

Въведение

Понятие за софтуерни технологии. Продукти и услуги.

Съдържание

- ▶ Въведение в курса
 - ▶ Понятие за софтуерни технологии/софтуерно инженерство.
- ▶ Понятие за софтуер.
- ▶ Продукти и услуги.



Какво означава софтуерни технологии?

- ▶ А софтуерно инженерство?
- ▶ А компютърни науки?
- ▶ Защо този курс е в програмата на специалността?

Софтуерен инженер vs. програмист

► Soft skills

- ▶ Communication
- ▶ Multitasking
- ▶ Organization
- ▶ Attention to detail

► Hard skills

- ▶ Programming languages
- ▶ Software testing
- ▶ Data structures
- ▶ Computer science



Видове професии

- ▶ Information Security Analyst
- ▶ Computer Programmer
- ▶ Data Scientist
- ▶ Systems Analyst
- ▶ Video game designer
- ▶ SOA engineer
- ▶ Cyber security engineer
- ▶ Application security engineer
- ▶ Software project manager
- ▶ Software test engineer
- ▶ Full-stack engineer
- ▶ Software developer
- ▶ DevOps engineer
- ▶ Software architect
- ▶ Machine learning engineer
- ▶ Etc...

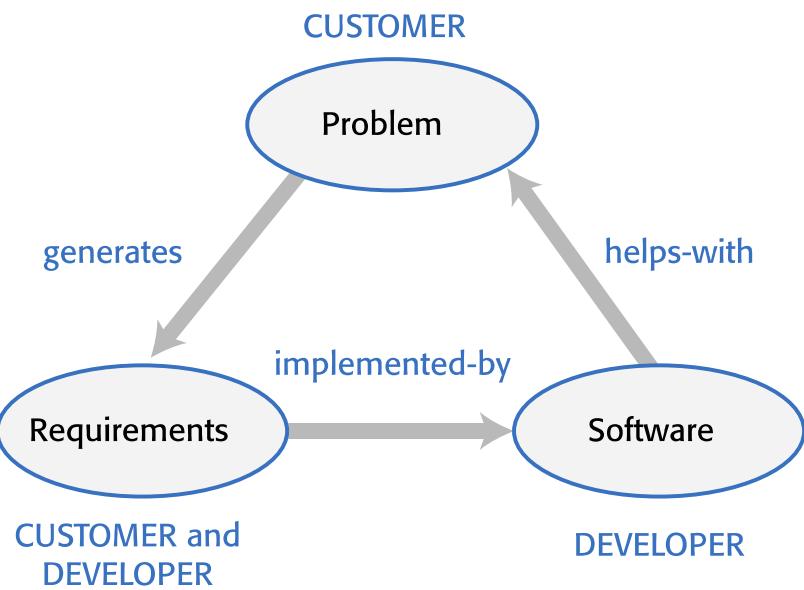


Предизвикателства пред разработката на софтуер

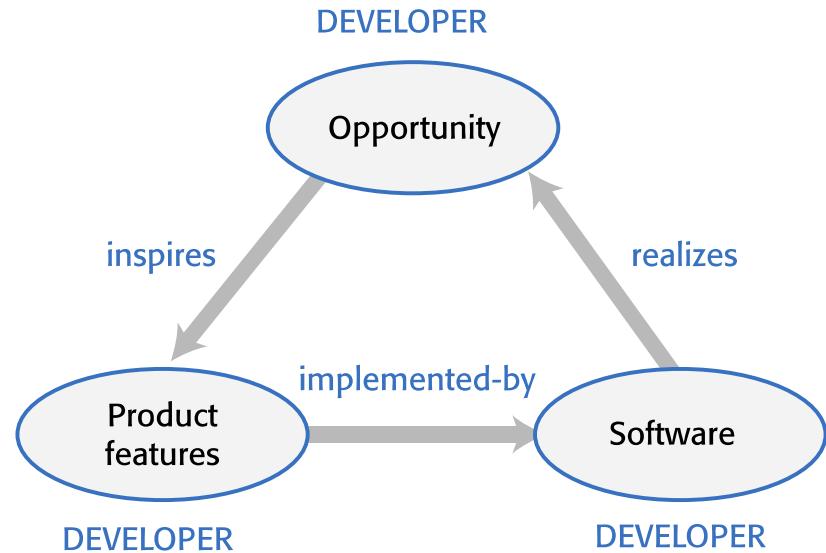
- ▶ Как да разберем какво точно искаме да разработим?
- ▶ Как да осигурим качеството на софтуера, който разработваме?
 - ▶ Как да избегнем големи закъснения във времето
- ▶ Как да удовлетворим нарастващото търсене, като продължим да управляваме бюджета?
 - ▶ Как бързо и лесно да включваме нови хора в екипа?
- ▶ Как успешно да въведем нови технологии?
 - ▶ Какви промени ще предизвикат те?
 - ▶ Искаме ли да разработим всичко отначало

Два типа подходи

Project-based approach



Product-based approach

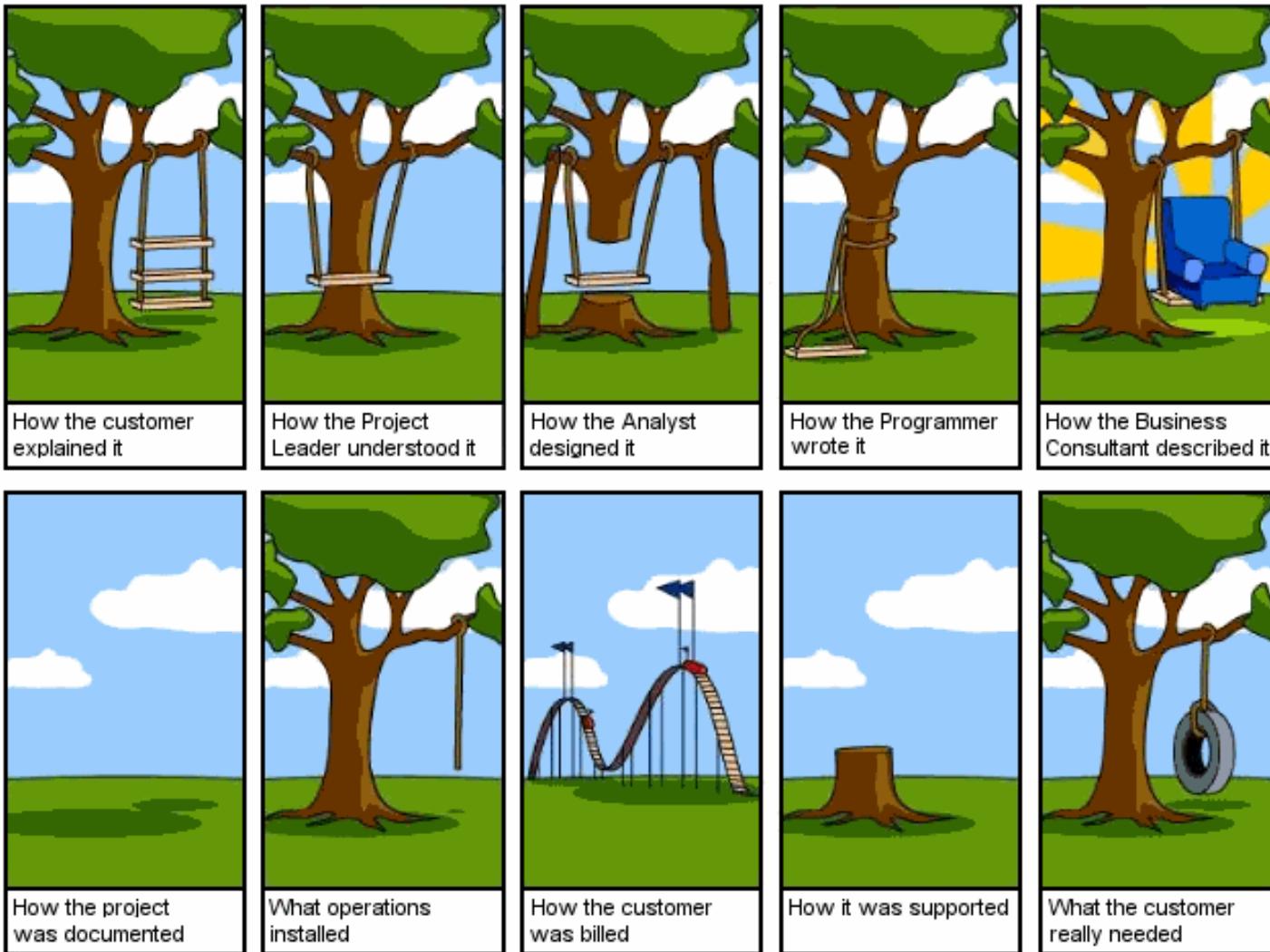


Project-based software engineering

- ▶ The starting point for the software development is a set of ‘software requirements’ that are owned by an external client and which set out what they want a software system to do to support their business processes.
- ▶ The software is developed by a software company (the contractor) who design and implement a system that delivers functionality to meet the requirements.
- ▶ The customer may change the requirements at any time in response to business changes (they usually do). The contractor must change the software to reflect these requirements changes.
- ▶ Custom software usually has a long-lifetime (10 years or more) and it must be supported over that lifetime.



Project-based software engineering



Product software engineering

- ▶ The starting point for product development is a business opportunity that is identified by individuals or a company. They develop a software product to take advantage of this opportunity and sell this to customers.
- ▶ The company who identified the opportunity design and implement a set of software features that realize the opportunity and that will be useful to customers.
- ▶ The software development company are responsible for deciding on the development timescale, what features to include and when the product should change.
- ▶ Rapid delivery of software products is essential to capture the market for that type of product.



Наследен (legacy) софтуер

- ▶ Съществуващ софтуер, който е разработен преди няколко десетилетия и продължава да се използва, като непрекъснато е бил променян, за да задоволи промени в бизнес изискванията и платформите (хардуерни и софтуерни).
- ▶ Наследеният софтуер се характеризира с
 - ▶ Дълъг живот
 - ▶ Критична важност за бизнеса
 - ▶ Лошо качество

Софтуерна продуктова/поточна (product) линия

- ▶ Набор от софтуерни системи, които имат общи функционалности – т.нар ядро (*core*). Към ядрото може да се добавят допълнителни функционалности (*features*), които удовлетворяват специфичните нужди на определен пазарен сегмент и са разработени от набор основни ресурси според предварително предначертан план

Софтуерна продуктова линия (аналогия с хардуерното производство)

- ▶ За да се получи автомобил от „ПЛ“ за автомобили може да се избира от следните „функционалности“: въздушна възглавница (за безопасност), ABS, CD player, комби, двигател 1500cc, турбокомпресор и т.н.
- ▶ Въпреки избраните функционалности, за да се произведе продукта (автомобил) може да се наложи да се уточнят определени вариации, например : брой въздушни възглавници.

Софтуерна продуктова линия (софтуерни системи и ПЛ)

- ▶ В ПЛ за банкови системи може да се избира от следните функционалности: внасяне, теглене, заем, връщане на заем, обмен на валута и т.н.
- ▶ Според избраните функционалности, може да се получат различни банкови системи. Подобно на предишния пример, може да се наложи да се уточнят някои вариации, като например: какви валути да се обменят.

Други видове софтуерни системи

- ▶ Системен софтуер (System software)
- ▶ Приложен софтуер (Application software)
- ▶ Уеб приложения (Web applications)
- ▶ Научен софтуер (Engineering/scientific software)
- ▶ Вграден софтуер (Embedded software)
 - ▶ Internet-of-Things (IoT)
 - ▶ Кибер-физични системи
- ▶ Изкуствен интелект (Artificial intelligence software)
- ▶ Системи от системи

Допълнителни материали

- ▶ Ian Sommerville, Product and System Engineering,
<https://www.dropbox.com/s/alwkfx6y9fqnirl/Appendix%20I.pdf>





Софтуерен процес



Софтуерен процес. Модели на софтуерен процес. Примери.

Съдържание

- ▶ Софтуерен процес
- ▶ Модели на софтуерен процес
- ▶ Примери

Процес

- ▶ *Последователност от стъпки включващи дейности, ограничения и ресурси, които осигуряват постигането на някакъв вид резултат.*

- ▶ Софтуерен процес
 - ▶ *Структуриран набор от дейности, необходими за разработване на софтуерна система в срок и с високо качество.*

Процес

► Основни цели на процеса

- ▶ *Ефикасност (Effectiveness)*
- ▶ *Възможност за поддръжка (Maintainability)*
- ▶ *Предсказуемост (Predictability)*
- ▶ *Повторяемост (Repeatability)*
- ▶ *Качество (Quality)*
- ▶ *Усъвършенстване (Improvement)*
- ▶ *Проследяване (Tracking)*

Процес

- ▶ Основни дейности
 - ▶ Комуникация (*Communication*)
 - ▶ Планиране (*Planning*)
 - ▶ Моделиране (*Modeling*)
 - ▶ Конструиране (*Construction*)
 - ▶ Внедряване (*Deployment*)

Процес

► Допълнителни дейности

- ▶ Следене и управление на софтуерния продукт (*Software product tracking and control*)
- ▶ Управление на риска (*Risk management*)
- ▶ Осигуряване на качеството (*Software quality assurance*)
- ▶ Измерване (*Measurement*)
- ▶ Управление на софтуерната конфигурация (*Software configuration management*)
- ▶ Управление на повторното използване (*Reusability management*)
- ▶ Подготовка и генериране на работни продукти (*Work product preparation and production*)

Модел на софтуерен процес

- ▶ A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.
- ▶ Опростено описание на софтуерен процес, което е представено от определена перспектива

Модел на софтуерен процес

► Цели

- ▶ *Формиране на общо разбиране за участниците в разработването на софтуер за дейностите, ресурсите и ограниченията*
- ▶ *Намиране на несъответствия, излишества и пропуски в процеса от разработващия екип*
- ▶ *Намиране и оценяване на подходящи дейности за постигане на целите на процеса*
- ▶ *Адаптиране на общ процес към отделна ситуация, в която ще се приложи*

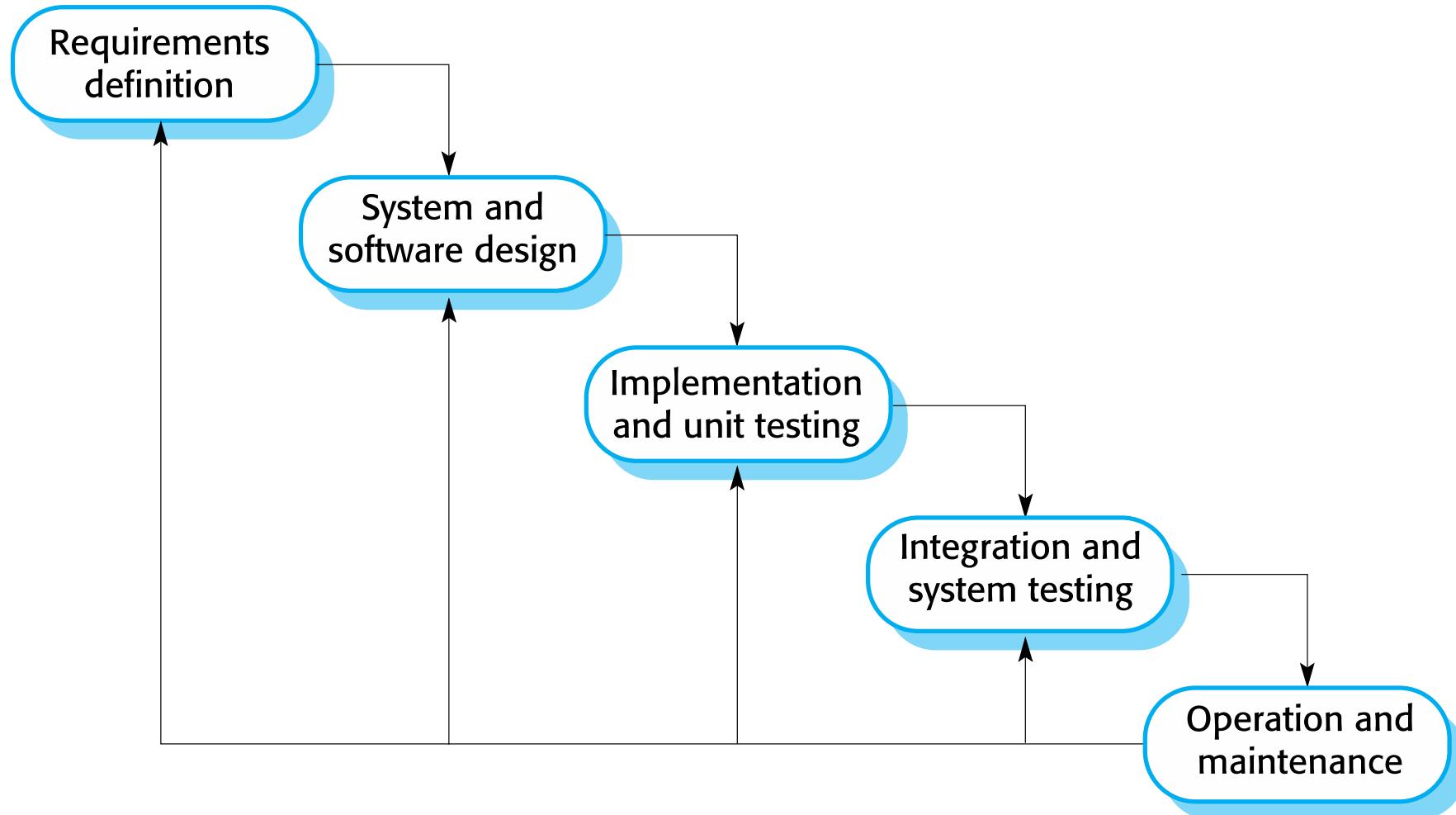
Plan-driven and agile processes

- ▶ Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- ▶ In agile processes, planning is incremental and it is 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!***

