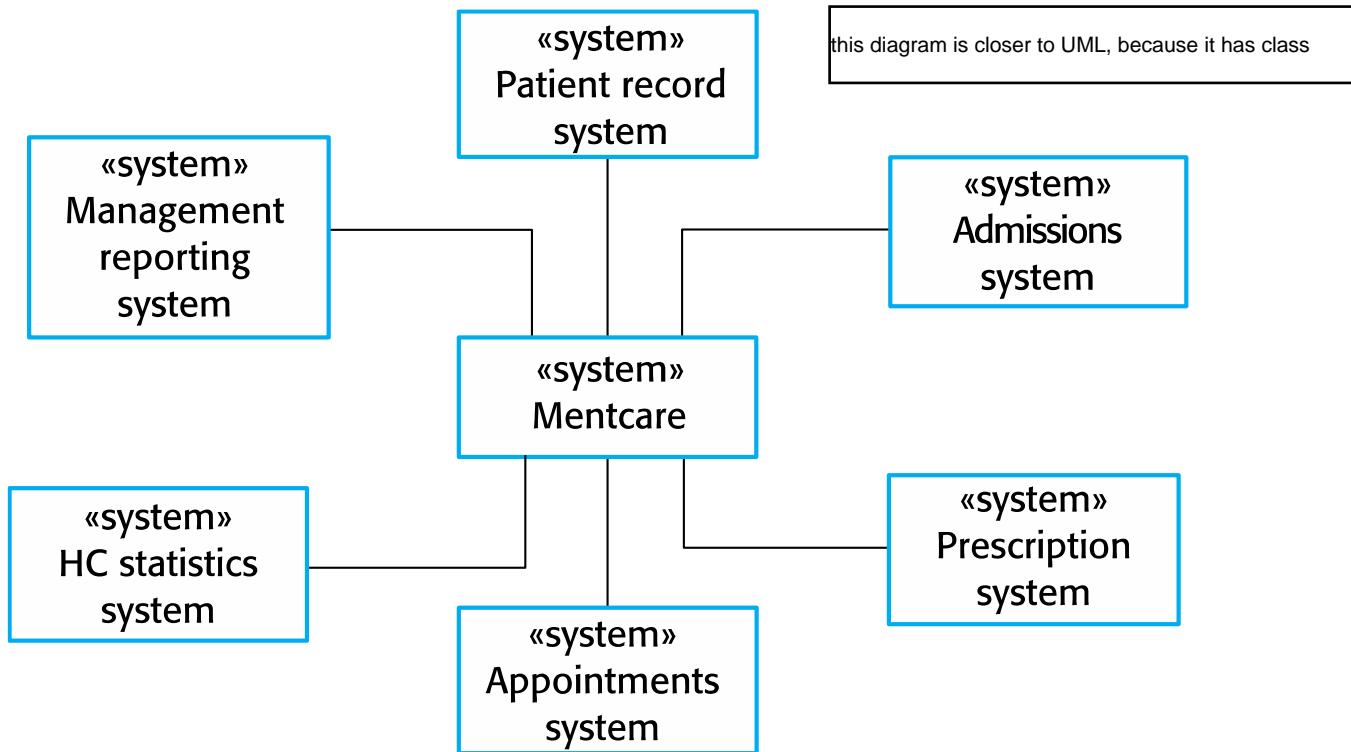




Data Flow Diagram (Level 0 – Context-level DFD)



The context of the Mentcare system

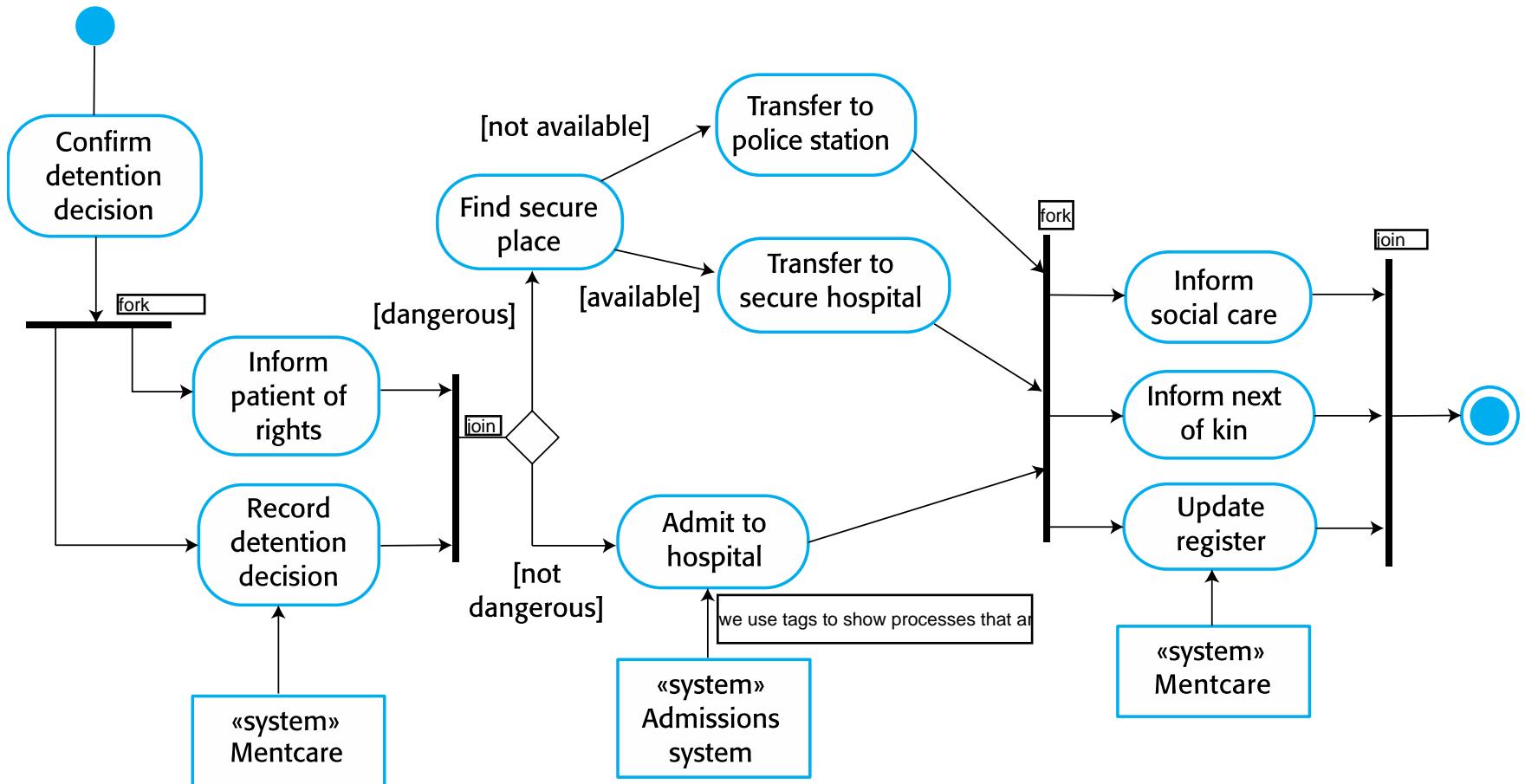


Process perspective



- ✧ Context models simply **show the other systems in the environment**
 - Not how the system being developed is used in that environment.
- ✧ **Process models** reveal **how the system being developed is used** in broader business processes.
- ✧ **UML activity diagrams** may be used to define business process models.

Process model of involuntary detention



Activity diagram

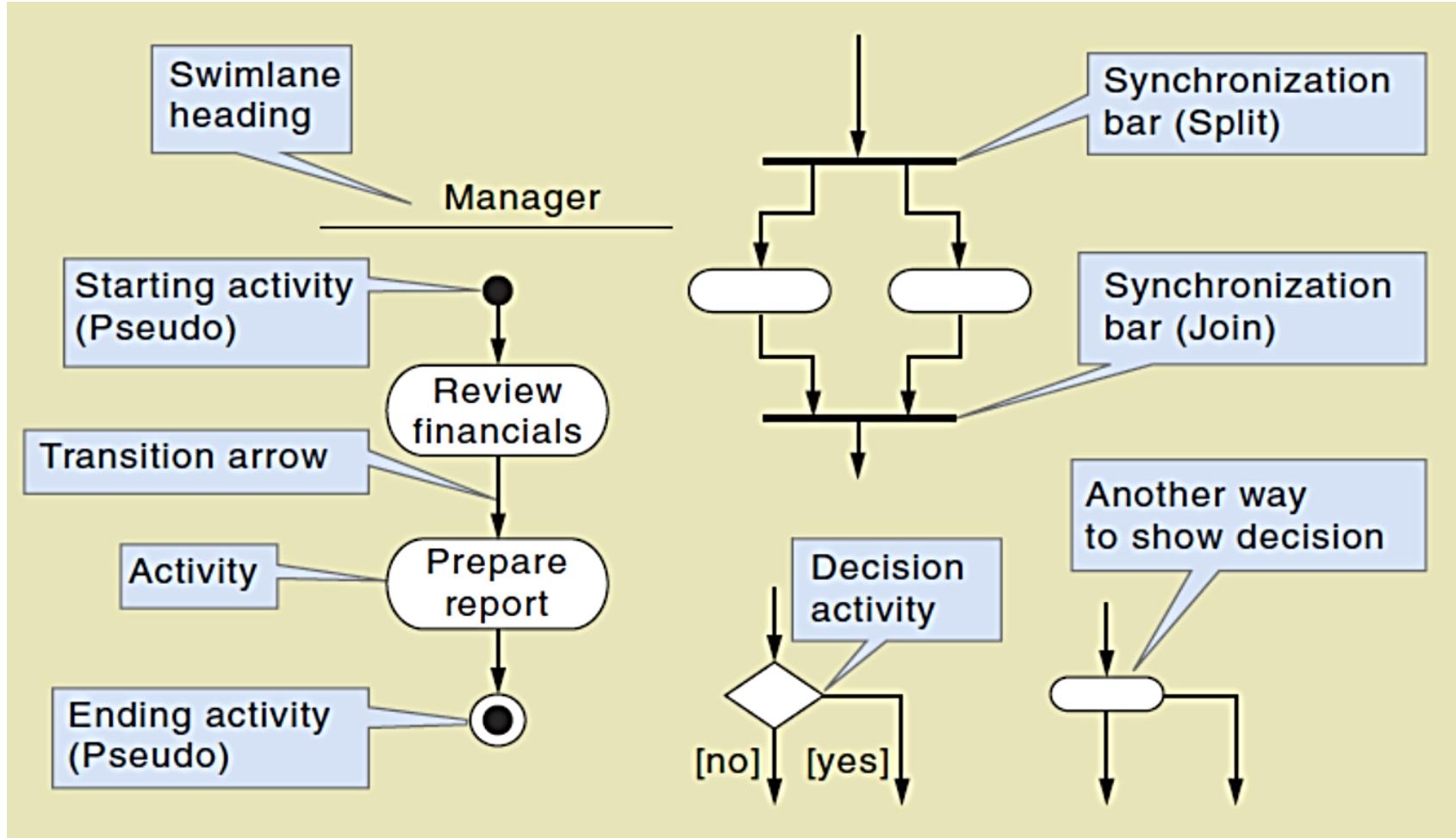


- Activities that make up a system process
- Starts with 
- Ends with 
- Rounded rectangles show activities
- Arrows represent the flow of work
- Solid bar is used to indicate activity coordination (synchronization)
 - Fork (split) and Join

Activity Diagram Symbols



Software Engineering
Ian Sommerville



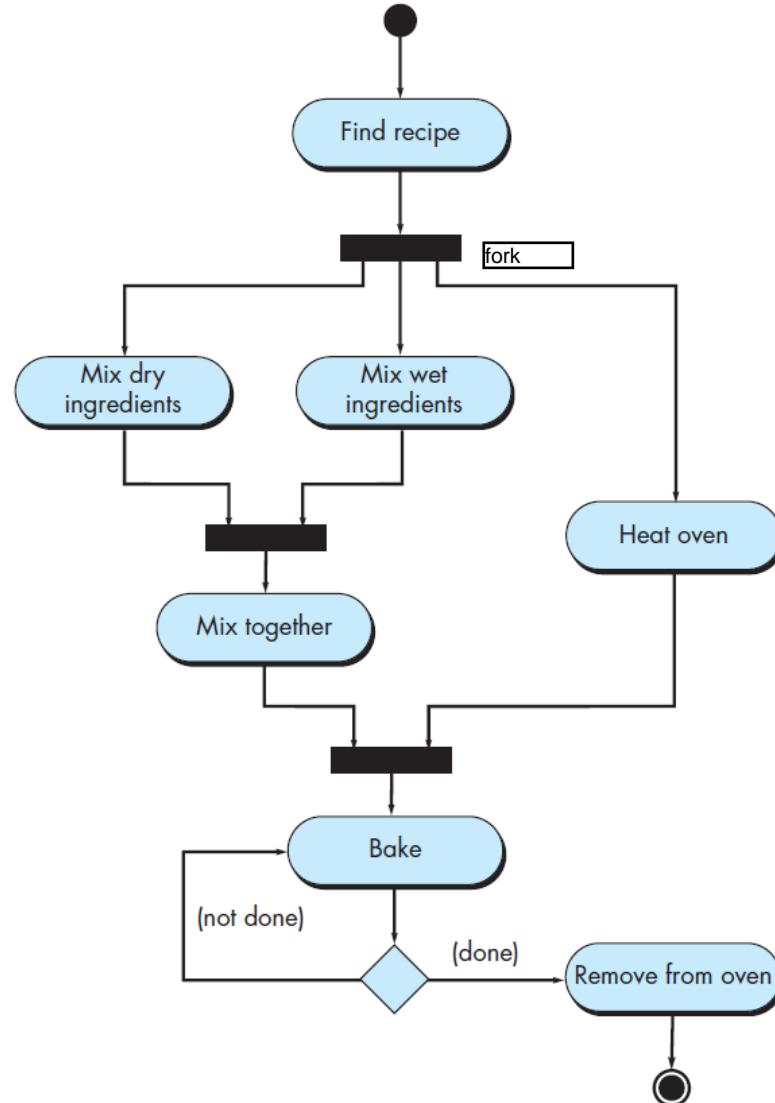
Activity diagram



- Draw the activity diagram of coming to school in the morning
- From wake up time until you arrive to the school
- Consider parallel activities, conditions

Activity diagram - another example

making a cake

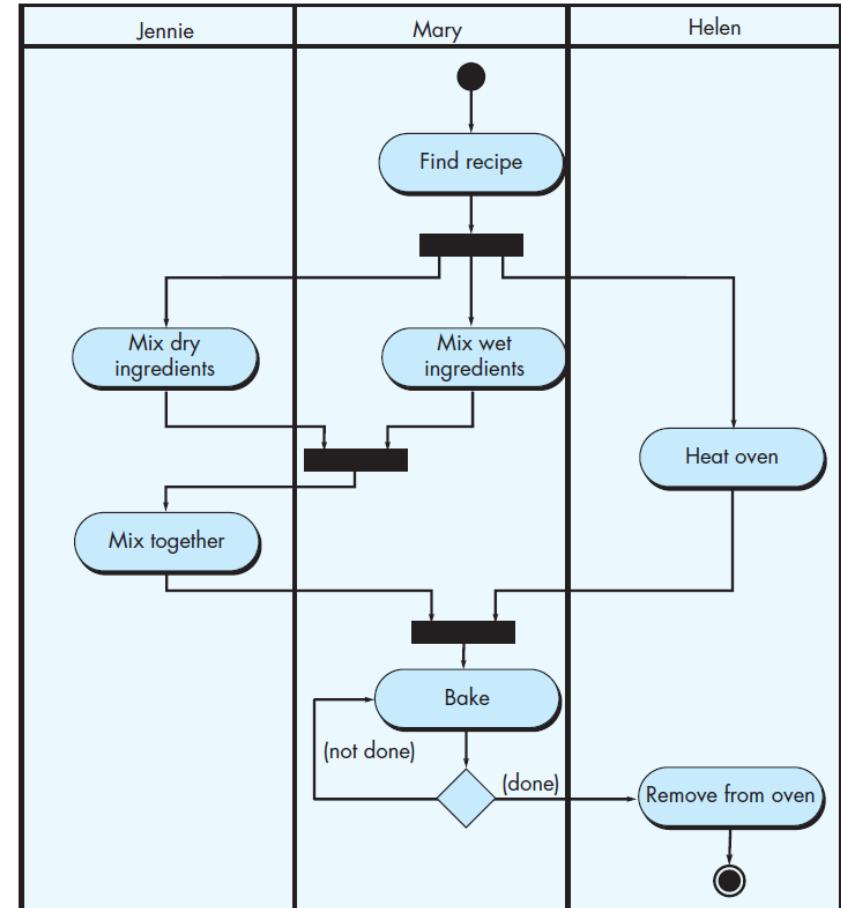


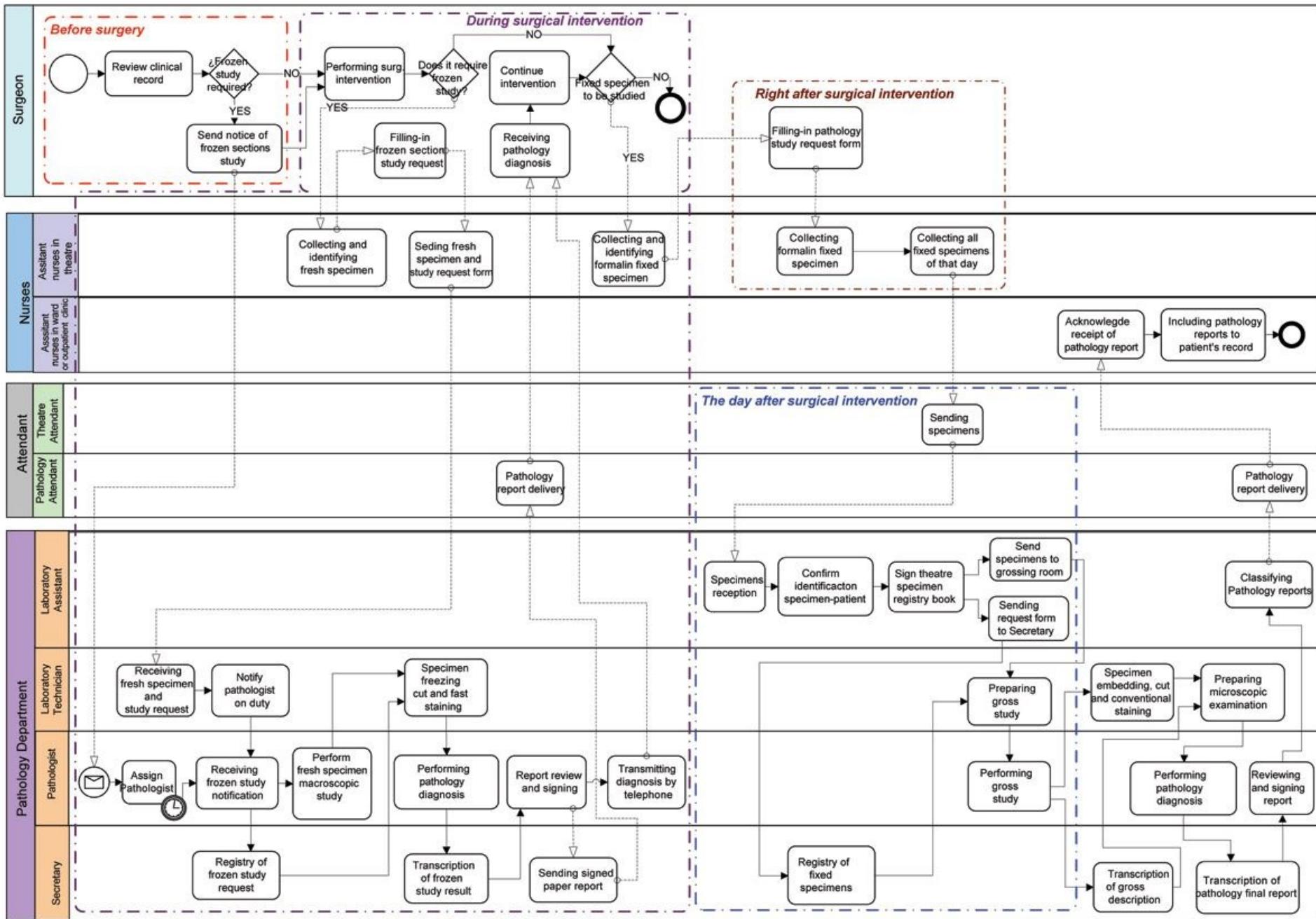
Activity diagram - swimlanes



Software Engineering
Ian Sommerville

- Allows to show who or what is responsible for each action
- Each individual/thing has its own lane







Interaction models



Interaction models

- ✧ Modeling **user** interaction is important as it helps to identify **user requirements**.
- ✧ Modeling **system-to-system** interaction **highlights the communication problems that may arise**.
- ✧ Modeling **component interaction** helps us understand if a proposed system structure is likely to deliver the required **system performance and dependability**.
- ✧ **Use case diagrams and sequence diagrams** may be used for interaction modelling.

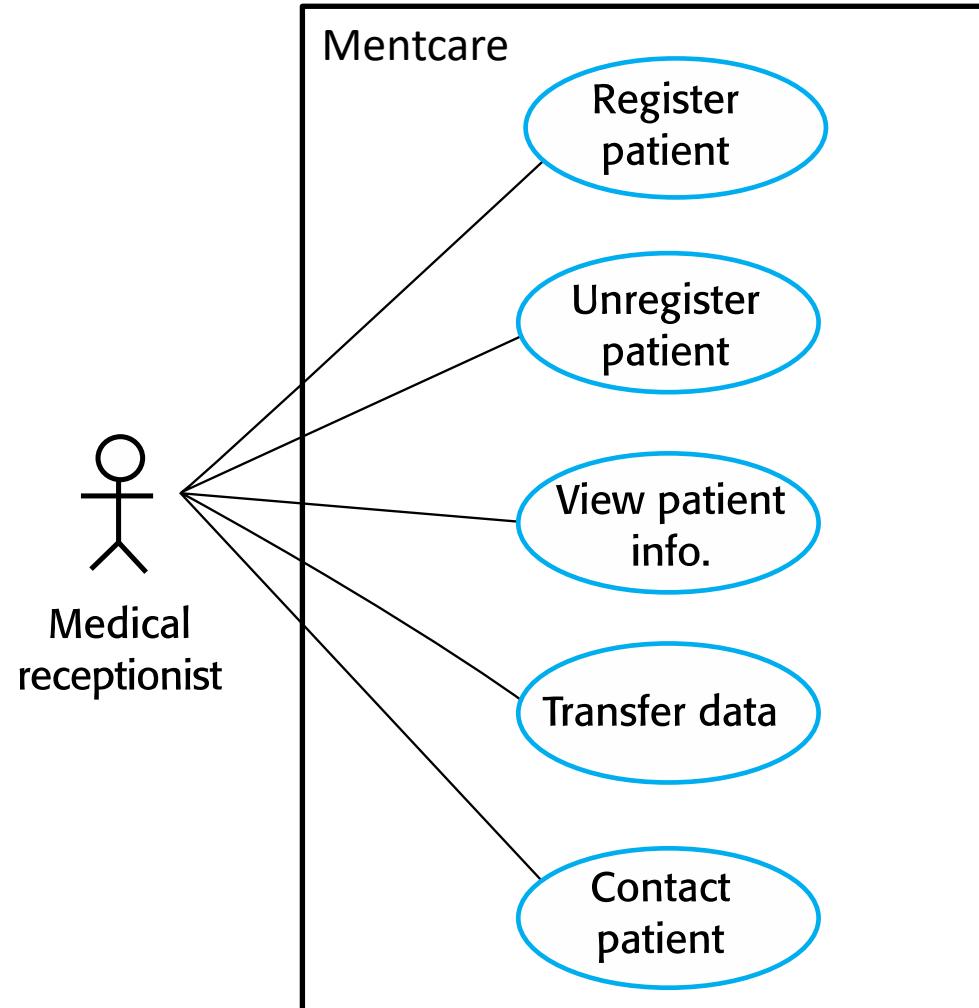
when the components are external we use sequence diagrams

Use case modeling



- ✧ Use cases were developed originally to **support requirements elicitation** and now incorporated into the UML.
- ✧ Each use case represents a discrete task that involves **external interaction with a system**.
- ✧ Actors in a use case may be **people** or other **systems**.
- ✧ Represented diagrammatically to provide an overview of the use case and in a **more detailed textual form**.

Use cases in the Mentcare system involving the role 'Medical Receptionist'

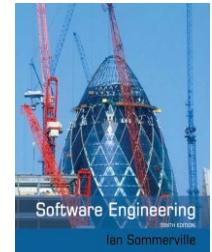


Example



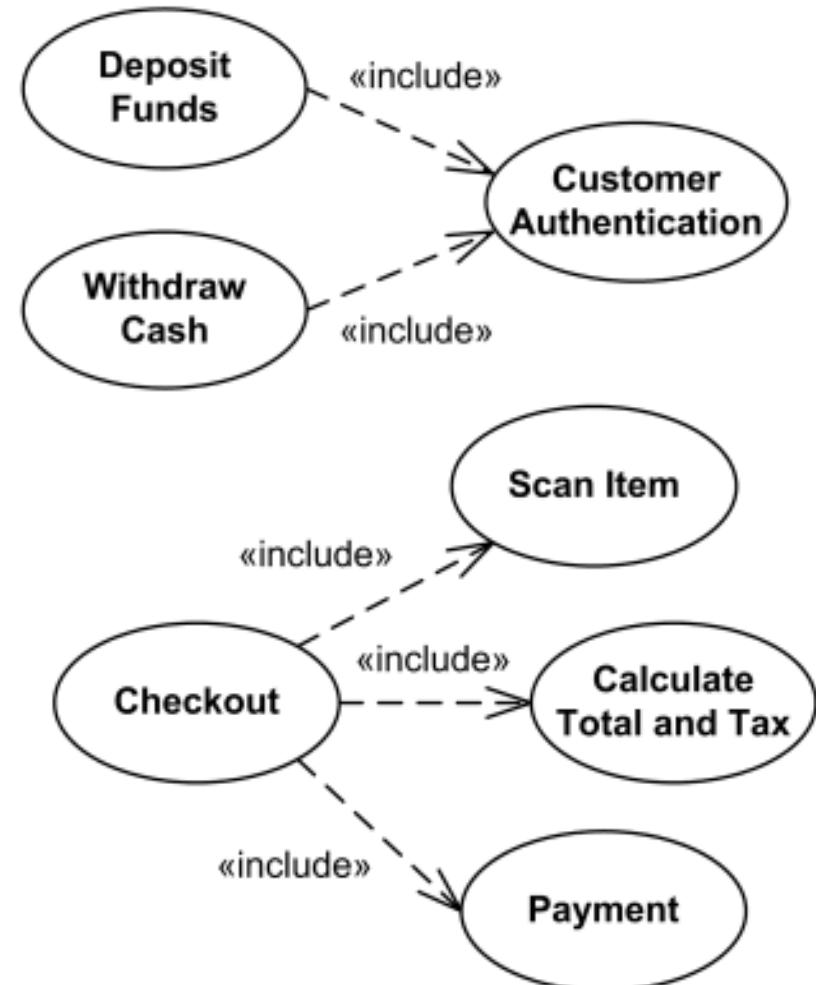
- ✧ Draw a use case diagram for a simple ATM
- ✧ Include use cases such:
 - Withdraw
 - Deposit
 - Transfer
 - User authentication
- ✧ Who are the actors?
 - Include two actors

Include relationship



❖ Use:

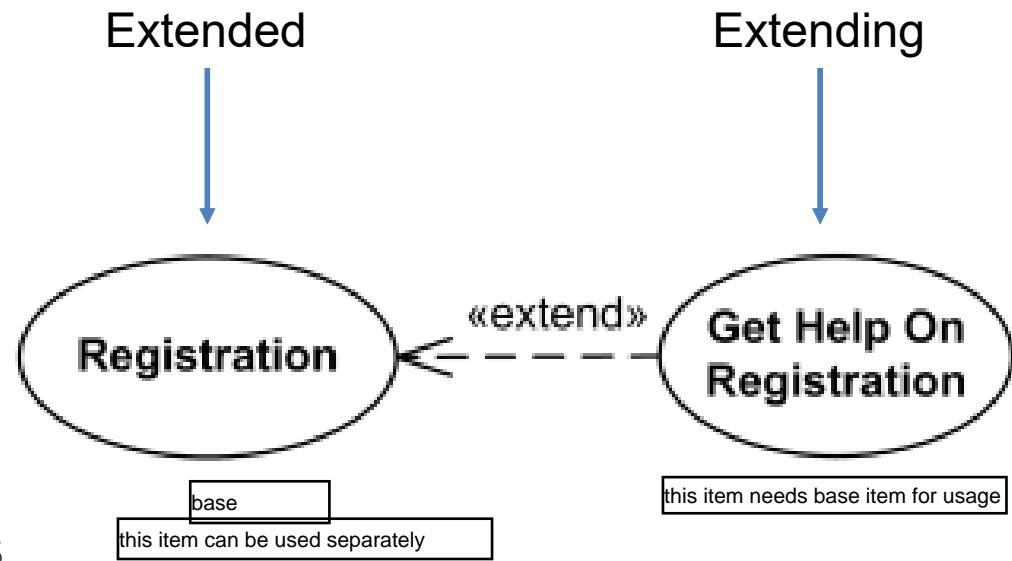
- when there are **common parts** of the behavior of two or more use cases,
- to simplify large use case by splitting it into several use cases.





Extend relationship

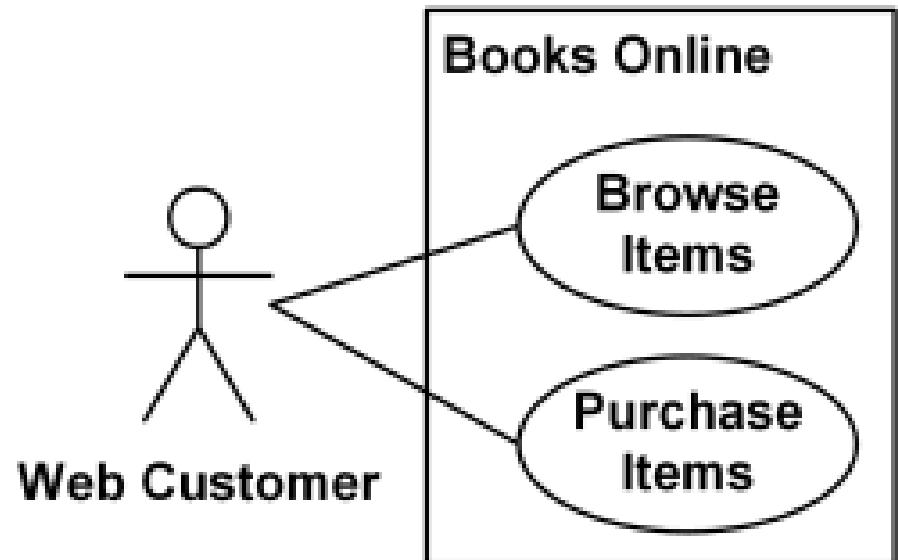
- ✧ **Extended** use case is meaningful on its own
 - it is **independent** of the extending use case.
- ✧ **Extending** use case typically defines **optional** behavior that is not necessarily meaningful by itself.



Subject – system boundary



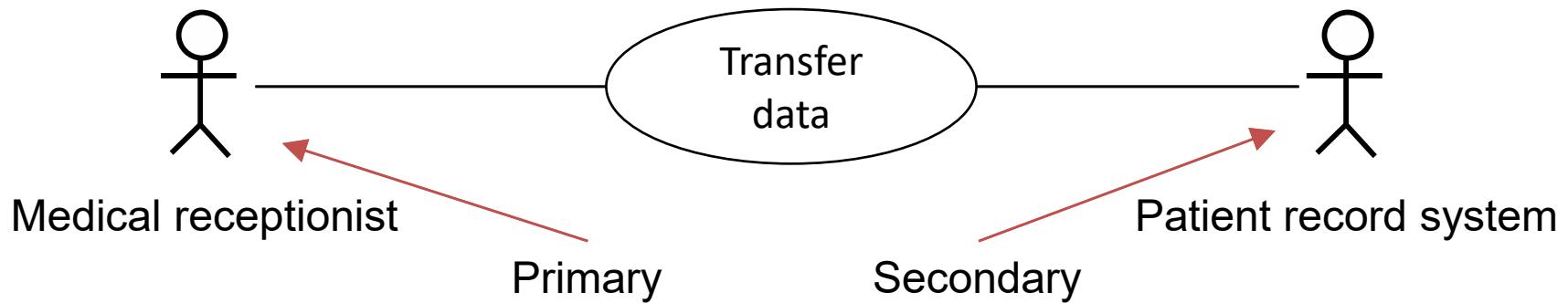
- ✧ **System** is presented by a rectangle with its name in upper corner
- ✧ **Use cases** inside the rectangle and **actors** - outside of the system boundaries.