Модел на софтуерен процес

- ▶ Каскаден (водопад) - *The Waterfall model*
- ▶ Модел на бързата разработка - *The Rapid Application Development (RAD) model*
- ▶ Фазови (еволюционни) модели - *Evolutionary process models*
 - ▶ Постъпков (инкрементален) модел
 - ▶ Итеративен модел
- ▶ Прототипен модел - *The Prototyping model*
- ▶ Спираловиден модел - *The spiral model*

Модел на водопада – *The Waterfall model*



Модел на водопада – *The Waterfall model*

► Характеристики

- ▶ ясно разграничен процес, който е лесен за разбиране
- ▶ всяка дейност трябва да бъде напълно завършена, преди да се премине към следваща
- ▶ ясно са дефинирани входовете и изходите на дейностите, както и интерфейсите между отделните стъпки
- ▶ ясно са дефинирани ролите на разработчиците на софтуер

Модел на водопада – *The Waterfall model*

► Прилагане

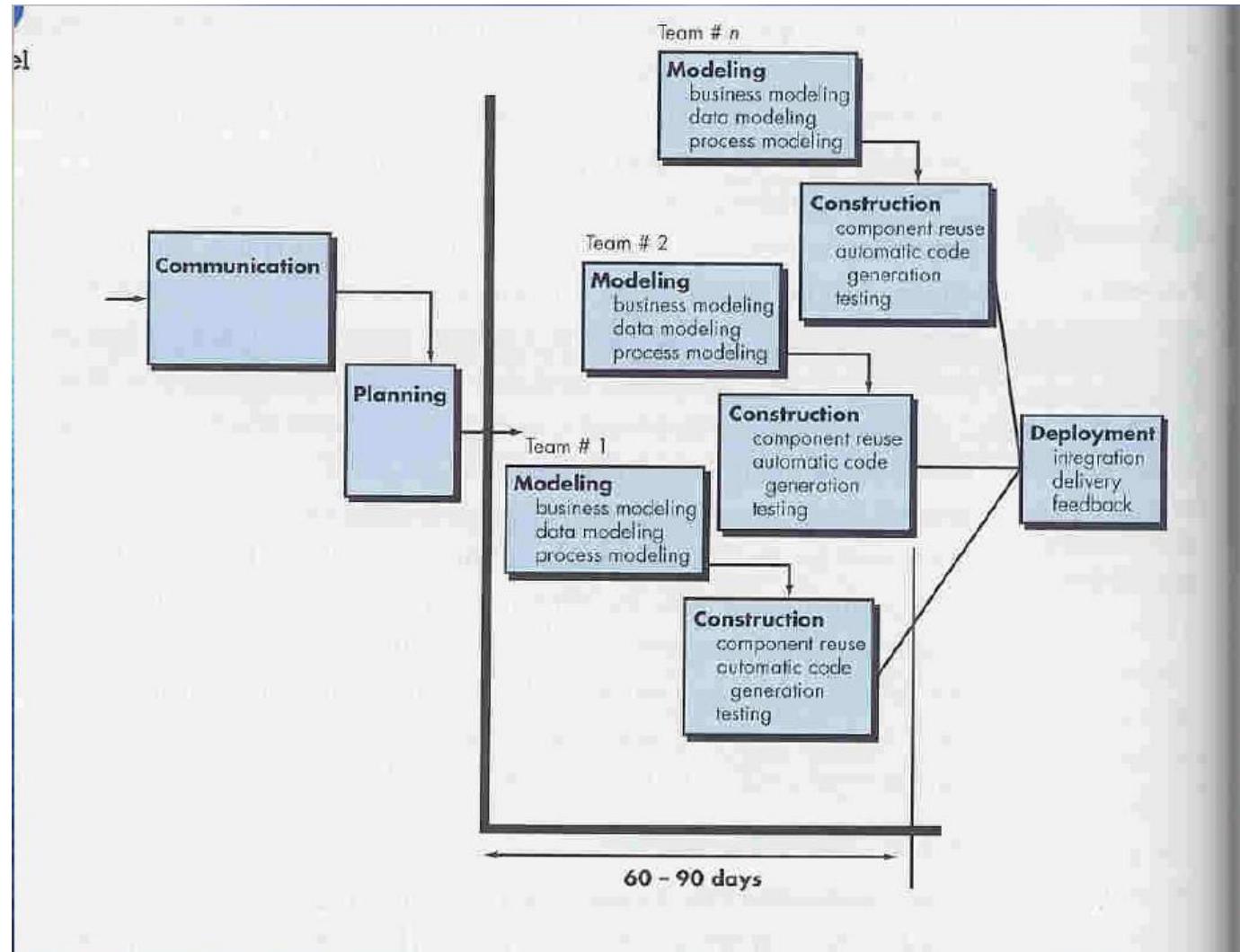
- ▶ Когато изискванията са осъзнати и ясно формулирани в началото
- ▶ Когато проектите са ясно организирани
- ▶ При повторяеми проекти и/или големи проекти, за които времето и бюджетът не са критични

Модел на водопада – *The Waterfall model*

▶ Проблеми

- ▶ Трудно е за потребителя да формулира всичките си изисквания в началото
- ▶ Разделянето на проекта на отделни етапи не е гъвкаво - трудно е да се реагира на променящите се изисквания на клиента
- ▶ Моделът е произлязъл от областта на хардуера и не отчита същността на софтуера като творчески процес на решаване на проблем (с итерации и връщане назад)

Модел на бързата разработка



Модел на бързата разработка

- ▶ **Комуникация / Планиране**
- ▶ **Моделиране**
 - ▶ *бизнес моделиране (Business modeling)*
 - ▶ *моделиране на данните (Data modeling)*
 - ▶ *моделиране на процес (Process modeling)*
- ▶ **Конструиране**
- ▶ **Внедряване**

Модел на бързата разработка

► Предимства

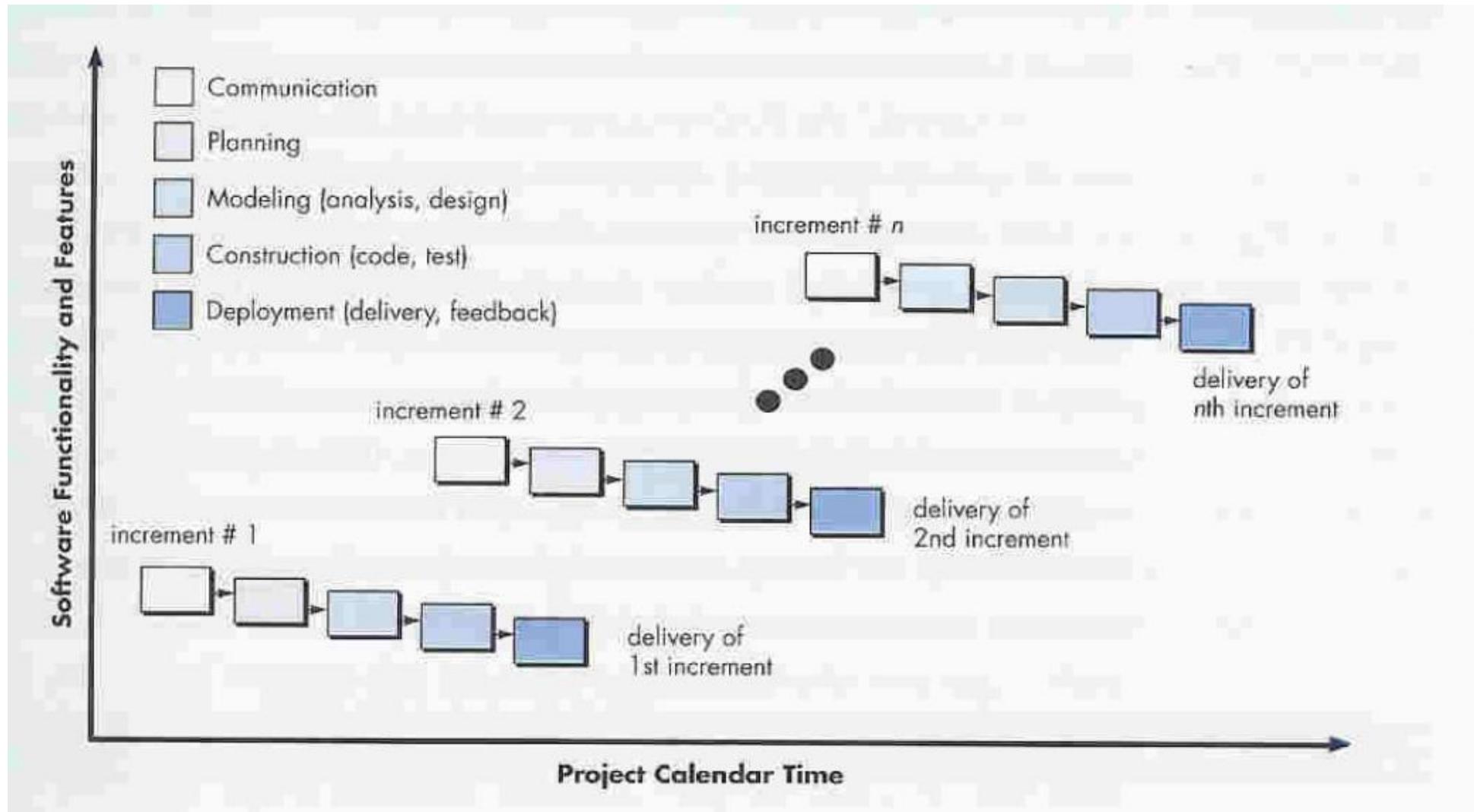
- ▶ *Подобрена гъвкавост, тъй като разработчиците могат да се адаптират към необходимите промени по време на изграждане*
- ▶ *Може да се създават бързи итерации, които намаляват времевите рамки*
- ▶ *Може да се извършват интеграции в началните етапи и да се използва повторно кода във всеки момент (по-кратко време за тестване)*
- ▶ *По-добро качество (от гледна точка на потребителите - бизнес функционалността) тъй като взаимодействат с развиващи се прототипи*

Модел на бързата разработка

► Недостатъци

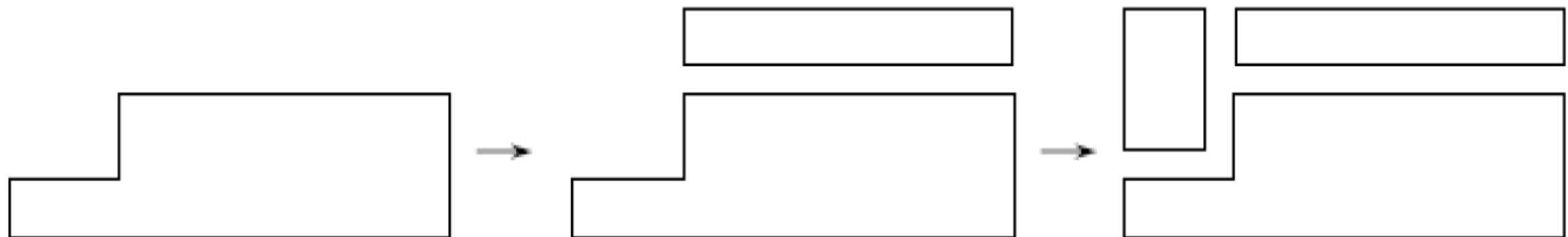
- ▶ За големи приложения, подлежащи на разделяне на модули (значителни човешки ресурси)
- ▶ Липса на акцент върху качествените атрибути на софтуера
- ▶ Фокусът върху прототипите може да доведе до непрекъснато незначителни промени в отделни компоненти и игнориране проблемите на системната архитектура.
- ▶ RAD изиска участие на клиента на много етапи, което прави процеса по-сложен от други методологии.

Постъпков (инкрементален) модел



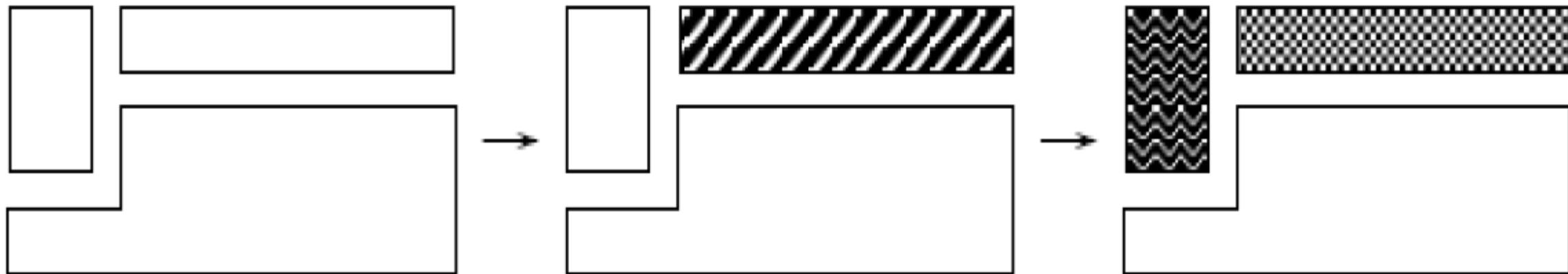
Постъпков (инкрементален) модел

- ▶ Системата не се доставя като едно цяло, а вместо това процесът на разработка и доставянето са разделени на стъпки, като всяка стъпка доставя само част от цялата функционалност
- ▶ На идентифицираните потребителски изисквания се присвояват приоритети и тези с по-висок се реализират в първите стъпки
- ▶ След като започне разработката на една стъпка, изискванията не се променят



Итеративен модел

- ▶ В самото начало доставя цялостната софтуерна система, макар и част от функционалността да е в примитивна форма
- ▶ При всяка следваща итерация не се добавя нова функционалност, а само се усъвършенства съществуващата



Фазови (еволюционни) модели

▶ Предимства

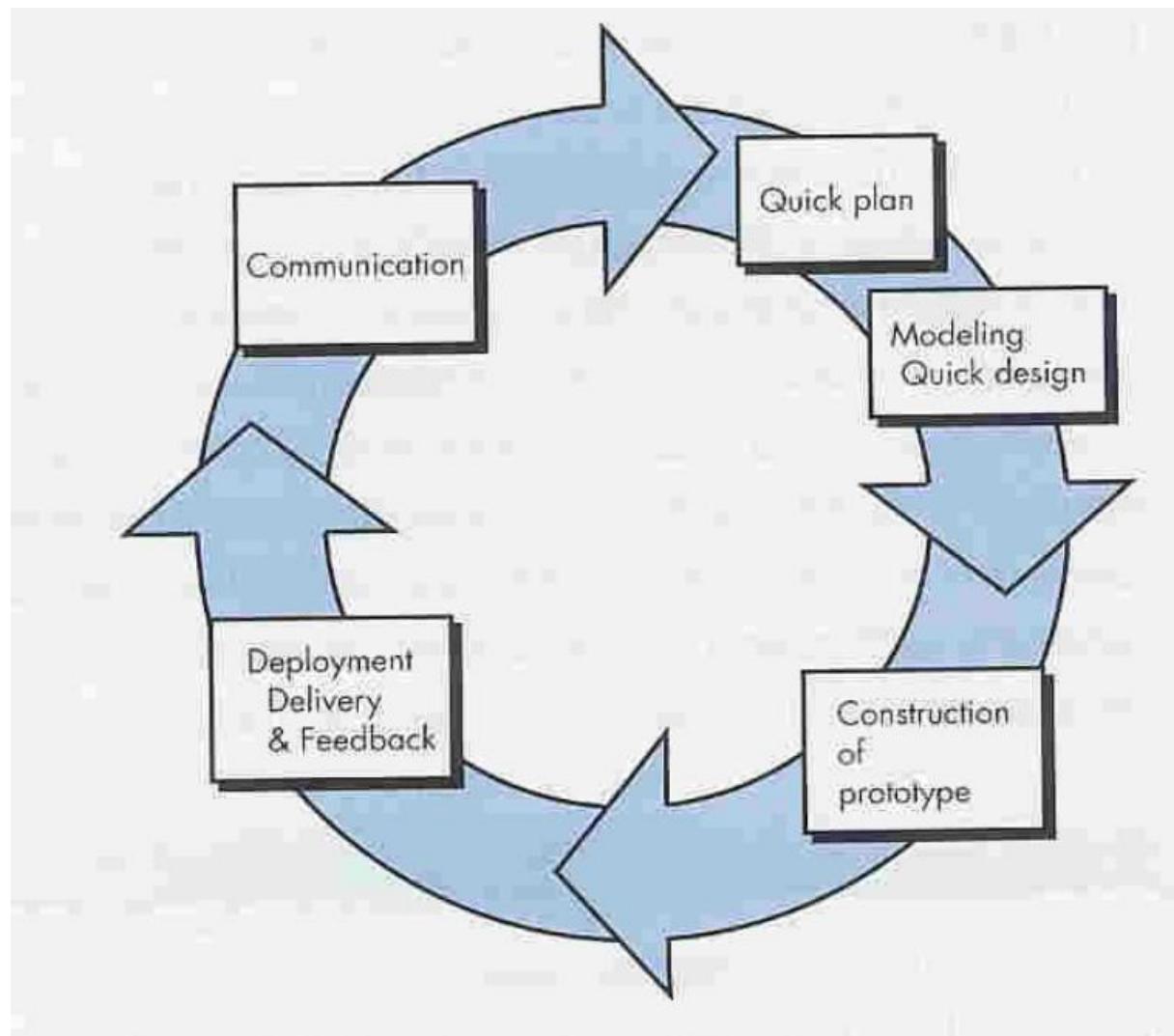
- ▶ *Клиентът може да използва системата, преди да е готов целият продукт*
- ▶ *Функционалностите от цялата система, които са с най-висок приоритет, са тествани най-много*
- ▶ *В разработването на по-ранните версии участват по-малко хора и в зависимост от обратната връзка, могат да се присъединят още разработчици на следващите итерации*
- ▶ *Итерация, която изисква използването на нова технология или продукт може да се планира по-късно*

Фазови (еволюционни) модели

▶ Проблеми

- ▶ *Необходимостта от активно участие на клиентите по време на изпълнение на проекта може да доведе до закъснения*
- ▶ *Уменията за комуникация и координация са от особено голямо значение при разработката*
- ▶ *Неформалните заявки за подобрения след завършването на всяка стъпка могат да доведат до объркване*
- ▶ *Може да доведе до т. нар. “scope-creep” – бавно и постепенно разширяване на обхвата на приложението*

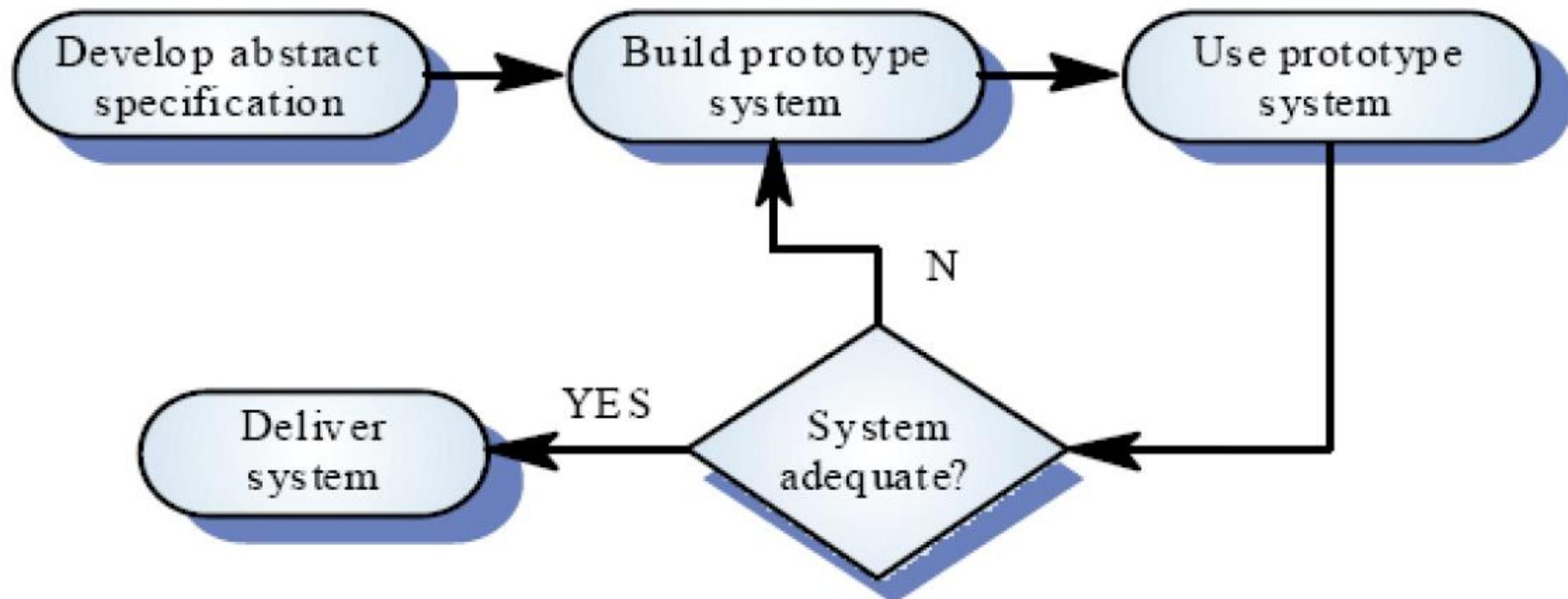
Прототипен модел



Прототипен модел

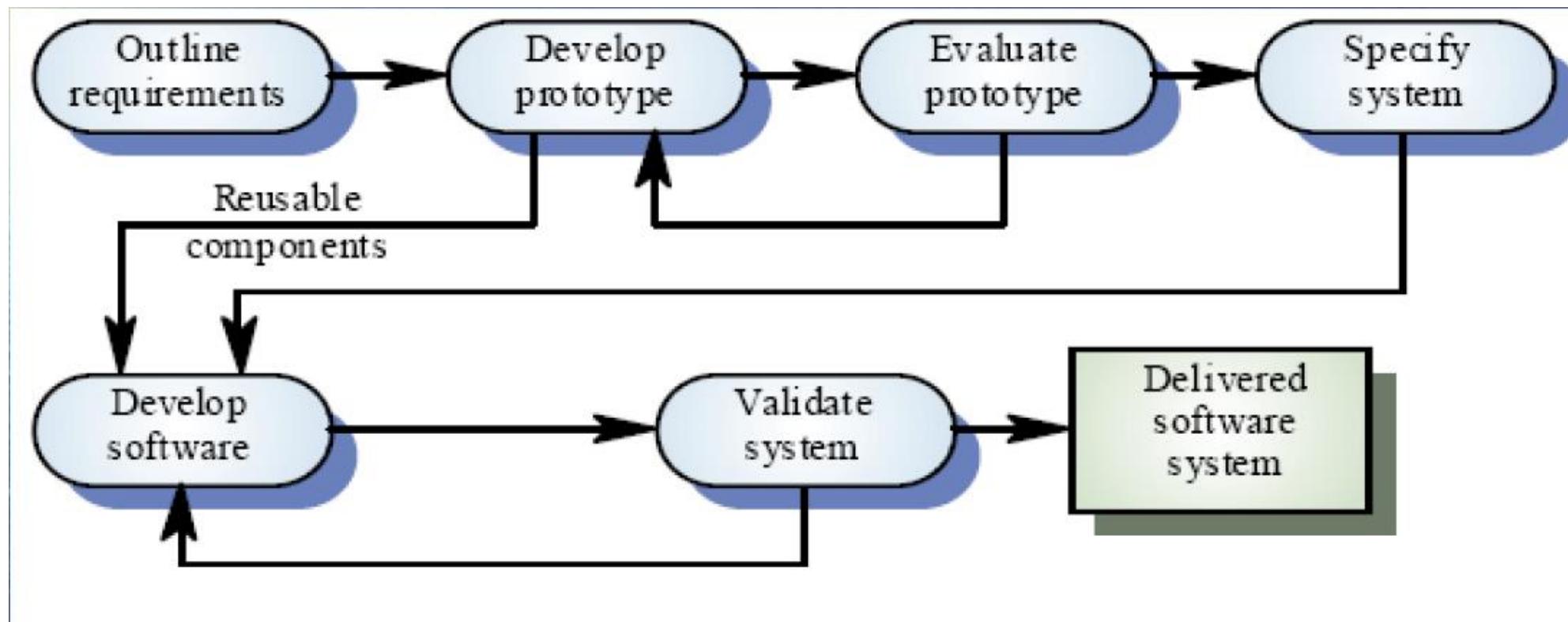
► Еволюционен прототип

► Цел: да достави работеща система на края потребител



Прототипен модел

- ▶ Изхвърлен (throw-away) прототип
 - ▶ Цел: да подпомогне специфицирането на изискванията към софтуера



Прототипен модел

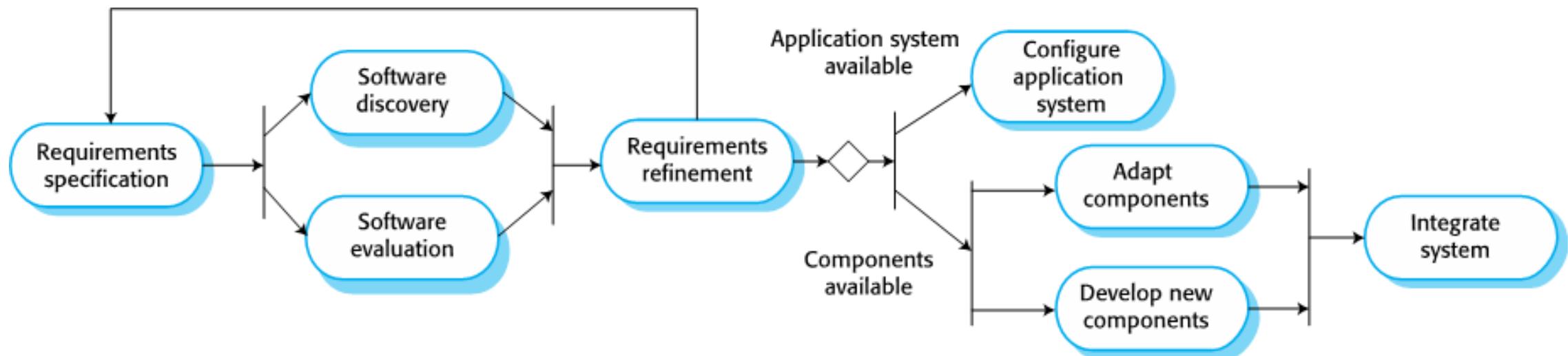
- ▶ Създаване на основните потребителски интерфейси, без да има никакво значително кодиране
- ▶ Разработване на съкратена версия на системата, която изпълнява ограничено подмножество от функции
- ▶ Използване на съществуваща система или компоненти от система, за да се демонстрират някои функции, които ще бъдат включени
- ▶ Прилага се в проекти, където не са достатъчно ясни изискванията на потребителите и дизайнът на софтуерната система
- ▶ *Както самостоятелно, така и в комбинация с други модели на процеси*

Прототипен модел

▶ Проблеми

- ▶ Прототипният модел може да използва значителни ресурси, а като резултат прототипът да не успее да удовлетвори очакванията
- ▶ Прототипът може да доведе до лошо проектирана система, ако самият той стане част от крайния продукт
- ▶ Прототипният модел не е подходящ за използване при разработване на софтуерни системи, където проблемът е добре разбран и интерфейсът е ясен и прост

Reuse-oriented software engineering



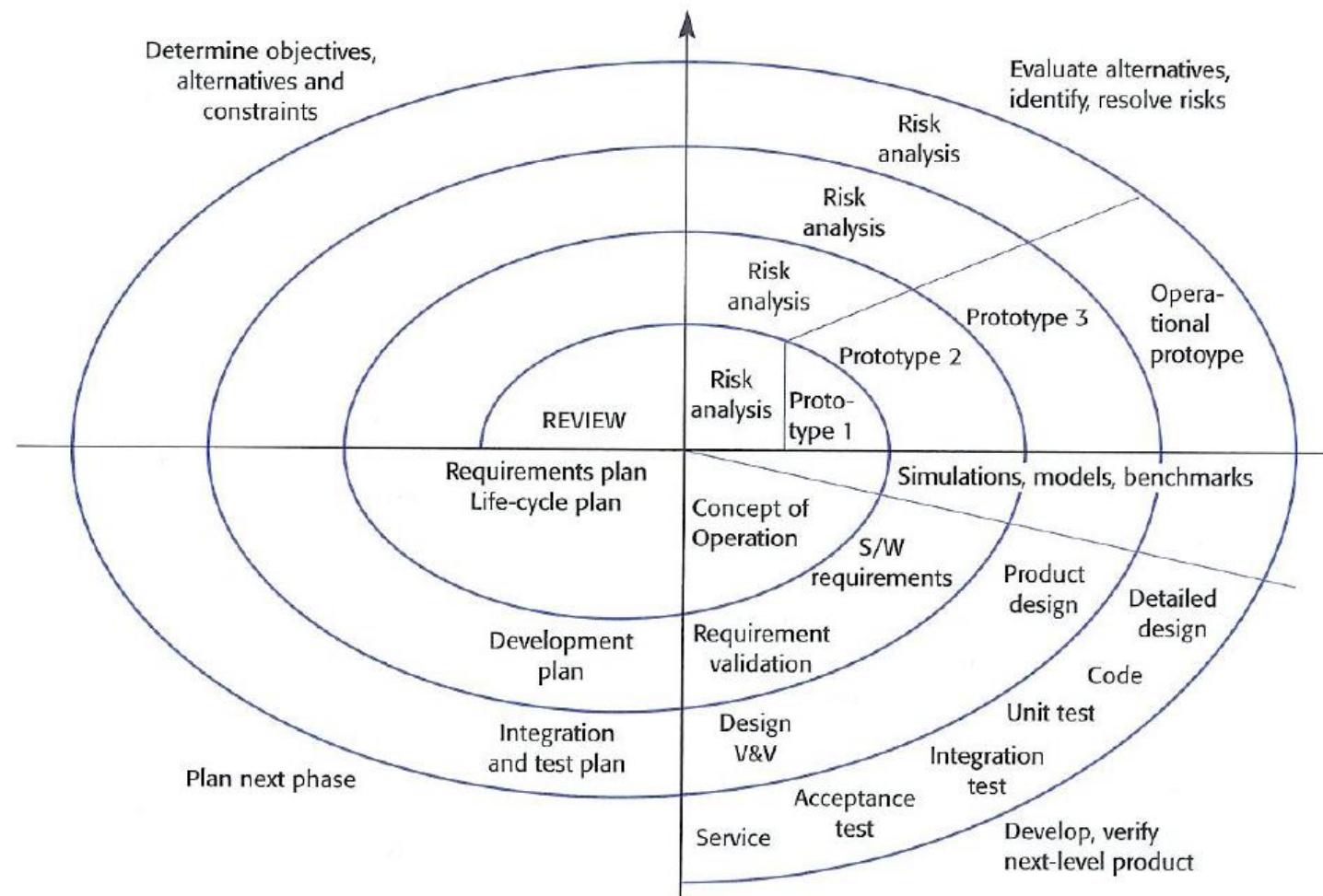
Integration and configuration

- ▶ 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 a standard approach for building many types of business system

Types of reusable software

- ▶ Stand-alone application systems (sometimes called 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.