Transfer data use case



- ✧ A use case in the Mentcare system



- ✧ Use case diagram gives high level overview of interactions

- ✧ Details can be provided as

- text (read Ch. 9 of UML distilled), table (forms), sequence diagram

Tabular description of the ‘Transfer data’ use-case



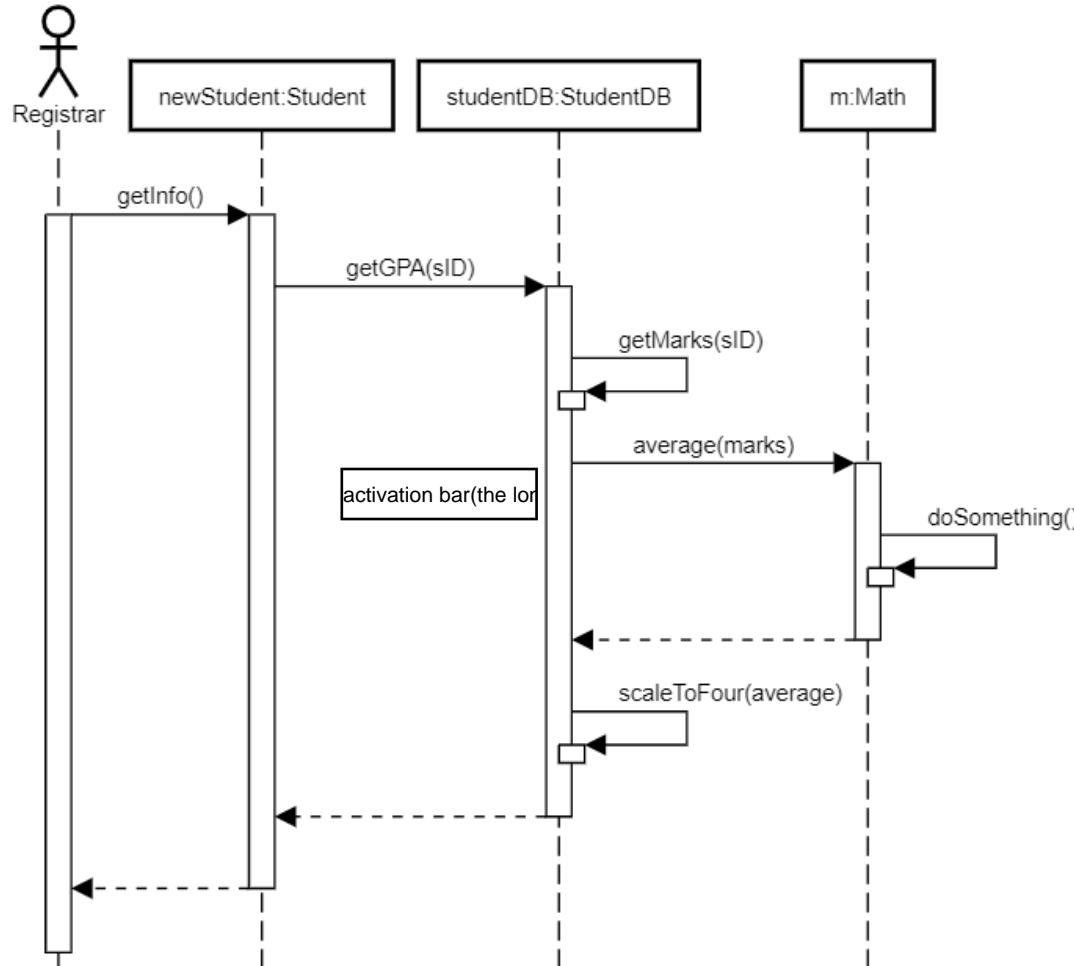
MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the Mentcase system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

Sequence diagrams



- ✧ Sequence diagrams are part of the UML and are used to model the interactions **between the actors and the objects within a system**.
- ✧ A sequence diagram shows the **sequence of interactions** that take place during a particular **use case or use case instance**.
- ✧ The objects and actors involved are listed **along the top** of the diagram, with a dotted line (known as life line) drawn vertically from these.
- ✧ Interactions between objects are indicated by **annotated arrows**.

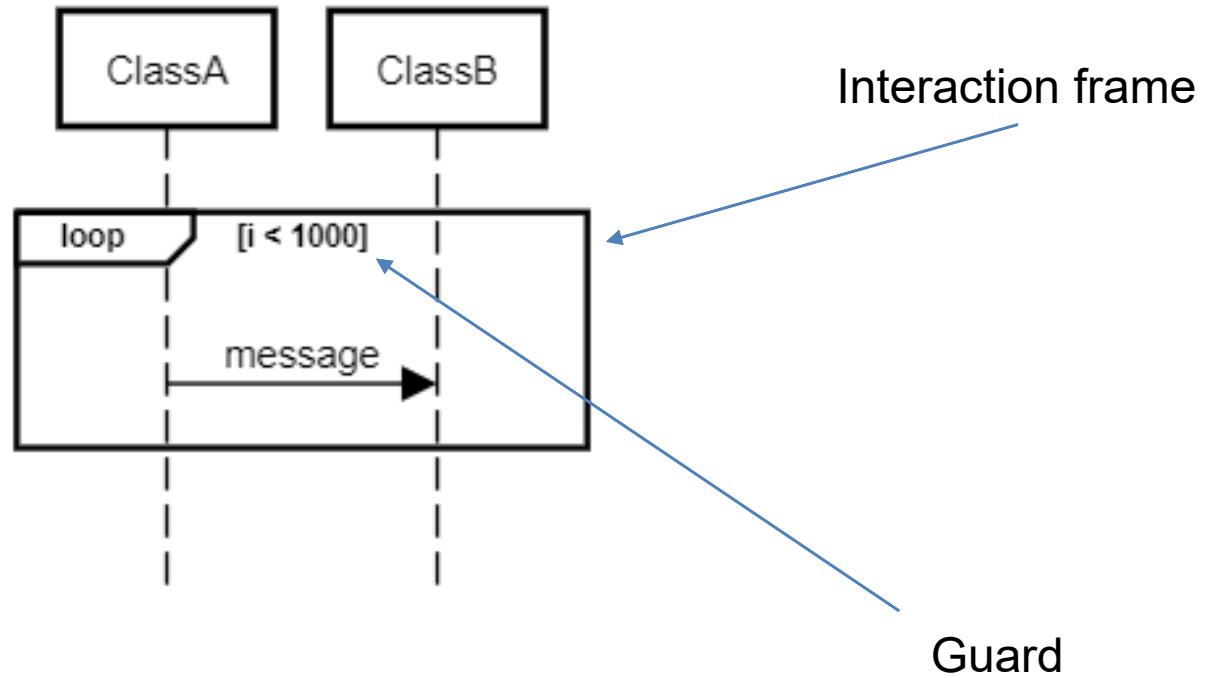
An example



actor Registrar
participant "newStudent:Student" as student
participant "studentDB:StudentDB" as DB
participant "m:Math" as Math
participant student
participant DB
participant Math

Registrar->student:`getInfo()`
activate Registrar
activate student
student->DB:`getGPA(sID)`
activate DB
DB->DB:`getMarks(sID)`
activate DB
deactivateafter DB
DB->Math:`average(marks)`
activate Math
Math->Math:`doSomething()`
activate Math
deactivateafter Math
Math-->DB:
deactivate Math
DB->DB:`scaleToFour(average)`
activate DB
deactivateafter DB
DB-->student:
deactivate DB
student-->Registrar:
deactivate student

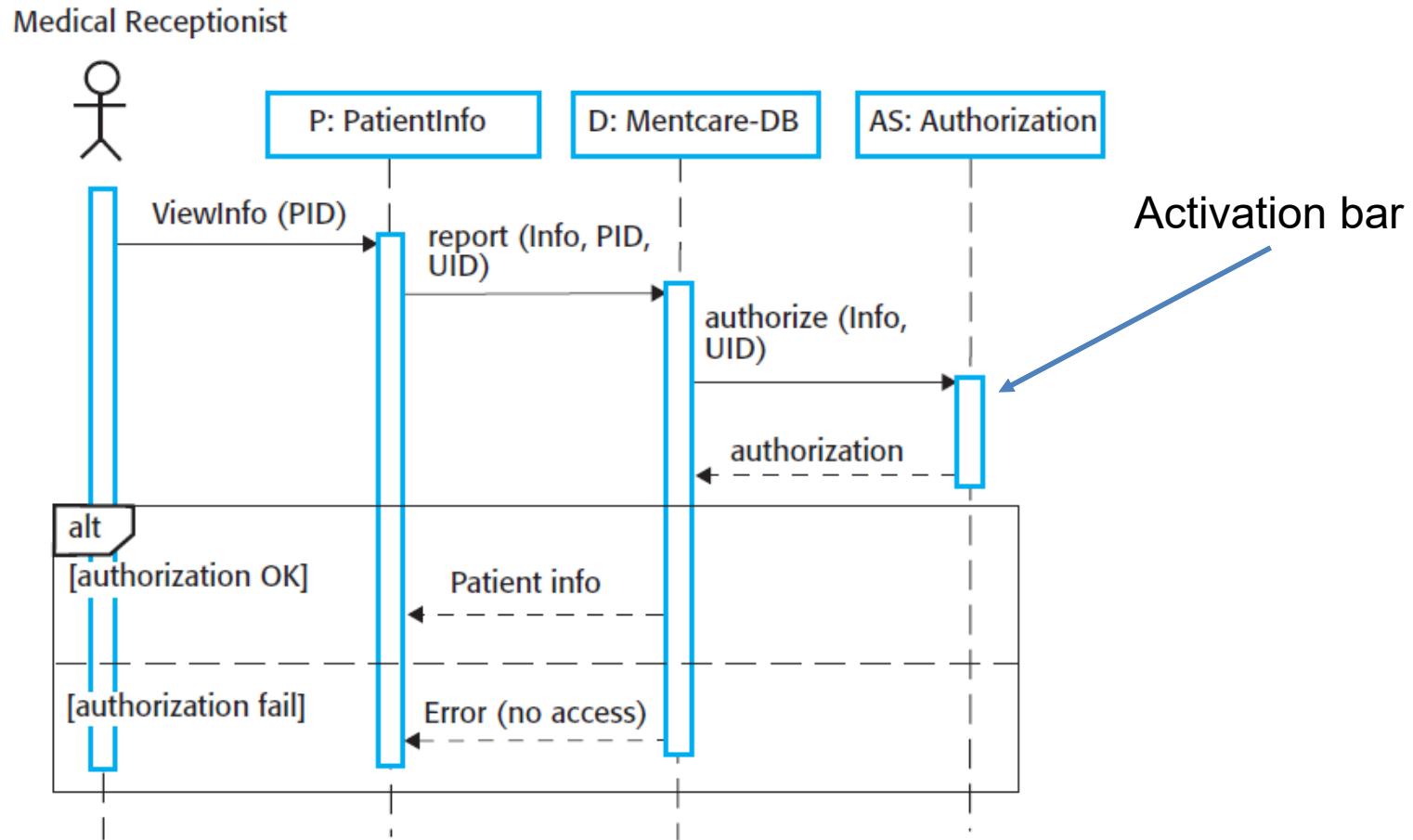
Loop

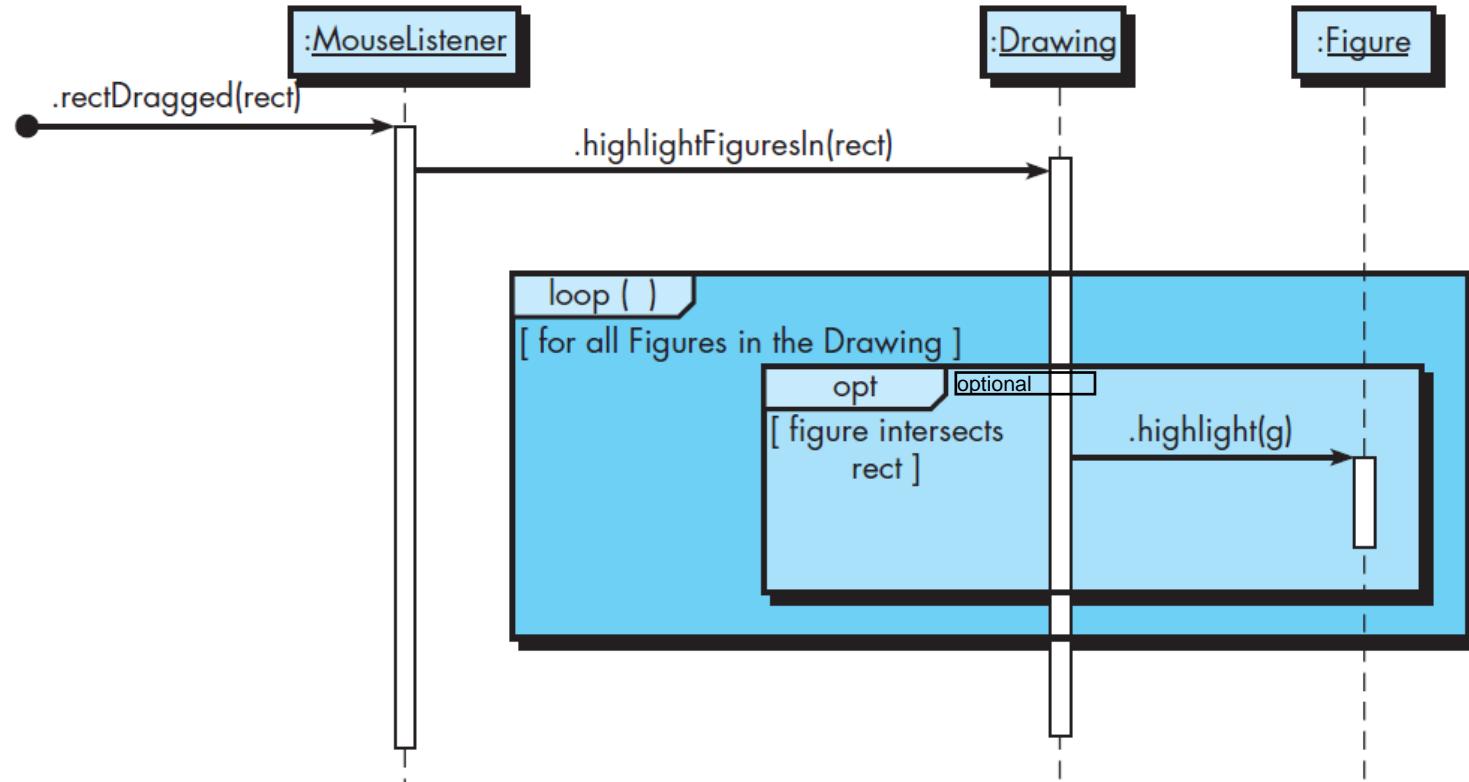


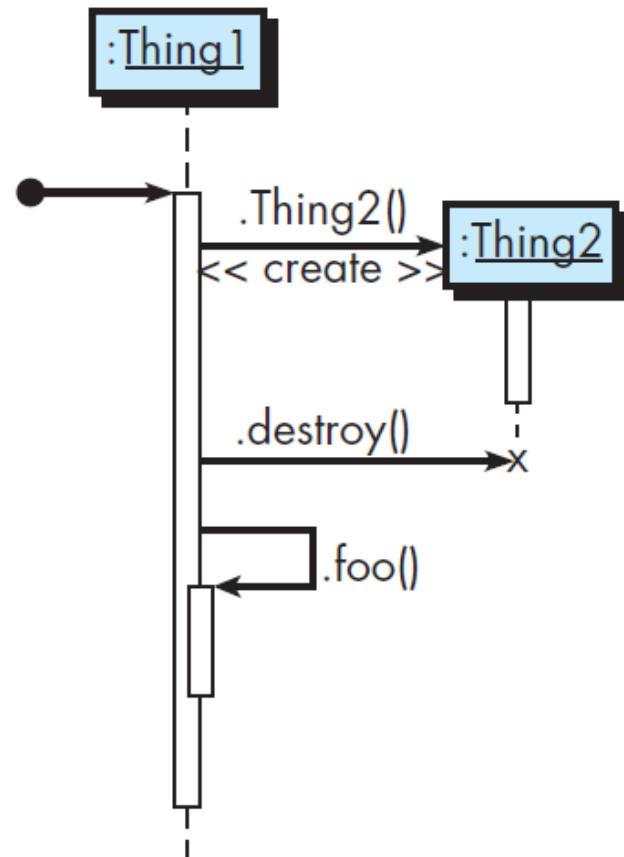
<https://sequencediagram.org/>

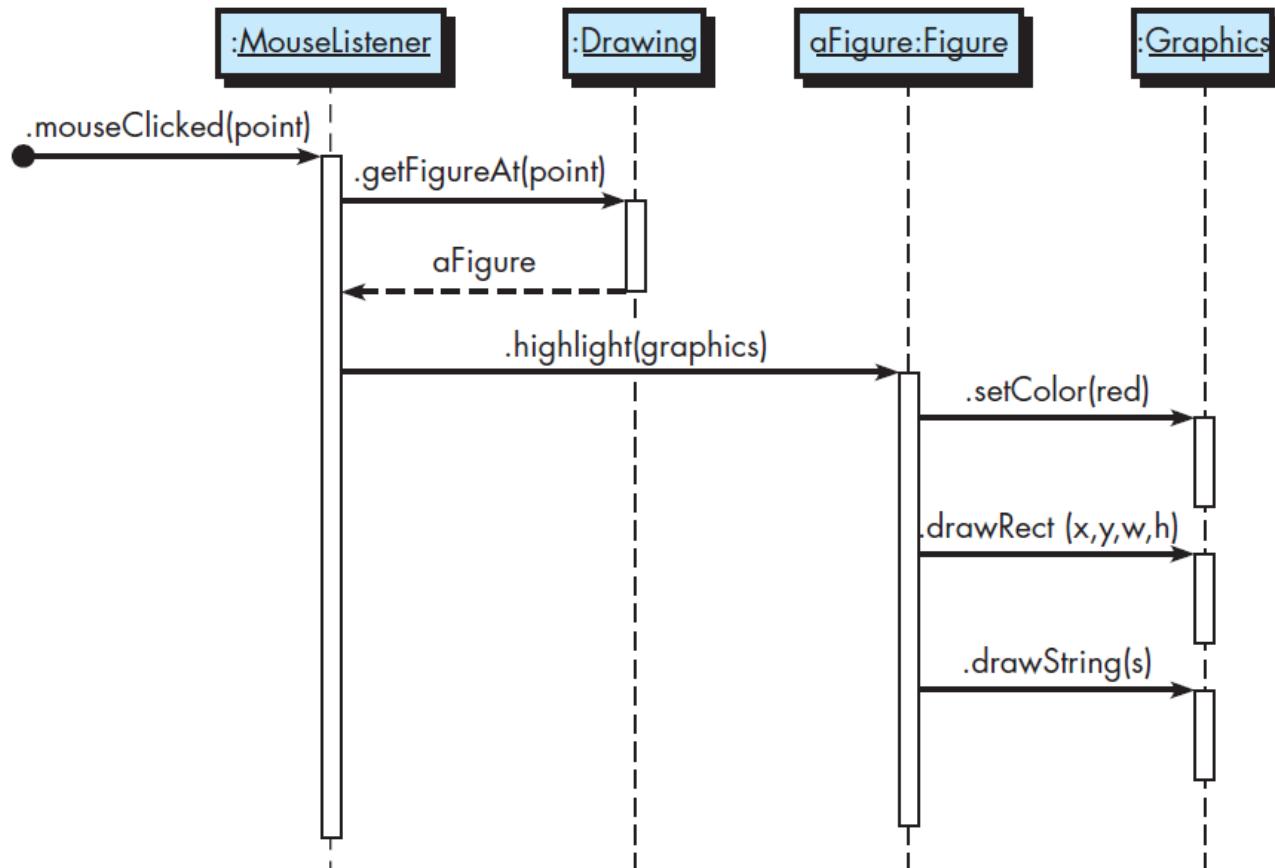


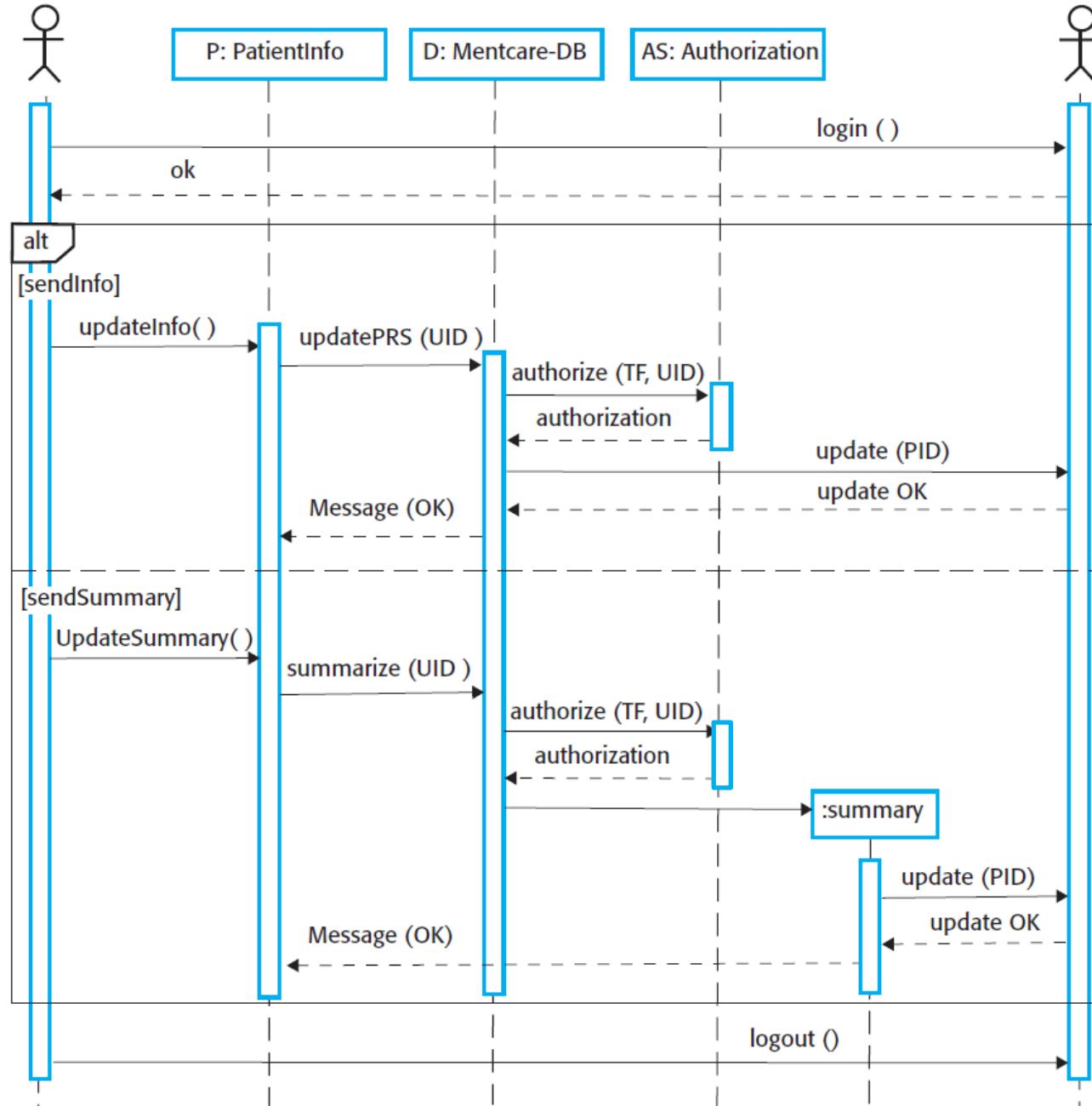
Sequence diagram for View patient information











Sequence diagram for Transfer Data



Structural models

Structural models



- ✧ Structural models of software display the **organization** of a system in terms of the **components** that make up that system and their **relationships**.
- ✧ Structural models may be **static** models, which show the structure of the system design, or **dynamic** models, which show the organization of the system when it is executing.
- ✧ Structural models are useful in **discussing** and **designing** the system architecture.

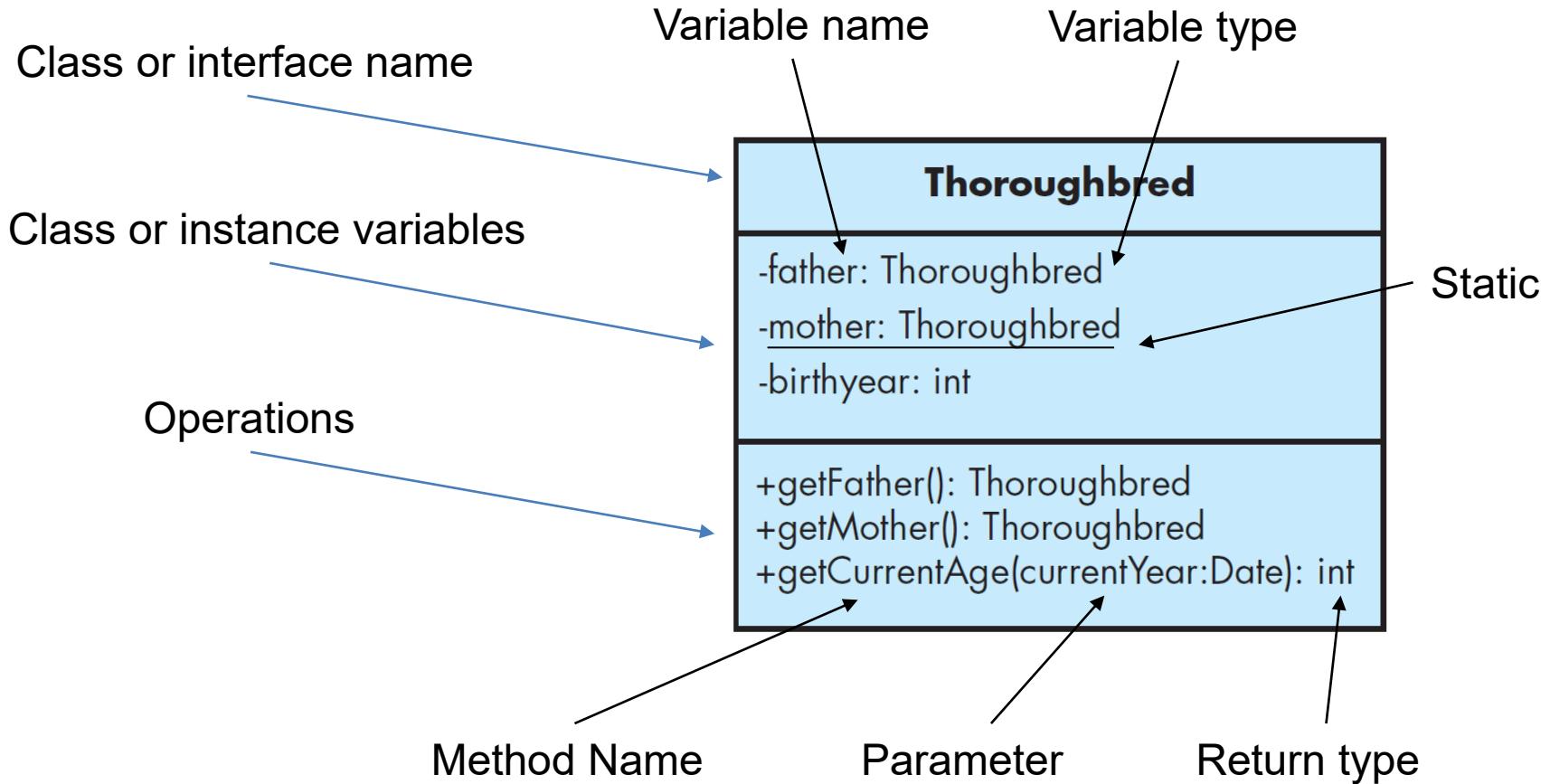


Class diagrams

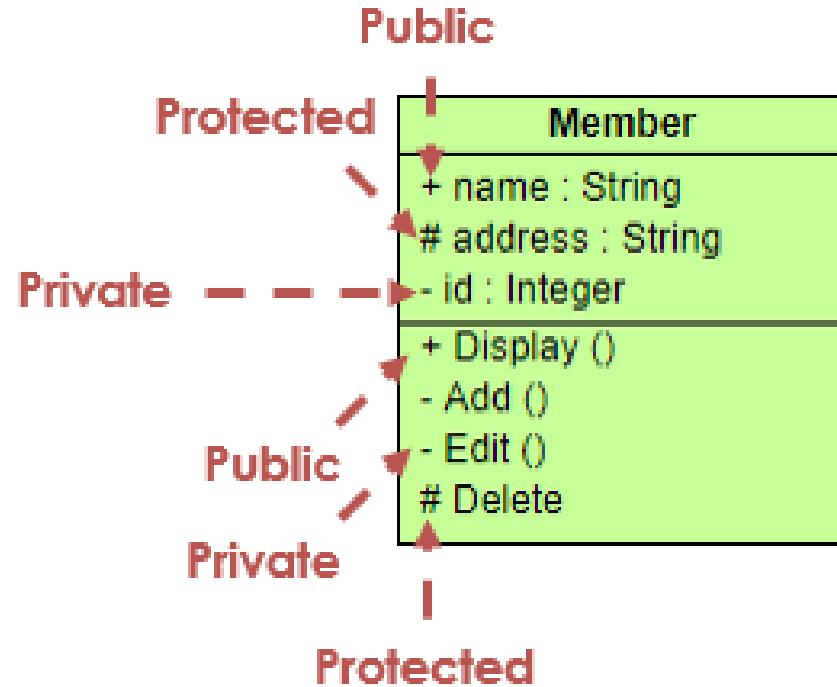
class diagram is static, because before executing we know the relation between classes

- ✧ Used when developing an **object-oriented system** model to show the **classes** in a system and the **associations** between these classes.
- ✧ An object **class** can be thought of as a **general definition of one kind** of system object.
- ✧ An **association** is a link between classes that indicates that there is some **relationship** between these classes.
- ✧ In early stage models of the software engineering process, objects **represent something in the real world**, such as a patient, a prescription, doctor, etc.

UML class diagram



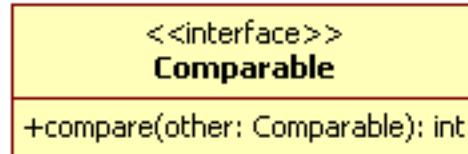
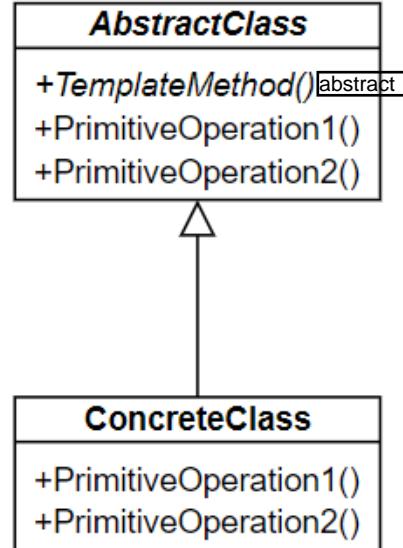
Visibility



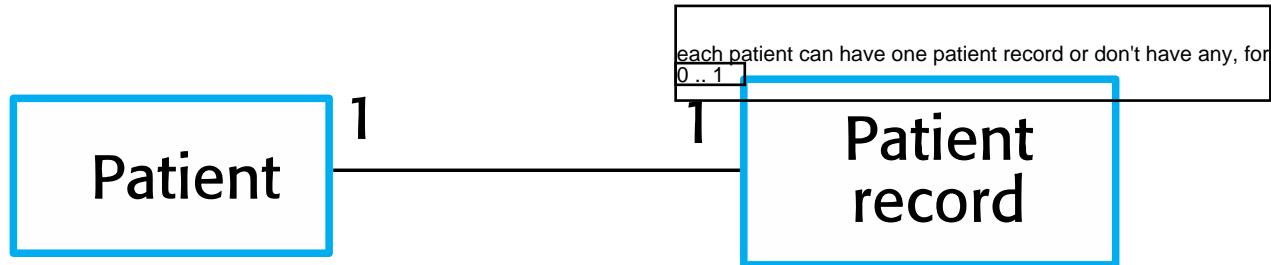


Abstract class and interface

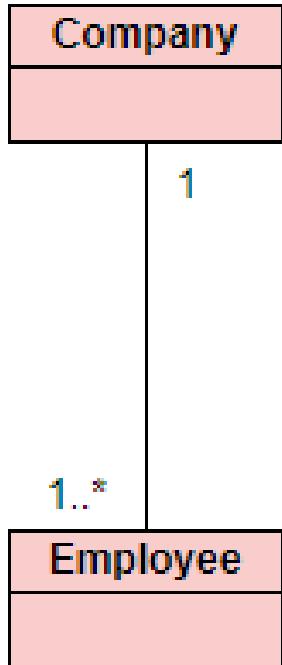
We write abstracts with italic



UML classes and association

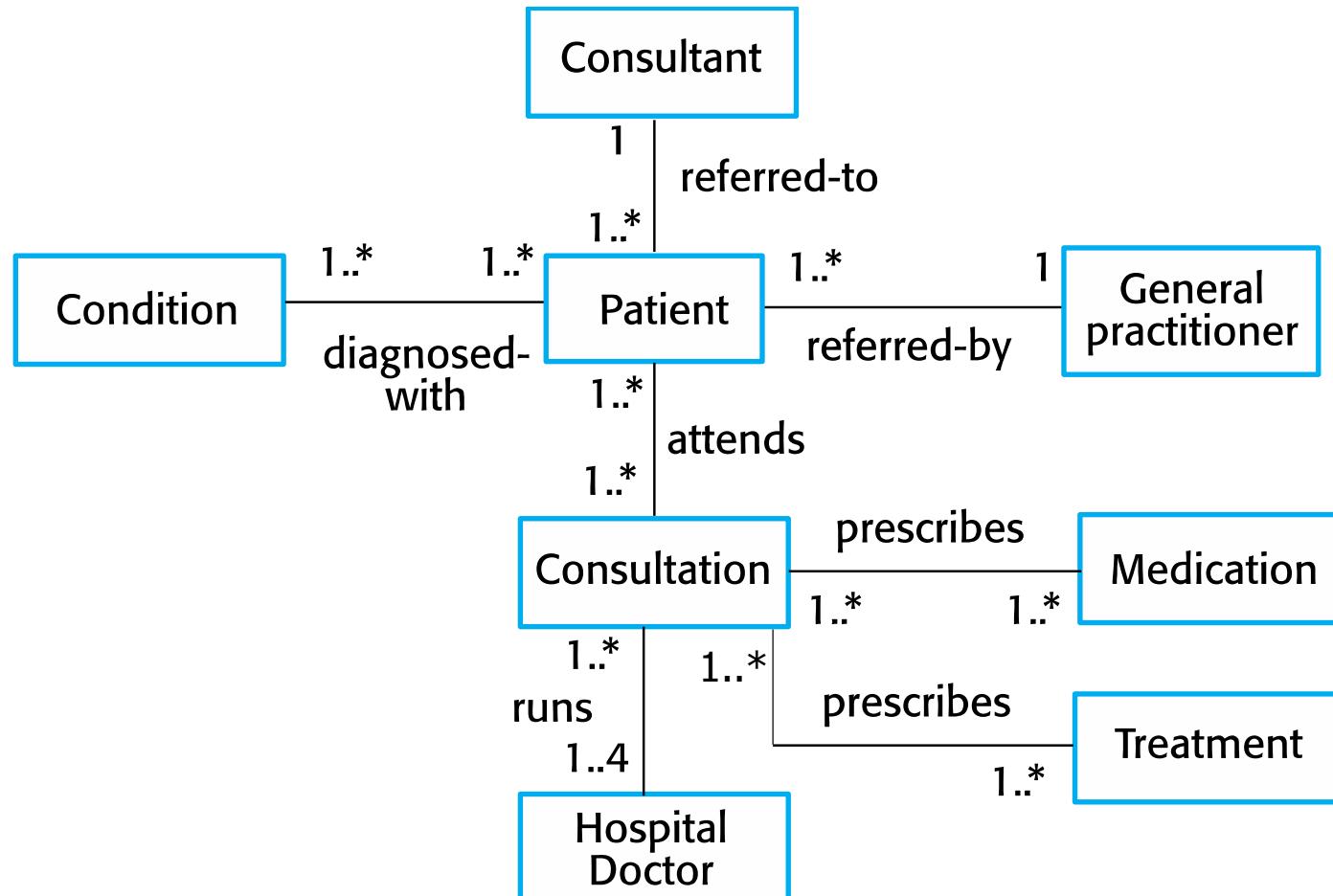


Multiplicity



Multiplicity	Option	Cardinality
0..0	0	Collection must be empty
0..1		No instances or one instance
1..1	1	Exactly one instance
0..*	*	Zero or more instances
1..*		At least one instance
5..5	5	Exactly 5 instances
m..n		At least m but no more than n instances

Classes and associations in the MHC-PMS



The Consultation class



Consultation

Doctors
Date
Time
Clinic
Reason
Medication prescribed
Treatment prescribed
Voice notes
Transcript
...