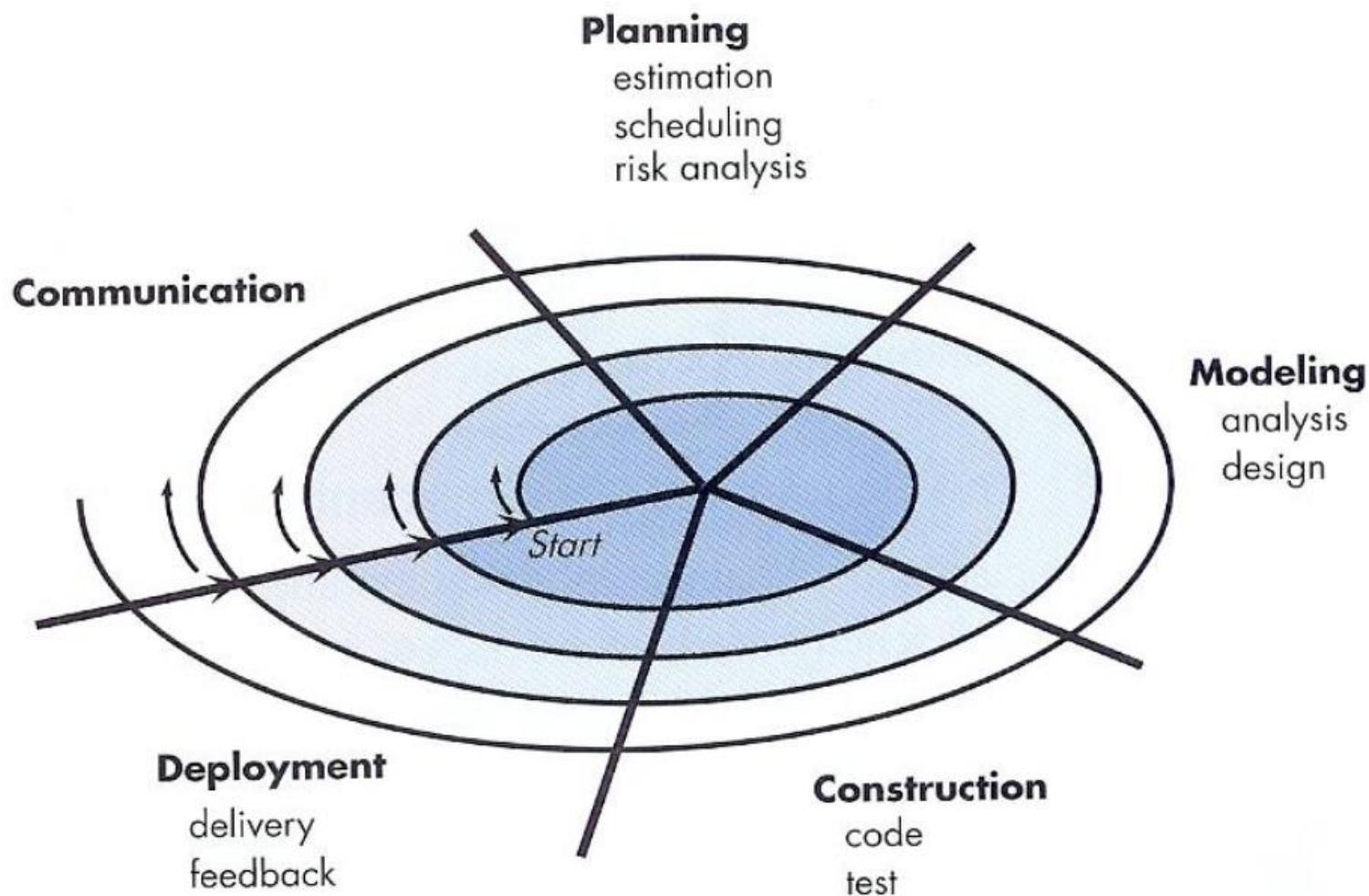
Спираловиден модел



Сpirаловиден модел

- ▶ Спираловидният модел е еволюционен модел на софтуерен процес, който съчетава прототипния модел и модела на водопада
- ▶ Движещият фактор е анализ на риска
- ▶ Основни характеристики:
 - ▶ *итеративен/цикличен подход*
 - ▶ *има множество от точки на прогреса (anchor point milestones)*

Спираловиден модел



Спираловиден модел

- ▶ Установяване на целите
 - ▶ определят се цели, алтернативи и ограничения на текущата фаза
- ▶ Оценка на рисковете и намаляването им
 - ▶ идентифицират се и се анализират потенциалните рискове
 - ▶ предприемат се действия за намаляването или елиминирането им
- ▶ Разработване и валидиране
 - ▶ избира се модел за разработване на текущата фаза
- ▶ Планиране
 - ▶ преглежда се и се анализира текущото състояние
 - ▶ планира се следващото завъртане по спиралата

Спираловиден модел

▶ Проблеми

- ▶ *Изискава се участието на разработчици с компетентност за оценка на рисковете*
- ▶ *Ако не се идентифицира и открие някой основен риск, това може да доведе до неуспех*
- ▶ *Може да се окаже трудно да се убедят клиентите, че процесът на разработка е контролиран, а не е безкраен цикъл*

Метод на Формална трансформация

- ▶ Основава се на математическо трансформиране на спецификацията на системните изисквания до изпълнима програма
 - ▶ *При трансформирането трябва да се запази коректността и ясно да се покаже, че изпълнимата програма съответства на спецификацията*
- ▶ Спецификацията на софтуерните изисквания се усъвършенства в детайлна формална спецификация, изразена с математическа нотация
 - ▶ *Дейностите моделиране и конструиране са заменени с разработване и прилагане на трансформации*

Метод на Формална трансформация

In the transformational development process. The formal specification is refined through a series of transformational steps into a program as shown fig 2.17.

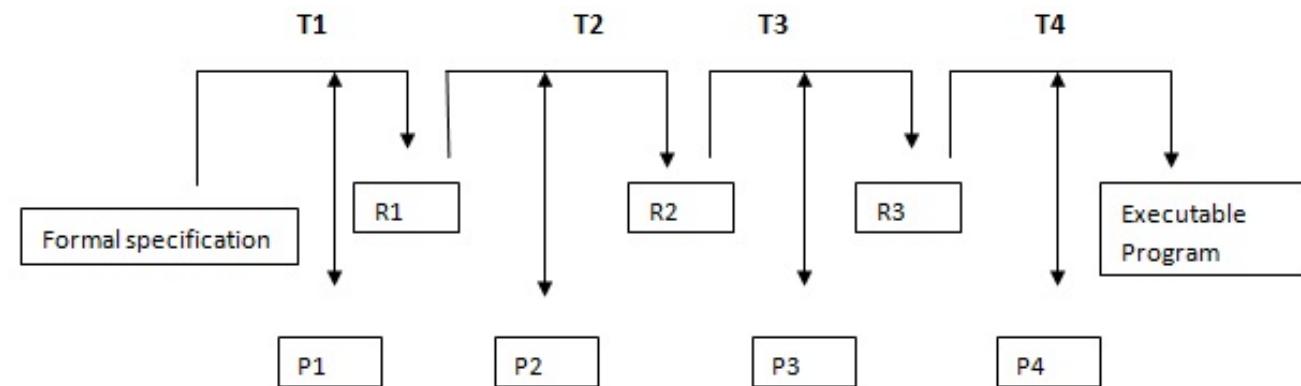


Fig. 2.17, Formal Transformation

T₁, T₂, T₃, T₄, Transformational steps

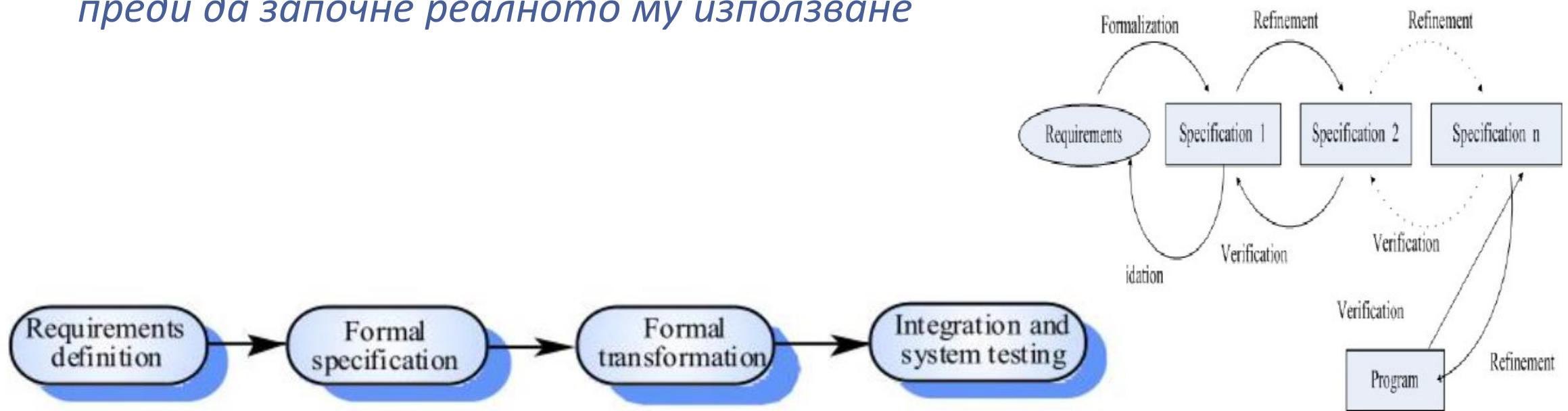
P₁, P₂, P₃, P₄, Process steps to transform the formal specification

R₁, R₂, R₃, Refinements

Метод на Формална трансформация

► Прилагане

► *При разработването на софтуерни системи, които са свързани с поддръжката на човешки живот и за които трябва да са гарантирани сигурността и отказоустойчивостта на софтуера, преди да започне реалното му използване*



Метод на Формална трансформация

▶ Проблеми

- ▶ *разработването на формални модели е скъп и бавен процес*
- ▶ *необходими са разработчици със специализирани умения, както и обучение за това, как да се прилага формалната трансформация*
- ▶ *поради сложността си изготвените модели трудно могат да се използват*
- ▶ *някои от аспектите на софтуерна система е трудно е да се специфицират формално – например потребителският интерфейс*

Модел на софтуерен процес

- ▶ Изборът зависи основно от два фактора:
 - ▶ *Организационната среда*
 - ▶ *Същността на приложението*

Въпроси ?



Допълнителни материали

- ▶ Ian Sommerville, *Software Engineering, 10th Ed.*, **Chapter 2**

Agile Software Engineering

Agile software engineering

- Software products must be brought to market quickly so rapid software development and delivery is essential.
- Virtually all software products are now developed using an agile approach.
- Agile software engineering focuses on delivering functionality quickly, responding to changing product specifications and minimizing development overheads.
- A large number of ‘agile methods’ have been developed.
 - There is no ‘best’ agile method or technique.
 - It depends on who is using the technique, the development team and the type of product being developed

Agile methods

- Plan-driven development evolved to support the engineering of large, long-lifetime systems (such as aircraft control systems) where teams may be geographically dispersed and work on the software for several years.
 - This approach is based on controlled and rigorous software development processes that include detailed project planning, requirements specification and analysis and system modelling.
 - However, plan-driven development involves significant overheads and documentation and it does not support the rapid development and delivery of software.
- Agile methods were developed in the 1990s to address this problem.
 - These methods focus on the software rather than its documentation, develop software in a series of increments and aim to reduce process bureaucracy as much as possible.

Table 2.1 The agile manifesto

We are uncovering better ways of developing software by doing it and helping others to 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.

While there is value on the items on the right, we value the items on the left more.

Incremental development

- All agile methods are based around incremental development and delivery.
- Product development focuses on the software features, where a feature does something for the software user.
- With incremental development, you start by prioritizing the features so that the most important features are implemented first.
 - You only define the details of the feature being implemented in an increment.
 - That feature is then implemented and delivered.
- Users or surrogate users can try it out and provide feedback to the development team. You then go on to define and implement the next feature of the system.

Figure 2.1 Incremental development

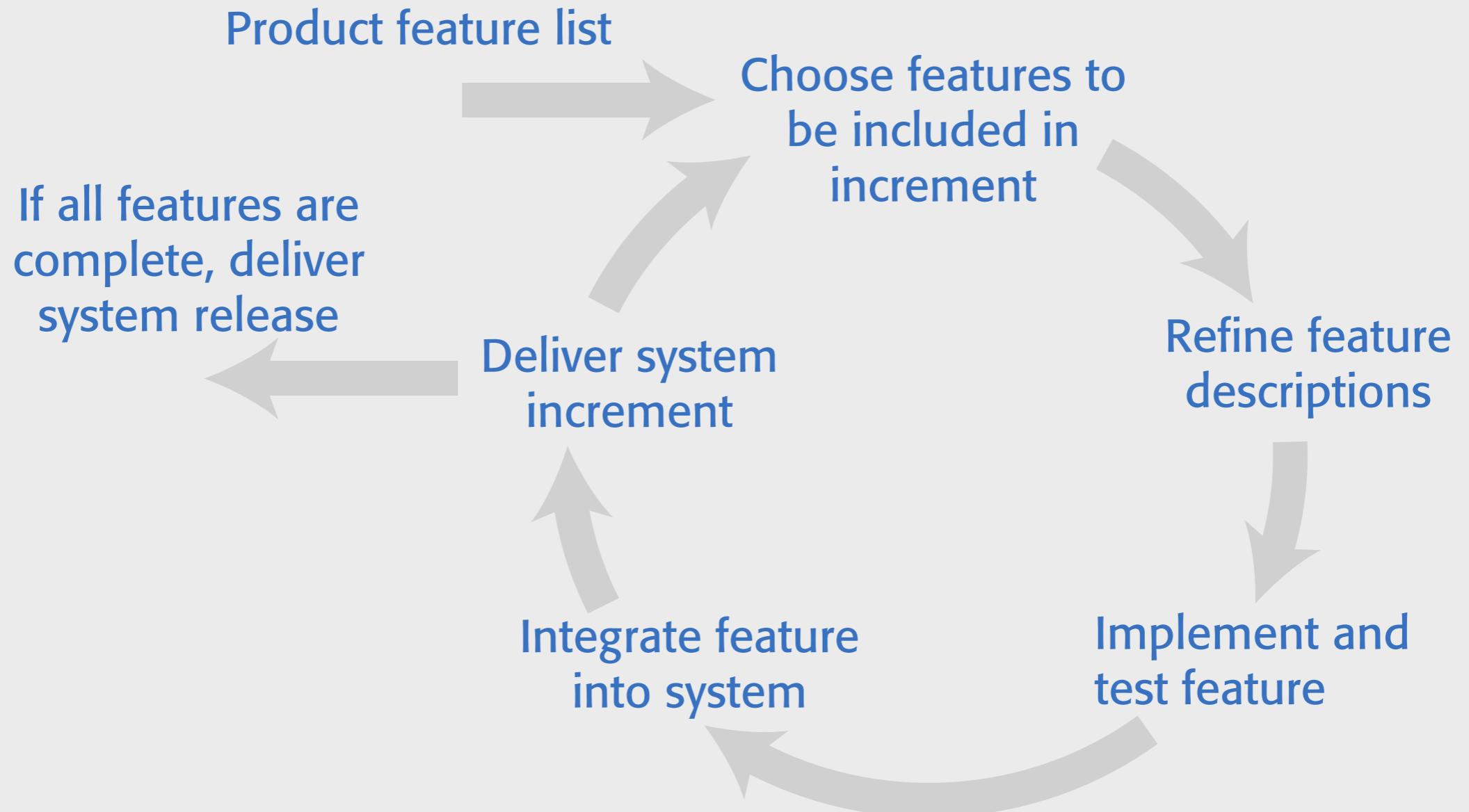


Table 2.2 Incremental development activities

Choose features to be included in an increment

Using the list of features in the planned product, select those features that can be implemented in the next product increment.

Refine feature descriptions

Add detail to the feature descriptions so that the team have a common understanding of each feature and there is sufficient detail to begin implementation.

Implement and test

Implement the feature and develop automated tests for that feature that show that its behaviour is consistent with its description.

Integrate feature and test

Integrate the developed feature with the existing system and test it to check that it works in conjunction with other features.

Deliver system increment

Deliver the system increment to the customer or product manager for checking and comments. If enough features have been implemented, release a version of the system for customer use.

Table 2.3 Agile development principles

Involve the customer

Involve customers closely with the software development team. Their role is to provide and prioritize new system requirements and to evaluate each increment of the system.

Embrace change

Expect the features of the product and the details of these features to change as the development team and the product manager learn more about it. Adapt the software to cope with changes as they are made.

Develop and deliver incrementally

Always develop software products in increments. Test and evaluate each increment as it is developed and feed back required changes to the development team.

Table 2.3 Agile development principles

Maintain simplicity

Focus on simplicity in both the software being developed and in the development process. Wherever possible, do what you can to eliminate complexity from the system.

Focus on people, not things

Trust the development team and do not expect everyone to always do the development process in the same way. Team members should be left to develop their own ways of working without being limited by prescriptive software processes.

Extreme programming

- The most influential work that has changed software development culture was the development of Extreme Programming (XP).
- The name was coined by Kent Beck in 1998 because the approach was developed by pushing recognized good practice, such as iterative development, to ‘extreme’ levels.
- Extreme programming focused on 12 new development techniques that were geared to rapid, incremental software development, change and delivery.
- Some of these techniques are now widely used; others have been less popular.

Figure 2.2 Extreme programming practices

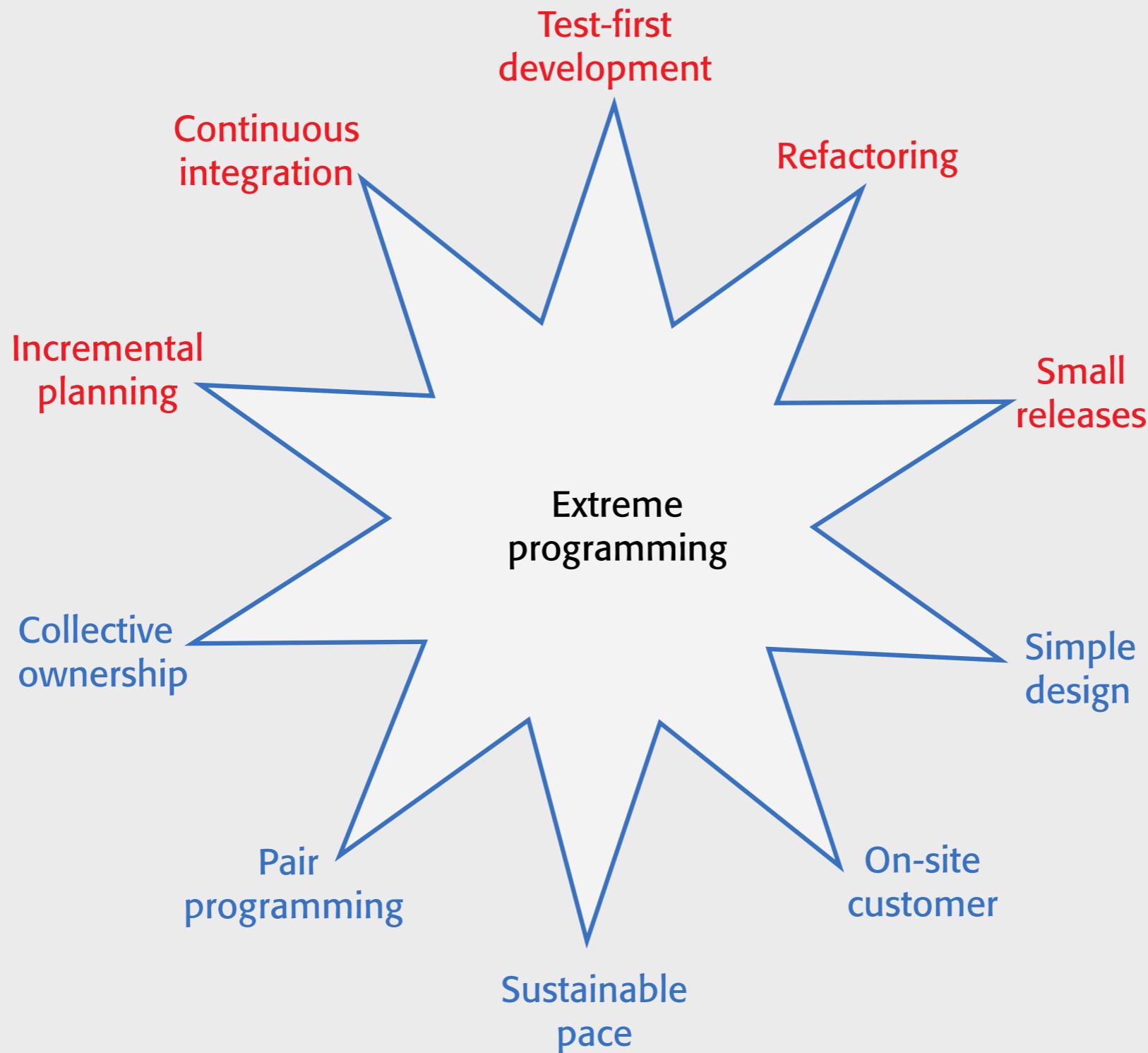


Table 2.4 Widely adopted XP practices

Incremental planning/user stories

There is no ‘grand plan’ for the system. Instead, what needs to be implemented (the requirements) in each increment are established in discussions with a customer representative. The requirements are written as user stories. The stories to be included in a release are determined by the time available and their relative priority.

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 previous release.

Test-driven development

Instead of writing code then tests for that code, developers write the tests first. This helps clarify what the code should actually do and that there is always a ‘tested’ version of the code available. An automated unit test framework is used to run the tests after every change. New code should not ‘break’ code that has already been implemented.

Table 2.4 Widely adopted XP practices

Continuous integration

As soon as the work on a task is complete, it is integrated into the whole system and a new version of the system is created. All unit tests from all developers are run automatically and must be successful before the new version of the system is accepted.

Refactoring

Refactoring means improving the structure, readability, efficiency and security of a program. All developers are expected to refactor the code as soon as potential code improvements are found. This keeps the code simple and maintainable.

Simple design

Enough design is carried out to meet the current requirements and no more.

Table 2.4 Widely adopted XP practices

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.

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.

Scrum

- Software company managers need information that will help them understand how much it costs to develop a software product, how long it will take and when the product can be brought to market.
- Plan-driven development provides this information through long-term development plans that identify deliverables - items the team will deliver and when these will be delivered.
- Plans always change so anything apart from short-term plans are unreliable.
- Scrum is an agile method that provides a framework for agile project organization and planning. It does not mandate any specific technical practices.

Table 2.5 Scrum terminology

Product

The software product that is being developed by the Scrum team.

Product owner

A team member who is responsible for identifying product features and attributes. They review work done and help to test the product.

Product backlog

A to-do list of items such as bugs, features and product improvements that the Scrum team have not yet completed.

Development team

A small self-organising team of five to eight people who are responsible for developing the product.

Sprint

A short period, typically two to four weeks, when a product increment is developed.

Table 2.5 Scrum terminology

Scrum

A daily team meeting where progress is reviewed and work to be done that day as discussed and agreed.

ScrumMaster

A team coach who guides the team in the effective use of Scrum.

Potentially shippable product increment

The output of a sprint which should be of high enough quality to be deployed for customer use.

Velocity

An estimate of how much work a team can do in a single sprint.

Key roles in Scrum

- **The Product Owner** is responsible for ensuring that the development team are always focused on the product they are building rather than diverted into technically interesting but less relevant work.
 - In product development, the product manager should normally take on the Product Owner role.
- **The ScrumMaster** is a Scrum expert whose job is to guide the team in the effective use of the Scrum method. The developers of Scrum emphasize that the ScrumMaster is not a conventional project manager but is a coach for the team. They have authority within the team on how Scrum is used.
 - In many companies that use Scrum, the ScrumMaster also has some project management responsibilities.

Scrum and sprints

- In Scrum, software is developed in sprints, which are fixed-length periods (2 - 4 weeks) in which software features are developed and delivered.
- During a sprint, the team has daily meetings (Scrums) to review progress and to update the list of work items that are incomplete.
- Sprints should produce a ‘shippable product increment’. This means that the developed software should be complete and ready to deploy.

Figure 2.3 Scrum cycles

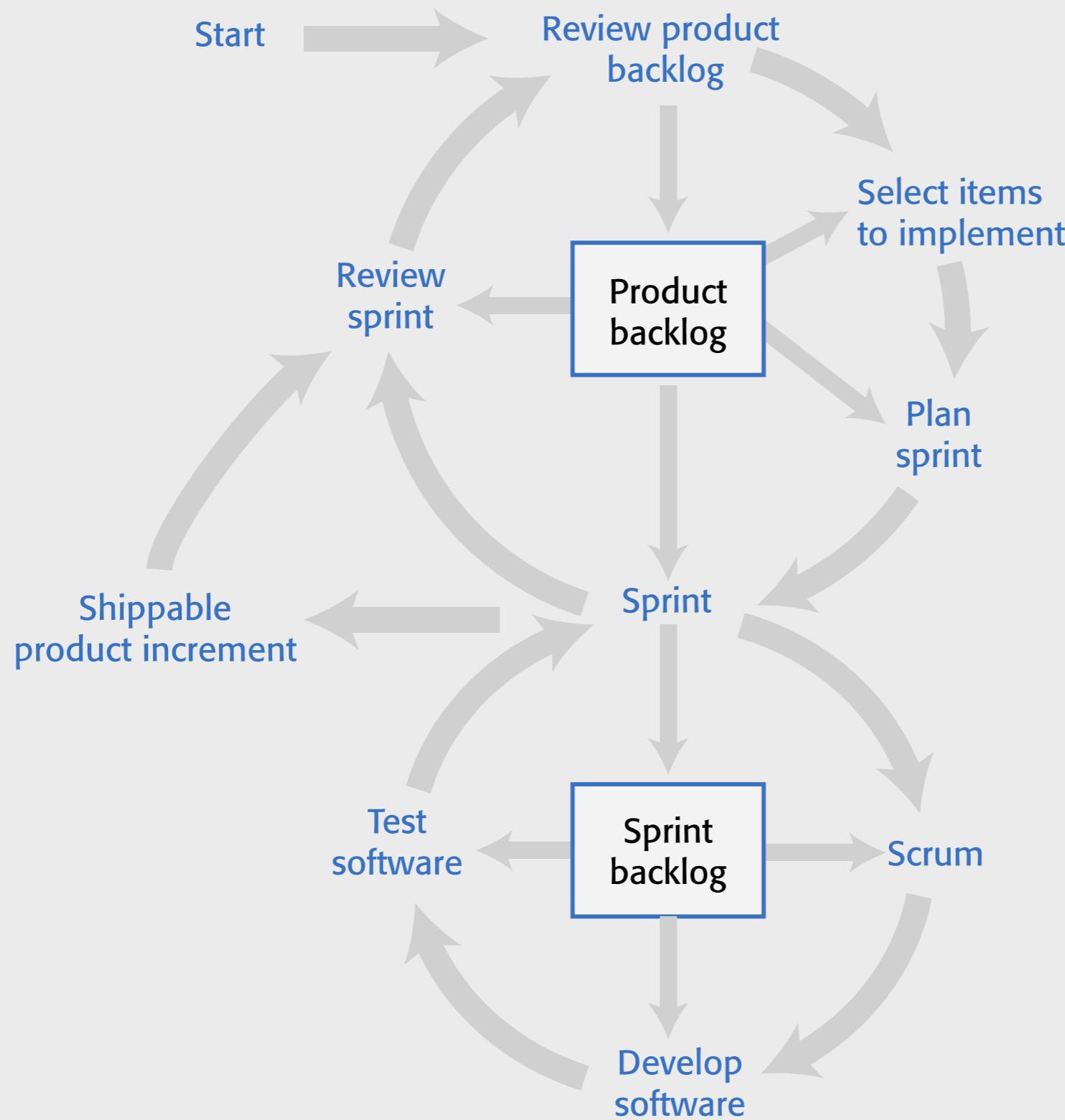
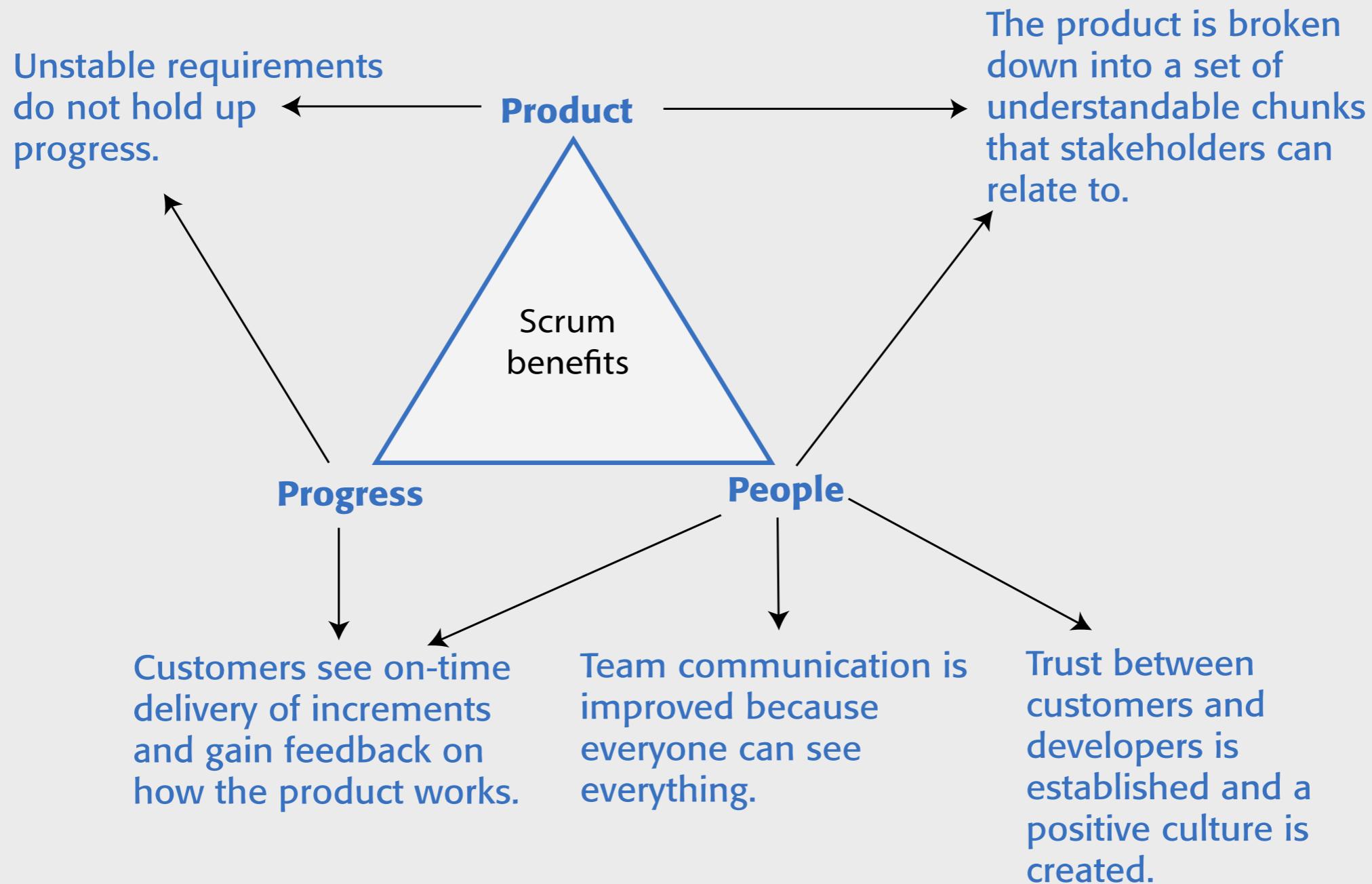


Figure 2.4 The top five benefits of using Scrum



Key Scrum practices

- ***Product backlog***

This is a to-do list of items to be implemented that is reviewed and updated before each sprint.

- ***Timeboxed sprints***

Fixed-time (2-4 week) periods in which items from the product backlog are implemented,

- ***Self-organizing teams***

Self-organizing teams make their own decisions and work by discussing issues and making decisions by consensus.

Product backlogs

- The product backlog is a list of what needs to be done to complete the development of the product.
- The items on this list are called product backlog items (PBIs).
- The product backlog may include a variety of different items such as product features to be implemented, user requests, essential development activities and desirable engineering improvements.
- The product backlog should always be prioritized so that the items that be implemented first are at the top of the list.

Table 2.6 Examples of product backlog items

1. As a teacher, I want to be able to configure the group of tools that are available to individual classes. (feature)
2. As a parent, I want to be able to view my childrens' work and the assessments made by their teachers. (feature)
3. As a teacher of young children, I want a pictorial interface for children with limited reading ability. (user request)
4. Establish criteria for the assessment of open source software that might be used as a basis for parts of this system. (development activity)
5. Refactor user interface code to improve understandability and performance. (engineering improvement)
6. Implement encryption for all personal user data. (engineering improvement)

Table 2.7 Product backlog item states

Ready for consideration

These are high-level ideas and feature descriptions that will be considered for inclusion in the product. They are tentative so may radically change or may not be included in the final product.

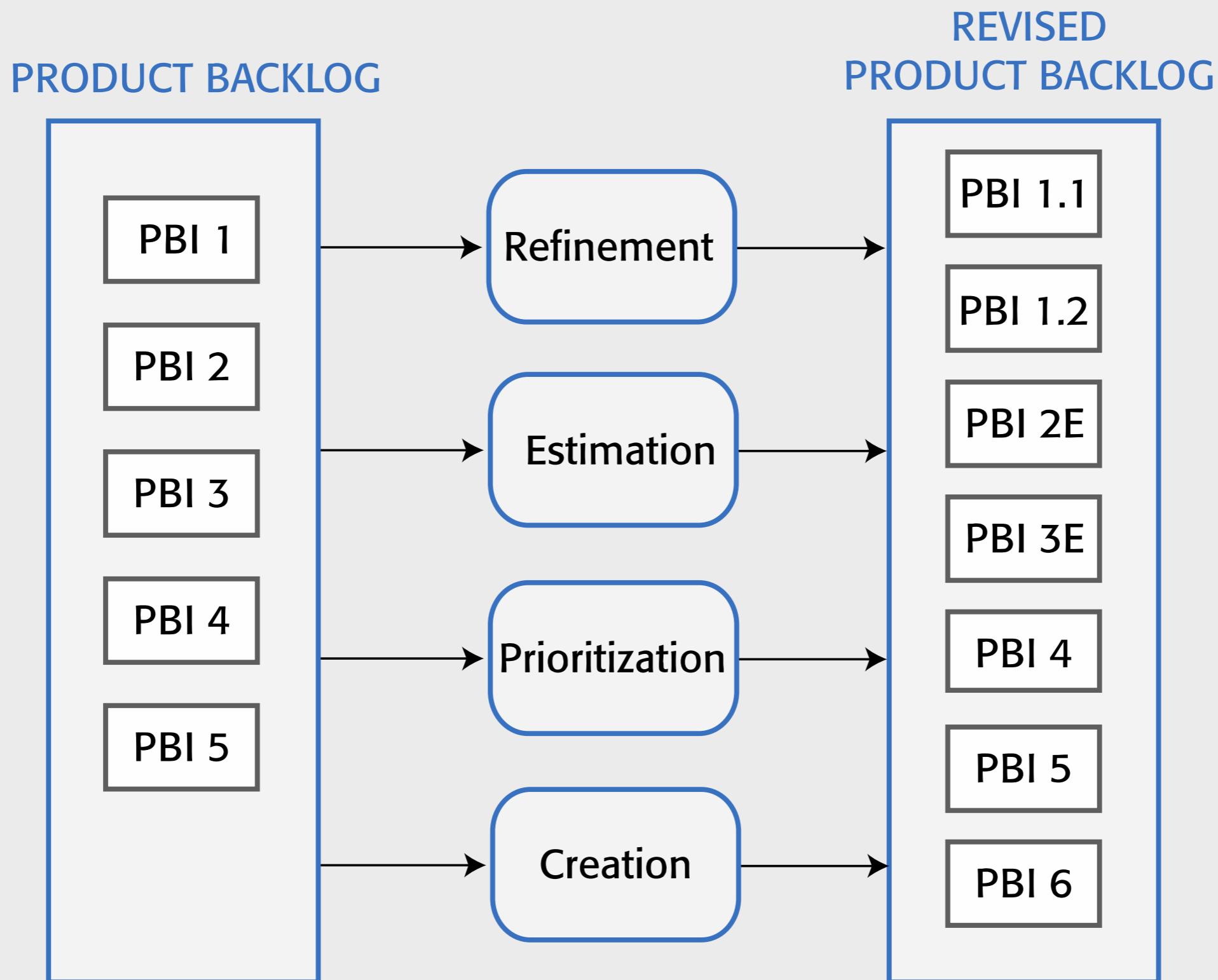
Ready for refinement

The team has agreed that this is an important item that should be implemented as part of the current development. There is a reasonably clear definition of what is required. However, work is needed to understand and refine the item.