New ()
Prescribe ()
RecordNotes ()
Transcribe ()
...

Generalization



- ✧ Generalization is an everyday technique that we use to **manage complexity**.
- ✧ Rather than learn the detailed characteristics of every entity that we experience, we **place these entities in more general classes** (animals, cars, houses, etc.) and learn the characteristics of these classes.
- ✧ This allows us to infer that different members of these classes have some **common characteristics** e.g. squirrels and rats are rodents.

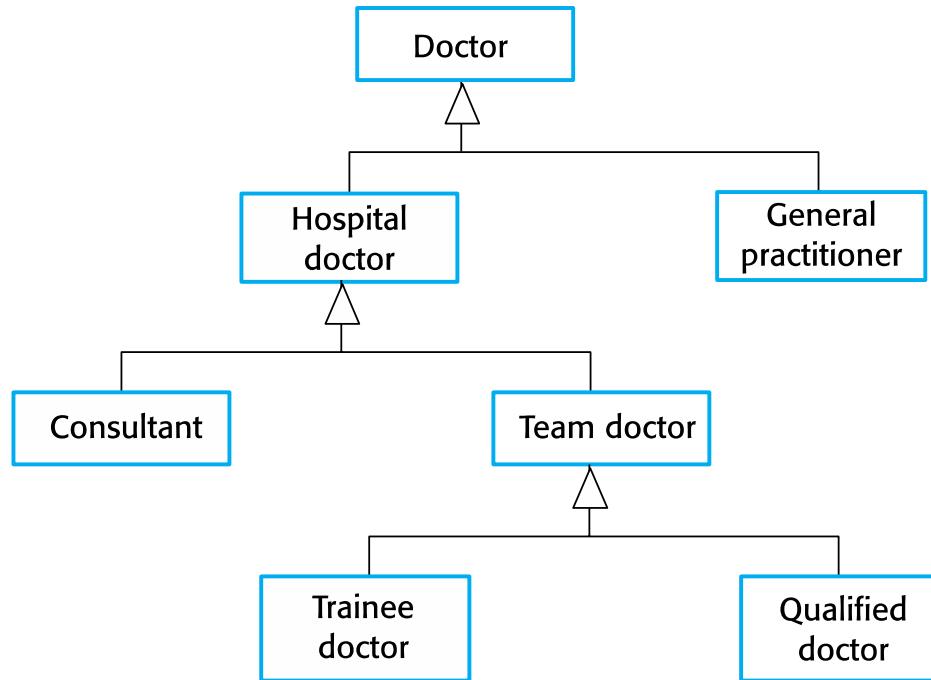
Generalization



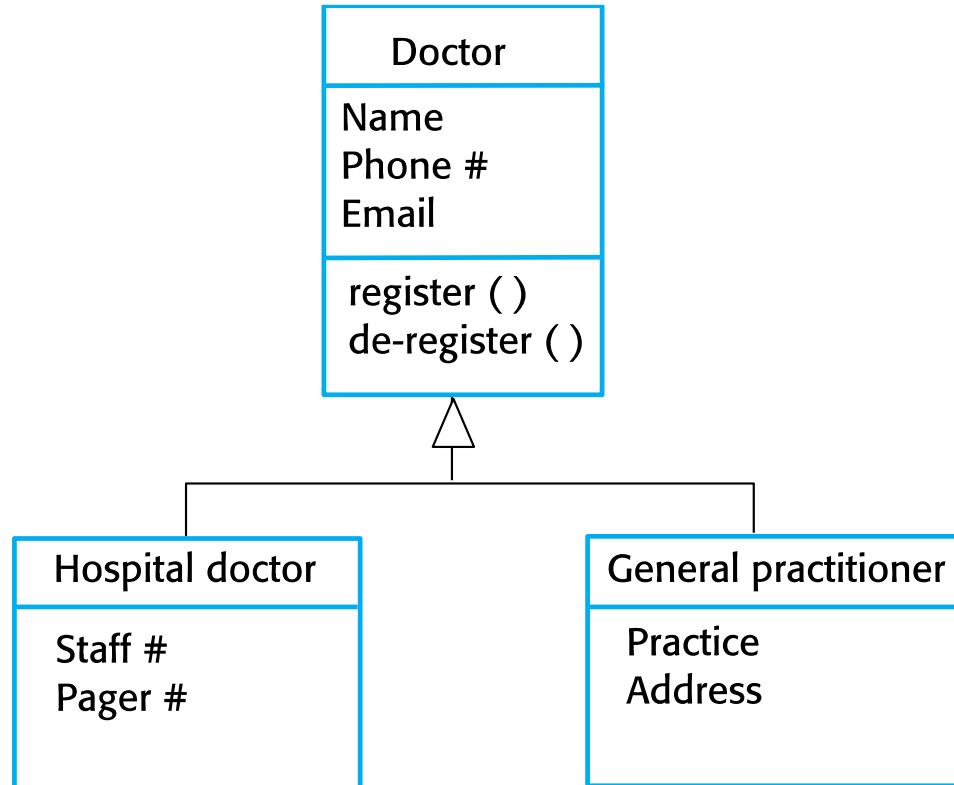
- ✧ In modeling systems, it is often useful to **examine the classes** in a system to see if there is **scope for generalization**.
 - If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
- ✧ In **object-oriented languages**, such as **Java**, generalization is implemented using the class inheritance mechanisms built into the language.
- ✧ In a generalization, the **attributes and operations** associated with higher-level classes are also associated with the lower-level classes.
- ✧ The lower-level classes are **subclasses inherit** the attributes and operations **from their superclasses**. These lower-level classes then add more specific attributes and operations.



A generalization hierarchy



A generalization hierarchy with added detail

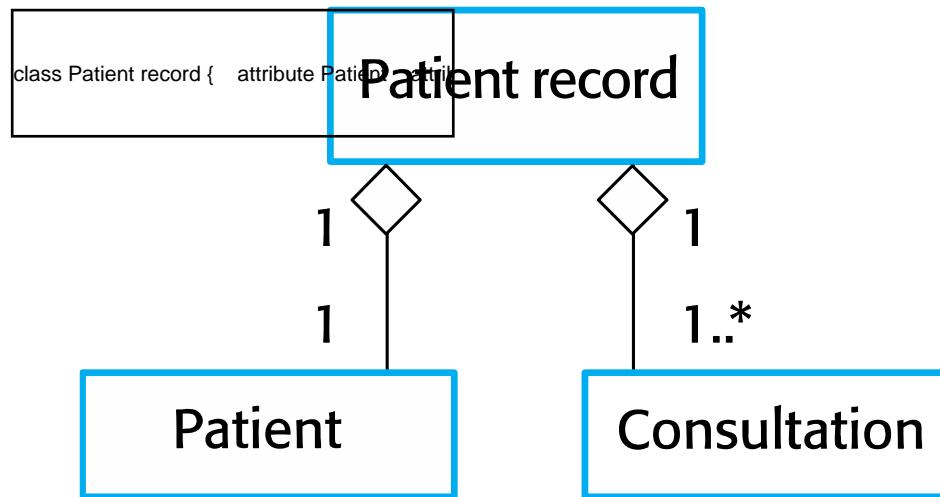


Object class aggregation models



- ✧ An aggregation model shows how **classes that are collections** are composed of other classes.
- ✧ Aggregation models are similar to the **part-of relationship** in semantic data models.

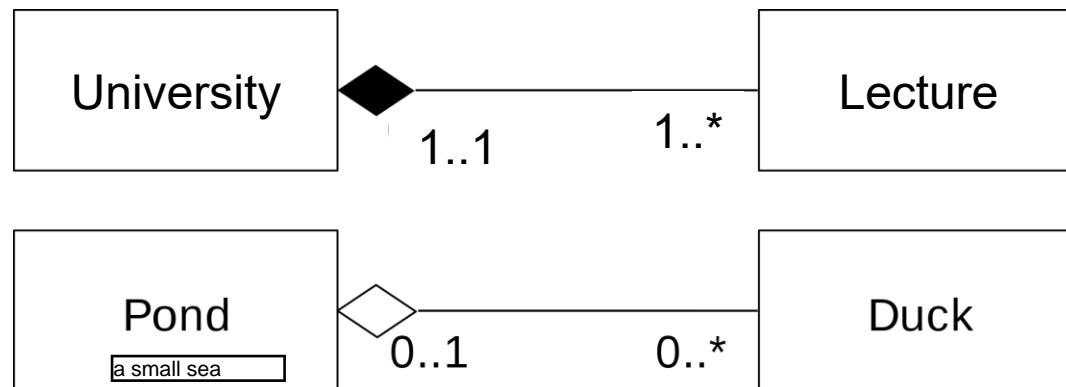
The aggregation association



Aggregation vs Composition



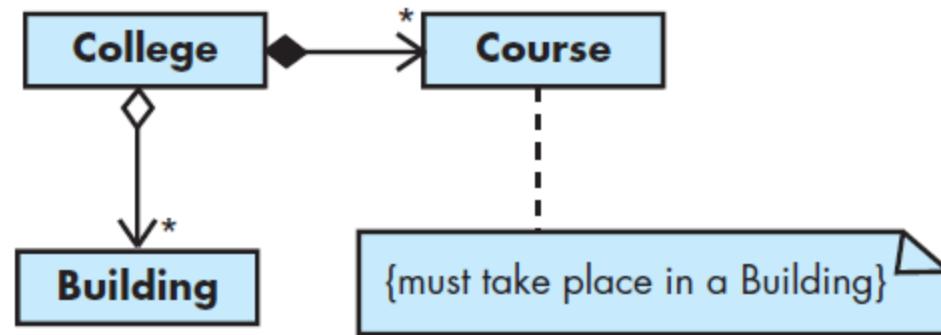
- ✧ A university is a composition of classes not physical classes, to be specific sections
 - University ceases to exist -> classes cease to exist
- ✧ A university is an aggregation of professors/students
 - University ceases to exist -> professors/students continue to exist



aggregation --> components are not destroyed
composition



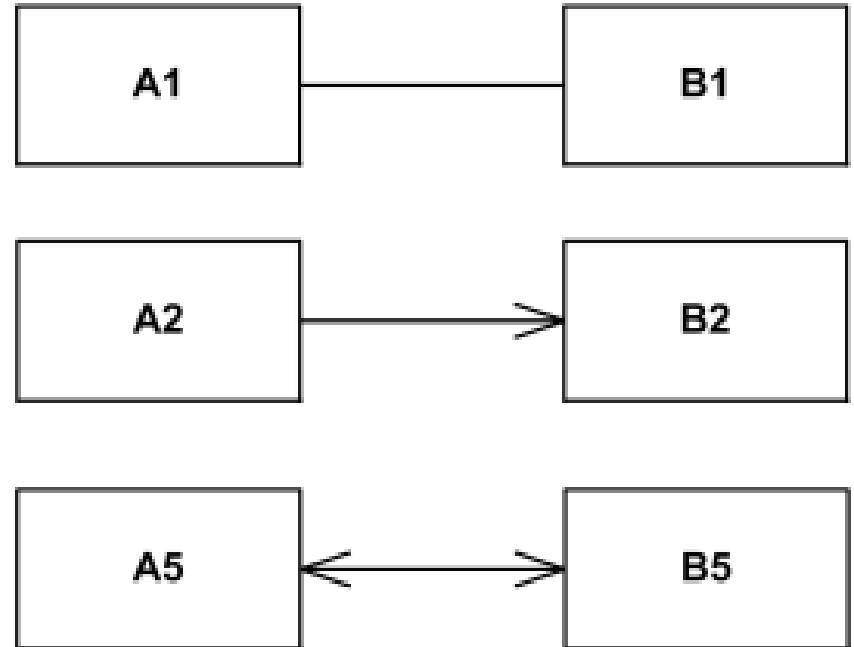
Aggregation vs Composition - 2



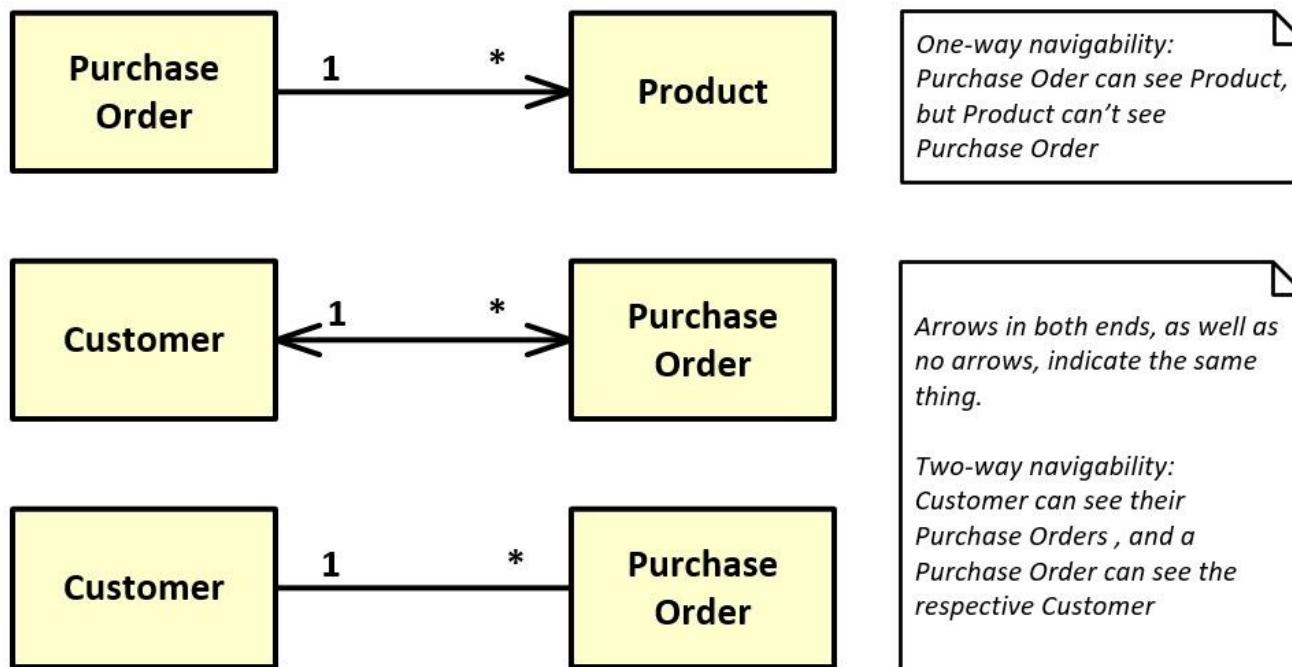
Association navigability



- ✧ End property of association is navigable from the opposite end(s) of association if instances of the classifier at this end of the link can be accessed efficiently at runtime from instances at the other ends of the link.



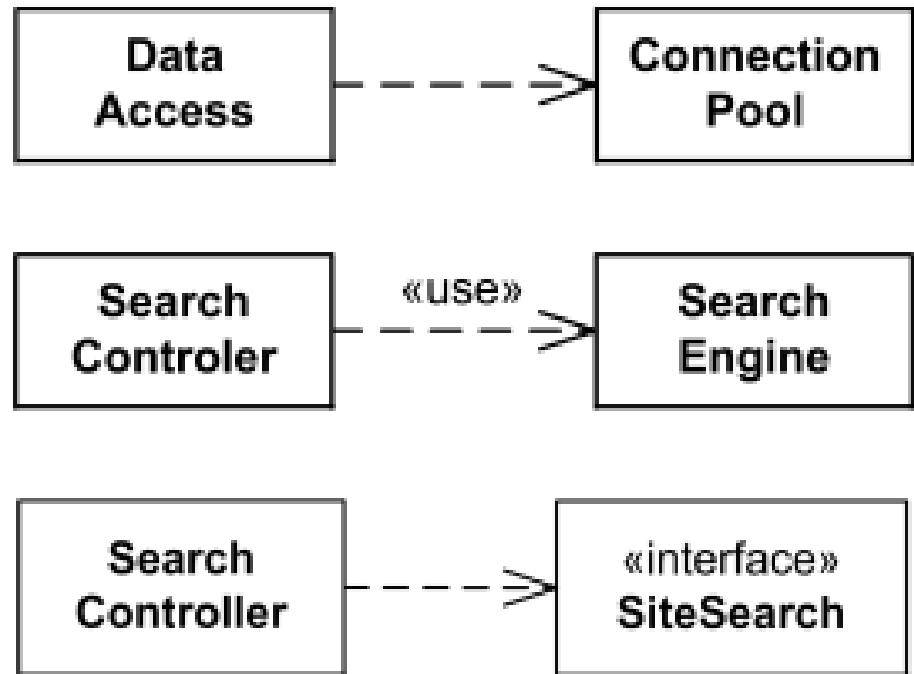
Association navigability



Dependency

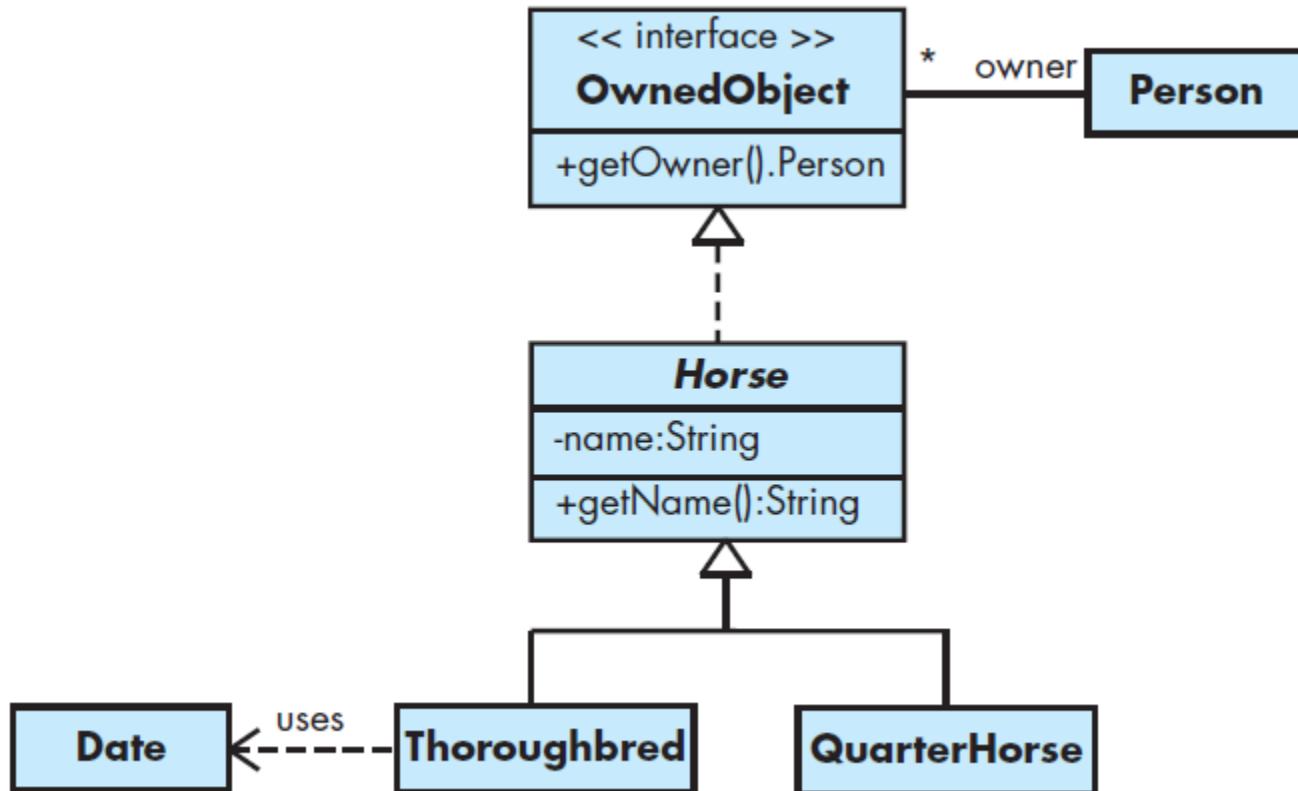


- ❖ The model element at the tail of the arrow (the client) depends on the model element at the arrowhead (the supplier)





Another example





Behavioral models

Behavioral models



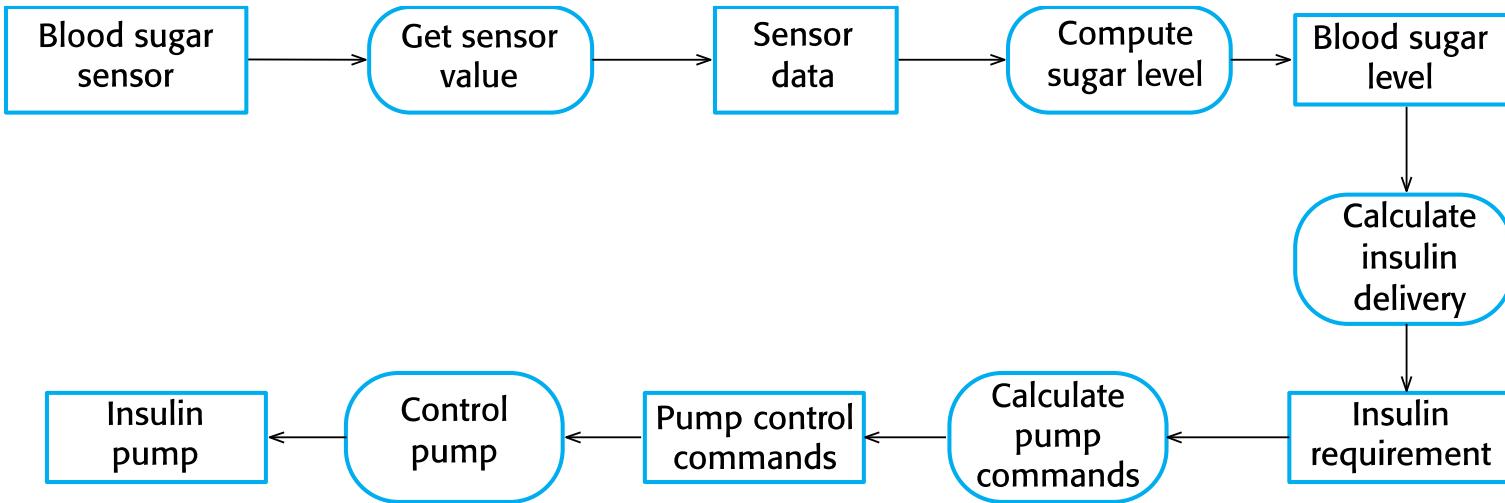
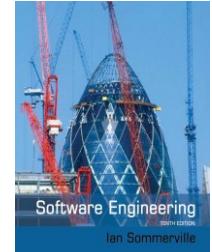
- ✧ Models of the dynamic behavior of a system as it is executing. They show **what happens or what is supposed to happen** when a **system responds to a stimulus** from its environment.
- ✧ You can think of these stimuli as being of two types:
 - **Data** Some data arrives that has to be processed by the system.
 - **Events** Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

Data-driven modeling



- ✧ Many business systems are **data-processing systems** that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
- ✧ Data-driven models show the **sequence of actions** involved in **processing input data** and generating an **associated output**.
- ✧ They are particularly useful during the **analysis of requirements** as they can be used to show **end-to-end processing** in a system.

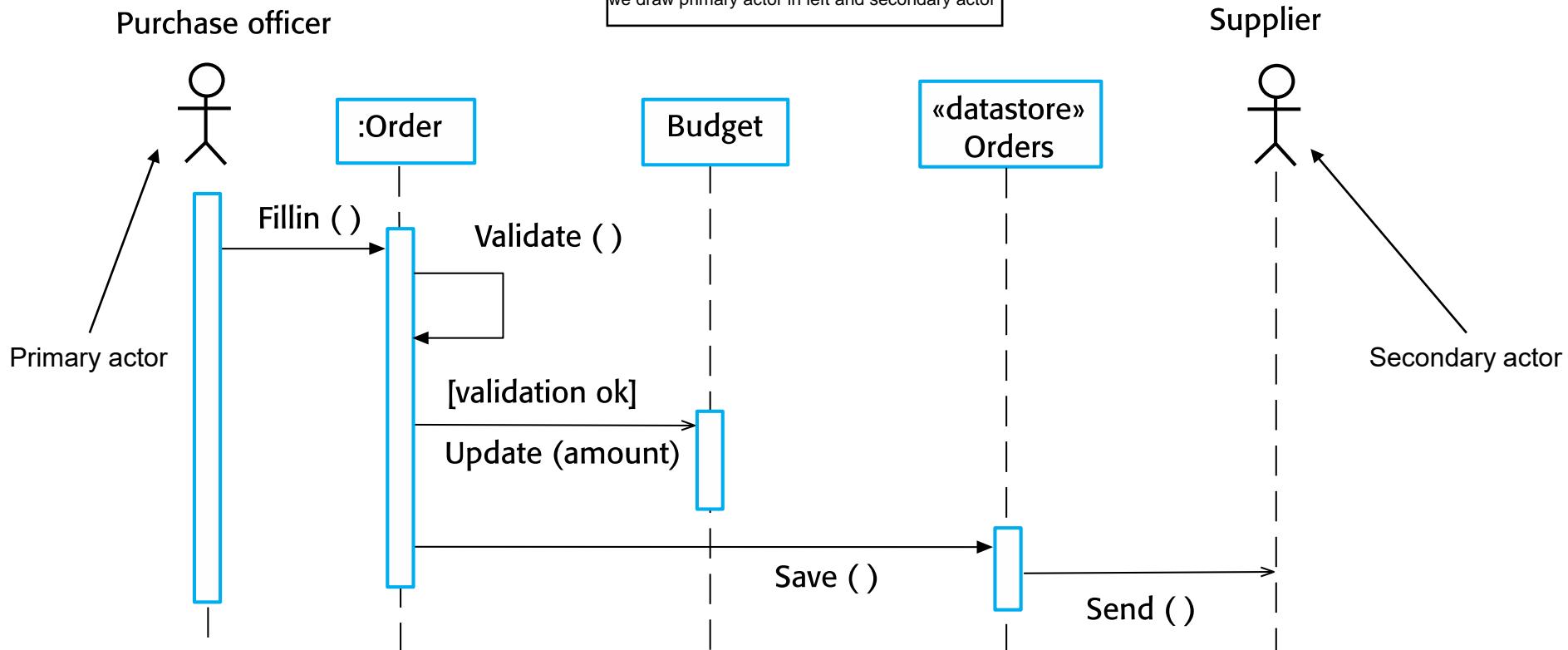
An activity model of the insulin pump's operation



Order processing



we draw primary actor in left and secondary actor



Event-driven modeling



- ✧ Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as 'receiver off hook' by generating a dial tone.
- ✧ Event-driven modeling shows how a system responds to external and internal events.
- ✧ It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

State machine models

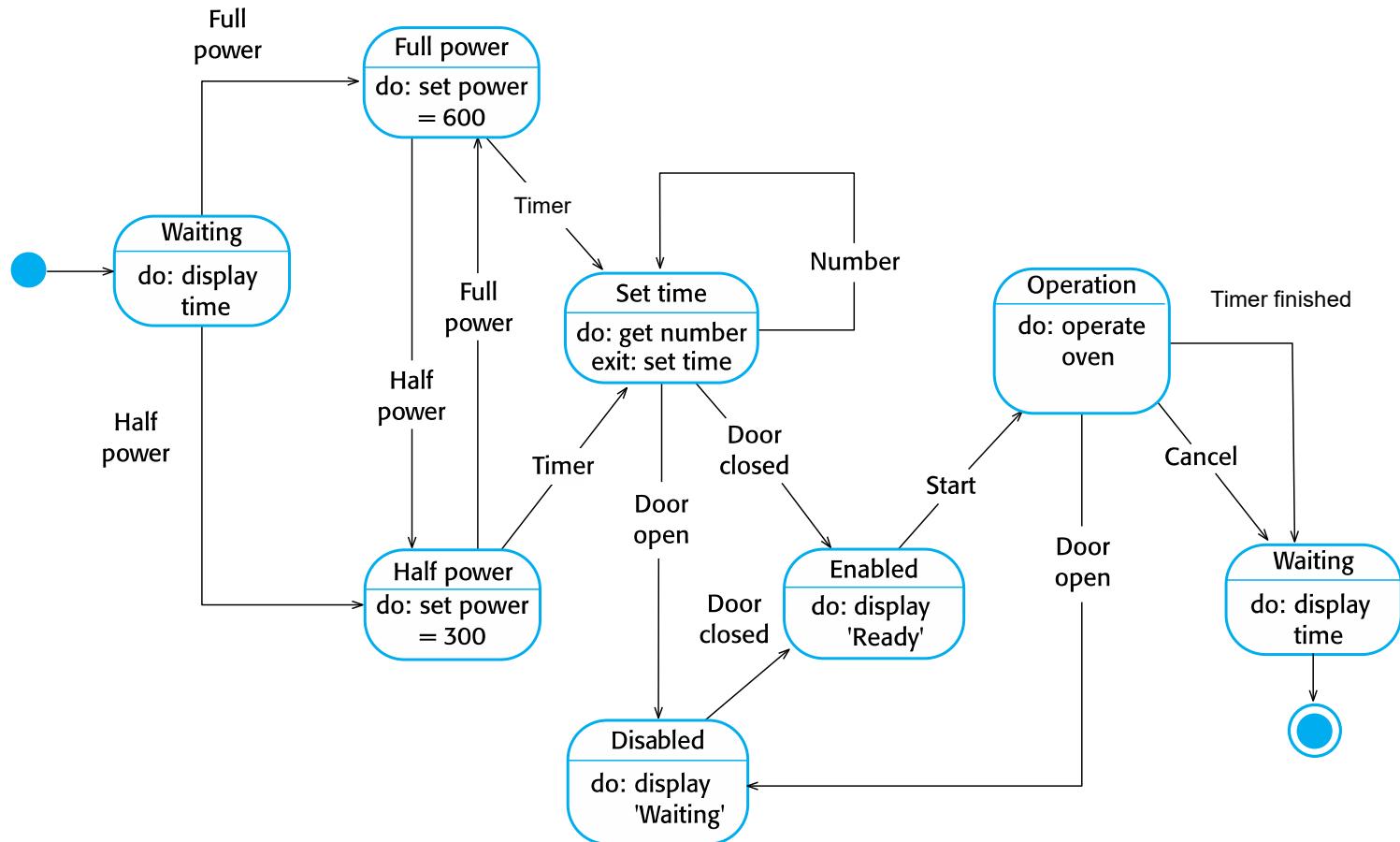


- ✧ These model the **behaviour of the system** in response to external and internal **events**.
- ✧ They show the system's responses to stimuli so are **often used for modelling real-time systems**.
- ✧ State machine models show system **states as nodes** and **events as arcs** between these nodes. When an event occurs, the system moves from one state to another.
- ✧ State charts are an integral part of the UML and are used to represent state machine models.