Ready for implementation

The PBI has enough detail for the team to estimate the effort involved and to implement the item. Dependencies on other items have been identified.

Figure 2.5 Product backlog activities



Product backlog activities

- **Refinement**

Existing PBIs are analysed and refined to create more detailed PBIs. This may lead to the creation of new product backlog items.

- **Estimation**

The team estimate the amount of work required to implement a PBI and add this assessment to each analysed PBI.

- **Creation**

New items are added to the backlog. These may be new features suggested by the product manager, required feature changes, engineering improvements, or process activities such as the assessment of development tools that might be used.

- **Prioritization**

The product backlog items are reordered to take new information and changed circumstances into account.

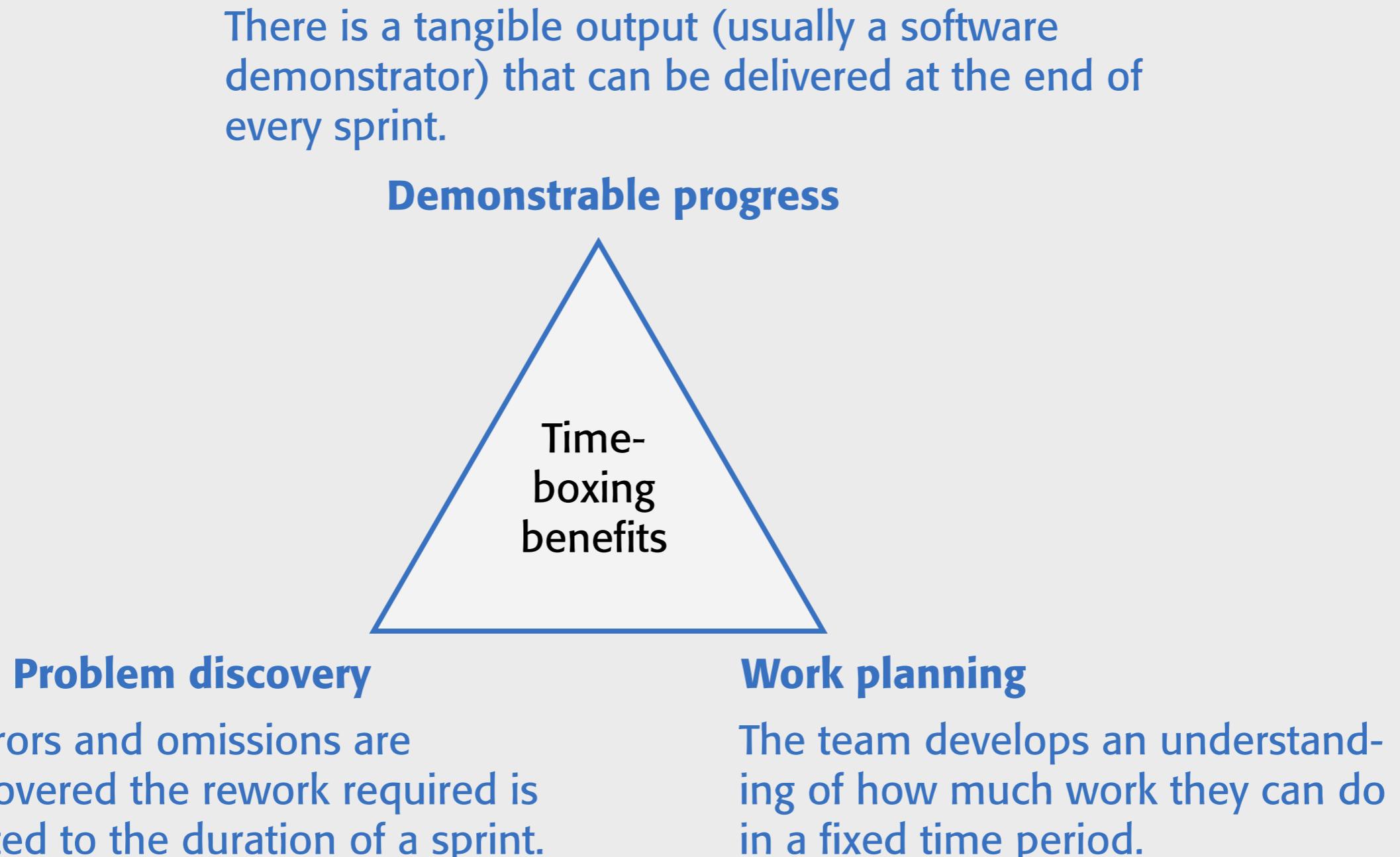
PBI estimation metrics

- Effort required
 - This may be expressed in person-hours or person-days i.e. the number of hours or days it would take one person to implement that PBI. This is not the same as calendar time. Several people may work on an item, which may shorten the calendar time required.
- Story points
 - Story points are an arbitrary estimate of the effort involved in implementing a PBI, taking into account the size of the task, its complexity, the technology that may be required and the ‘unknown’ characteristics of the work.
 - They were derived originally by comparing user stories, but they can be used for estimating any kind of PBI.
 - Story points are estimated relatively. The team agree on the story points for a baseline task and other tasks are estimated by comparison with this e.g. more/less complex, larger/smaller etc.

Timeboxed sprints

- Products are developed in a series of sprints, each of which delivers an increment of the product or supporting software.
- Sprints are short duration activities (1-4 weeks) and take place between a defined start and end date. Sprints are timeboxed, which means that development stops at the end of a sprint whether or not the work has been completed.
- During a sprint, the team work on the items from the product backlog.

Figure 2.6 Benefits of using timeboxed sprints



Sprint activities

- ***Sprint planning***

Work items to be completed in that sprint are selected and, if necessary, refined to create a sprint backlog. This should not last more than a day at the beginning of the sprint.

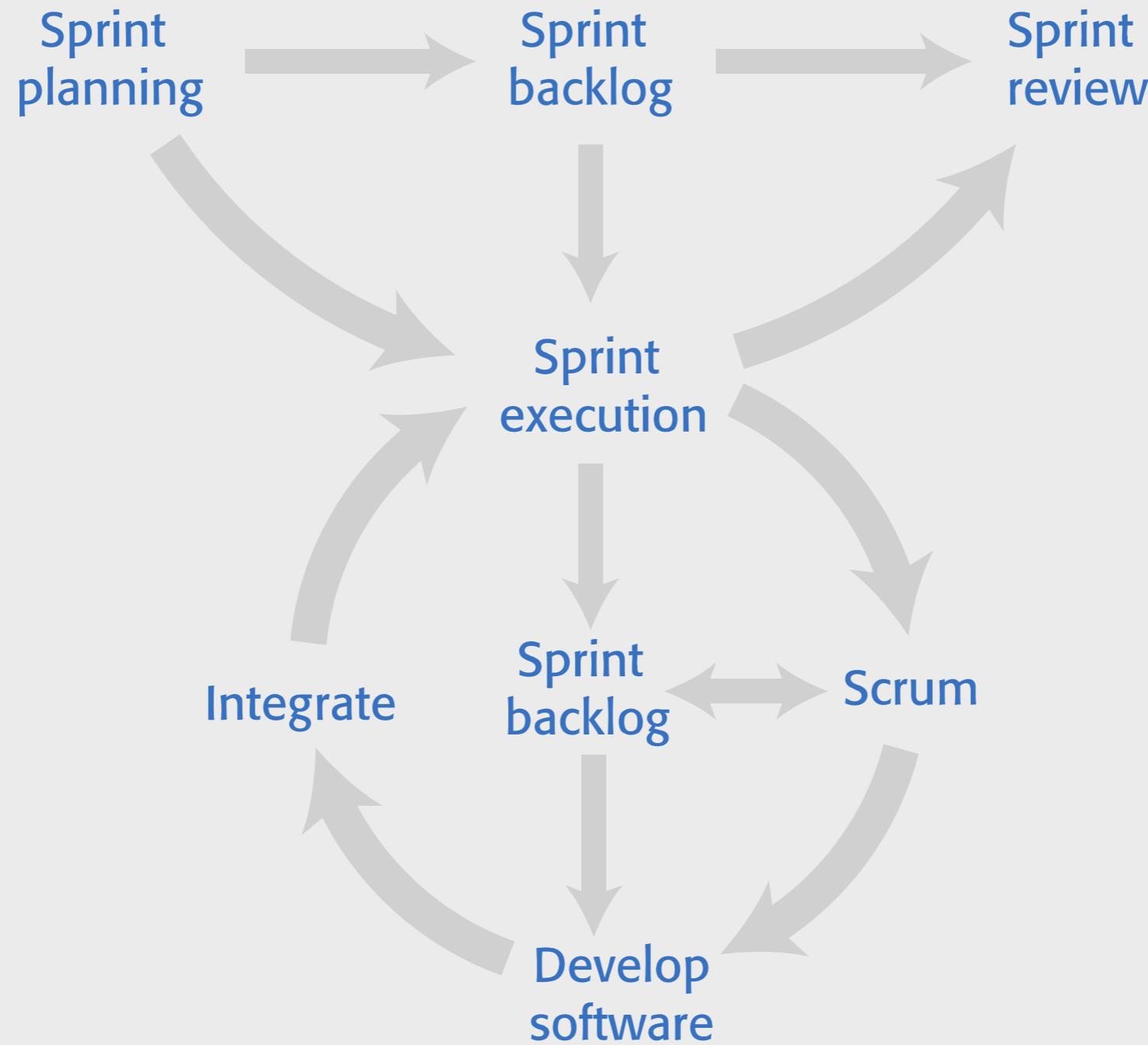
- ***Sprint execution***

The team work to implement the sprint backlog items that have been chosen for that sprint. If it is impossible to complete all of the sprint backlog items, the sprint is not extended. The unfinished items are returned to the product backlog and queued for a future sprint.

- ***Sprint reviewing***

The work done in the sprint is reviewed by the team and (possibly) external stakeholders. The team reflect on what went well and what went wrong during the sprint with a view to improving their work process.

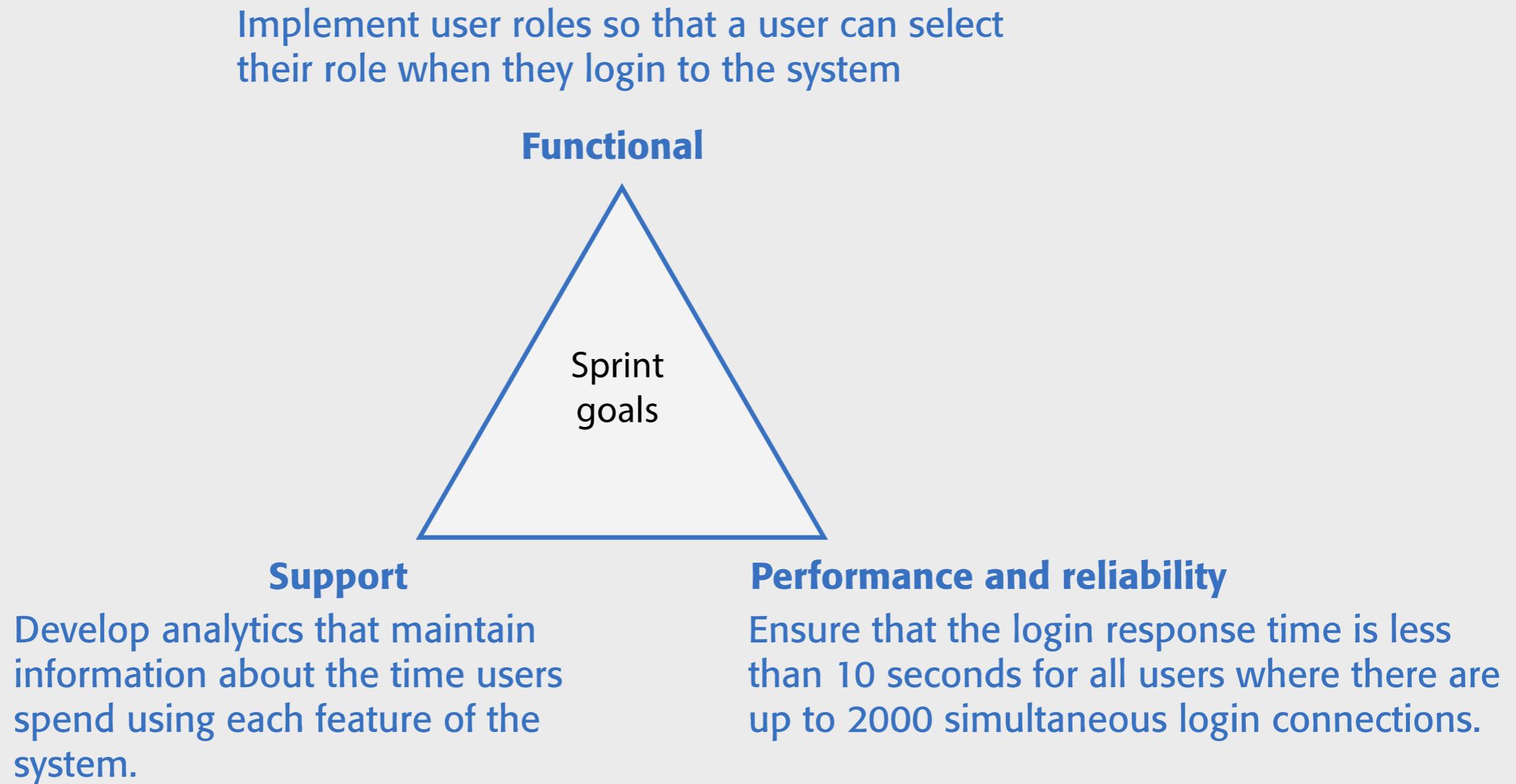
Figure 2.7 Sprint activities



Sprint planning

- Establish an agreed sprint goal
 - Sprint goals may be focused on software functionality, support or performance and reliability,..
- Decide on the list of items from the product backlog that should be implemented
- Create a sprint backlog.
 - This is a more detailed version of the product backlog that records the work to be done during the sprint

Figure 2.8 Sprint goals



Sprint planning

- In a sprint plan, the team decides which items in the product backlog should be implemented during that sprint.
 - Key inputs are the effort estimates associated with PBIs and the team's velocity
- The output of the sprint planning process is a sprint backlog.
 - The sprint backlog is a breakdown of PBIs to show the what is involved in implementing the PBIs chosen for that sprint.
- During a sprint, the team have daily meetings (scrums) to coordinate their work.

Table 2.8 Scrums

A scrum is a short, daily meeting that is usually held at the beginning of the day. During a scrum, all team members share information, describe their progress since the previous day's scrum, problems that have arisen and plans for the coming 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.

Scrum meetings should be short and focused. To dissuade team members from getting involved in long discussions, they are sometimes organized as 'stand-up' meetings where there are no chairs in the meeting room.

During a scrum, the sprint backlog is reviewed. Completed items are removed from it. New items may be added to the backlog as new information emerges. The team then decide who should work on sprint backlog items that day.

Agile activities

- Scrum does not suggest the technical agile activities that should be used. However, I think there are two practices that should always be used in a sprint.

- ***Test automation***

As far as possible, product testing should be automated. You should develop a suite of executable tests that can be run at any time.

- ***Continuous integration***

Whenever anyone makes changes to the software components they are developing, these components should be immediately integrated with other components to create a system. This system should then be tested to check for unanticipated component interaction problems.

Table 2.9 Code completeness checklist

Reviewed

The code has been reviewed by another team member who has checked that it meets agreed coding standards, is understandable, includes appropriate comments, and has been refactored if necessary.

Unit tested

All unit tests have been run automatically and all tests have executed successfully.

Integrated

The code has been integrated with the project codebase and no integration errors have been reported.

Integration tested

All integration tests have been run automatically and all tests have executed successfully.

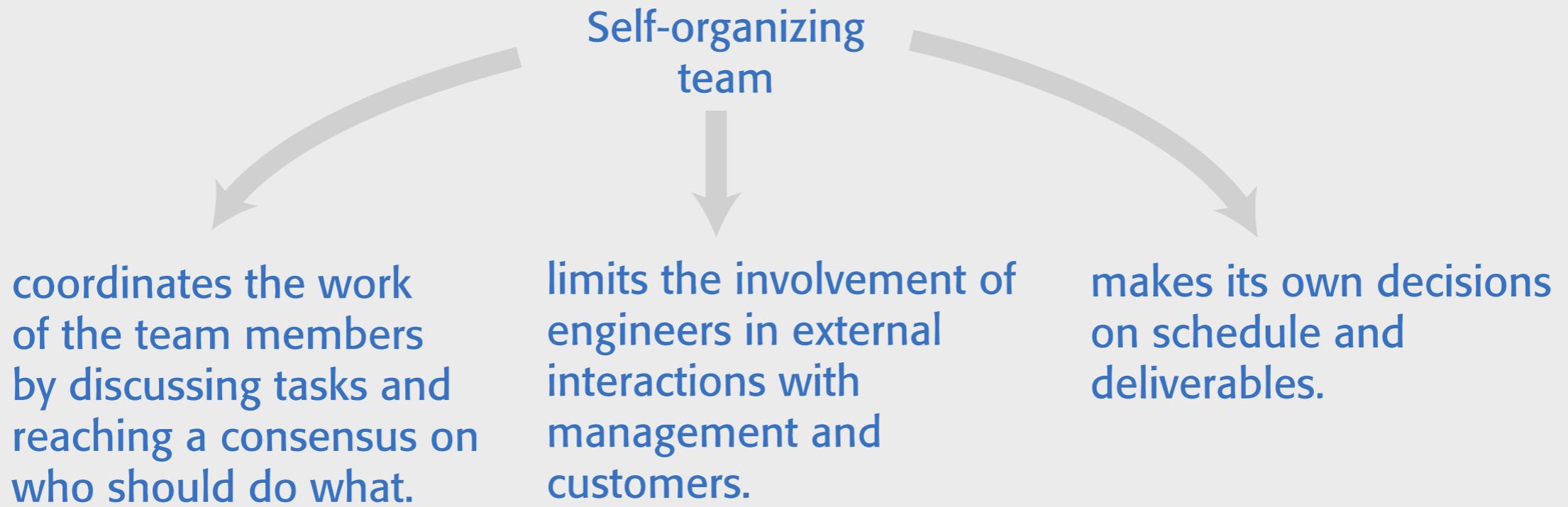
Accepted

Acceptance tests have been run if appropriate and the product owner or the development team have confirmed that the product backlog item has been completed.

Sprint reviews

- At the end of each sprint, there is a review meeting, which involves the whole team. This meeting:
 - reviews whether or not the sprint has met its goal.
 - sets out any new problems and issues that have emerged during the sprint.
 - is a way for a team to reflect on how they can improve the way they work.
- The product owner has the ultimate authority to decide whether or not the goal of the print has been achieved. They should confirm that the implementation of the selected product backlog items is complete.
- The sprint review should include a process review, in which the team reflects on its own way of working and how Scrum has been used.
 - The aim is to identify ways to improve and to discuss how to use Scrum more productively.

Figure 2.9 Self-organizing teams



Team size and composition

- The ideal Scrum team size is between 5 and 8 people.
 - Teams have to tackle diverse tasks and so usually require people with different skills, such as networking, user experience, database design and so on.
 - They usually involve people with different levels of experience.
 - A team of 5-8 people is large enough to be diverse yet small enough to communicate informally and effectively and to agree on the priorities of the team.
- The advantage of a self-organizing team is that it can be a cohesive team that can adapt to change.
 - Because the team rather than individuals take responsibility for the work, they can cope with people leaving and joining the team.
 - Good team communication means that team members inevitably learn something about each other's areas

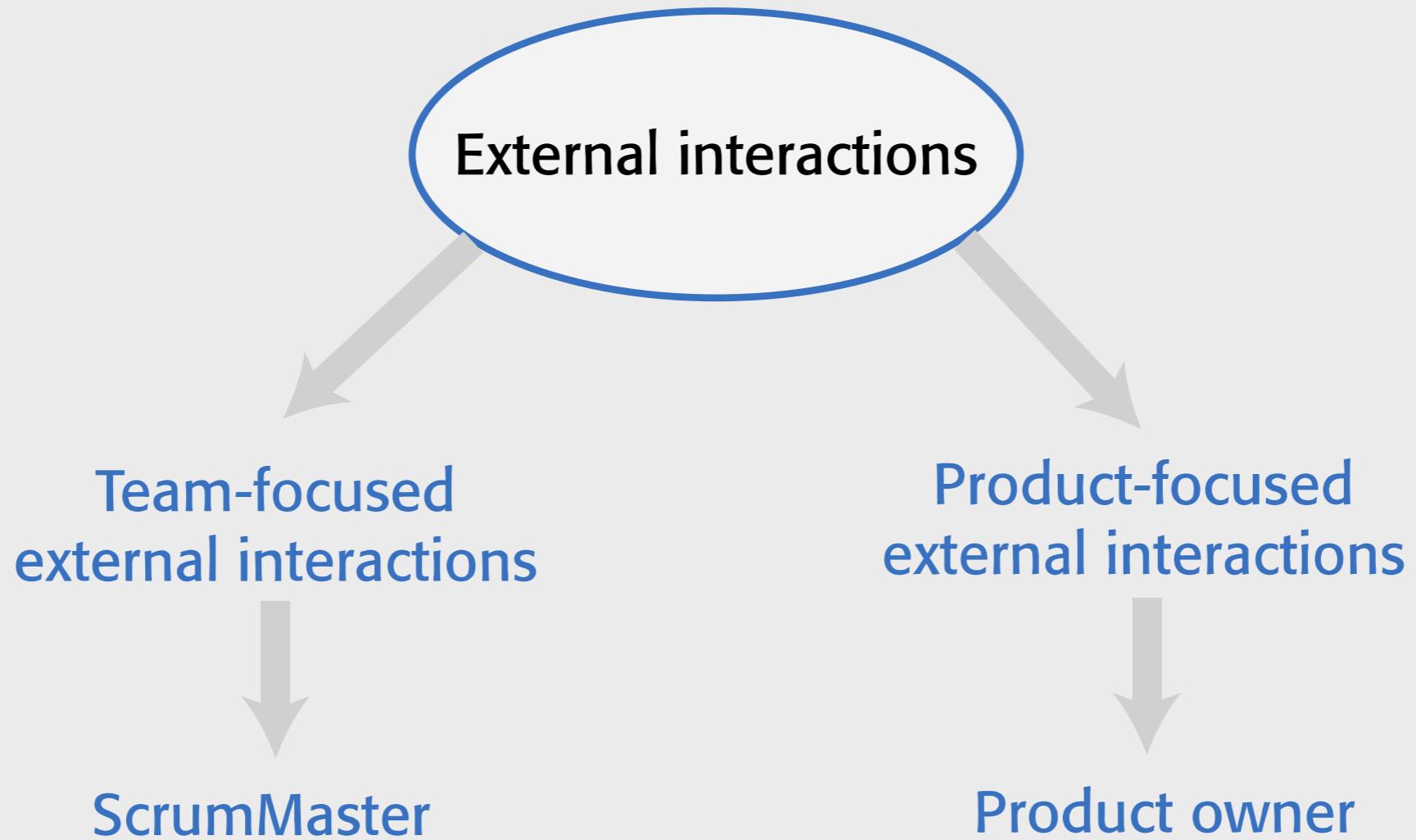
Team coordination

- The developers of Scrum assumed that teams would be co-located. They would work in the same room and could communicate informally.
 - Daily scrums mean that the team members know what's been done and what others are doing.
- However, the use of daily scrums as a coordination mechanism is based on two assumptions that are not always correct:
 - Scrum assumes that the team will be made up of full-time workers who share a workspace. In reality, team members may be part-time and may work in different places. For a student project team, the team members may take different classes at different times.
 - Scrum assumes that all team members can attend a morning meeting to coordinate the work for the day. However, some team members may work flexible hours (e.g. because of childcare responsibilities) or may work on several projects at the same time.

External interactions

- External interactions are interactions that team members have with people outside of the team.
- In Scrum, the idea is that developers should focus on development and only the ScrumMaster and Product Owner should be involved in external interactions.
- The intention is that the team should be able to work on software development without external interference or distractions.

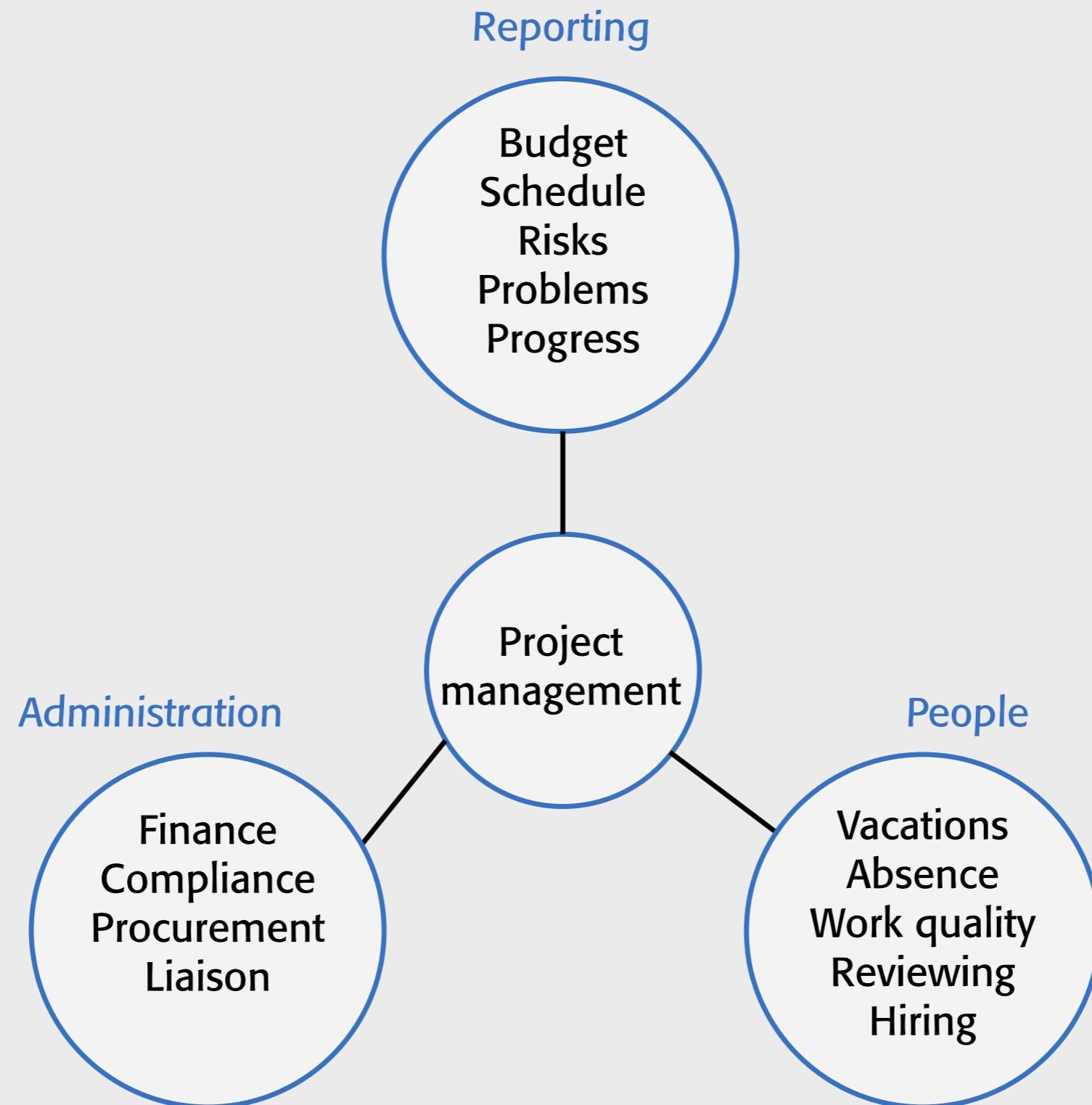
Figure 2.10 Managing external interactions



Project management

- In all but the smallest product development companies, there is a need for development teams to report on progress to company management.
- A self-organizing team has to appoint someone to take on these responsibilities.
 - Because of the need to maintain continuity of communication with people outside of the group, rotating these activities around team members is not a viable approach.
- The developers of Scrum did not envisage that the ScrumMaster should also have project management responsibilities.
 - In many companies, however, the ScrumMaster has to take on project management responsibilities.
 - They know the work going on and are in the best position to provide accurate information and project plans and progress.

Figure 2.11 Project management responsibilities



Key points 1

- The best way to develop software products is to use agile software engineering methods that are geared to rapid product development and delivery.
- Agile methods are based around iterative development and the minimization of overheads during the development process.
- Extreme programming (XP) is an influential agile method that introduced agile development practices such as user stories, test-first development and continuous integration. These are now mainstream software development activities.
- Scrum is an agile method that focuses on agile planning and management. Unlike XP, it does not define the engineering practices to be used. The development team may use any technical practices that they believe are appropriate for the product being developed.
- In Scrum, work to be done is maintained in a product backlog – a list of work items to be completed. Each increment of the software implements some of the work items from the product backlog.

Key points 2

- Sprints are fixed-time activities (usually 2–4 weeks) where a product increment is developed. Increments should be ‘potentially shippable’ i.e. they should not need further work before they are delivered.
- A self-organizing team is a development team that organizes the work to be done by discussion and agreement amongst team members.
- Scrum practices such as the product backlog, sprints and self-organizing teams can be used in any agile development process, even if other aspects of Scrum are not used.



Архитектура ориентирана към услуги.
Микроуслуги



MicroServices evolution

1990s and earlier

Coupling

Pre-SOA (monolithic)

Tight coupling



2000s

Traditional SOA

Looser coupling



2010s

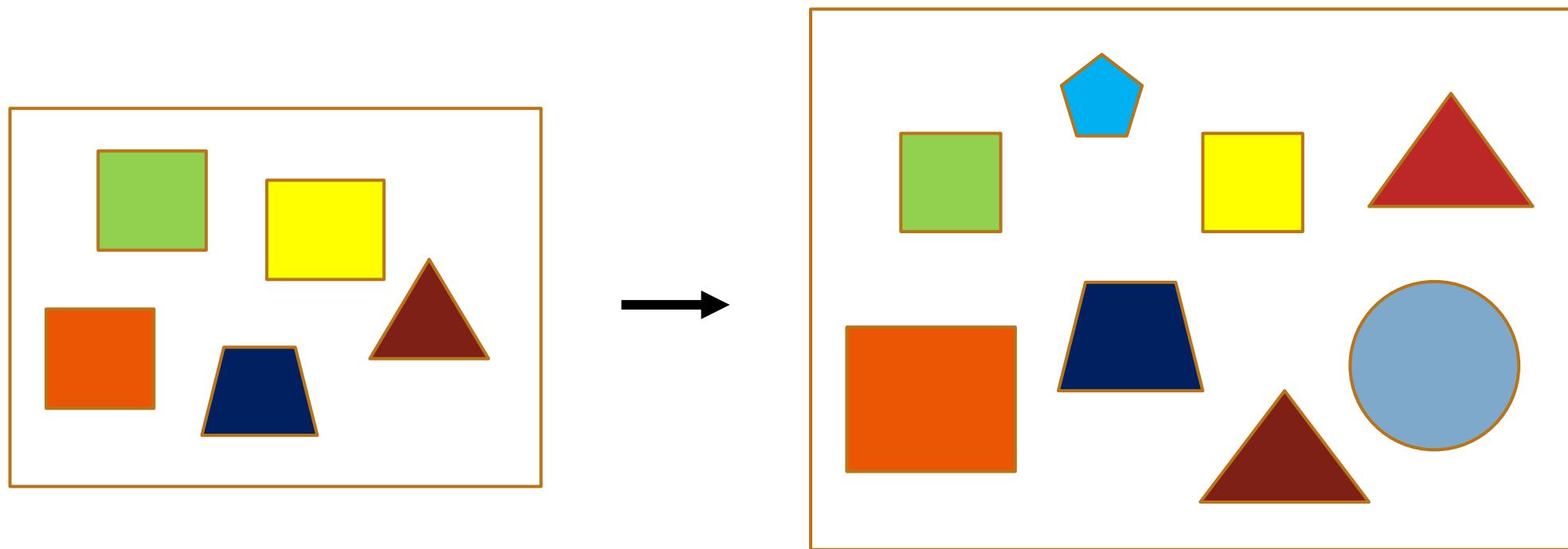
Microservices

Decoupled

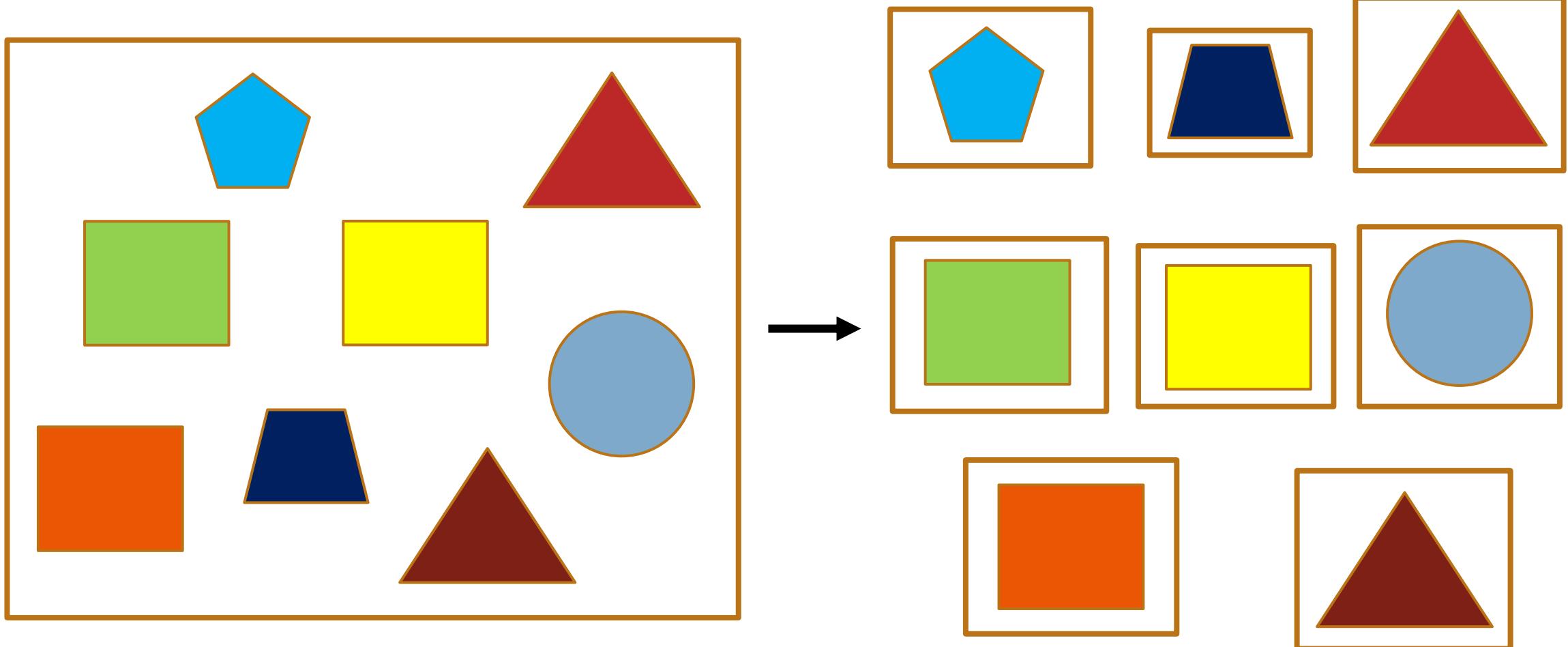


Monolith

- ▶ Добавянето на нови функционалности, с течение на времето, увеличава значимо размера и сложността му.
- ▶ Усилието за добавяне на нови функции към монолита също нараства значително.