State diagram of a microwave oven



Software Engineering
Ian Sommerville



States and stimuli for the microwave oven (a)



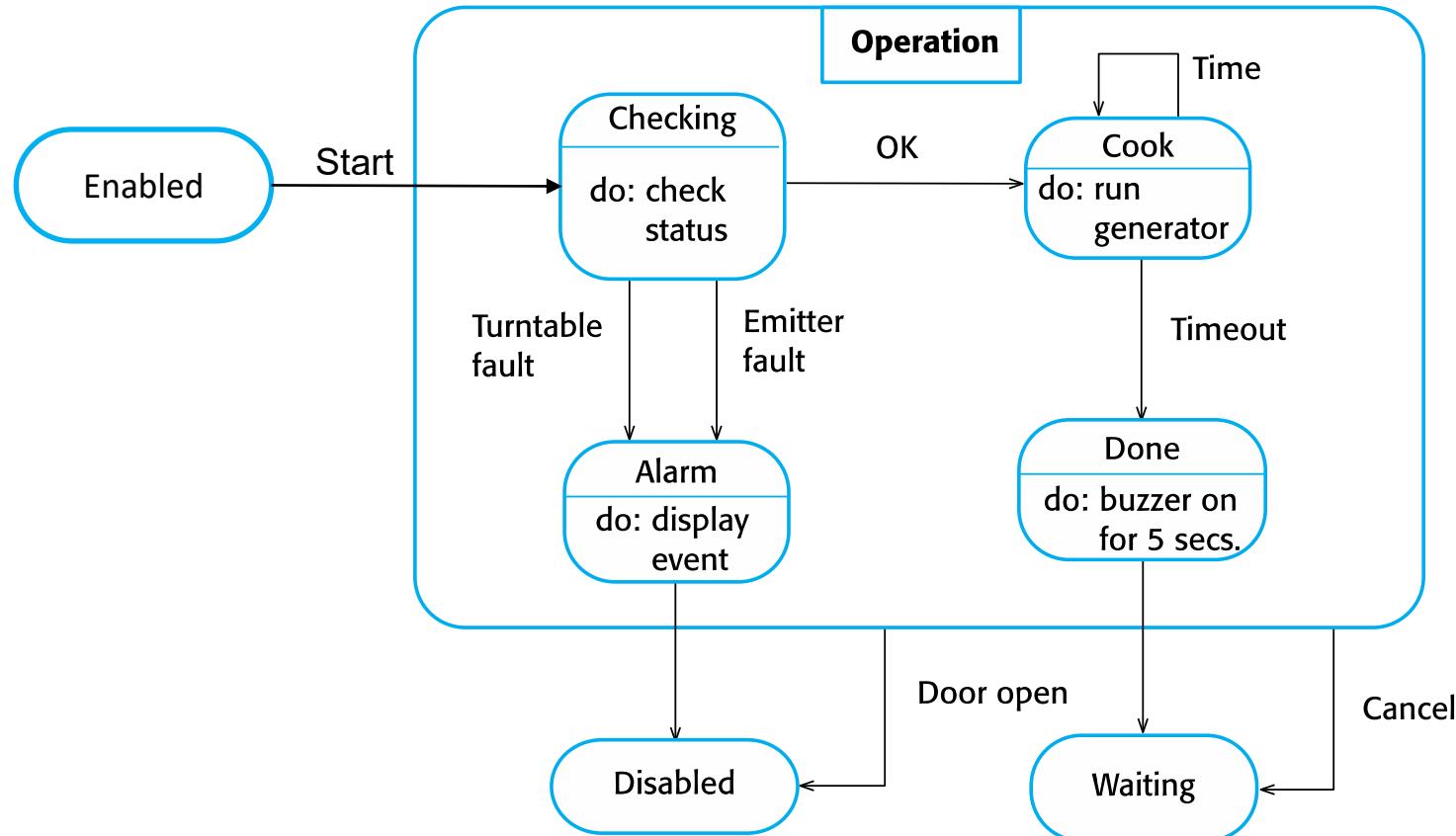
State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Waiting'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

States and stimuli for the microwave oven (b)



Stimulus	Description
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

Microwave oven operation

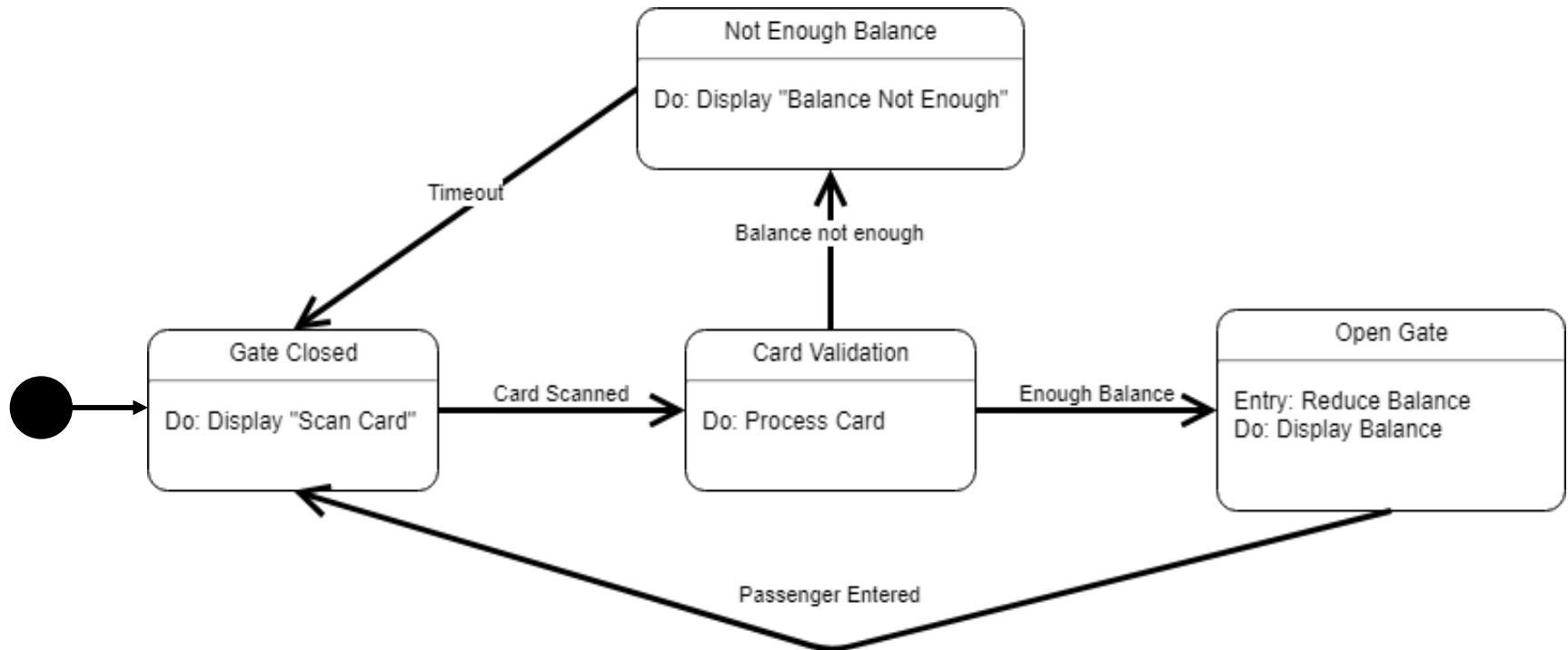


Metro pay gate



- ✧ Draw state diagram for metro gates in Tehran.

Metro pay gate





Model-driven engineering

Model-driven engineering



- ✧ Model-driven engineering (MDE) is an approach to software development where **models rather than programs** are the principal outputs of the development process.
- ✧ The programs that execute on a hardware/software platform are then **generated automatically from the models**.
- ✧ Proponents of MDE argue that this **raises the level of abstraction** in software engineering so that engineers **no longer have to be concerned with programming language details or the specifics of execution platforms**.



Usage of model-driven engineering

✧ Model-driven engineering is **still at an early stage of development**, and it is unclear whether or not it will have a significant effect on software engineering practice.

✧ Pros

- Allows systems to be considered at **higher levels of abstraction**
- Generating code automatically means that it **is cheaper to adapt systems to new platforms.**

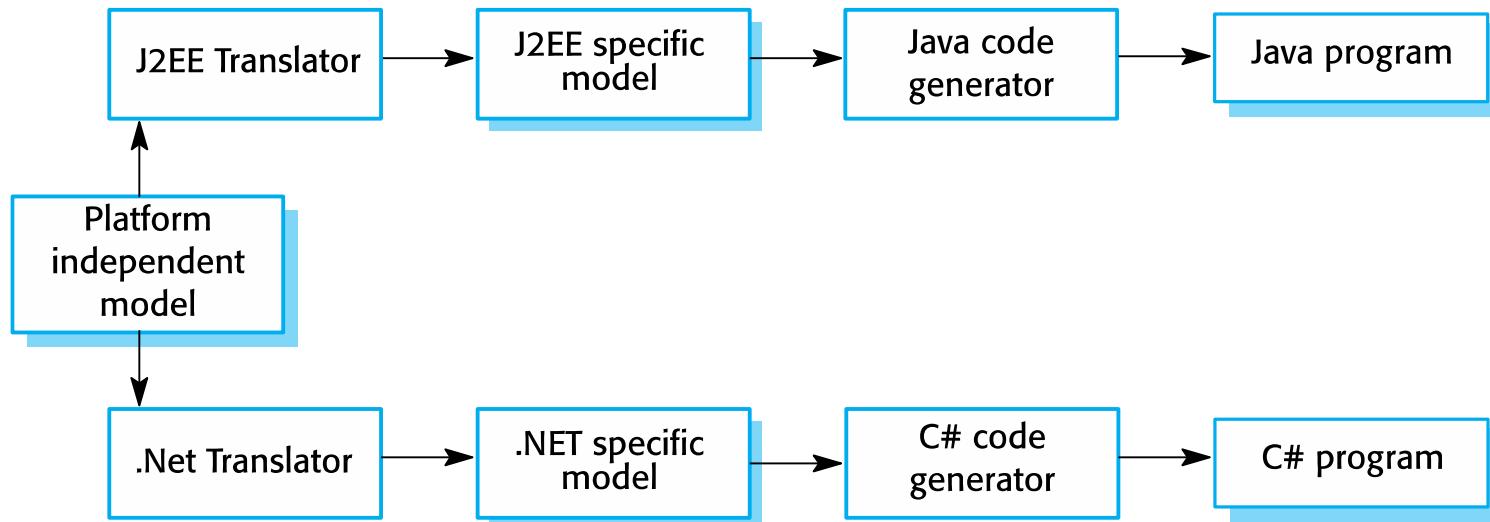
if we have tools, it is cheaper to use this approach

✧ Cons

- Models for abstraction and not necessarily right for implementation.
- Savings from generating code may be outweighed by the **costs of developing translators for new platforms.**

if we don't have tools, developing tools costs a lot

Multiple platform-specific models





Chapter 6 – Architectural Design

Architectural design



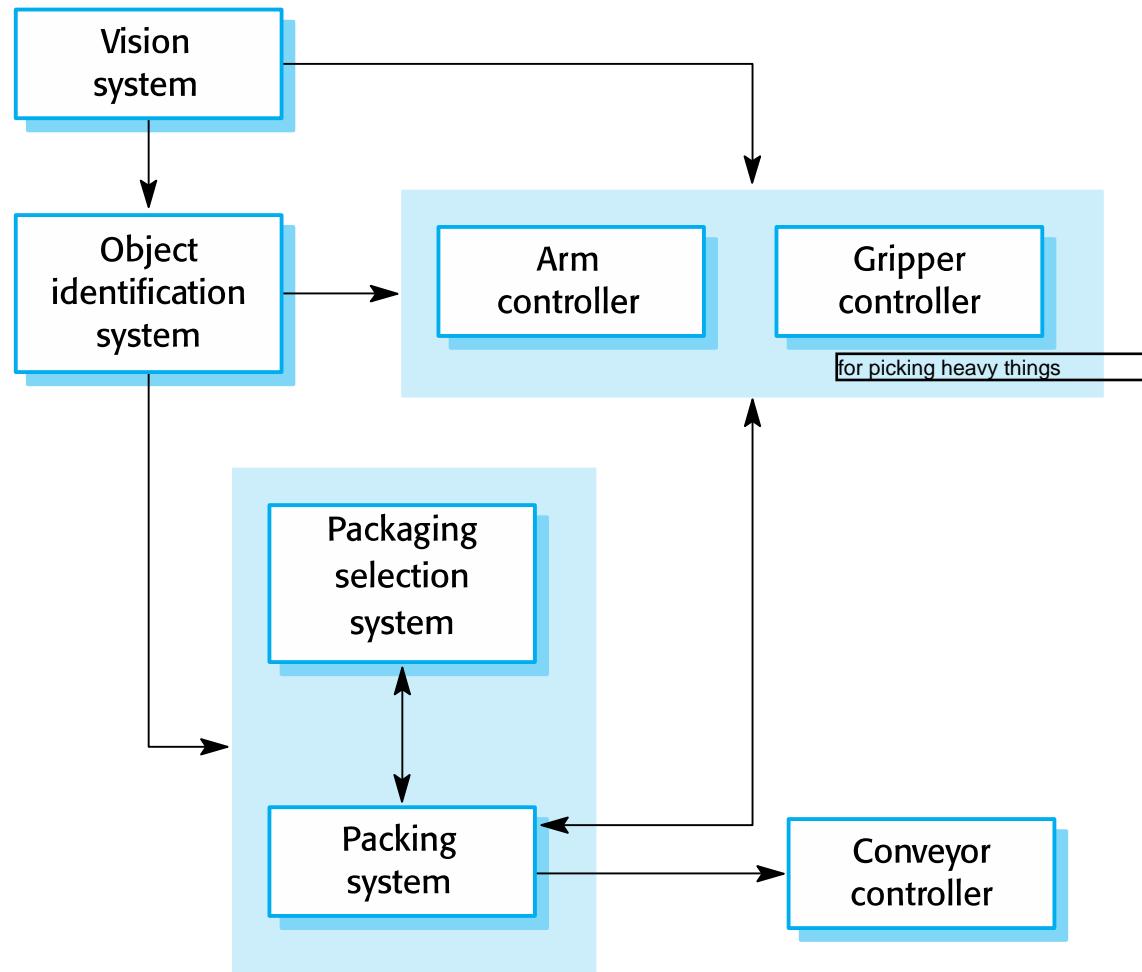
- ✧ Architectural design is concerned with **understanding how a software system should be organized** and designing the **overall structure** of that system.
- ✧ Architectural design is the critical **link between design and requirements engineering**, as it identifies the **main structural components** in a system and the **relationships** between them.
- ✧ The output of the architectural design process is an **architectural model** that describes how the **system is organized** as a set of communicating components.

Agility and architecture



- ✧ It is generally accepted that an **early stage of agile** processes is to **design an overall systems architecture**.
- ✧ **Refactoring** the system architecture is usually **expensive** because it affects so many components in the system

The architecture of a packing robot control system



Architectural abstraction



- ✧ **Architecture in the small** is concerned with the architecture of **individual programs**. At this level, we are concerned with the way that an individual program is **decomposed into components**.
- ✧ **Architecture in the large** is concerned with the architecture of **complex enterprise systems that include other systems, programs, and program components**. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Advantages of explicit architecture



✧ Stakeholder communication

- Architecture may be used as a focus of **discussion by system stakeholders**.

✧ System analysis

- Means that **analysis** of whether the system can **meet its non-functional requirements (performance, reliability, maintainability)** is possible.

✧ Large-scale reuse

- The architecture may be reusable across a range of systems
 - Large-scale software reuse
- Product-line architectures may be developed.

Architectural representations



- ✧ Simple, informal **block diagrams** showing entities and relationships are the **most frequently used** method for documenting software architectures.
- ✧ But these have been criticized because they **lack semantics**, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
- ✧ Depends on the **use of architectural models**. The requirements for model semantics depends on **how the models are used**.

Use of architectural models



- ✧ As a way of **facilitating discussion** about the system design
 - A high-level architectural view of a system is **useful for communication with system stakeholders** and **project planning** because it is **not cluttered with detail**. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole **without being confused by detail**.
- ✧ As a way of **documenting an architecture** that has been designed
 - The aim here is to produce a complete system model that shows the different **components in a system**, their **interfaces** and **their connections**.



Architectural design decisions

Architectural design decisions



- ✧ Architectural design is a creative process; Also, the **process differs depending on the type of system** being developed.

- ✧ However, a number of common decisions span all design processes and these decisions **affect the non-functional** characteristics of the system.

Architectural design decisions



Is there a generic application architecture that can act as a template for the system that is being designed?

How will the system be distributed across hardware cores or processors?

What architectural patterns or styles might be used?

What will be the fundamental approach used to structure the system?

What strategy will be used to control the operation of the components in the system?

How will the structural components in the system be decomposed into sub-components?

What architectural organization is best for delivering the non-functional requirements of the system?

How should the architecture of the system be documented?

Architecture reuse



- ✧ Systems in the **same domain** often have **similar architectures** that reflect domain concepts.
- ✧ Application **product lines** are built around a **core architecture with variants** that satisfy particular customer requirements.
- ✧ The architecture of a system may be designed around one of more **architectural patterns** or 'styles'.
 - These **capture the essence of an architecture** and can be instantiated in different ways.

Architecture and system characteristics



✧ Performance

- Localize critical operations and minimize communications. Use large rather than fine-grain components. Or allow replications.

✧ Security

- Use a layered architecture with critical assets in the inner layers.

✧ Safety

- Localize safety-critical features in a small number of subsystems.

✧ Availability

- Include redundant components and mechanisms for fault tolerance.

✧ Maintainability

- Use fine-grain, replaceable components.



Architectural views



Architectural views

- ✧ What views or perspectives are useful when designing and documenting a system's architecture?
- ✧ What notations should be used for describing architectural models?
- ✧ Each architectural model only shows one view or perspective of the system.
 - It might show how a system **is decomposed into modules**, how the **run-time processes** interact or the different ways in which system **components are distributed across a network**. For both design and documentation, you usually need to present multiple views of the software architecture.

4 + 1 view model of software architecture



- ✧ A **logical view**, which **shows the key abstractions** in the system as objects or object classes.
- ✧ A **process view**, which shows how, at run-time, the system is composed of **interacting processes**.
- ✧ A **development view**, which shows how the software is **decomposed for development**.
- ✧ A **physical view**, which shows the **system hardware** and **how software components are distributed across the processors** in the system.
- ✧ Related together using use cases or scenarios (+1)

+1: connects other views together



Architectural patterns

Architectural patterns



- ✧ Patterns are a means of **representing, sharing and reusing knowledge**.
- ✧ An **architectural pattern** is a stylized description of **good design practice**, which has been tried and **tested in different environments**.
- ✧ Patterns should include information about **when they are and when they are not useful**.
- ✧ Patterns may be represented using tabular and graphical descriptions.

Layered architecture



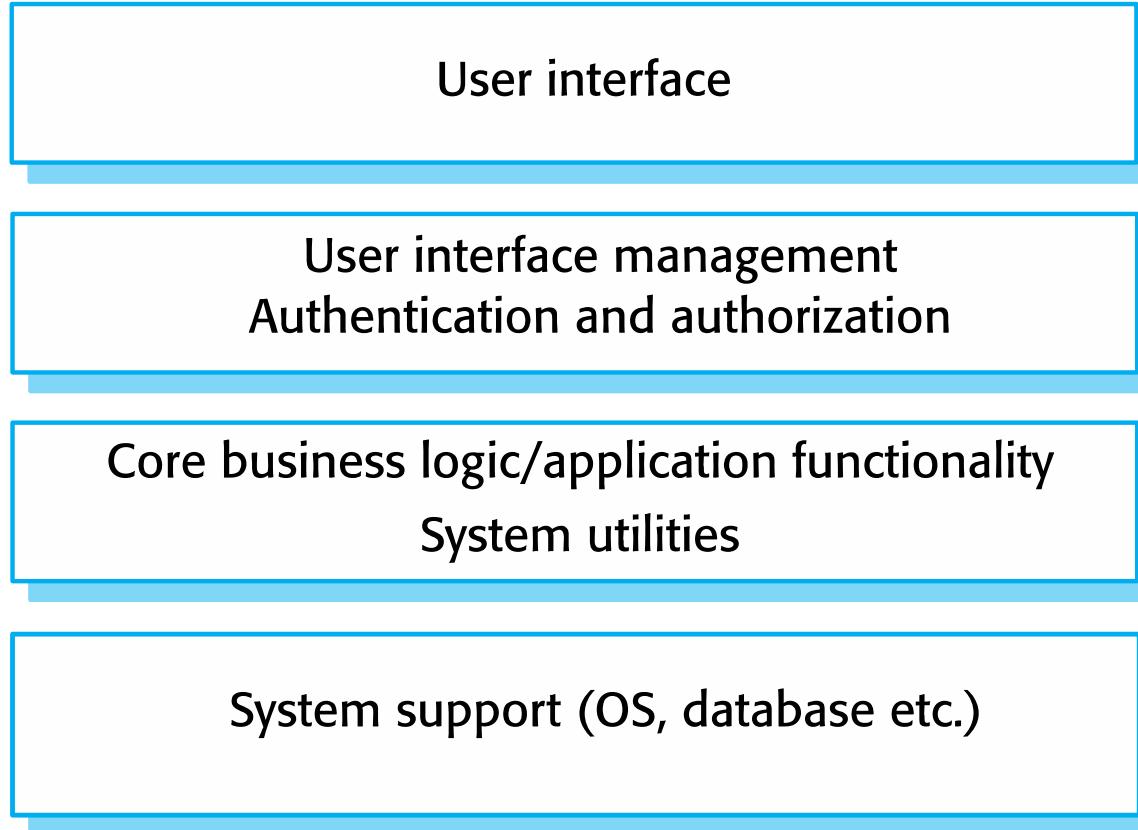
- ✧ Used to model the **interfacing of sub-systems**.
- ✧ Organises the system **into a set of layers** (or abstract machines) **each of which provide a set of services**.
- ✧ Supports the **incremental development of sub-systems** in different layers. When a layer interface changes, only the adjacent layer is affected.
- ✧ However, often artificial to structure systems in this way.

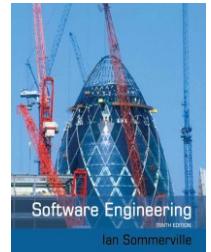
The Layered architecture pattern



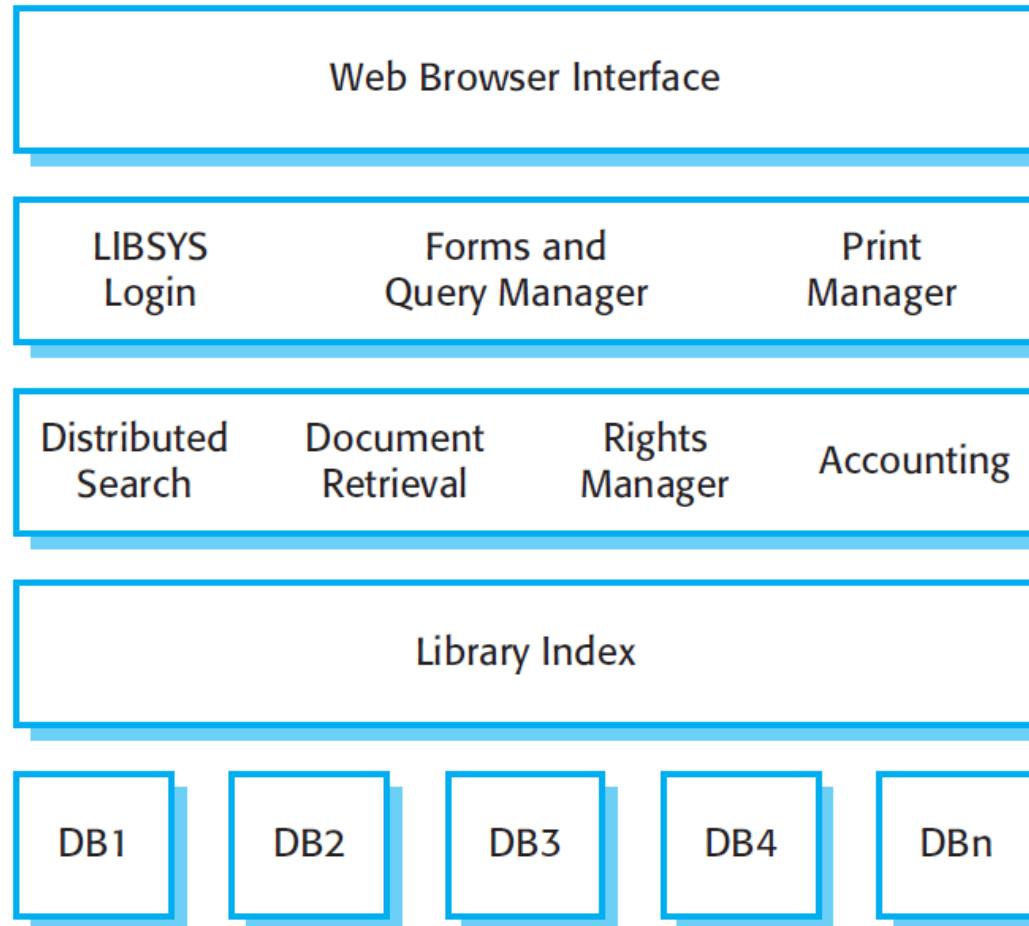
Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems ; when the development is spread across several teams with each team responsibility for a layer of functionality ; when there is a requirement for multi-level security .
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

A generic layered architecture





The architecture of the LIBSYS system



Client-server architecture



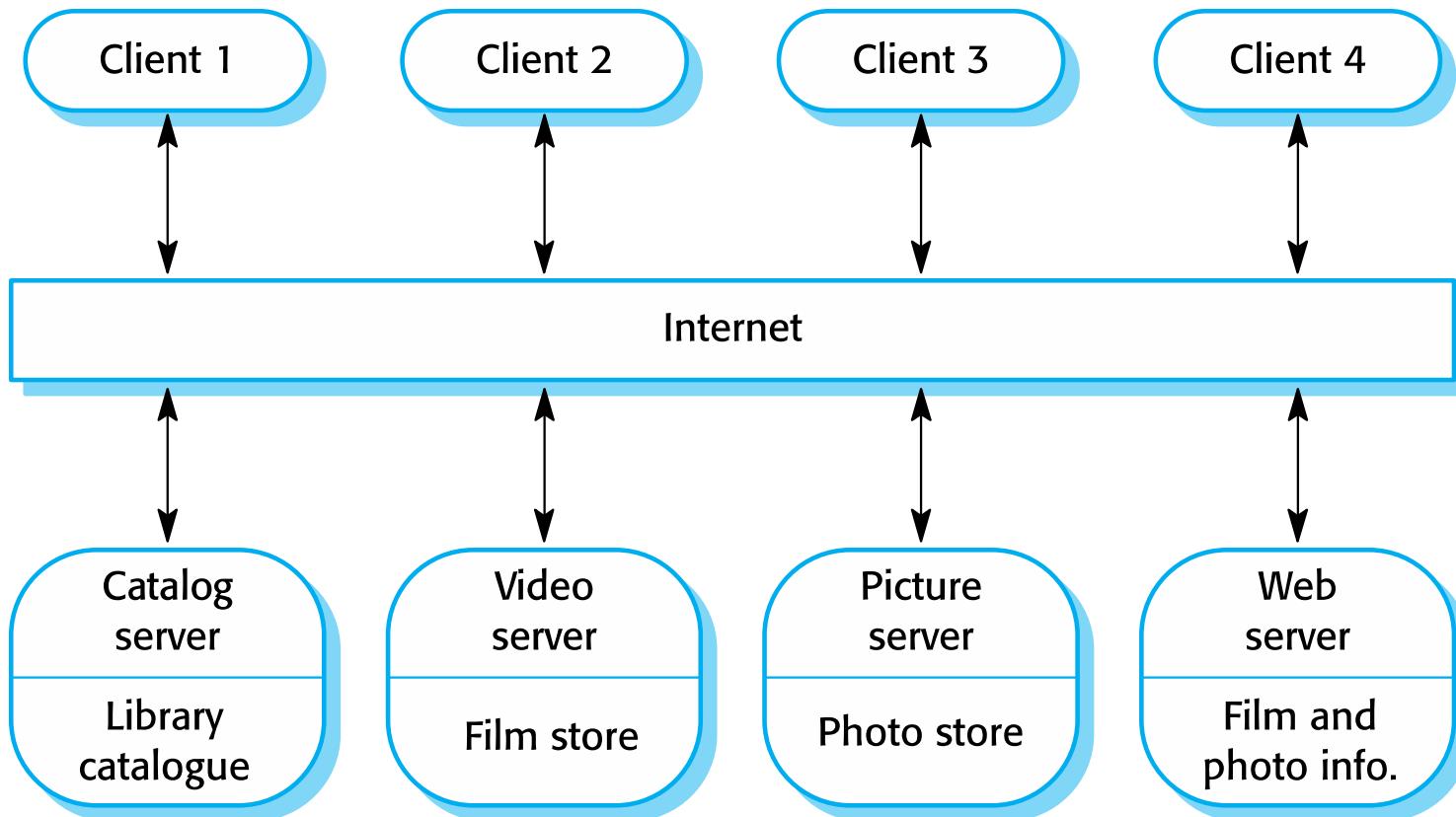
- ✧ Distributed system model which shows **how data and processing is distributed across a range of components.**
 - Can be implemented on a single computer.

Components:

- ✧ Set of **stand-alone servers** which provide specific services such as printing, data management, etc.
- ✧ Set of **clients which call on these services.**
- ✧ Network which allows clients to access servers.



A client–server architecture for a film library



The Client–server pattern

location factor is important in this approach



Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services , with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations . Because servers can be replicated , may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network . General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system . May be management problems if servers are owned by different organizations.

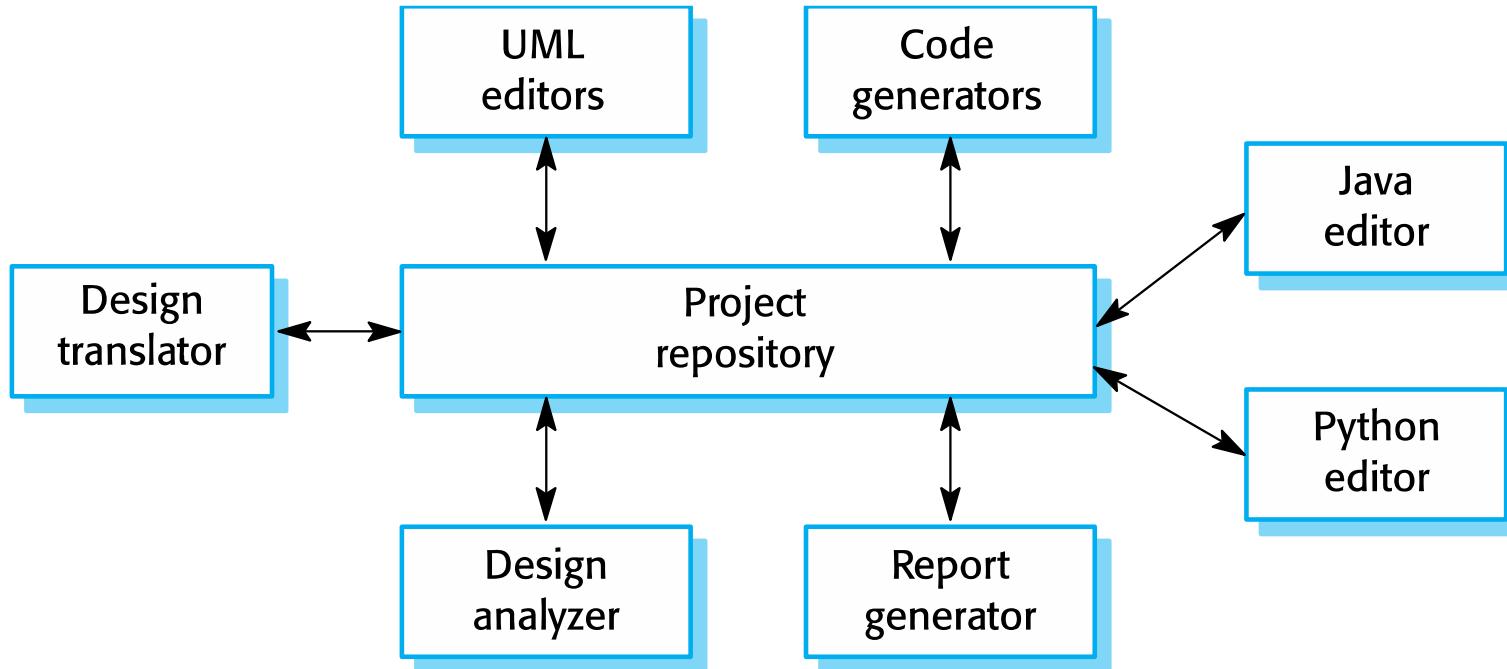
we don't have a lot of coupling

Repository architecture



- ✧ Sub-systems must exchange data. This may be done in two ways:
 - Each sub-system **maintains its own database** and passes data explicitly to other sub-systems.
 - Shared data is held in a **central database or repository** and may be accessed by all sub-systems;
- ✧ When **large amounts of data** are to be shared, the **repository model of sharing is most commonly used** as this is an efficient data sharing mechanism.

A repository architecture for an IDE





The Repository pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. There may be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

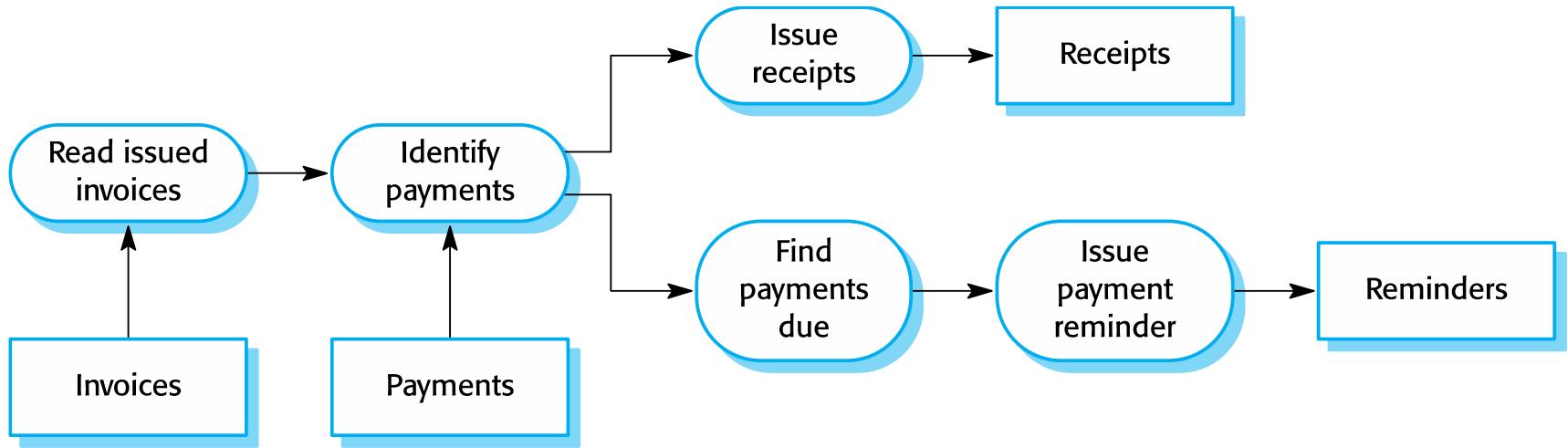
Pipe and filter architecture



- ✧ Functional transformations **process their inputs to produce outputs.**
- ✧ May be referred to as a pipe and filter model (as in UNIX shell).
- ✧ Variants of this approach are very common. When transformations are sequential, this is a **batch sequential model** which is extensively used in **data processing systems**.
- ✧ Not really suitable for interactive systems.

because it takes a lot of time to do its work, so that may take two days long and user may be tired

An example of the pipe and filter architecture used in a payments system

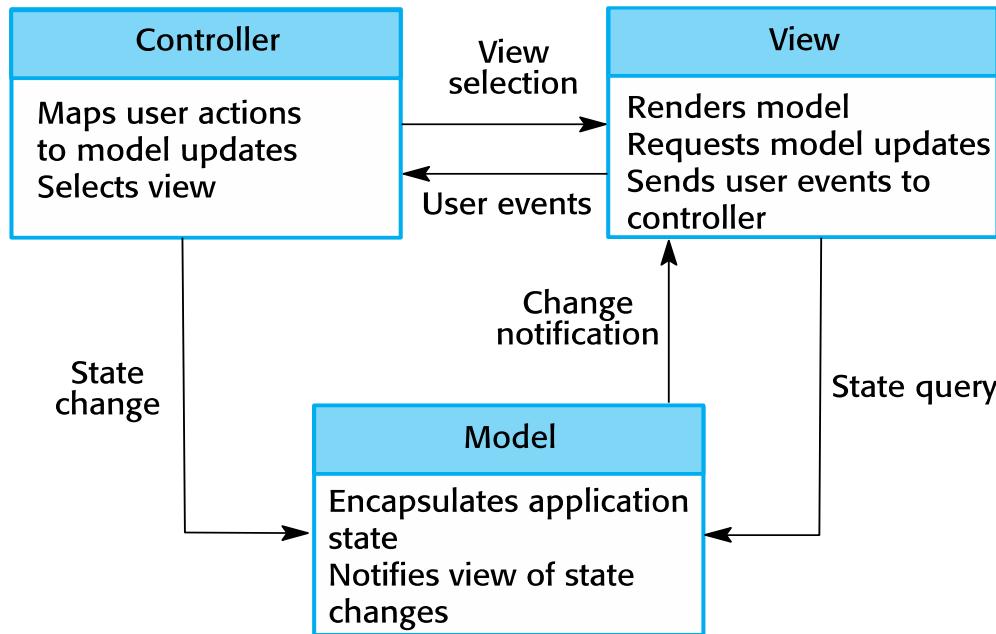
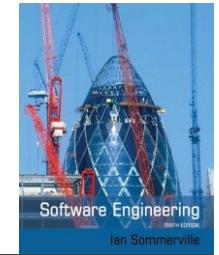


The pipe and filter pattern



Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation . The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

The organization of the Model-View-Controller



The Model-View-Controller (MVC) pattern



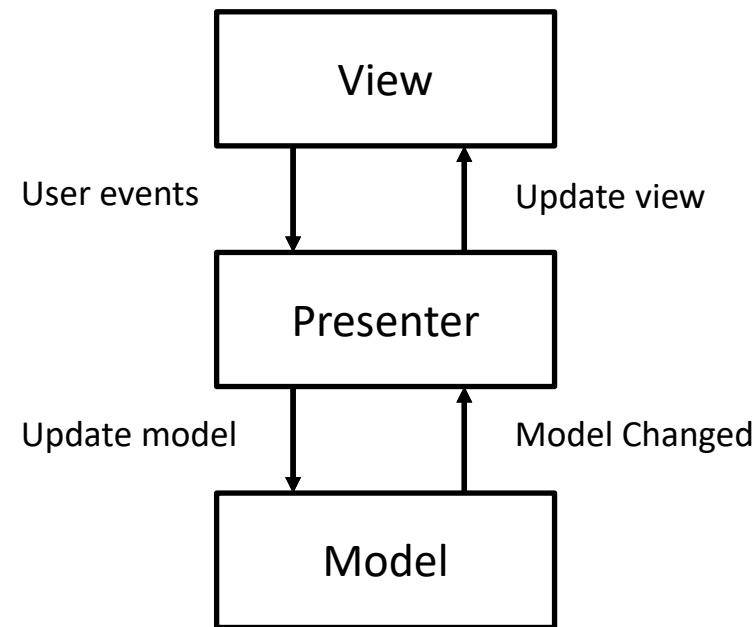
Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

Model-View-Presenter



✧ MVP

- Model doesn't interact with view directly and vice versa
 - Less coupling between view and model (easier to do unit testing)
- One-to-one relationship between presenter and view
 - In MVC, controller has a many-to-one relationship with views.





Application architectures

Application architectures



- ✧ Application systems are designed to meet an organizational need.
- ✧ As **businesses have much in common**, their application systems also tend to have a common architecture that reflects the application requirements.
- ✧ A **generic application architecture** is **an architecture for a type of software system** that may be configured and adapted to create a system that meets specific requirements.

Use of application architectures



- ✧ As a starting point for architectural design.
- ✧ As a design checklist.
- ✧ As a way of organizing the work of the development team.
- ✧ As a means of assessing components for reuse.
- ✧ As a vocabulary for talking about application types.



Examples of application types

✧ Data processing applications

- Data driven applications that **process data in batches** without explicit user intervention during the processing. pipe and filter

✧ Transaction processing applications

- Data-centred applications that process **user requests and update information** in a system database. repository, client server

✧ Event processing systems

- Applications where **system actions depend on interpreting events** from the system's environment.

✧ Language processing systems

- Applications where the **users' intentions are specified in a formal language** that is processed and interpreted by the system.

Application type examples



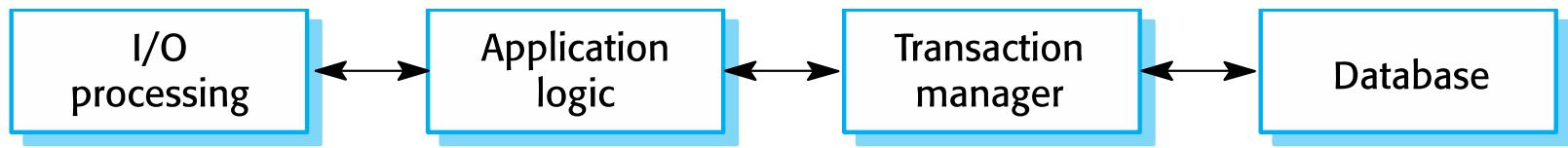
- ✧ Focus here is on transaction processing and language processing systems.
- ✧ Transaction processing systems
 - E-commerce systems
 - Reservation systems
- ✧ Language processing systems
 - Compilers
 - Command interpreters

Transaction processing systems



- ✧ Process user **requests for information** from a database or **requests to update** the database.
- ✧ From a user perspective a transaction is:
 - Any coherent sequence of operations that satisfies a goal;
 - For example - find the times of flights from London to Paris.
- ✧ Users make asynchronous requests for service which are then processed by a transaction manager.

The structure of transaction processing applications



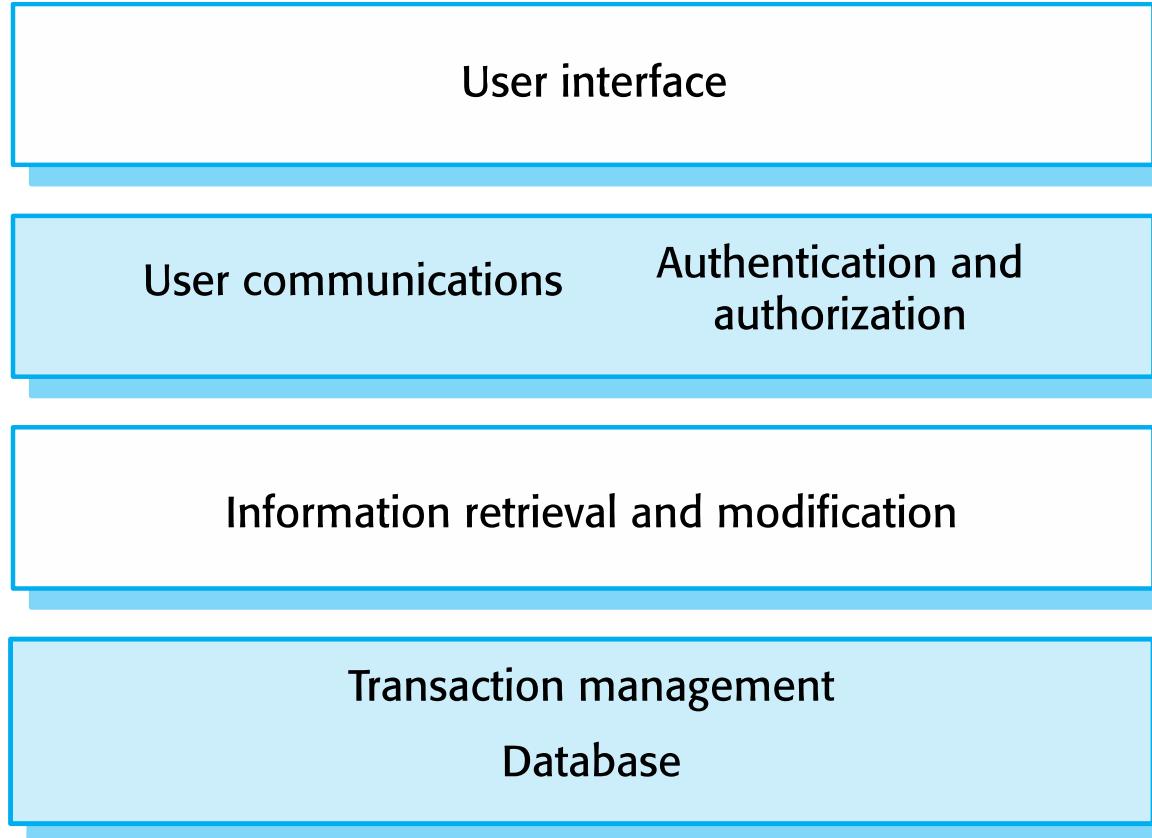
Information systems architecture



- ✧ Information systems have a generic architecture that can be organized as a layered architecture.
- ✧ These are transaction-based systems as interaction with these systems generally involves database transactions.
- ✧ Layers include:
 - The user interface
 - User communications
 - Information retrieval
 - System database

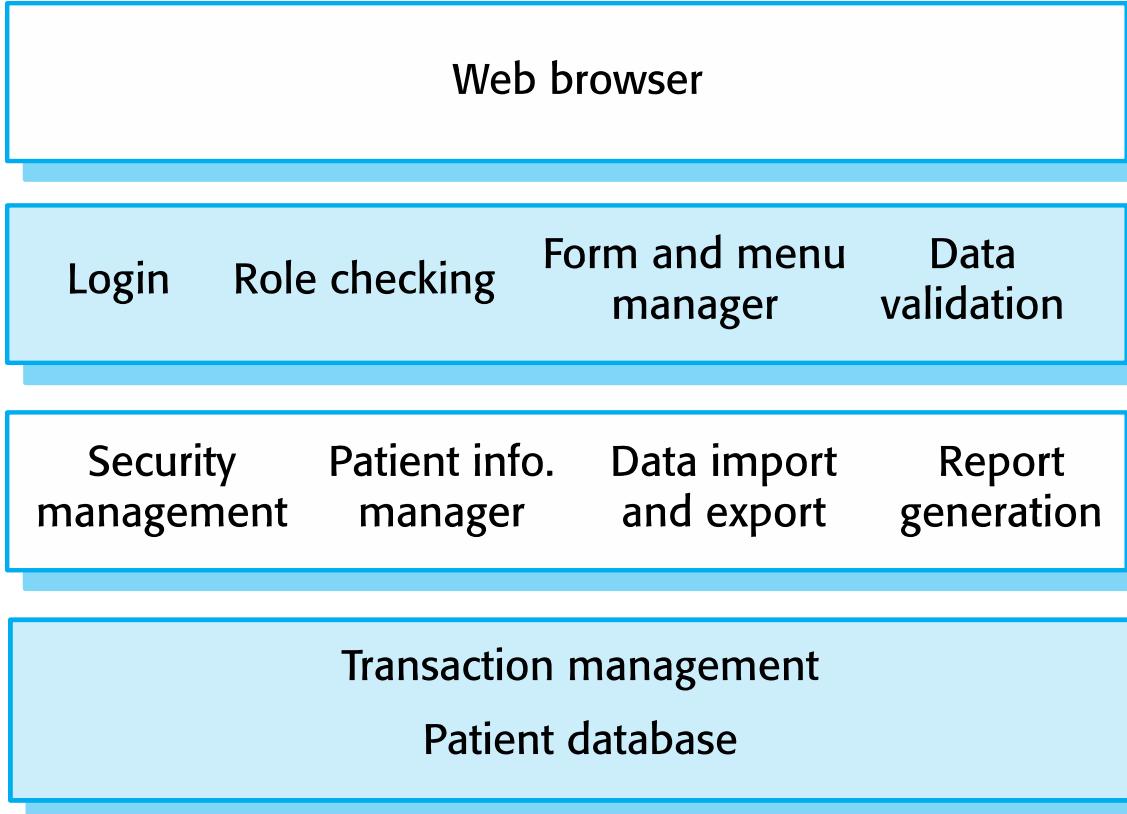


Layered information system architecture





The architecture of the Mentcare system



Web-based information systems



- ✧ Information and resource management systems are now **usually web-based systems** where the **user interfaces** are implemented using a **web browser**.
- ✧ For example, e-commerce systems are Internet-based resource management systems
 - accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer.
- ✧ In an e-commerce system, the application-specific layer includes additional functionality **supporting a 'shopping cart'** in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

Server implementation



- ✧ These systems are often implemented as multi-tier client server/architectures
 - The web server is **responsible for all user communications**, with the user interface implemented using a web browser;
 - The application server is responsible for implementing **application-specific logic** as well as information storage and retrieval requests;
 - The database server **moves information to and from the database** and handles transaction management.

Key points



- ✧ A software architecture is a description of how a software system is organized.
- ✧ Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- ✧ Architectures may be documented from several different perspectives or views such as a conceptual view, a logical view, a process view, and a development view.
- ✧ Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.

Key points



- ✧ Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
- ✧ Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users.
- ✧ Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.

Compiler components



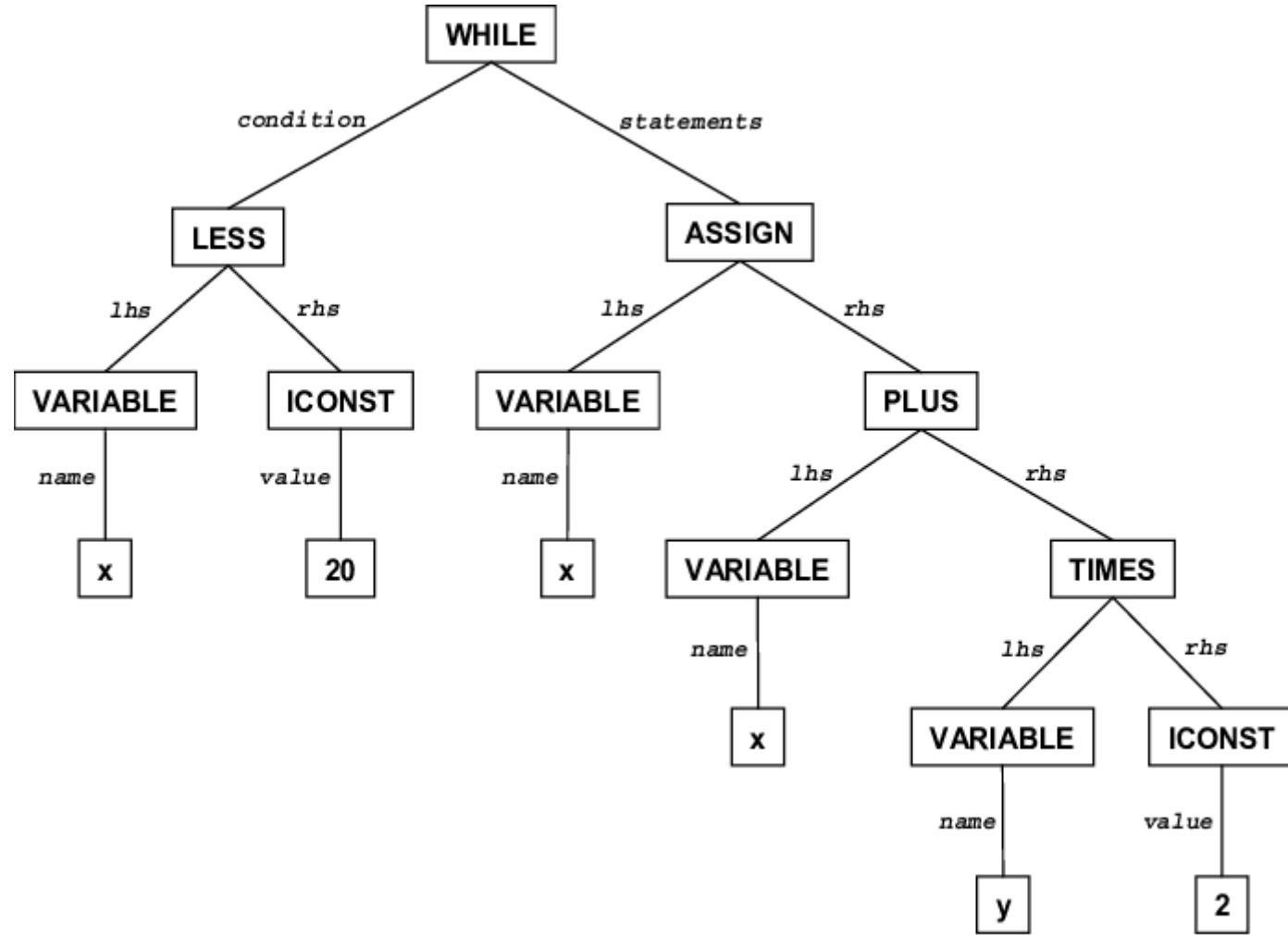
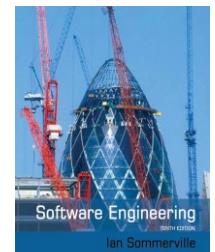
- ✧ A **lexical analyzer**, which **takes input language tokens** and **converts them to an internal form**.
- ✧ A **symbol table**, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
- ✧ A **syntax analyzer**, which checks the syntax of the language being translated.
- ✧ A **syntax tree**, which is an internal structure representing the program being compiled.

Compiler components

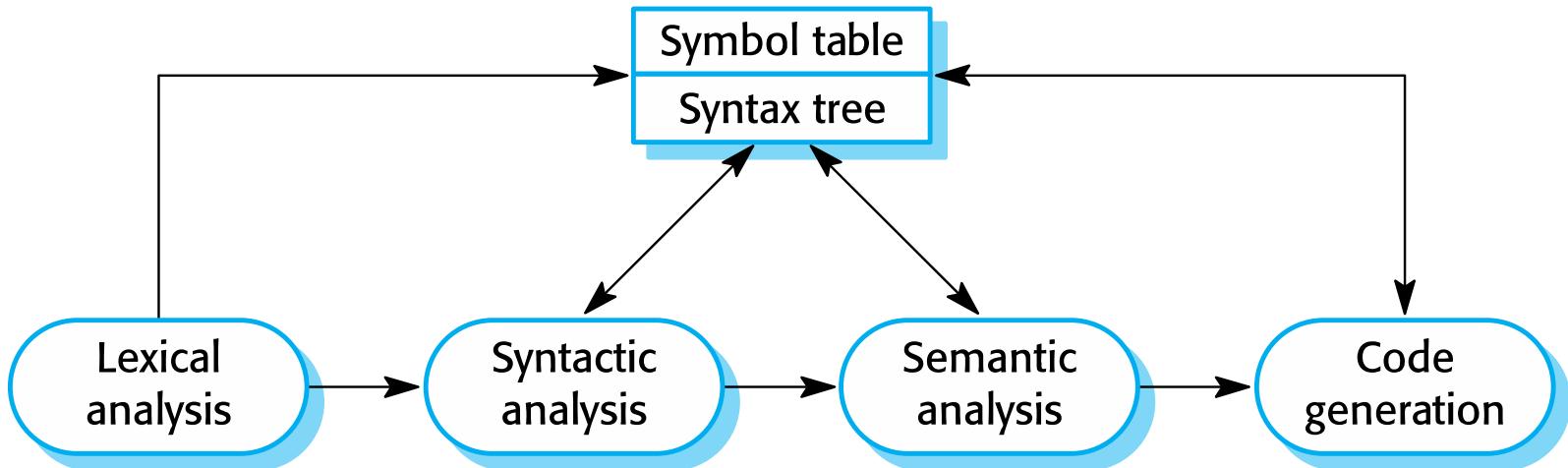


- ✧ A **semantic analyzer** that uses information from the syntax tree and the symbol table to **check the semantic correctness** of the input language text.
- ✧ A **code generator** that ‘walks’ the syntax tree and generates abstract machine code.

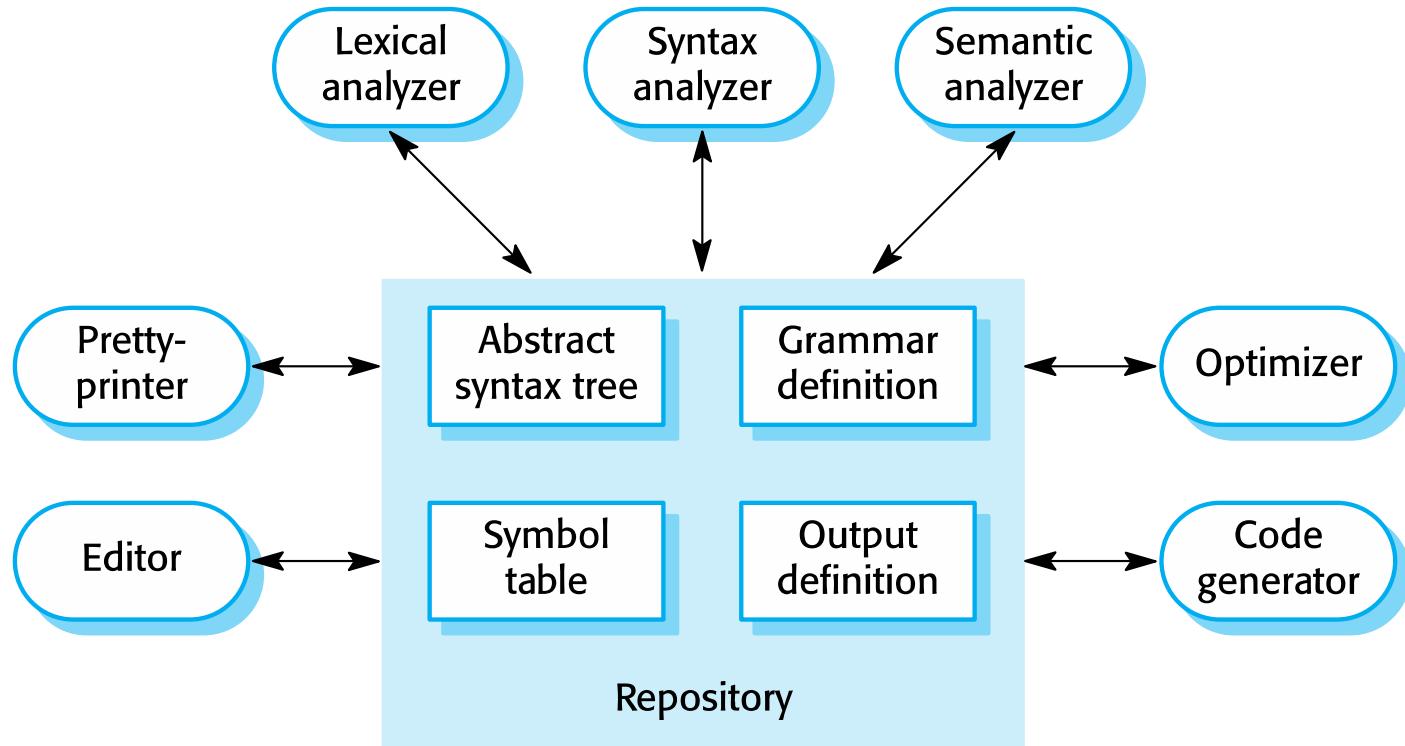
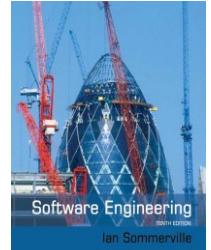
Syntax tree example

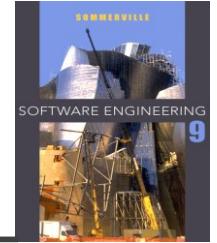


A pipe and filter compiler architecture

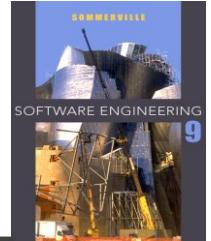


A repository architecture for a language processing system



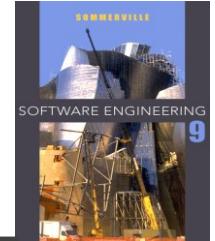


Chapter 7 – Design and Implementation



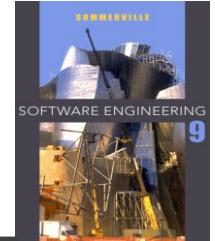
Topics covered

- ✧ Object-oriented design using the UML
- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open source development



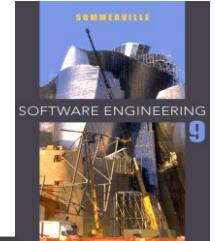
Design and implementation

- ✧ Software **design and implementation** is the stage in the software engineering process at which an **executable software system is developed**.
- ✧ Software design and implementation activities are invariably **inter-leaved**.
 - Software design is a creative activity in which you **identify software components and their relationships**, based on a customer's requirements.
 - Implementation is the process of **realizing the design** as a program.



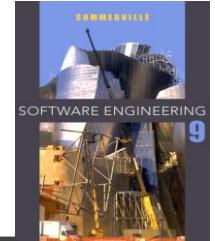
Build or buy

- ✧ In a wide range of domains, it is now possible to **buy off-the-shelf systems (COTS)** that can be adapted and tailored to the users' requirements.
 - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. **It can be cheaper and faster** to use this approach rather than developing a system in a conventional programming language.
- ✧ When you develop an application in this way, **the design process** becomes concerned with **how to use the configuration features** of that system to **deliver the system requirements**.



An object-oriented design process

- ✧ Structured object-oriented design processes involve **developing a number of different system models**.
- ✧ They require **a lot of effort** for development and maintenance of these models and, for small systems, this may not be cost-effective.
- ✧ However, for large systems developed by different groups design models are an **important communication mechanism**.



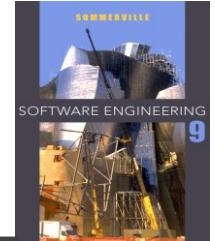
Process stages

✧ Common activities in these processes include:

- Define the **context** and **modes of use** of the system;
- Design the system architecture;
- Identify the **principal system object classes**;
- Develop design models;
- Specify object interfaces.

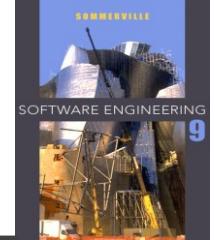
✧ Process illustrated here using a design for a wilderness weather station.

context model is used. sequence diagram is not useful here.

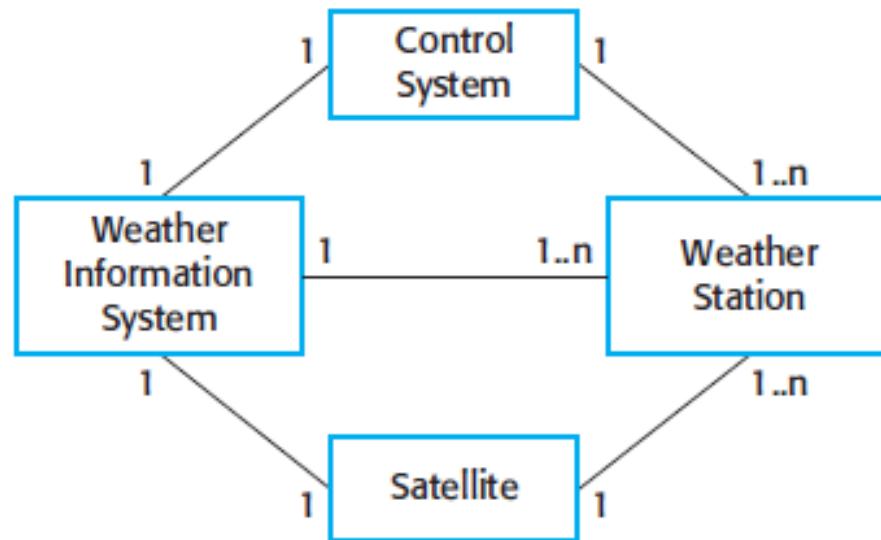


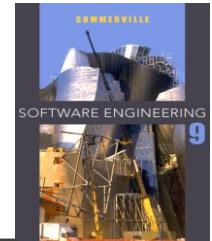
System context and interactions

- ✧ Understanding the **relationships between the software and its external environment** is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- ✧ Understanding of the context also lets you **establish the boundaries** of the system. Setting the system boundaries helps you decide **what features are implemented** in the system being designed and what features are in other **associated systems**.



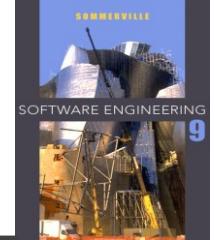
System context for the weather station



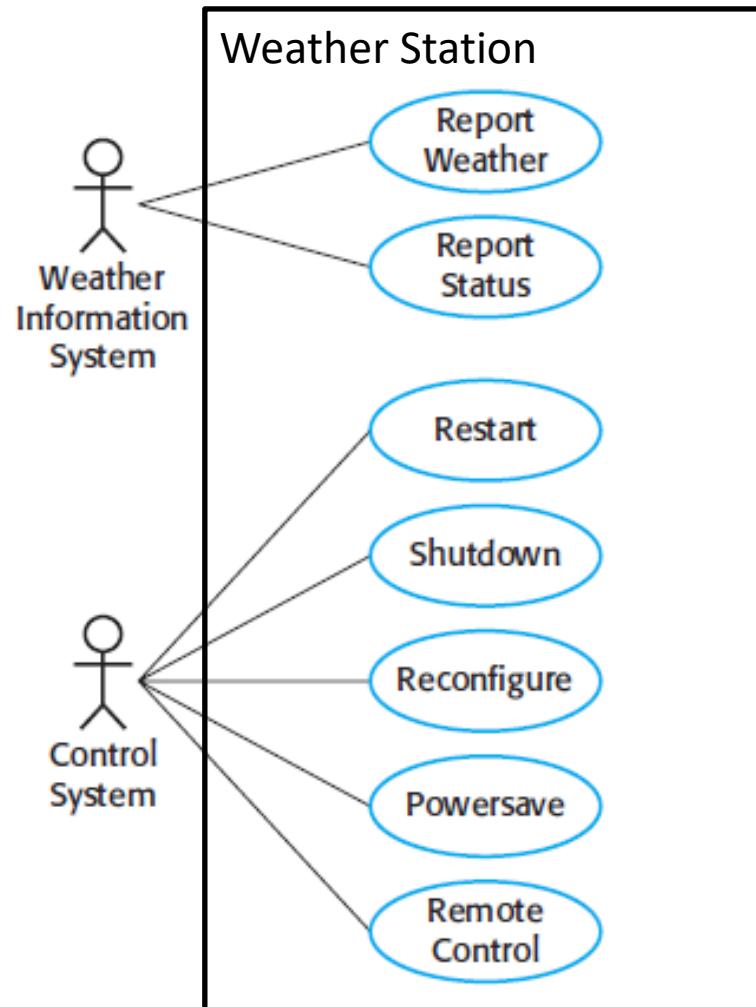


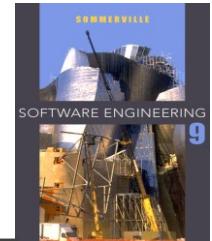
Context and interaction models

- ✧ A system context model is a **structural model** that demonstrates the **other systems** in the environment of the system being developed.
- ✧ An interaction model is a **dynamic model** that shows how the system interacts with its environment as it is used.
 - Users interactions
 - Systems interactions
 - eg., sequence and use case diagrams



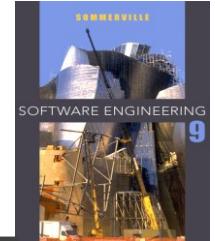
Weather station use cases





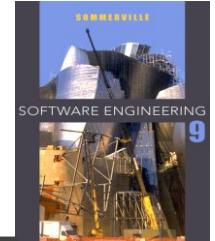
Use case description—Report weather

System	Weather station
Use case	Report weather
Actors	Weather information system
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.



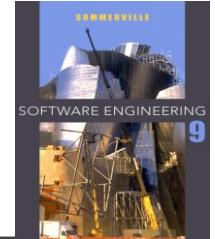
Process stages

- ✧ Define the context and modes of use of the system
- ✧ Design the system architecture
- ✧ Identify the principal system object classes
- ✧ Develop design models
- ✧ Specify object interfaces

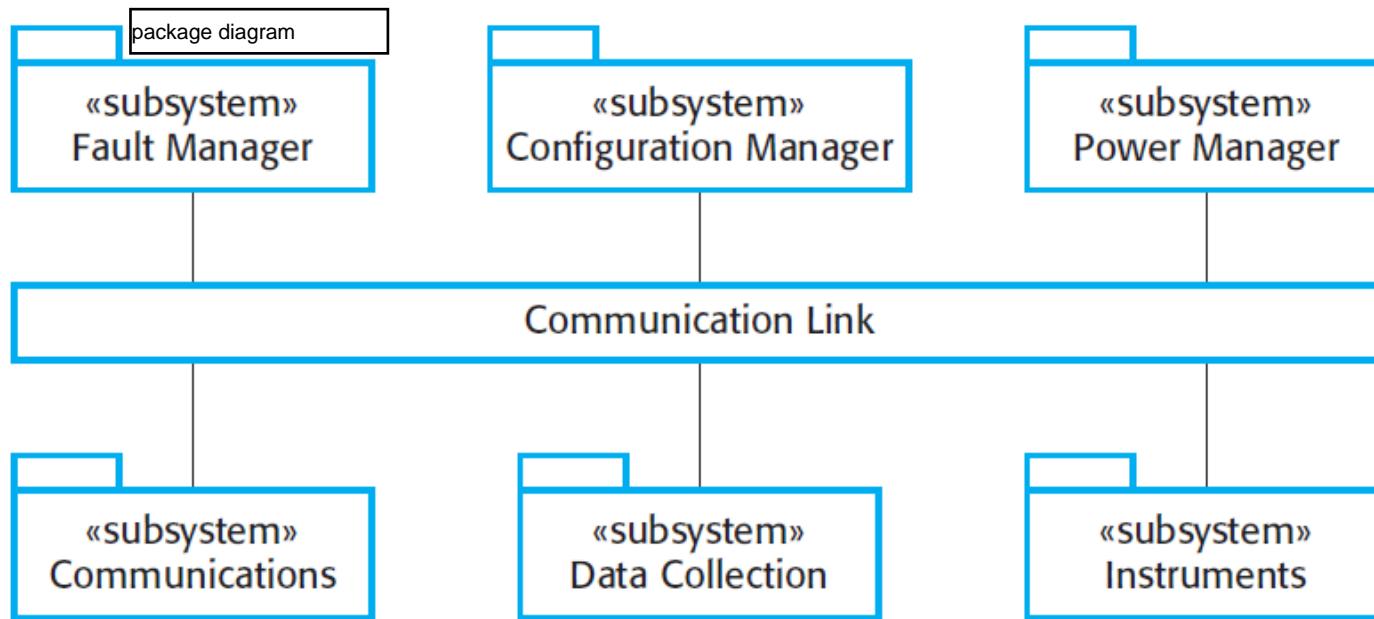


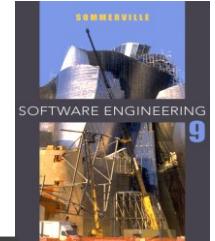
Architectural design

- ✧ Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- ✧ You identify the **major components** that make up the system and **their interactions**, and then **may organize the components using an architectural pattern** such as a layered or client-server model.
- ✧ The weather station is composed of independent subsystems that communicate by **broadcasting messages on a common infrastructure**.

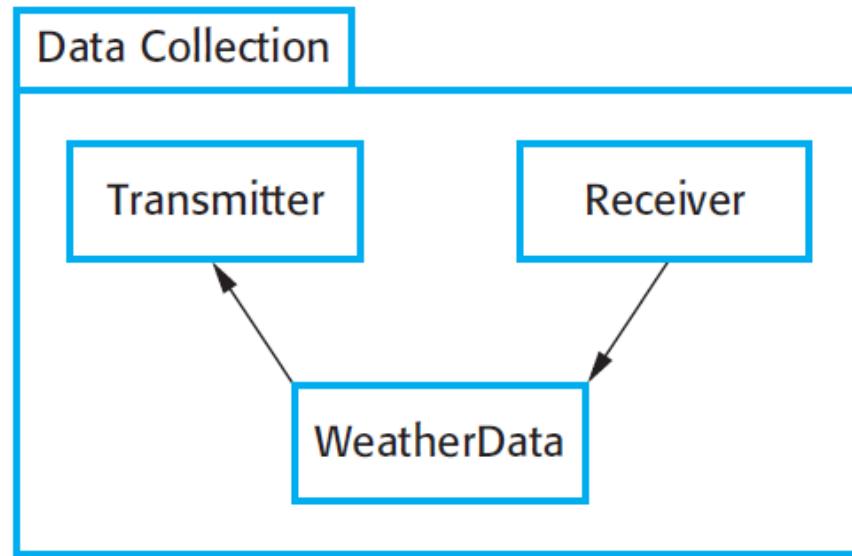


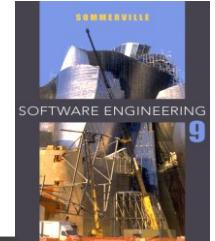
High-level architecture of the weather station





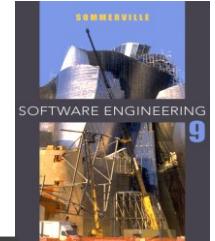
Architecture of data collection system





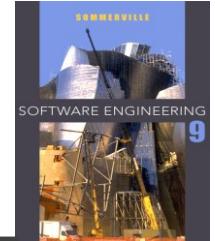
Process stages

- ✧ Define the context and modes of use of the system
- ✧ Design the system architecture
- ✧ Identify the principal system object classes
- ✧ Develop design models
- ✧ Specify object interfaces



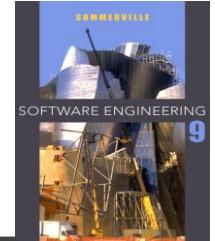
Object class identification

- ✧ Identifying **object classes** is often a **difficult** part of object oriented design.
- ✧ There is no 'magic formula' for object identification. It relies on the **skill, experience and domain knowledge** of system designers.
- ✧ Object identification is an **iterative process**. You are unlikely to get it right first time.



Approaches to identification

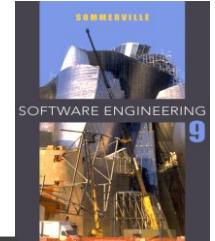
- ✧ Use a **grammatical approach** based on a natural language description of the system.
- ✧ Base the **identification on tangible** things in the application domain.
- ✧ Use a scenario-based analysis. The **objects, attributes and methods in each scenario** are identified.



Weather station description

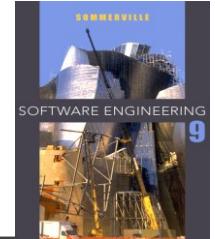
A **weather station** is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected periodically.

When a command is issued to transmit the weather data, the weather station processes and summarises the collected data. The summarised data is transmitted to the mapping computer when a request is received.



Weather station object classes

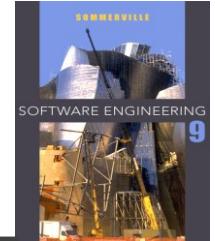
- ✧ Object class identification in the weather station system may be based on the tangible hardware and data in the system:
 - Ground thermometer, Anemometer, Barometer
 - Application domain objects that are ‘hardware’ objects related to the instruments in the system.
 - Weather station
 - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
 - Weather data
 - Encapsulates the summarized data from the instruments.



Weather station object classes

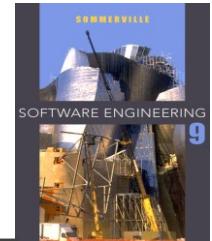
WeatherStation	WeatherData
identifier reportWeather() reportStatus() powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)	airTemperatures groundTemperatures windSpeeds windDirections pressures rainfall collect() summarize()

Ground Thermometer	Anemometer	Barometer
gt_Ident temperature get() test()	an_Ident windSpeed windDirection get() test()	bar_Ident pressure height get() test()



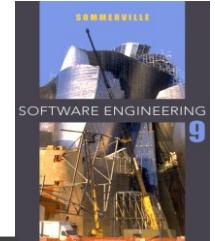
Process stages

- ✧ Define the context and modes of use of the system
- ✧ Design the system architecture
- ✧ Identify the principal system object classes
- ✧ Develop design models
- ✧ Specify object interfaces



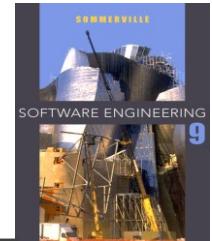
Design models

- ✧ Design models show the **objects and object classes** and **relationships** between these entities.
- ✧ Static models describe the **static structure** of the system in terms of object classes and relationships.
such as class diagrams
- ✧ Dynamic models describe the **dynamic interactions** between objects.
sequence diagrams and state diagrams



Examples of design models

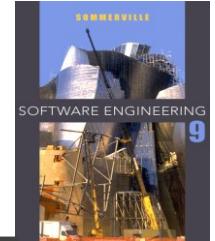
- ✧ **Subsystem models** that show logical groupings of objects into coherent subsystems. (Static)
- ✧ **Sequence models** that show the sequence of object interactions. (Dynamic)
- ✧ **State machine models** that show how individual objects change their state in response to events. (Dynamic)
- ✧ Other models include use-case models, aggregation models, generalisation models, etc.



Subsystem models

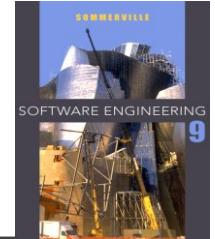
- ✧ Shows how the design is organised into **logically related groups of objects**.

- ✧ In the UML, these are **shown using packages** - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

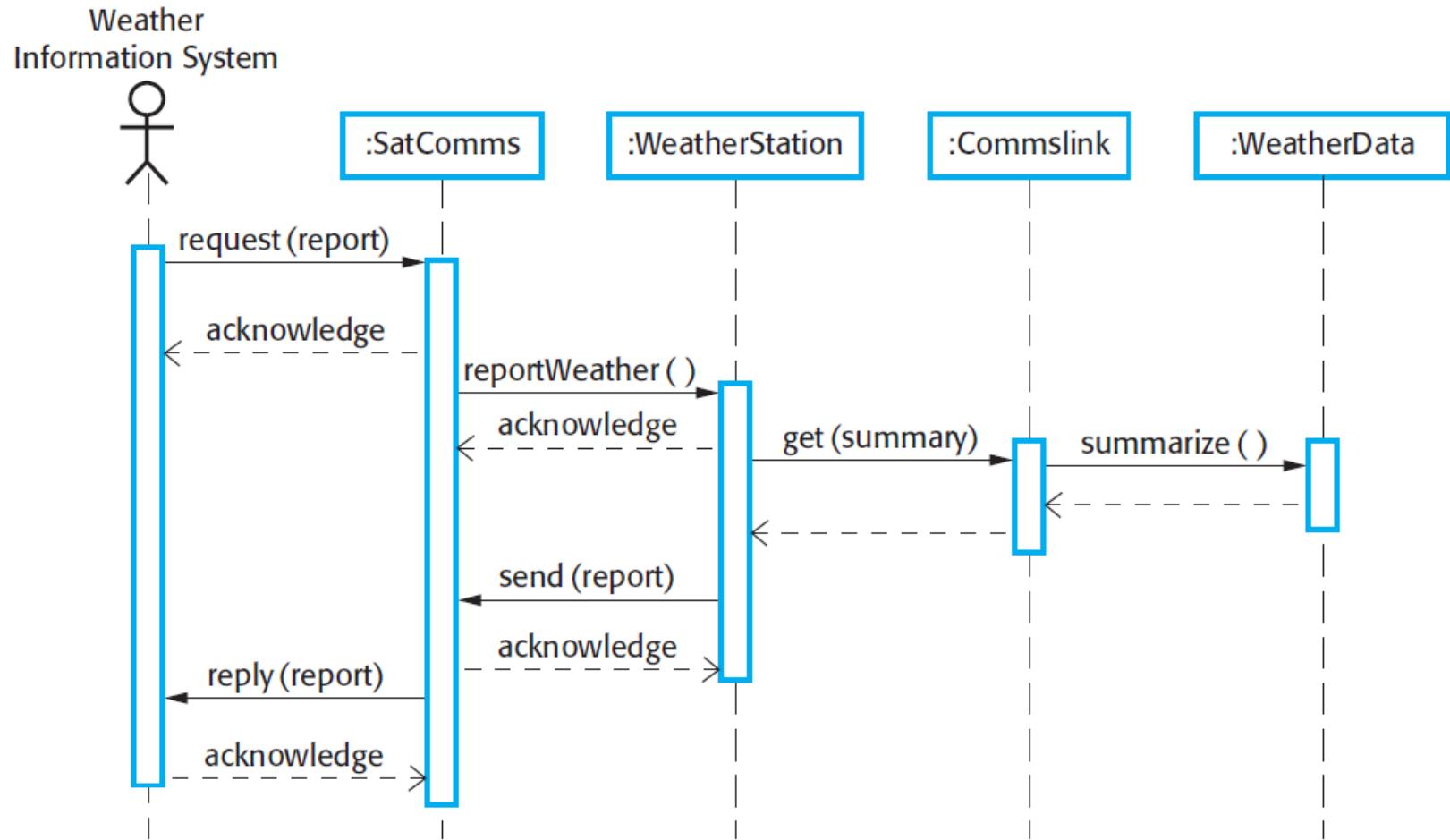


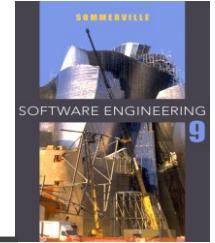
Sequence models

- ✧ Sequence models show the sequence of object interactions that take place
 - Objects are arranged horizontally across the top;
 - Time is represented vertically so models are read top to bottom;
 - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
 - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.



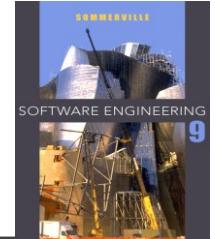
Sequence diagram describing data collection



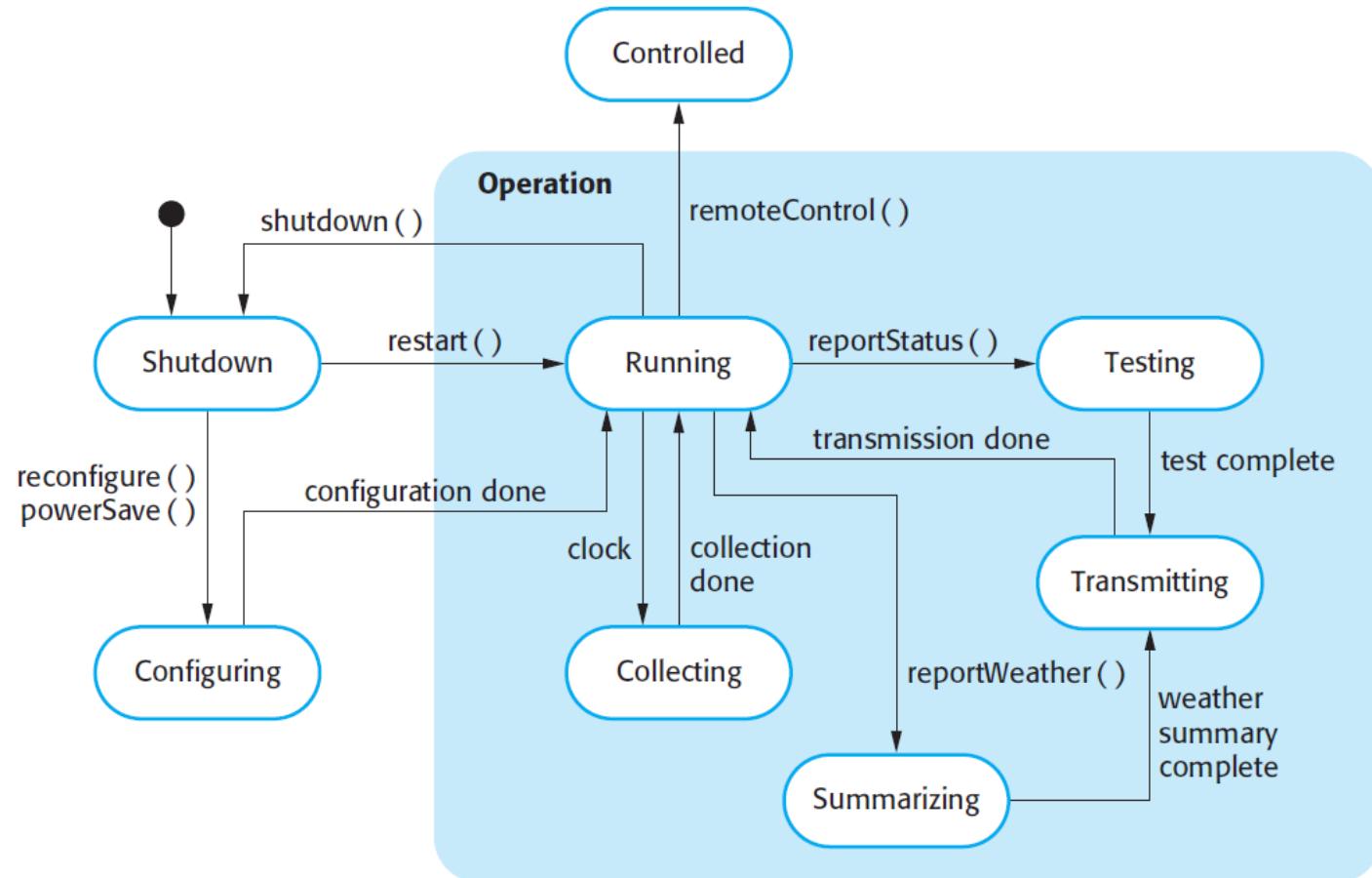


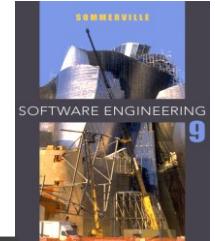
State diagrams

- ✧ State diagrams are used to show **how objects respond** to different **service requests** and the state transitions triggered by these requests.
- ✧ State diagrams are useful **high-level models of a system** or an **object's run-time behavior**.
- ✧ You don't usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.



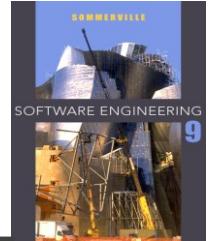
Weather station state diagram





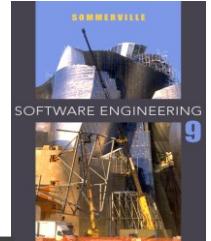
Process stages

- ✧ Define the context and modes of use of the system
- ✧ Design the system architecture
- ✧ Identify the principal system object classes
- ✧ Develop design models
- ✧ Specify object interfaces



Interface specification

- ✧ Object interfaces **have to be specified** so that the objects and other components can be **designed in parallel**.
- ✧ Designers should avoid designing the interface **implementation and data representation**.
- ✧ Objects may have several interfaces which are viewpoints on the methods provided.
- ✧ The UML uses **class diagrams** for interface specification.



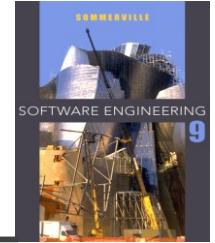
Weather station interfaces

«interface» Reporting

weatherReport (WS-Ident): Wreport
statusReport (WS-Ident): Sreport

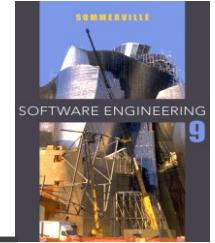
«interface» Remote Control

startInstrument (instrument): iStatus
stopInstrument (instrument): iStatus
collectData (instrument): iStatus
provideData (instrument): string



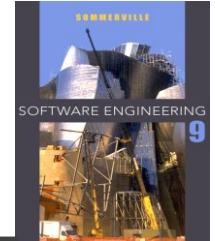
Key points

- ✧ Software design and implementation are **inter-leaved activities**. The **level of detail** in the design **depends on the type of system** and whether you are using a **plan-driven or agile approach**.
- ✧ The process of object-oriented design includes activities to **design the system architecture**, **identify objects** in the system, describe the design using different **object models** and document the **component interfaces**.
- ✧ A range of different models may be produced during an object-oriented design process. These include **static models** (class models, generalization models, association models) and **dynamic models** (sequence models, state machine models).
- ✧ Component **interfaces must be defined precisely** so that **other objects can use them**. A UML interface stereotype may be used to define interfaces.



Design patterns

- ✧ A design pattern is a way of **reusing abstract knowledge** about a problem and its solution.
- ✧ A pattern is a description of **the problem and the essence of its solution**.
- ✧ It should be **sufficiently abstract** to be reused in different settings.
- ✧ Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.



Pattern elements

✧ Name

- A meaningful pattern identifier.

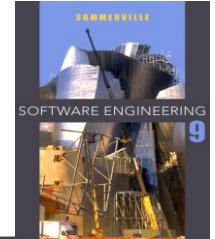
✧ Problem description.

✧ Solution description.

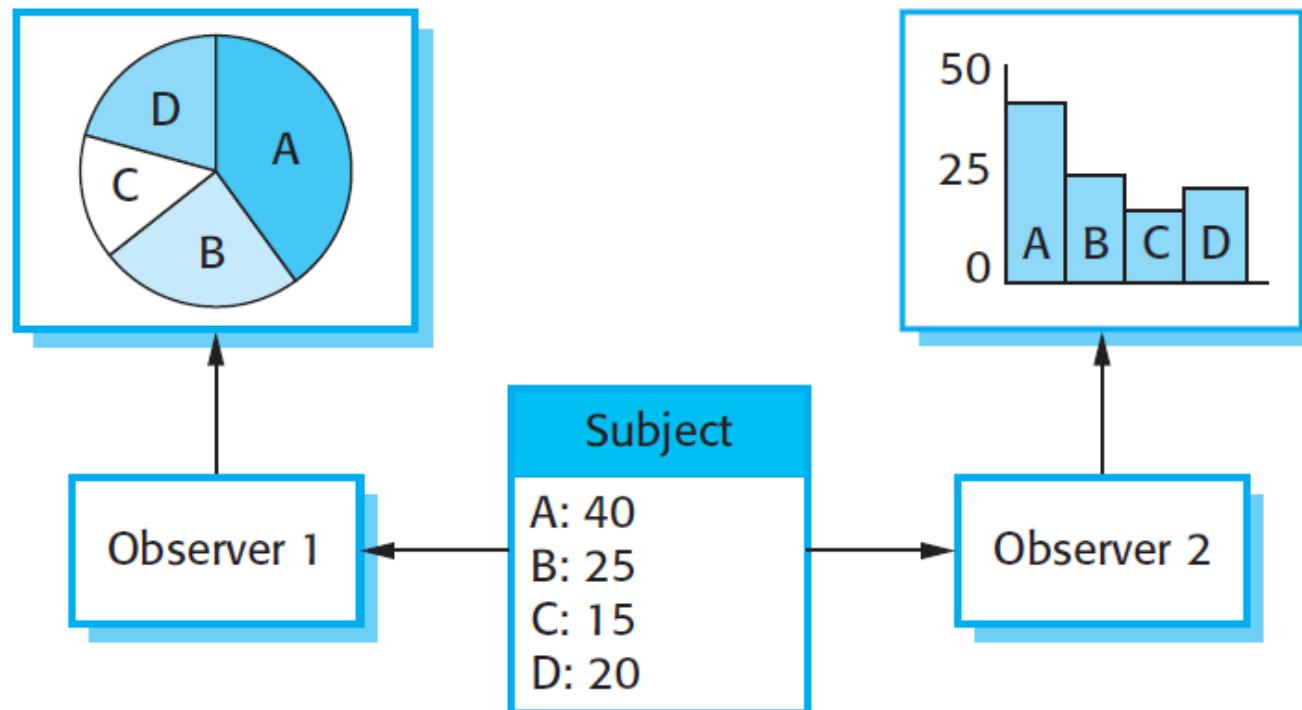
- Not a concrete design but a template for a design solution that can be instantiated in different ways.

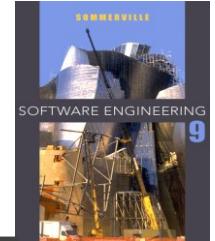
✧ Consequences

- The results and trade-offs of applying the pattern.



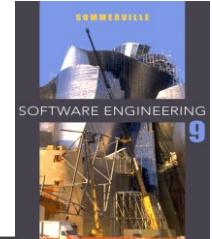
Multiple displays using the Observer pattern



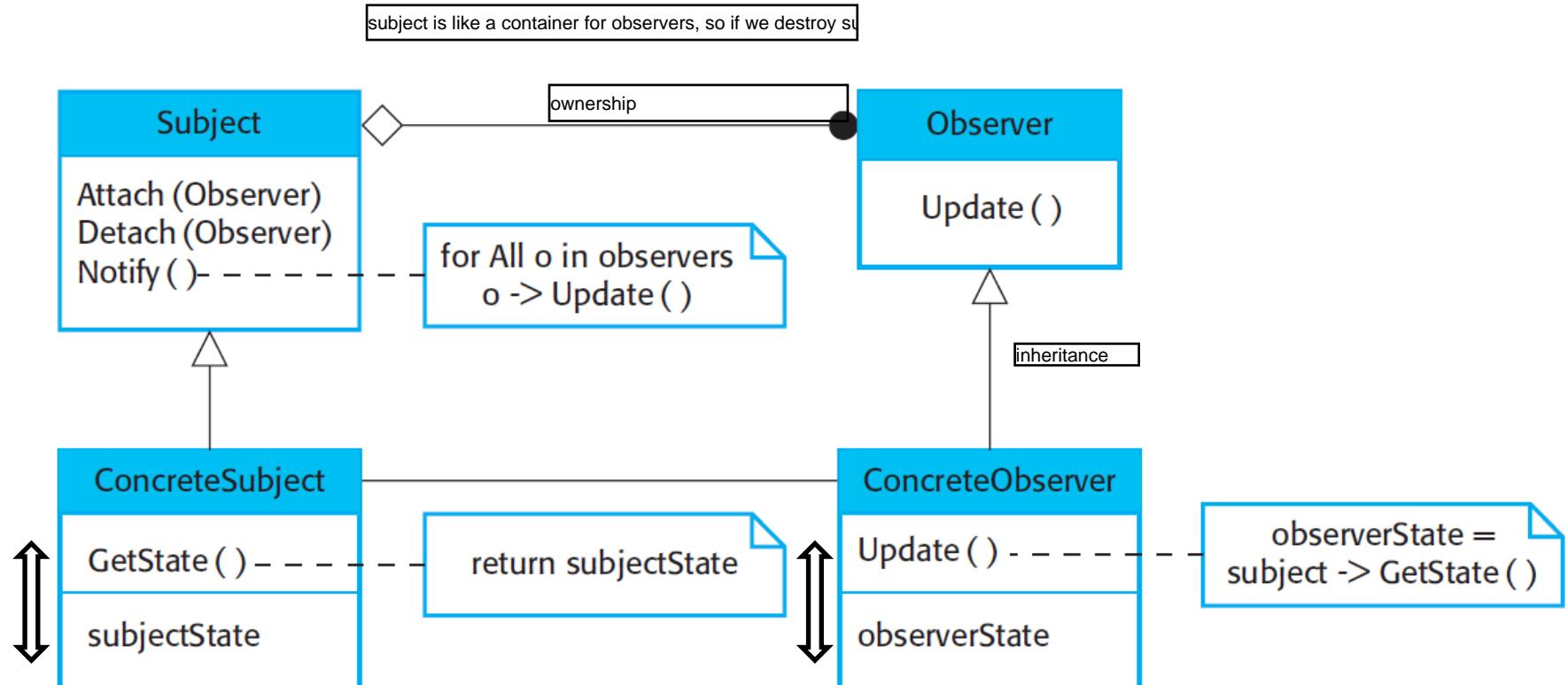


The Observer pattern

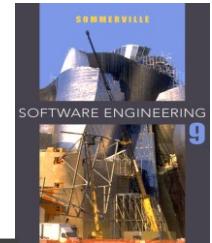
- ✧ Name
 - Observer.
- ✧ Description
 - Separates the display of object state from the object itself.
- ✧ Problem description
 - Used when multiple displays of state are needed.
- ✧ Solution description
 - See slide with UML description.
- ✧ Consequences
 - Optimisations to enhance display performance are impractical.



A UML model of the Observer pattern

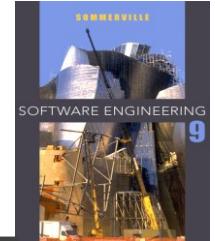


The Observer pattern (1)

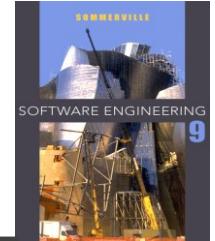


Pattern name	Observer
Description	<p>Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.</p> <p style="border: 1px solid black; padding: 2px;">low coupling, subject does not want to know about observers</p>
Problem description	<p>In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.</p> <p>This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.</p>

The Observer pattern (2)

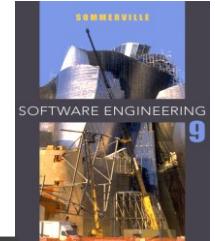


Pattern name	Observer
Solution description	<p>This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.</p> <p>The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.</p>
Consequences	<p>The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.</p>



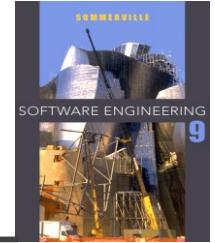
Design problems

- ✧ To use patterns in your design, you need to recognize that **any design problem you are facing may have an associated pattern** that can be applied.
 - Tell several objects that the state of some other object has changed (Observer pattern).
 - Tidy up the interfaces to a number of related classes that have often been developed incrementally (Façade pattern).
 - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
 - Allow for the possibility of extending the functionality of an existing object at run-time (Decorator pattern).



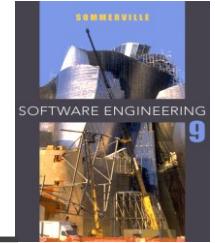
Implementation issues

- ✧ Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
 - **Reuse** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should **make as much use as possible of existing code**.
 - **Configuration management** During the development process, you have to **keep track of the many different versions** of each software component in a configuration management system.
 - **Host-target development** Production **software does not usually execute on the same computer as the software development environment**. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).



Reuse

- ✧ From the 1960s to the 1990s, most new software was developed from scratch, **by writing all code** in a high-level programming language.
 - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- ✧ **Costs and schedule pressure** mean that this approach became **increasingly unviable**, especially for commercial and Internet-based systems.
- ✧ An approach to **development based around the reuse** of existing software emerged and is now generally used for business and scientific software.



Reuse levels

✧ The abstraction level

- At this level, you **don't reuse software directly** but **use knowledge** of successful abstractions in the design of your software.

✧ The object level

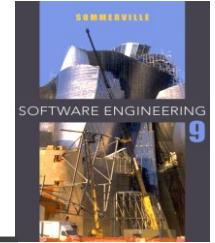
- At this level, you **directly reuse objects** from a library rather than writing the code yourself.

✧ The component level

- Components are **collections of objects and object classes** that you reuse in application systems.

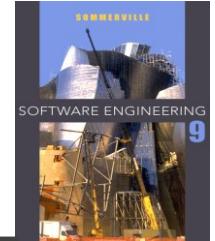
✧ The system level

- At this level, you **reuse entire application systems**.



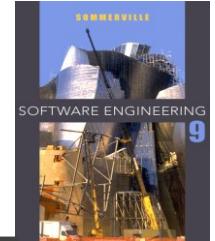
Reuse costs

- ✧ The costs of the **time spent in looking for software to reuse and assessing whether or not it meets your needs.**
- ✧ Where applicable, **the costs of buying the reusable software.** For large off-the-shelf systems, these **costs can be very high.**
- ✧ The costs of **adapting and configuring** the reusable software components or systems to reflect the requirements of the system that you are developing.
- ✧ The costs of **integrating reusable software elements** with each other (if you are using software from different sources) and with the new code that you have developed.



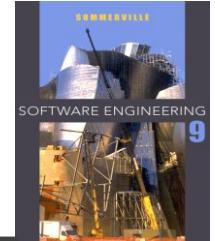
Configuration management

- ✧ Configuration management is the name given to the general process of **managing a changing software system**.
- ✧ The aim of configuration management is to **support the system integration process** so that all developers can **access the project** code and documents in a controlled way, find out **what changes** have been made, and **compile and link** components to create a system.
- ✧ For more information see also Chapter 25.



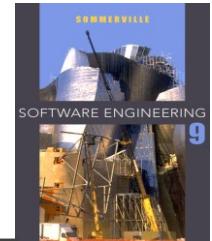
Host-target development

- ✧ Most software is developed on one computer (**the host**), but runs on a separate machine (**the target**).
- ✧ More generally, we can talk about a **development platform** and an **execution platform**.
 - A platform is **more than just hardware**.
 - It includes the **installed operating system plus other supporting software** such as a database management system or, for development platforms, an interactive development environment.
- ✧ Development platform usually has **different installed software than execution platform**; these platforms may have different architectures.



Open source development

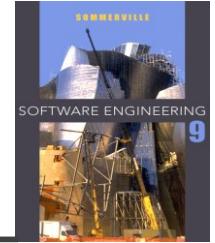
- ✧ Open source development is an approach to software development in which the **source code of a software system is published** and volunteers are invited to participate in the development process
- ✧ Its roots are in the **Free Software Foundation** (www.fsf.org), which advocates that source code **should not be proprietary but rather should always be available for users to examine and modify as they wish**.
- ✧ Open source software extended this idea by **using the Internet to recruit a much larger population of volunteer developers**. Many of them are also users of the code.



Open source systems

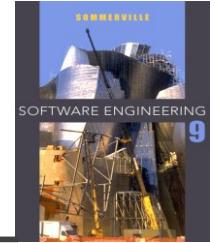
- ✧ The best-known open source product is, of course, the **Linux operating system** which is widely used as a server system and, increasingly, as a desktop environment.

- ✧ Other important open source products are **Java**, the **Apache web server** and the **MySQL** database management system.



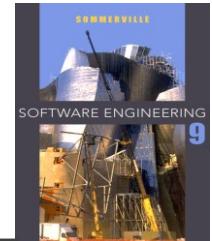
Open source issues

- ✧ Should the product that is being developed **make use of open source components?**
- ✧ Should an **open source approach** be used for the software's development?



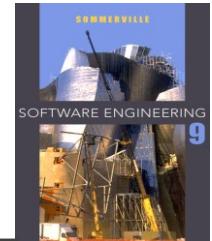
Open source business

- ✧ More and more product companies are using an open source approach to development.
- ✧ Their business model is **not reliant on selling** a software product **but on selling support** for that product.
- ✧ They believe that involving the open source community will allow software to be developed **more cheaply, more quickly** and will create a community of users for the software.



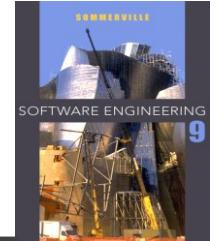
Open source licensing

- ✧ A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
 - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
 - Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
 - Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.



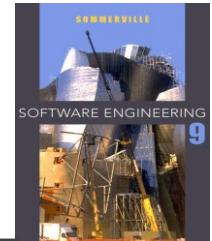
License models

- ✧ The **GNU General Public License (GPL)**. This is a so-called ‘reciprocal’ license that means **that if you use open source** software that is licensed under the GPL license, then you **must make that software open source**.
- ✧ The **GNU Lesser General Public License (LGPL)** is a variant of the GPL license where you can **write components that link to open source code without having to publish** the source of these components.
- ✧ The **Berkley Standard Distribution (BSD) License**. This is a **non-reciprocal** license, which means **you are not obliged to re-publish** any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.



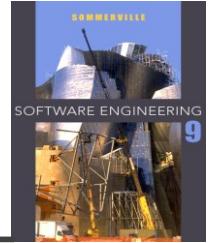
License management

- ✧ Establish a system for **maintaining information about open-source components** that are downloaded and used.
- ✧ Be aware of the **different types of licenses** and understand how a component is licensed **before it is used**.
- ✧ Be aware of **evolution pathways** for components.
- ✧ **Educate people** about open source.
- ✧ Have **auditing systems** in place.
- ✧ **Participate in the open source community**.

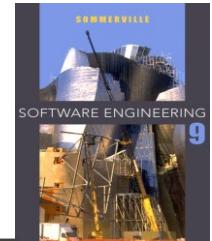


Key points

- ✧ When developing software, you should **always consider the possibility of reusing existing software**, either as components, services or complete systems.
- ✧ **Configuration management** is the process of **managing changes to an evolving software system**. It is essential when a team of people are cooperating to develop software.
- ✧ Most software development is **host-target development**. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- ✧ Open source development involves making the **source code of a system publicly available**. This means that many people can propose changes and improvements to the software.



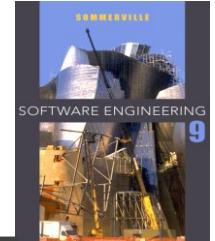
Chapter 8 – Software Testing



Program testing

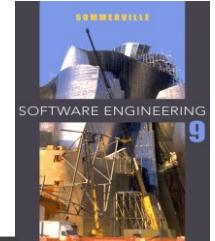
- ✧ Testing is intended **to show** that a **program does what it is intended to do** and to **discover program defects** before it is put into use.
- ✧ When you test software, you execute a program **using artificial data**.
- ✧ You **check the results** of the test run for **errors, anomalies** or information about the program's **non-functional attributes**.
- ✧ Can reveal the **presence of errors NOT their absence**.
- ✧ Testing is part of a more general **verification and validation** process.

V & V



Program testing goals

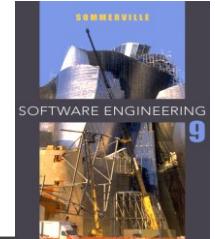
- ✧ To demonstrate to the developer and the customer that the software **meets its requirements**.
 - This means that there should be **at least one test for every requirement in the requirements document**. Also, it means that there should be tests for **all of the system features**, plus **combinations of these features**, that will be incorporated in the product release.
- ✧ To **discover situations** in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
 - Defect testing is concerned with rooting out undesirable system behavior such as **system crashes, unwanted interactions with other systems, incorrect computations and data corruption**.



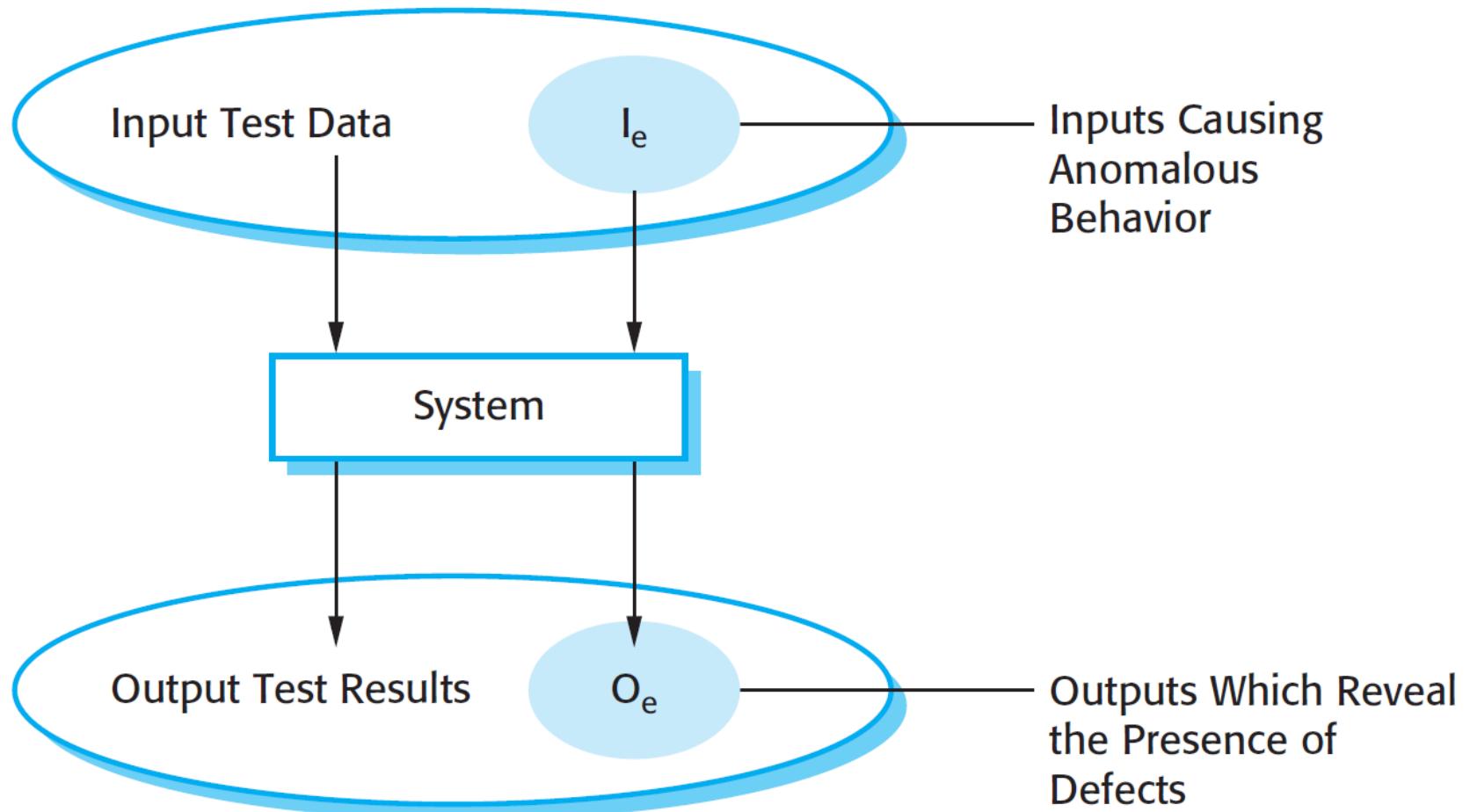
Validation and defect testing

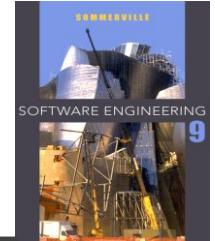
- ✧ The first goal leads to **validation testing**
 - You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.

- ✧ The second goal leads to **defect testing**
 - The test cases **are designed to expose defects**. The test cases in defect testing **can be deliberately obscure** and need not reflect how the system is normally used.



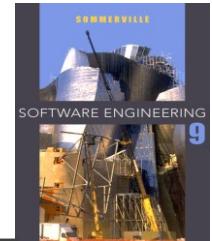
An input-output model of program testing



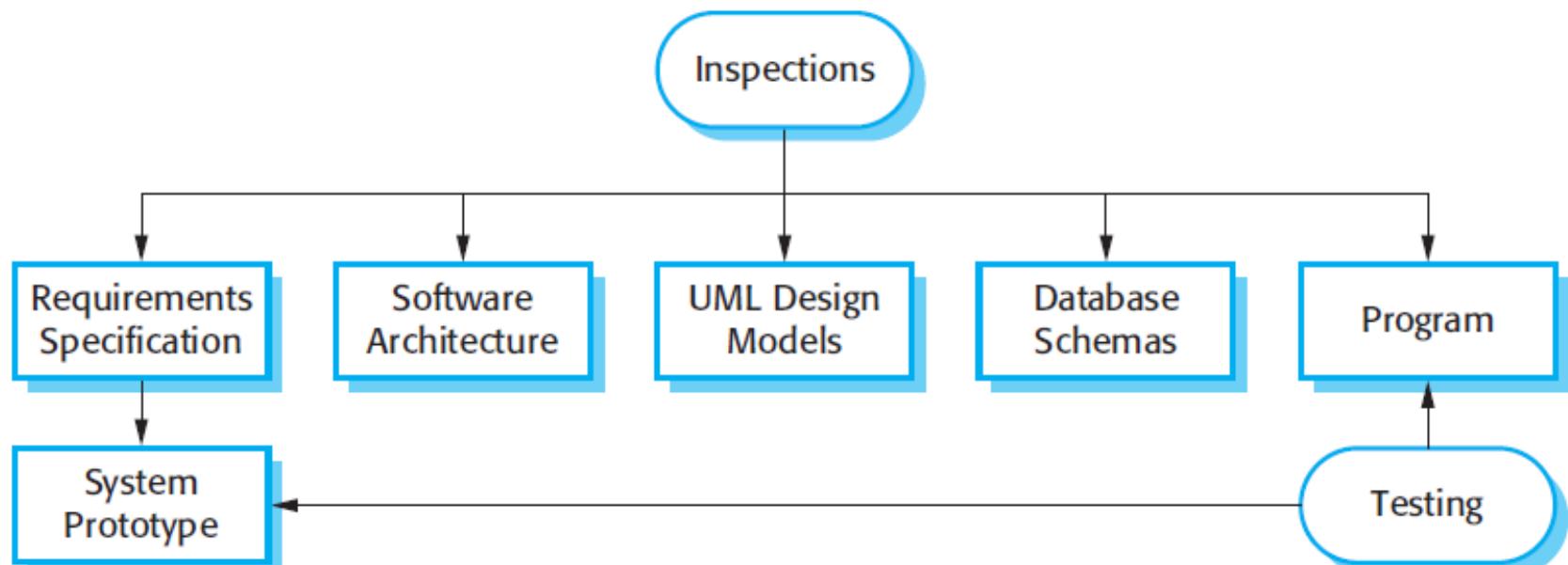


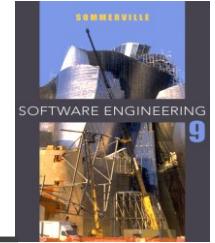
Inspections and testing

- ✧ **Software inspections** Concerned with **analysis of the static system representation** to discover problems (static verification)
 - May be supplement by tool-based document and code analysis.
 - See Chapter 15.
- ✧ **Software testing** Concerned with exercising and observing product behaviour (dynamic verification)
 - The system is **executed with test data** and its operational behaviour is observed.



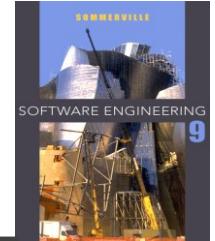
Inspections and testing





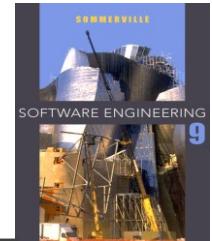
Software inspections

- ✧ These involve people **examining the source representation** with the aim of discovering anomalies and defects.
- ✧ Inspections **not require execution** of a system so may be used before implementation.
- ✧ They may be **applied to any representation** of the system (requirements, design, configuration data, test data, etc.).
- ✧ They have been shown to be an **effective technique for discovering program errors**.



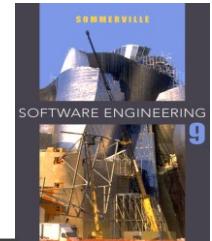
Advantages of inspections

- ✧ During testing, **errors can mask (hide) other errors.**
Because inspection is a static process, you don't have to be concerned with interactions between errors.
- ✧ **Incomplete versions** of a system **can be inspected** without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- ✧ As well as searching for program defects, an inspection can also consider **broader quality attributes** of a program, such as compliance with standards, portability and maintainability.

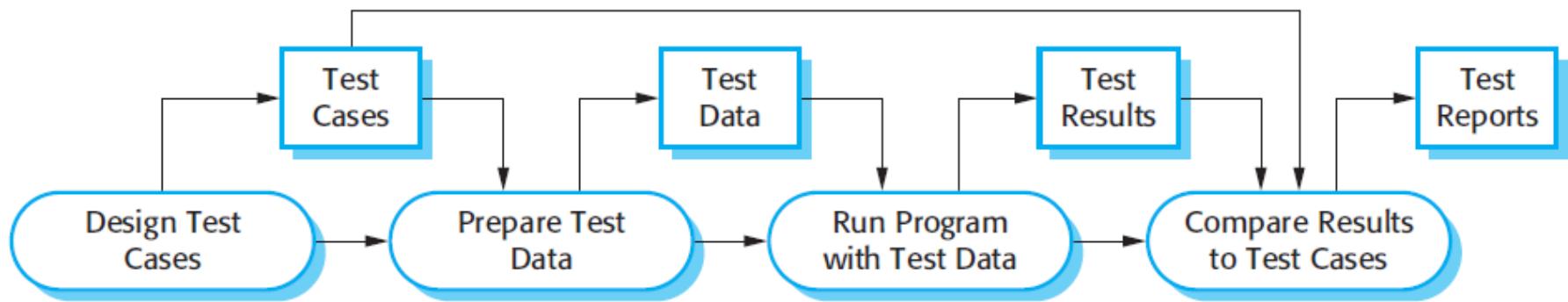


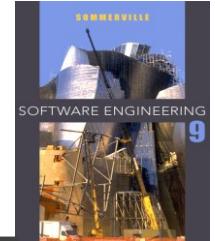
Inspections and testing

- ✧ Inspections and testing are complementary and not opposing verification techniques.
- ✧ Both should be used during the Verification & Validation process.
- ✧ Inspections may not be appropriate to check non-functional characteristics such as performance, usability, etc.



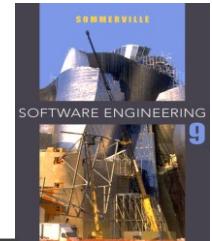
A model of the software testing process





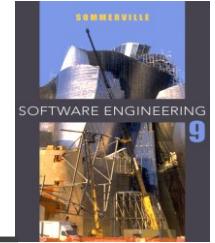
Stages of testing (typical commercial software)

- ✧ **Development testing**, where the system is tested during development to discover bugs and defects. (mostly done by the development team)
- ✧ **Release testing**, where a separate testing team tests a complete version of the system before it is released to users. (checking to meet requirements)
- ✧ **User testing**, where users or potential users of a system test the system in their own environment. (Acceptance testing is one type of user testing)



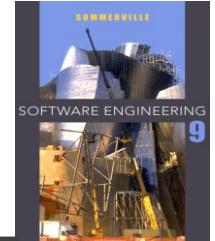
Development testing

- ✧ Development testing includes all testing activities that are carried out by the **team developing** the system.
 - **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the **functionality of objects or methods**.
 - **Component testing**, where several individual units are integrated to create composite components. Component testing should focus on testing **component interfaces**.
 - **System testing**, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing **component interactions**.



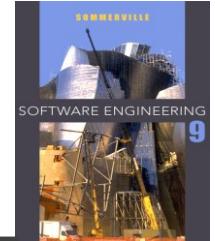
Unit testing

- ✧ Unit testing is the process of **testing individual part of code in isolation**. It is a defect testing process.
- ✧ Units may be:
 - Individual functions or methods
 - Object classes with several attributes and methods
- ✧ Complete test **coverage of a class** involves
 - Testing **all operations** associated with an object
 - **Setting and interrogating** all object attributes
 - Exercising the object in **all possible states**.
- ✧ Inheritance makes it more difficult to design object class tests as the **information to be tested is not localised**.



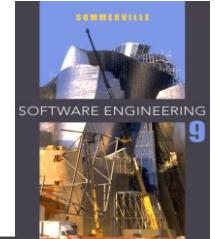
Automated testing

- ✧ Whenever possible, unit testing **should be automated** so that tests are run and checked without manual intervention.
- ✧ In automated unit testing, you make use of a **test automation framework** (such as JUnit) to write and run your program tests.
- ✧ Unit **testing frameworks** may provide generic test classes that you extend to create specific test cases. They can then **run all of the tests** that you have implemented and **report**, often through some GUI, on the success or otherwise of the tests.



Automated test components

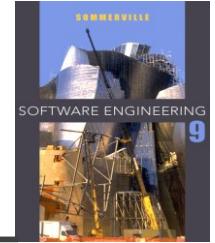
- ✧ A **setup part**, where you initialize the system with the test case, namely the inputs and expected outputs.
- ✧ A **call part**, where you call the object or method to be tested.
- ✧ An **assertion part**, where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.



Sample test in JUnit

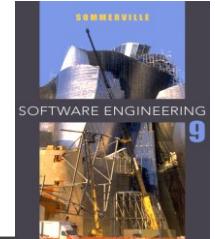
```
public class MyTests {  
  
    @Test  
    public void multiplicationOfZeroIntegersShouldReturnZero() {  
        Calculator calc = new Calculator(); // Class that is being tested  
  
        // assert statements  
        assertEquals(0, calc.multiply(10, 0), "10 x 0 must be 0");  
        assertEquals(0, calc.multiply(0, 10), "0 x 10 must be 0");  
        assertEquals(0, calc.multiply(0, 0), "0 x 0 must be 0");  
    }  
}
```

zero is our expected value



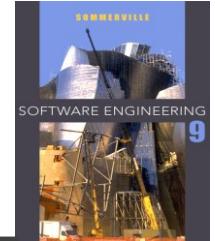
Unit test effectiveness

- ✧ The test cases should show that, when used as expected, the component that you are testing **does what it is supposed to do**.
- ✧ If there are **defects in the component**, these should be revealed by test cases.
- ✧ This leads to 2 types of unit test case:
 - The first of these should **reflect normal operation** of a program and should show that the **component works as expected**.
 - The other kind of test case should be based on testing experience of where common problems arise. It should use **abnormal inputs** to check that these are **properly processed** and do not crash the component.



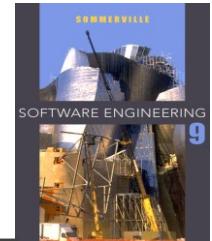
Testing strategies

- ✧ **Partition testing**, where you identify groups of inputs that have common characteristics and should be processed in the same way.
 - You should choose tests from within each of these groups.
- ✧ **Guideline-based testing**, where you use testing guidelines to choose test cases.
 - These guidelines reflect **previous experience of the kinds of errors that programmers often make** when developing components.

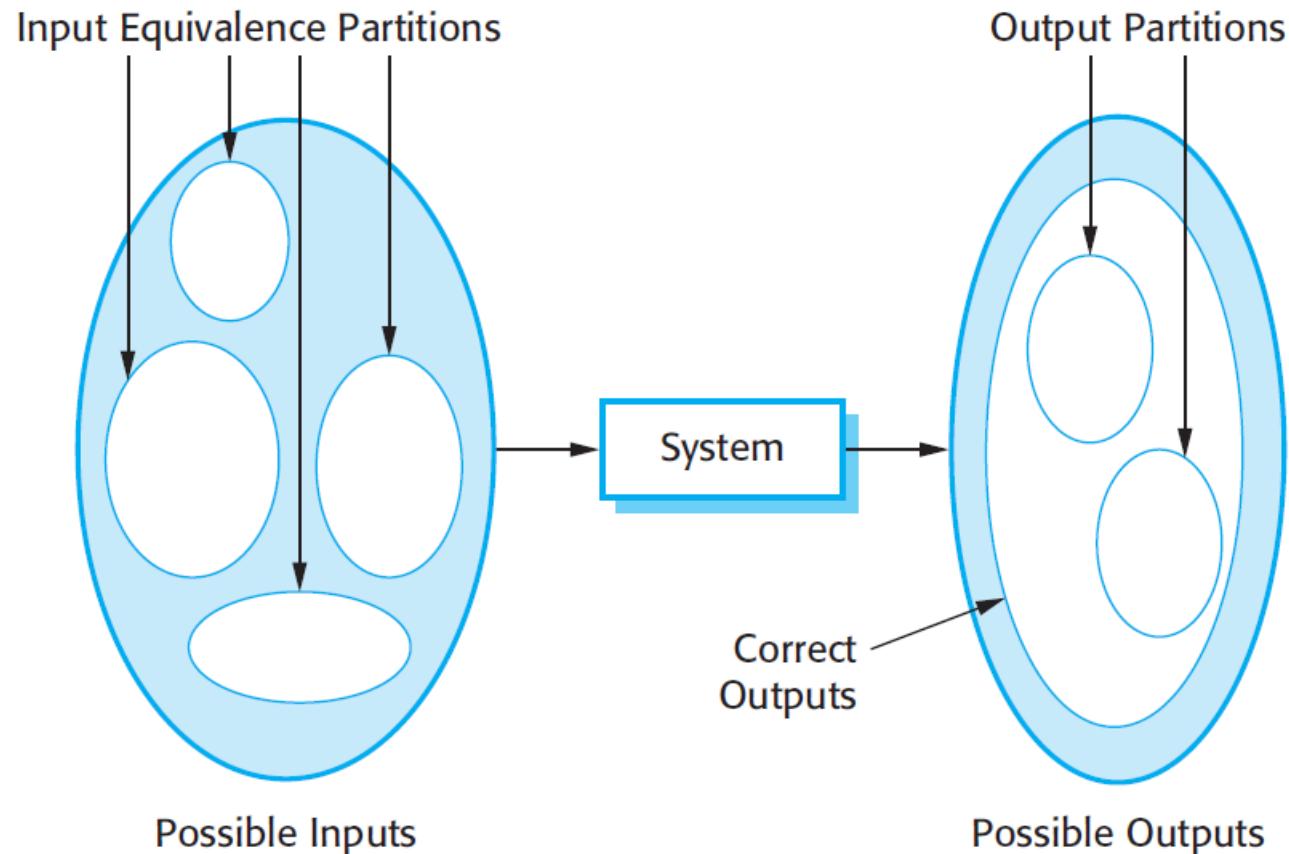


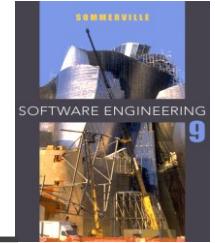
Partition testing

- ✧ Input data results often fall into different classes where all members of a class are related and result in similar output.
- ✧ Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- ✧ Test cases should be chosen from each partition.



Equivalence partitioning

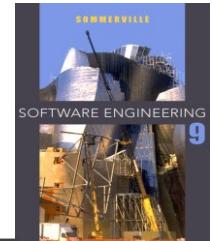




Equivalence partitions

✧ Exercise:

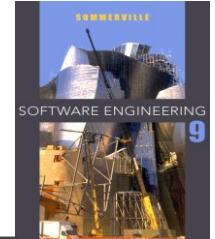
- identify equivalence partitions for a program that takes int value representing a month and returns the name of the month. 12 partitions
- identify equivalence partitions for a program that takes int value representing a month and returns the name of the season the month belongs to. 4 partitions
- Height categorizer! 3 valid partitions, small, medium, tall
if we separate based on country, we have $3 * \#country$ partitions



Boundary testing

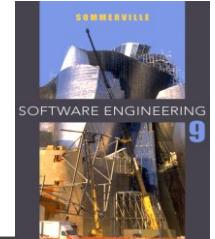
```
public char gradeRank(double studentGrade) {  
    char grade;  
    if (studentGrade >= 90) {  
        grade = 'A';  
    } else if (studentGrade >= 80) {  
        grade = 'B';  
    } else if (studentGrade >= 70) {  
        grade = 'B';  
    } else if (studentGrade >= 60) {  
        grade = 'D';  
    } else {  
        grade = 'F';  
    }  
    return grade;  
}
```

5 partitions here



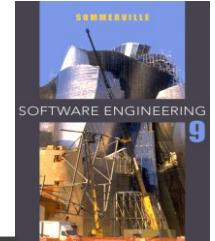
Testing guidelines (sequences, arrays, lists)

- ✧ Test software with sequences which have only a **single value**.
- ✧ Use sequences of **different sizes in different tests**.
- ✧ Derive tests so that the **first, middle and last elements** of the sequence are accessed. (partition boundaries)
- ✧ Test with sequences of zero length.



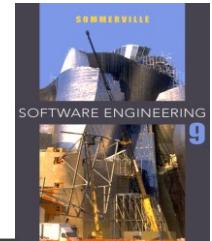
General testing guidelines (examples)

- ✧ Choose inputs that force the system to **generate all error messages**
- ✧ Design inputs that cause input **buffers to overflow**
- ✧ Repeat the same input or series of **inputs numerous times**
- ✧ Force computation results to be **too large or too small**

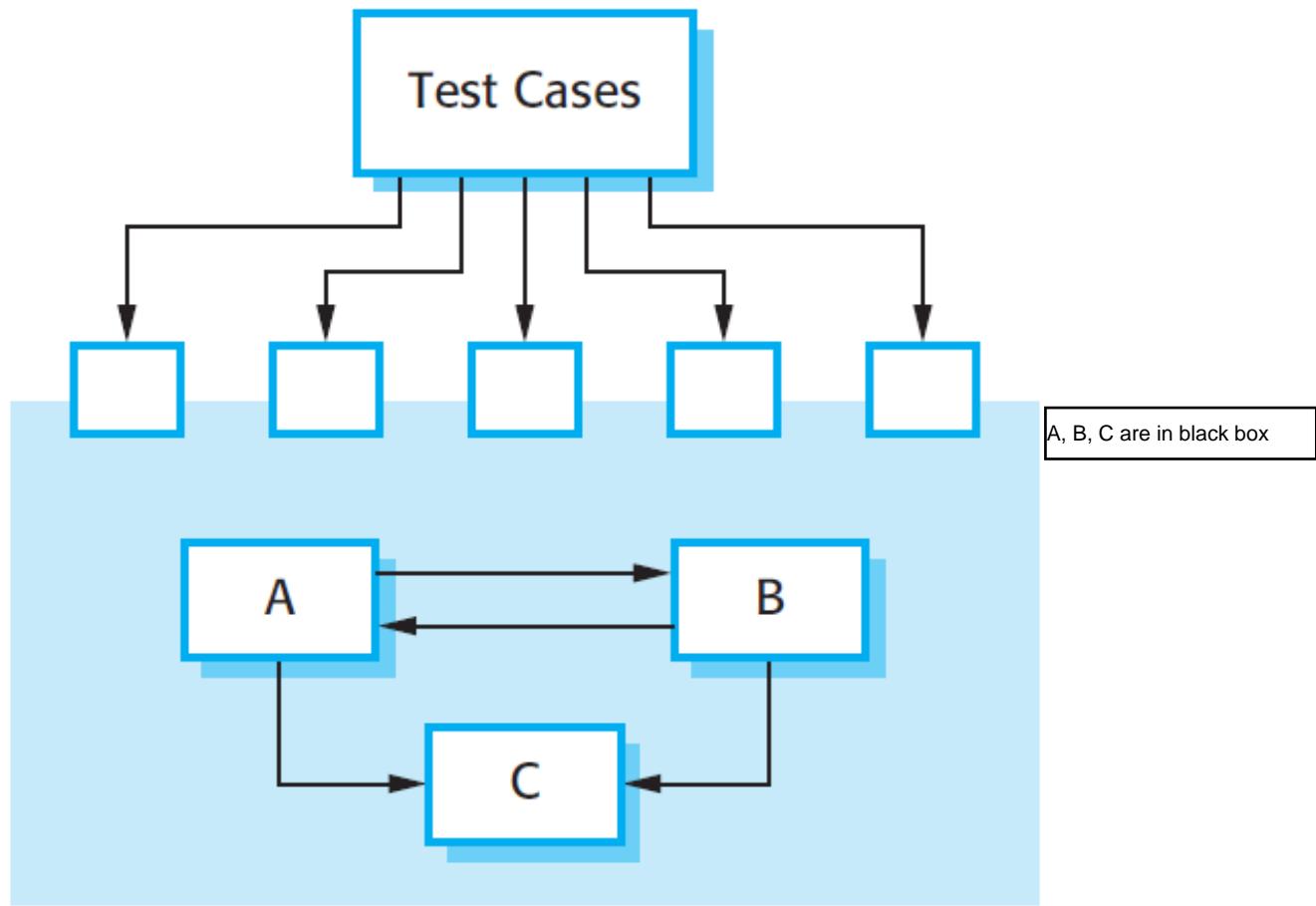


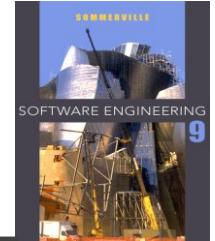
Component testing

- ✧ Software components are often **composite components** that are made up of **several interacting objects**.
- ✧ You **access the functionality** of these objects through the defined component **interface**.
- ✧ Testing composite components should therefore **focus on showing that the component interface** behaves according to its specification. (black box testing)
 - You can assume that unit tests on the individual objects within the component have been completed.



Interface testing





Interface errors

✧ Interface misuse

- A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.

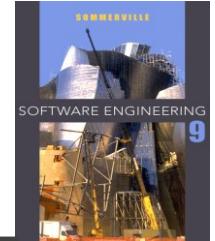
✧ Interface misunderstanding

- A calling component embeds assumptions about the behaviour of the called component which are incorrect.

✧ Timing errors

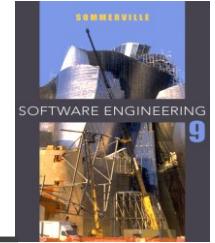
- The called and the calling component operate at different speeds and out-of-date information is accessed.

for example in client server architecture, before server runs



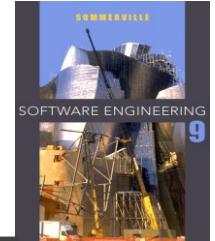
Interface testing guidelines

- ✧ Design tests so that parameters to a called procedure are at the **extreme ends of their ranges**.
- ✧ Always test **pointer** parameters with **null** pointers.
- ✧ Design tests which cause the **component to fail**.
- ✧ Use **stress testing** in message passing systems.
- ✧ **Vary the order** in which components are activated.



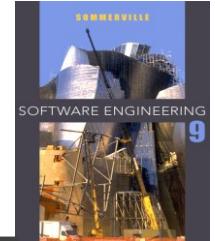
System testing

- ✧ System testing during development involves integrating components to create a **version of the system** and then **testing the integrated system**.
- ✧ The focus in system testing is testing the **interactions between components**.
- ✧ System testing checks that **components are compatible**, interact correctly and transfer the right data at the right time across their interfaces.



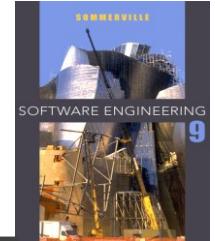
System and component testing

- ✧ During system testing, **reusable components that have been separately developed and off-the-shelf systems** may be integrated with newly developed components. The complete system is then tested.
- ✧ Components **developed by different team members** or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
 - In some companies, system testing **may involve a separate testing team** with no involvement from designers and programmers.



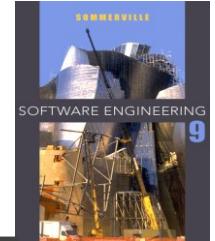
Use-case testing

- ✧ The use-cases developed to identify system interactions can be used as a **basis for system testing**.
- ✧ Each use case usually involves **several system components** so testing the use case **forces these interactions** to occur.
- ✧ The **sequence diagrams** associated with the use case documents the components and interactions that are being tested.



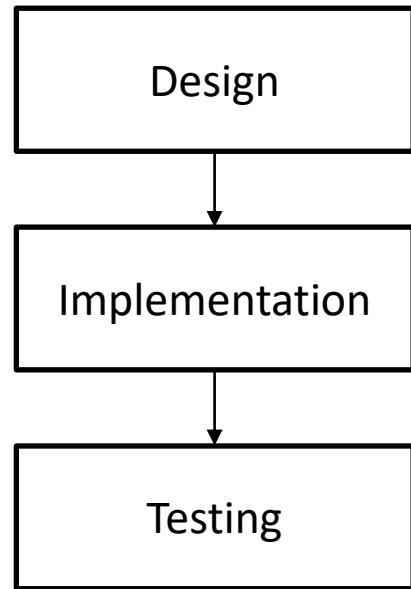
Test-driven development

- ✧ Test-driven development (TDD) is an approach to program development in which you **inter-leave testing and code development**.
- ✧ Tests are written **before code** and ‘passing’ the tests is the critical driver of development.
- ✧ You **develop code incrementally**, along **with a test** for that increment. You don’t move on to the next increment until the code that you have developed passes its test.
- ✧ TDD was introduced as **part of agile** methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

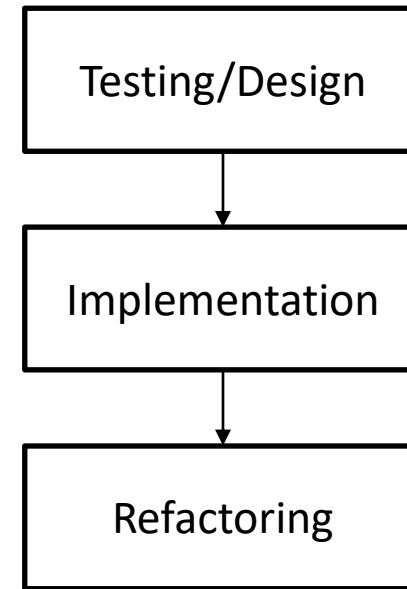


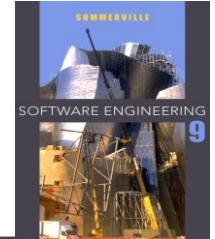
Test-driven development

Traditional Approach

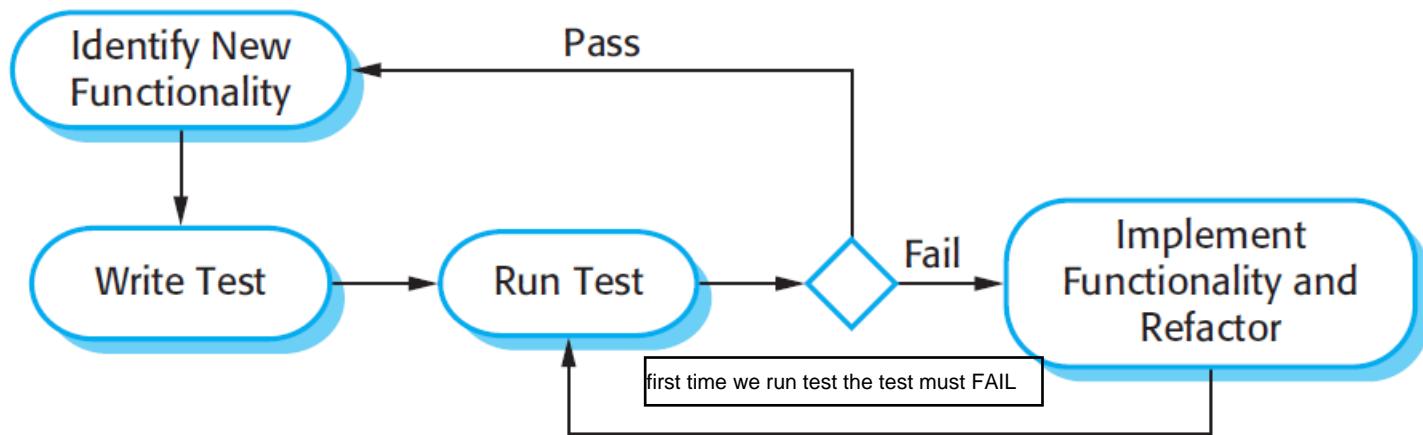


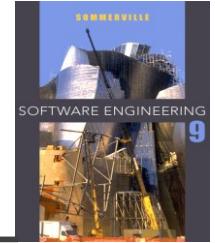
TDD Approach





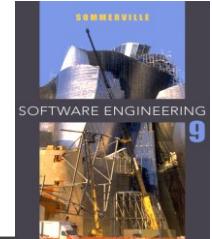
Test-driven development





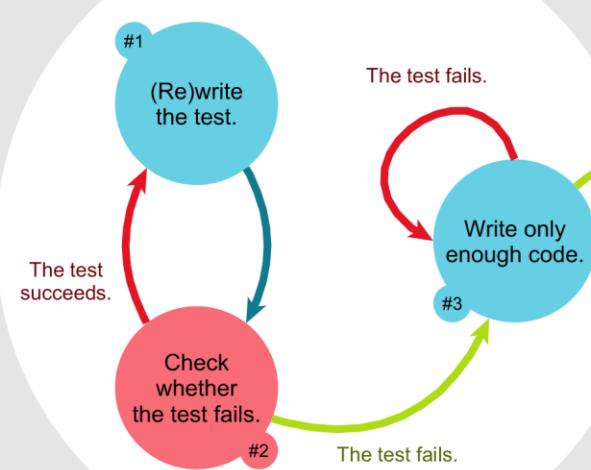
TDD process activities

- ✧ Start by **identifying the increment of functionality** that is required. This should normally be small and implementable in a few lines of code.
- ✧ Write a **test for this functionality** and implement this as an automated test.
- ✧ Run the test, along with all other tests that have been implemented. **Initially, you have not implemented the functionality so the new test will fail.**
- ✧ Implement the functionality and re-run the test.
- ✧ Once all tests run successfully, you move on to implementing the next chunk of functionality.

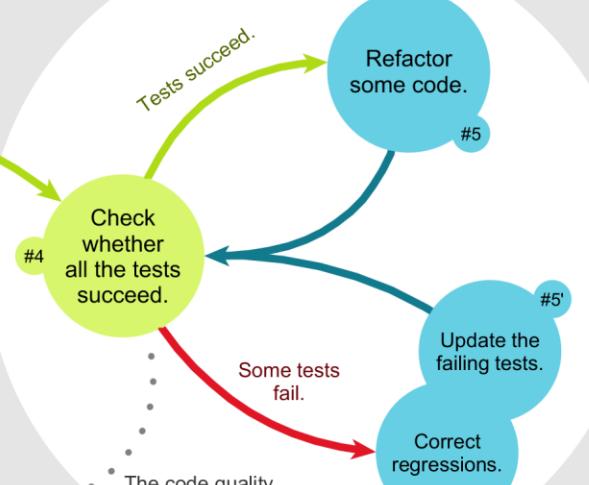


Test-driven development

CODE-DRIVEN TESTING



REFACTORING



Iterate

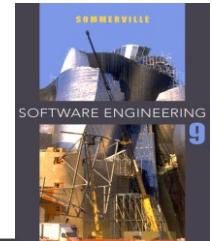
focus
Completion of the contract
as defined by the test

focus
Alignment of the design
with known needs

TEST-DRIVEN DEVELOPMENT



Xavier Pigeon



Benefits of test-driven development

✧ Code coverage

- Every code segment that you write has **at least one associated test** so all code written has at least one test.

✧ Regression testing

we run all the tests

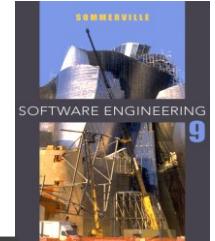
- A regression test suite is developed incrementally as a program is developed.

✧ Simplified debugging

- When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.

✧ System documentation

- The tests themselves are a form of documentation that describe what the code should be doing.

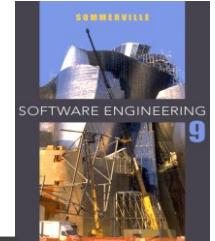


Regression testing

- ✧ Regression testing is testing the system to check that changes **have not ‘broken’ previously working code.**

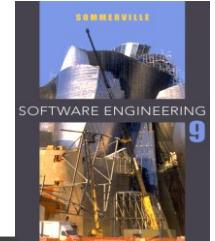
- ✧ In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are **rerun every time a change is made** to the program.

- ✧ Tests must run ‘successfully’ before the change is committed.



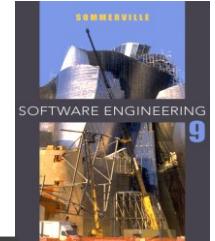
Release testing

- ✧ Release testing is the process of testing a particular release of a system that is **intended for use outside of the development team**.
- ✧ The primary goal of the release testing process is to **convince the supplier** of the system that it is **good enough for use**.
 - Release testing, therefore, has to show that the system **delivers its specified functionality, performance and dependability**, and that it **does not fail during normal use**.
- ✧ Release testing is usually a **black-box** testing process where tests are only **derived from the system specification**.



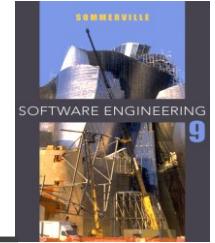
Release testing and system testing

- ✧ Release testing is a form of system testing.
- ✧ Important differences:
 - A **separate team** that **has not** been involved in the system development, should be responsible for release testing.
 - System testing by the **development team** should focus on **discovering bugs** in the system (**defect testing**). The objective of release testing is to check that the **system meets its requirements** and is good enough for external use (**validation testing**).



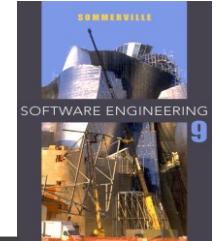
Requirements based testing

- ✧ Requirements-based testing involves **examining each requirement** and developing a test or tests for it.
- ✧ Mencare requirements:
 - If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
 - If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.



Requirements tests

- ✧ Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
- ✧ Set up a patient record with a known allergy. Prescribe the medication that the patient is allergic to, and check that the warning is issued by the system.
- ✧ Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
- ✧ Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
- ✧ Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.



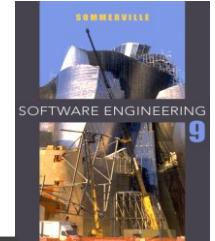
A usage scenario for the Mentcare

Kate is a nurse who specializes in mental health care. One of her responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side -effects.

On a day for home visits, Kate logs into the Mentcare and uses it to print her schedule of home visits for that day, along with summary information about the patients to be visited. She requests that the records for these patients be downloaded to her laptop. She is prompted for her key phrase to encrypt the records on the laptop.

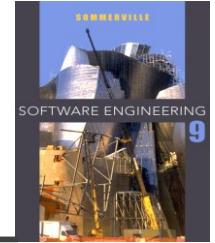
One of the patients that she visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side -effect of keeping him awake at night. Kate looks up Jim's record and is prompted for her key phrase to decrypt the record. She checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so she notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. He agrees so Kate enters a prompt to call him when she gets back to the clinic to make an appointment with a physician. She ends the consultation and the system re-encrypts Jim's record.

After, finishing her consultations, Kate returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for Kate of those patients who she has to contact for follow-up information and make clinic appointments.



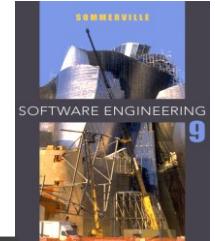
Features tested by scenario

- ✧ Authentication by logging on to the system.
- ✧ Downloading and uploading of specified patient records to a laptop.
- ✧ Home visit scheduling.
- ✧ Encryption and decryption of patient records on a mobile device.
- ✧ Record retrieval and modification.
- ✧ Links with the drugs database that maintains side-effect information.
- ✧ The system for call prompting.



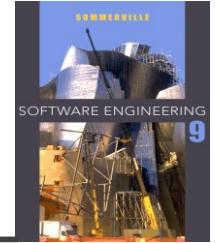
Performance testing

- ✧ Part of **release testing** may involve testing the emergent properties of a system, such as performance and reliability.
- ✧ Tests should **reflect the profile of use of the system**.
- ✧ Performance tests usually involve planning a series of tests where the **load is steadily increased** until the system performance becomes unacceptable.
- ✧ Stress testing is a form of performance testing where the system is **deliberately overloaded** to test its failure behavior.



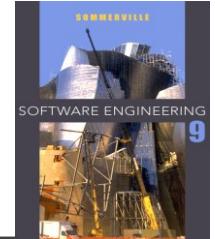
User testing

- ✧ User or customer testing is a stage in the testing process in which **users or customers provide input and advice** on system testing.
- ✧ User testing is essential, even when comprehensive system and release testing have been carried out.
 - The reason for this is that **influences from the user's working environment** have a major effect on the reliability, performance, usability and robustness of a system. These **cannot be replicated in a testing environment**.



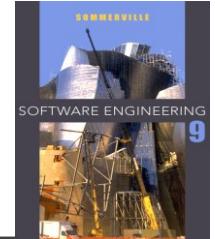
Types of user testing

- ✧ Alpha testing the system is not complete
 - Users of the software **work with the development team** to test the software **at the developer's site**.
- ✧ Beta testing system is almost complete but may have bugs
 - A release of the software is **made available to users** to allow them to experiment and to **raise problems that they discover** with the system developers.
- ✧ Acceptance testing
 - Customers test a system to decide **whether or not it is ready to be accepted from the system developers** and deployed in the customer environment. Primarily for custom systems.

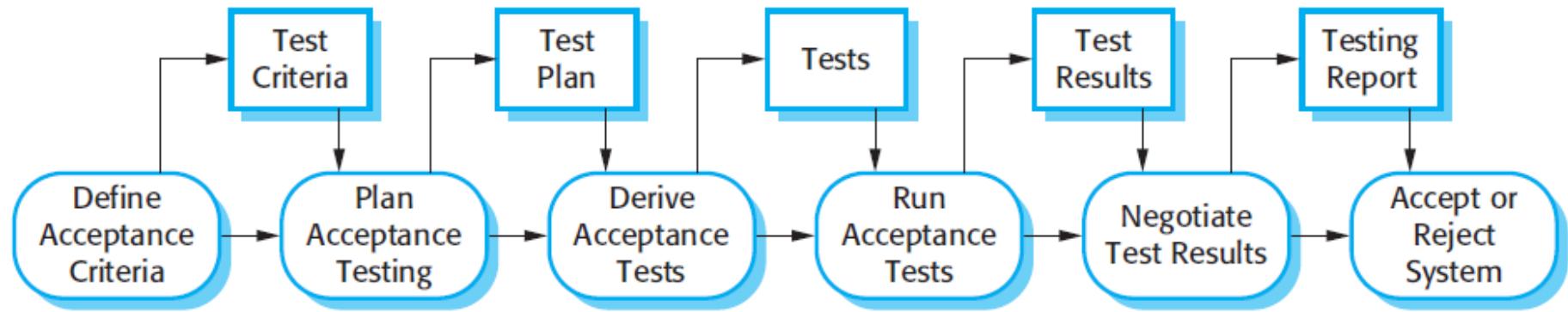


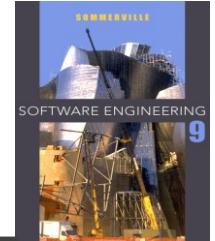
Stages in the acceptance testing process

- ✧ Define acceptance criteria
- ✧ Plan acceptance testing
- ✧ Derive acceptance tests
- ✧ Run acceptance tests
- ✧ Negotiate test results
- ✧ Reject/accept system



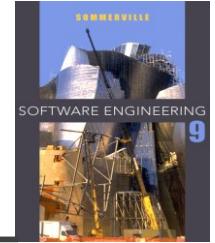
The acceptance testing process





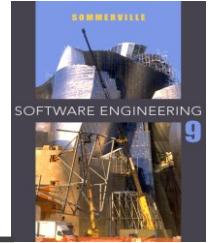
Agile methods and acceptance testing

- ✧ In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- ✧ Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- ✧ There is no separate acceptance testing process.
- ✧ Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.

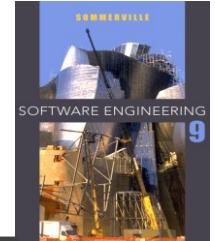


Key points

- ✧ When testing software, you should **try to 'break'** the software by using experience and guidelines to choose types of test case that have been effective in **discovering defects** in other systems.
- ✧ Wherever possible, you should **write automated tests**. The tests are embedded in a program that can be run every time a change is made to a system.
- ✧ Test-driven development is an **approach to development** where **tests are written before the code** to be tested.
- ✧ Scenario testing involves inventing a typical usage scenario and using this to derive test cases.
- ✧ Acceptance testing is a user testing process where the aim is to decide if the **software is good enough to be deployed** and used in its operational environment.



Chapter 9 – Software Evolution

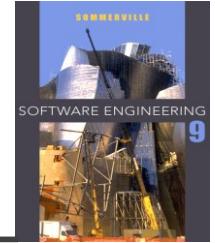


Software change

✧ Software change is **inevitable**

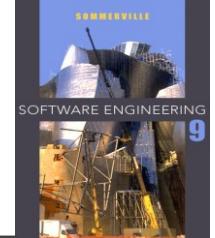
- New requirements emerge when the software is used;
- The business environment changes;
- Errors must be repaired;
- New computers and equipment is added to the system;
- The performance or reliability of the system may have to be improved.

✧ A key problem for all organizations is **implementing and managing change** to their existing software systems.

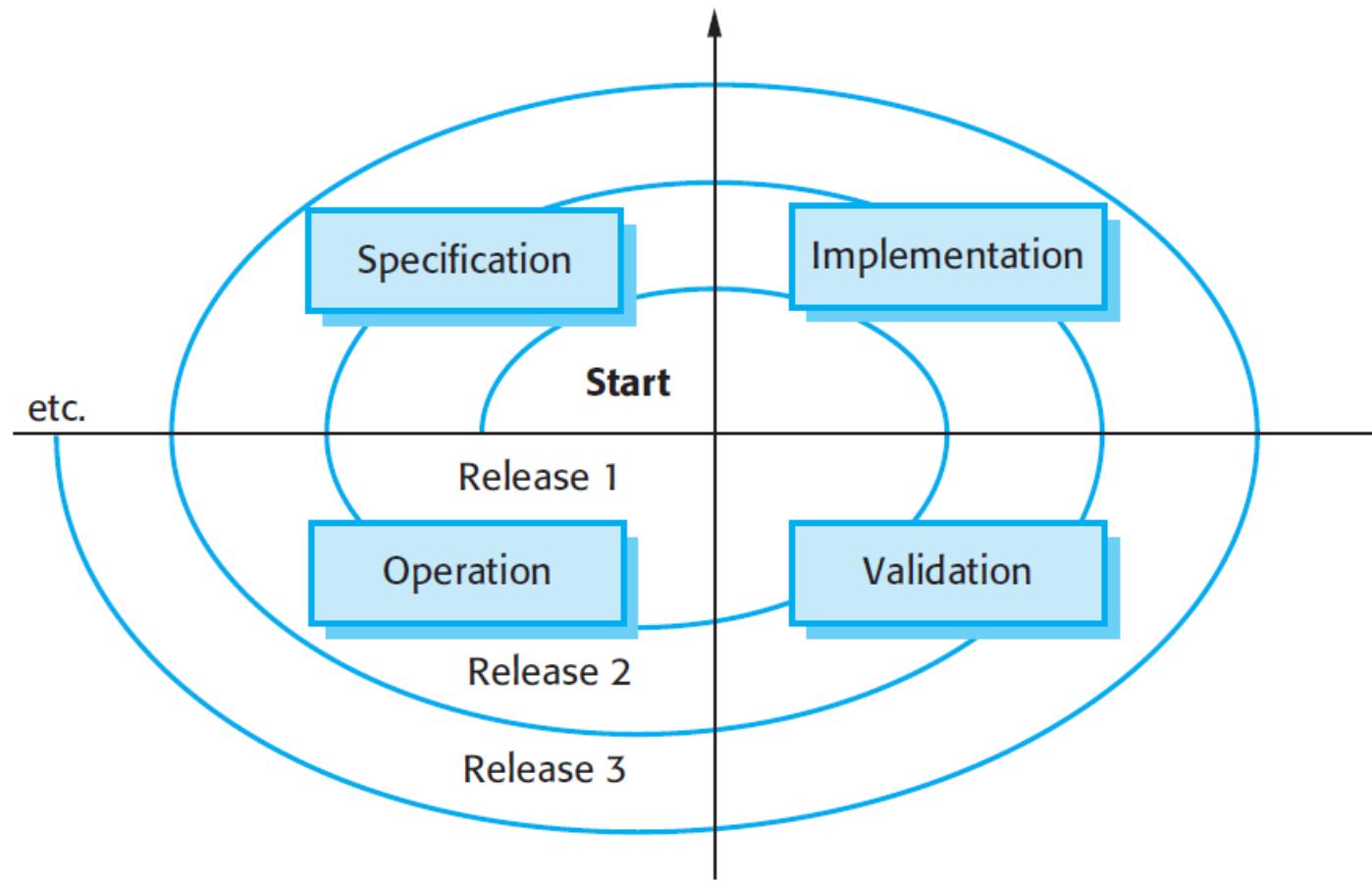


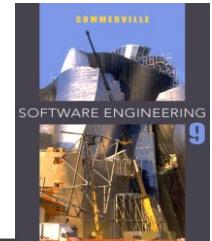
Importance of evolution

- ✧ Organizations have huge investments in their software systems - they are critical business assets.
- ✧ To **maintain** the value of these assets to the business, they **must be changed and updated**.
- ✧ The **majority of the software budget** in large companies is devoted to changing and evolving existing software rather than developing new software.

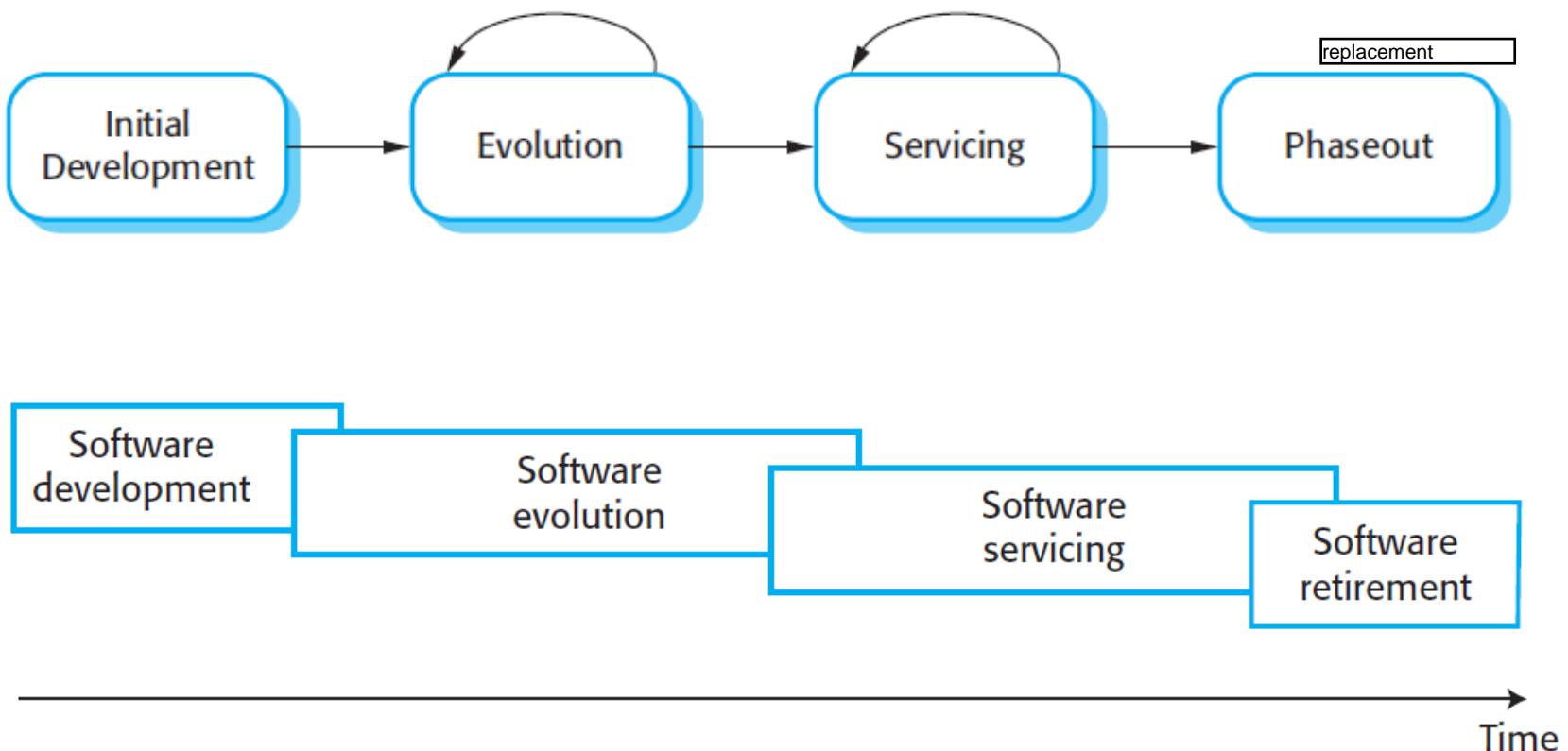


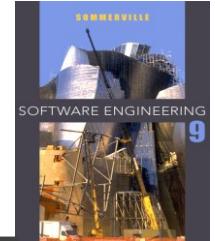
A spiral model of development and evolution





Evolution and servicing





Evolution and servicing

✧ Evolution

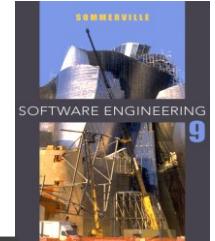
- The stage in a software system's life cycle where it is in operational use and **is evolving as new requirements are proposed** and implemented in the system.

✧ Servicing

- At this stage, the software remains useful but the only changes made are those **required to keep it operational** i.e. bug fixes and changes to reflect changes in the software's environment. **No new functionality is added.**

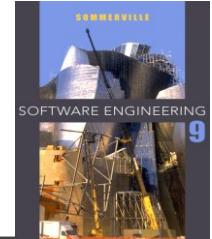
✧ Phase-out

- The software may still be used but **no further changes are made to it.**

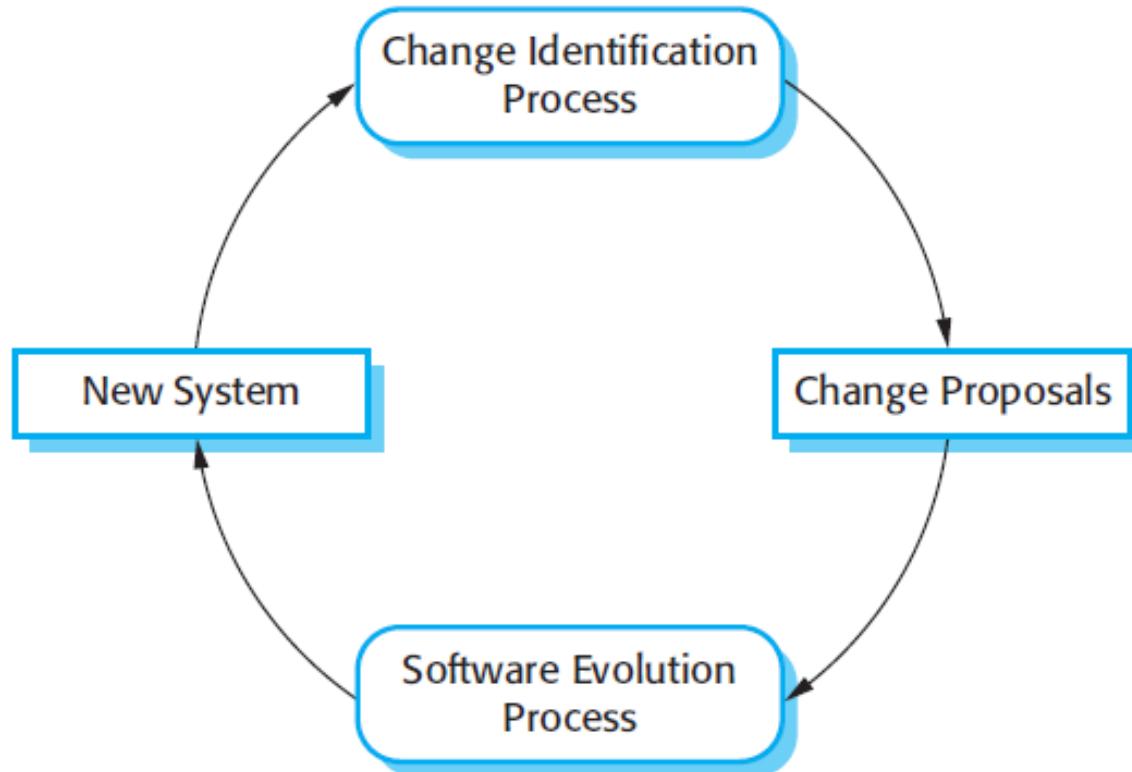


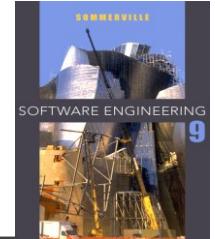
Evolution processes

- ✧ Proposals for change are the driver for system evolution.
 - Should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.
- ✧ Change identification and evolution continues throughout the system lifetime.

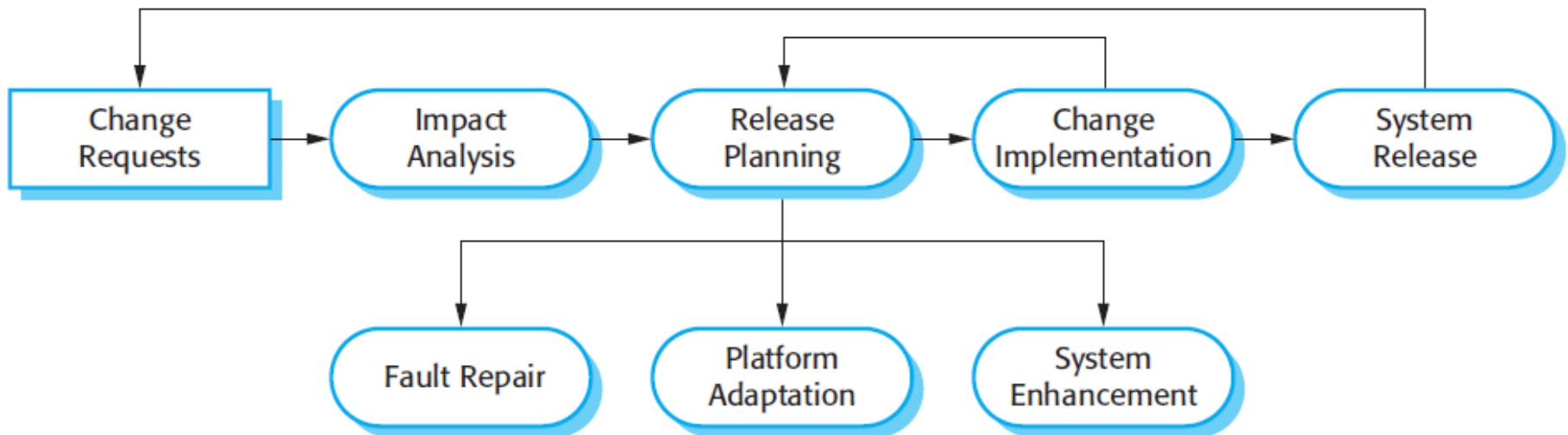


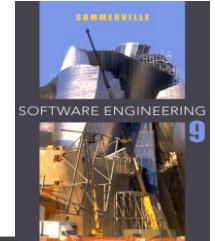
Change identification and evolution processes





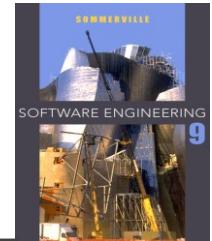
The software evolution process





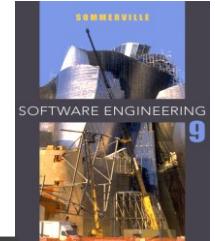
Change implementation

- ✧ Iteration of the development process where the revisions to the system are designed, implemented and tested.
- ✧ A critical difference is that the first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for the change implementation.
- ✧ During the program understanding phase, you have to understand how the program is structured, how it delivers functionality and how the proposed change might affect the program.

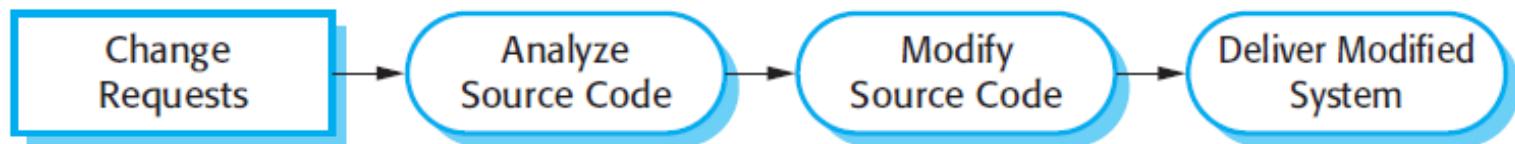


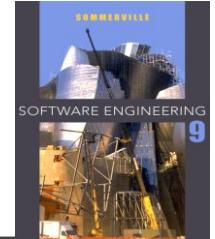
Urgent change requests

- ✧ Urgent changes **may have to be implemented without going through all stages** of the software engineering process
 - If a **serious system fault has to be repaired** to allow normal operation to continue;
 - If **changes to the system's environment** (e.g. an OS upgrade) have unexpected effects;
 - If there are **business changes that require a very rapid response** (e.g. the release of a competing product).



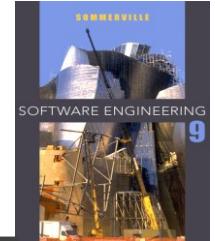
The emergency repair process





Agile methods and evolution

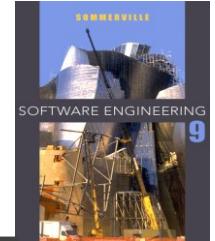
- ✧ Agile methods are based on **incremental development** so the transition from development to evolution is a seamless one.
 - Evolution is simply a continuation of the development process **based on frequent system releases**.
- ✧ Automated regression testing is particularly valuable when changes are made to a system.
- ✧ Changes may be expressed as **additional user stories**.



Handover problems

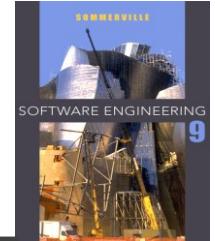
delivery

- ✧ Where the **development team** have used an agile approach but the **evolution team** is unfamiliar with agile methods and prefer a plan-based approach.
 - The evolution team may expect detailed documentation to support evolution and this is not produced in agile processes.
- ✧ Where a **plan-based approach** has been used for development but the **evolution team** prefer to use agile methods.
 - The evolution team may have to start from scratch developing automated tests and the code in the system may not have been refactored and simplified as is expected in agile development.



Software maintenance

- ✧ Modifying a program after it has been put into use.
- ✧ The term is mostly used for changing **custom software**.
Generic software products are said to **evolve** to create new versions.
- ✧ Maintenance does not normally involve major changes to the system's architecture.
because they are costly
- ✧ Changes are implemented by modifying existing components and adding new components to the system.



Types of maintenance

- ✧ Maintenance **to repair** software faults
 - Changing a system to correct deficiencies to meet its requirements.
- ✧ Maintenance **to adapt** software to a different operating environment
 - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- ✧ Maintenance **to add to or modify** the system's functionality
 - Modifying the system to satisfy new requirements.

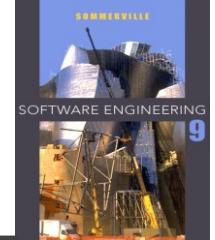
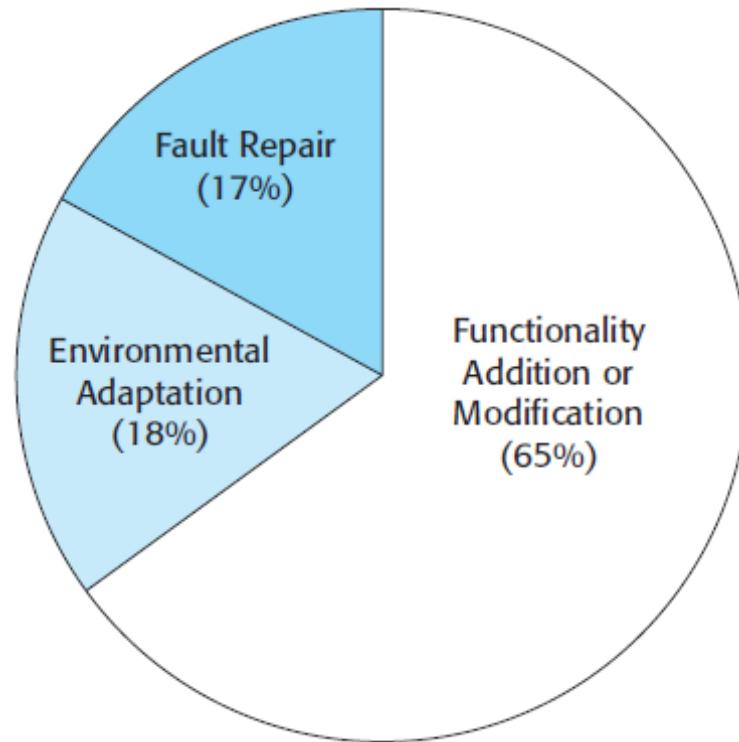
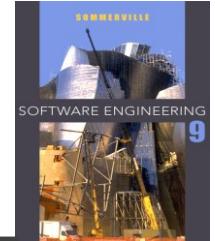


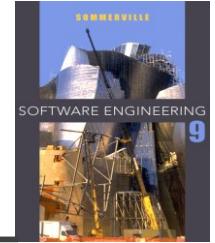
Figure 9.8 Maintenance effort distribution





Maintenance costs

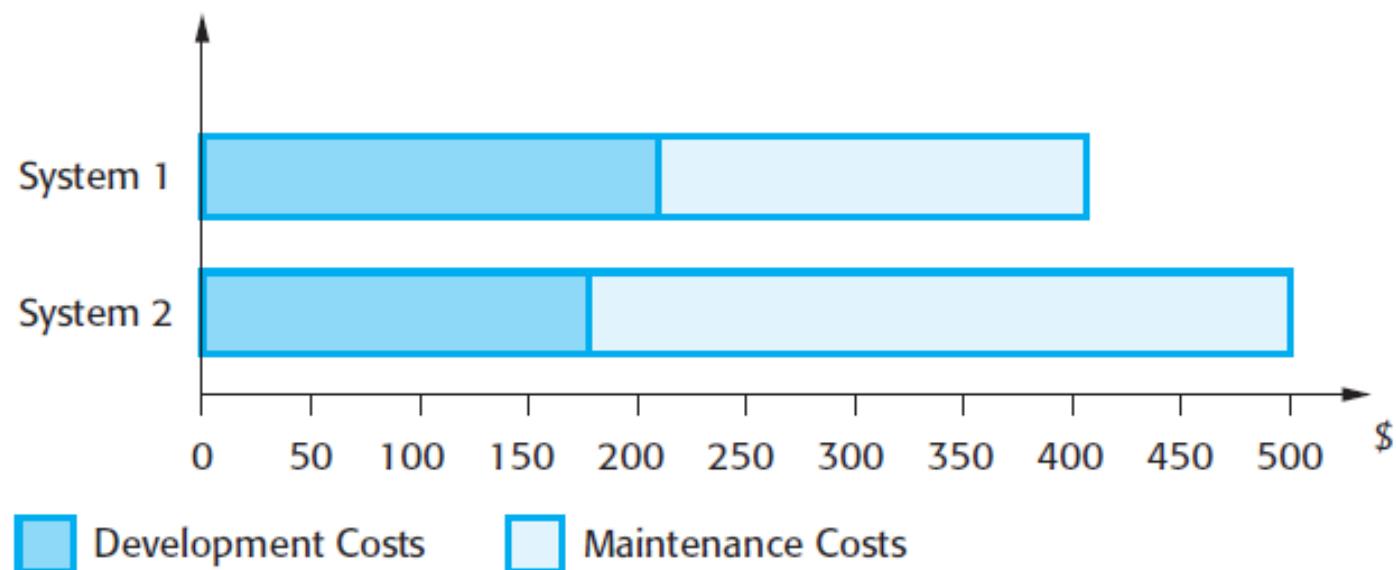
- ✧ Usually greater than development costs (2* to 100* depending on the application).
- ✧ Affected by both technical and non-technical factors.
- ✧ Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.
- ✧ Ageing software can have high support costs (e.g. old languages, compilers etc.).

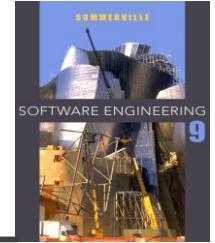


Maintenance cost factors

- ✧ Team stability
 - Maintenance costs are reduced if the same staff are involved with them for some time.
- ✧ Poor development practice
 - The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.
- ✧ Staff skills
 - Maintenance staff are often inexperienced and have limited domain knowledge.
- ✧ Program age and structure
 - As programs age, their structure is degraded and they become harder to understand and change.

Figure 9.9 Development and maintenance costs





'Bad smells' in program code

✧ Duplicate code

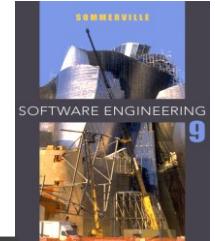
- The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.

✧ Long methods

- If a method is too long, it should be redesigned as a number of shorter methods.

✧ Switch (case) statements

- These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.



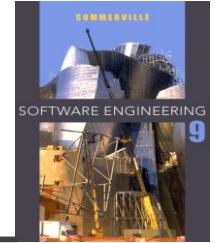
'Bad smells' in program code

✧ Data clumping

- Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. These can often be replaced with an object that encapsulates all of the data.

✧ Speculative generality

- This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.



Key points

- ✧ Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- ✧ For custom systems, the costs of software maintenance usually exceed the software development costs.
- ✧ The process of software evolution is driven by requests for changes and includes change impact analysis, release planning and change implementation.