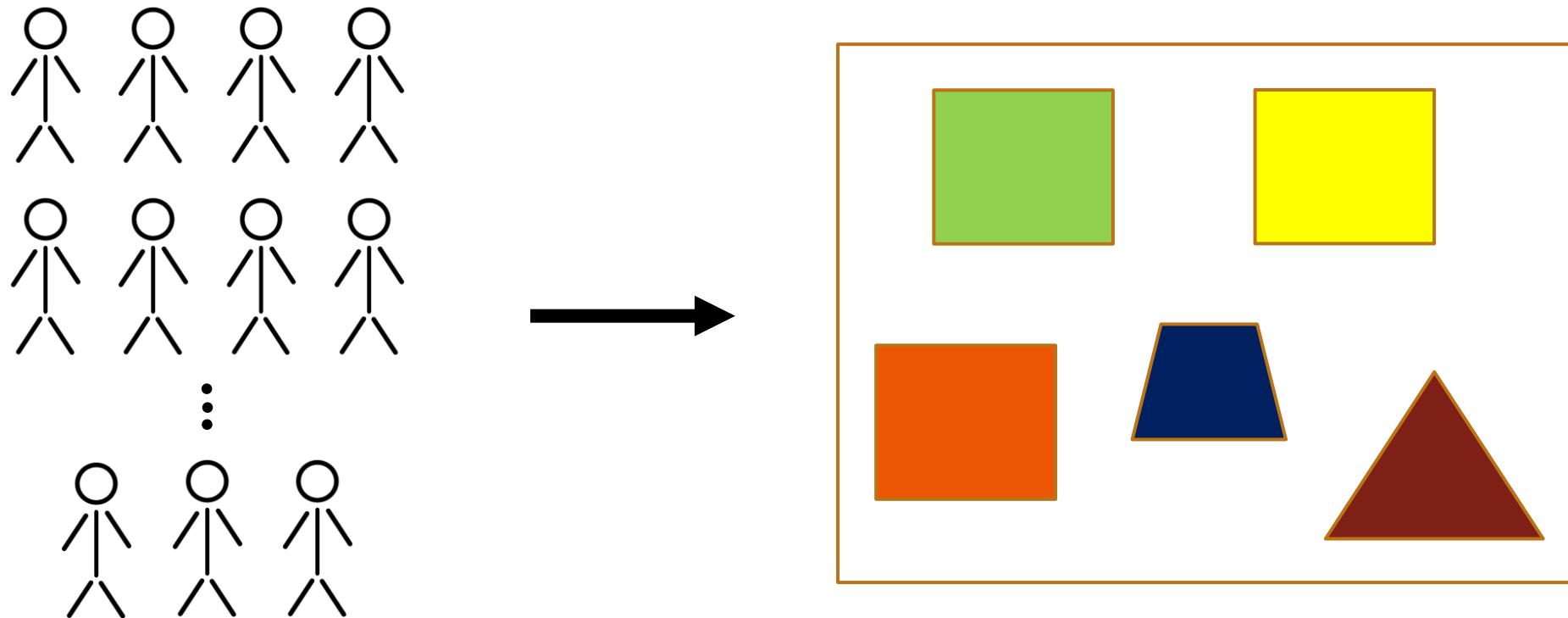
Monolith vs MicroServices



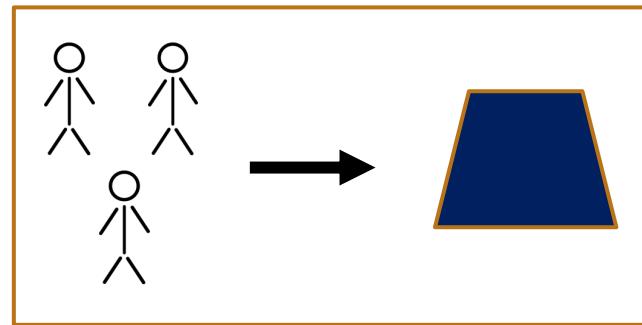
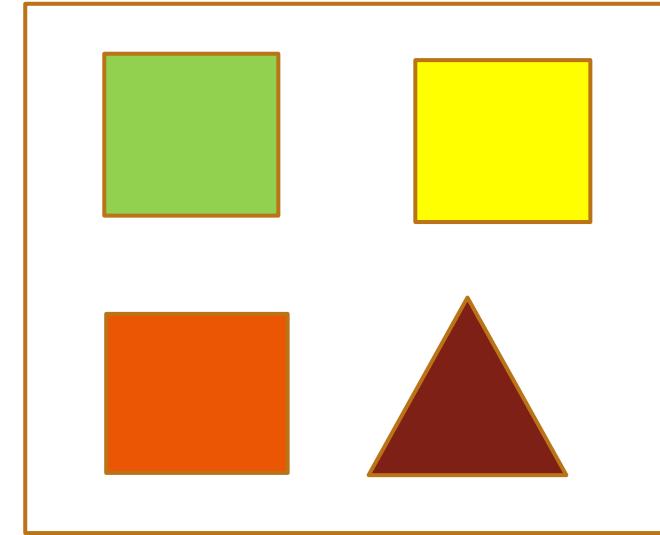
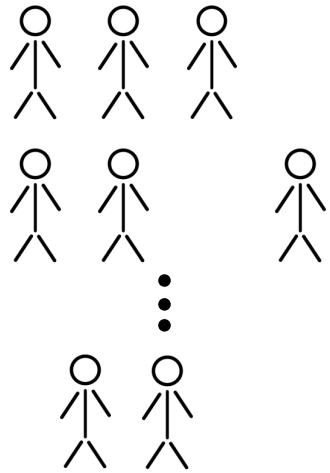
MicroServices

- ▶ Типични свойства на архитектурата на микроуслуги са:
 - ▶ *те са (в контекста им) малки.*
 - ▶ *могат да се „деплоинат“ независимо.*
 - ▶ *не са зависими от дадена технология.*
 - ▶ *кумуницират стриктно чрез своите API-та.*

Team Organization



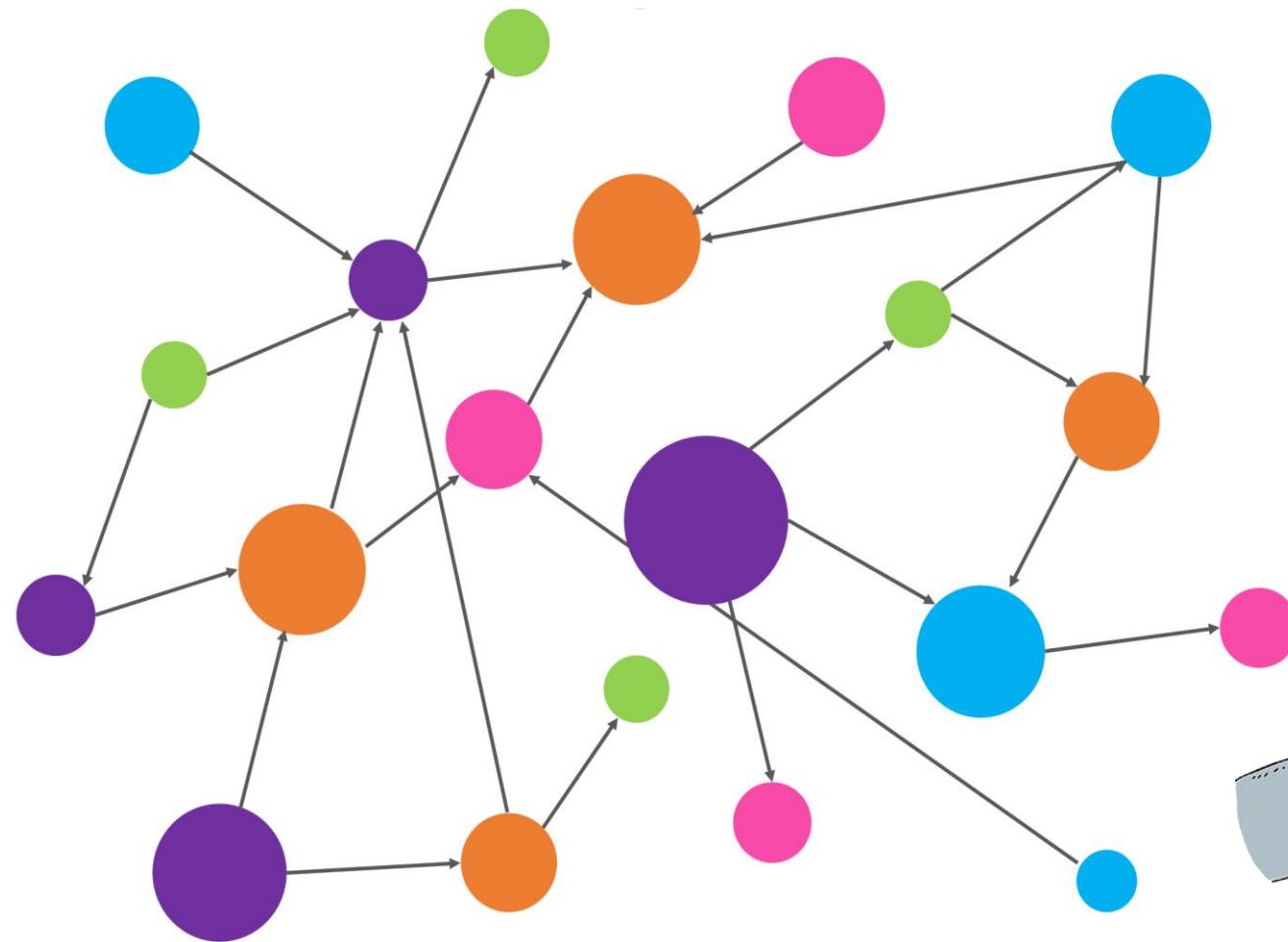
Team Organization



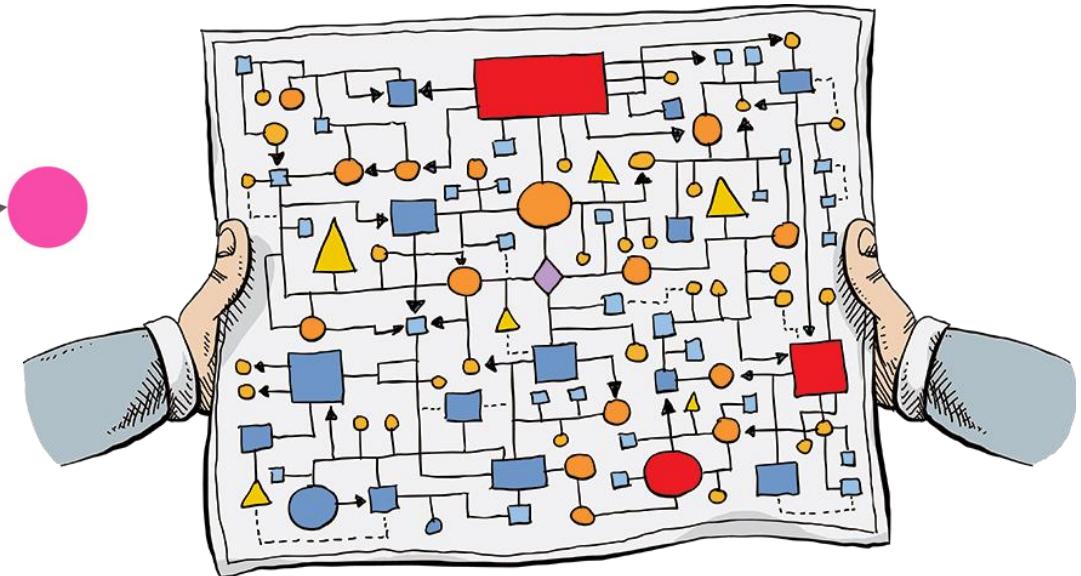
Team Organization

- ▶ Екипа се развива по-лесно – по-тесен обхват (*narrow scope*)
- ▶ Намалява риска за *bottleneck*
- ▶ По-лесно за нови разработчици да се присъединят
- ▶ Всеки екип може да взема изолирани архитектурни решения (*isolated impact*)
- ▶ Екипите могат да управляват работата си по различен начин
- ▶ Децентрализирано вземане на решения

MicroServices – Difficulties



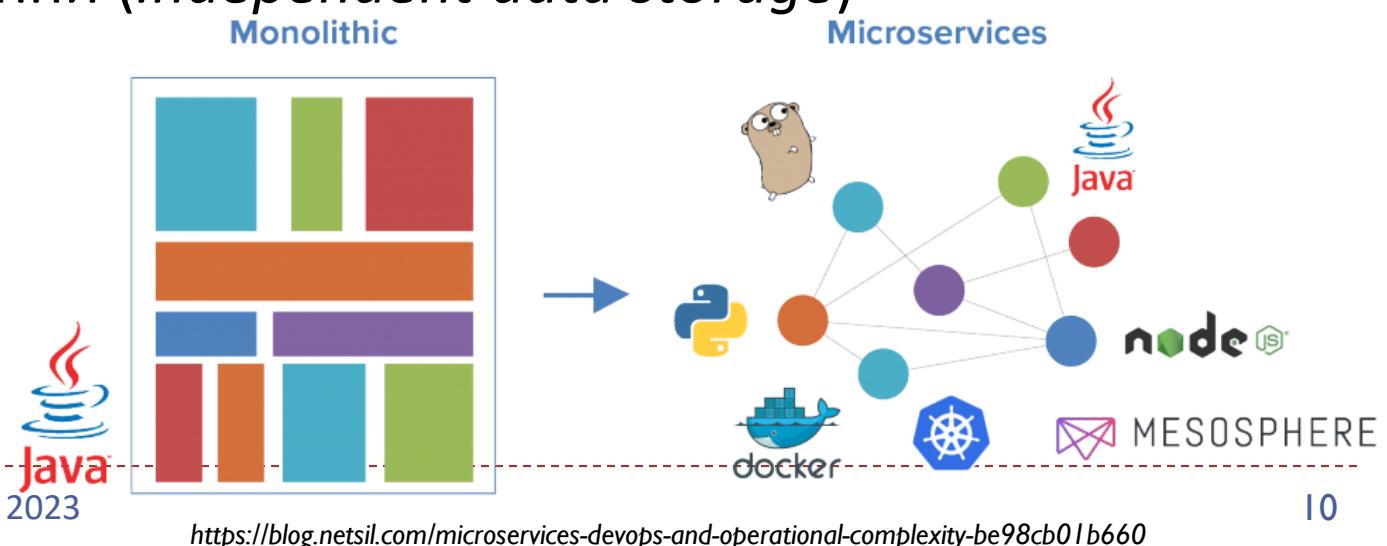
<https://medium.com/@marcus.cavalcanti/lessons-learned-about-run-microservices-b360347c8a77>



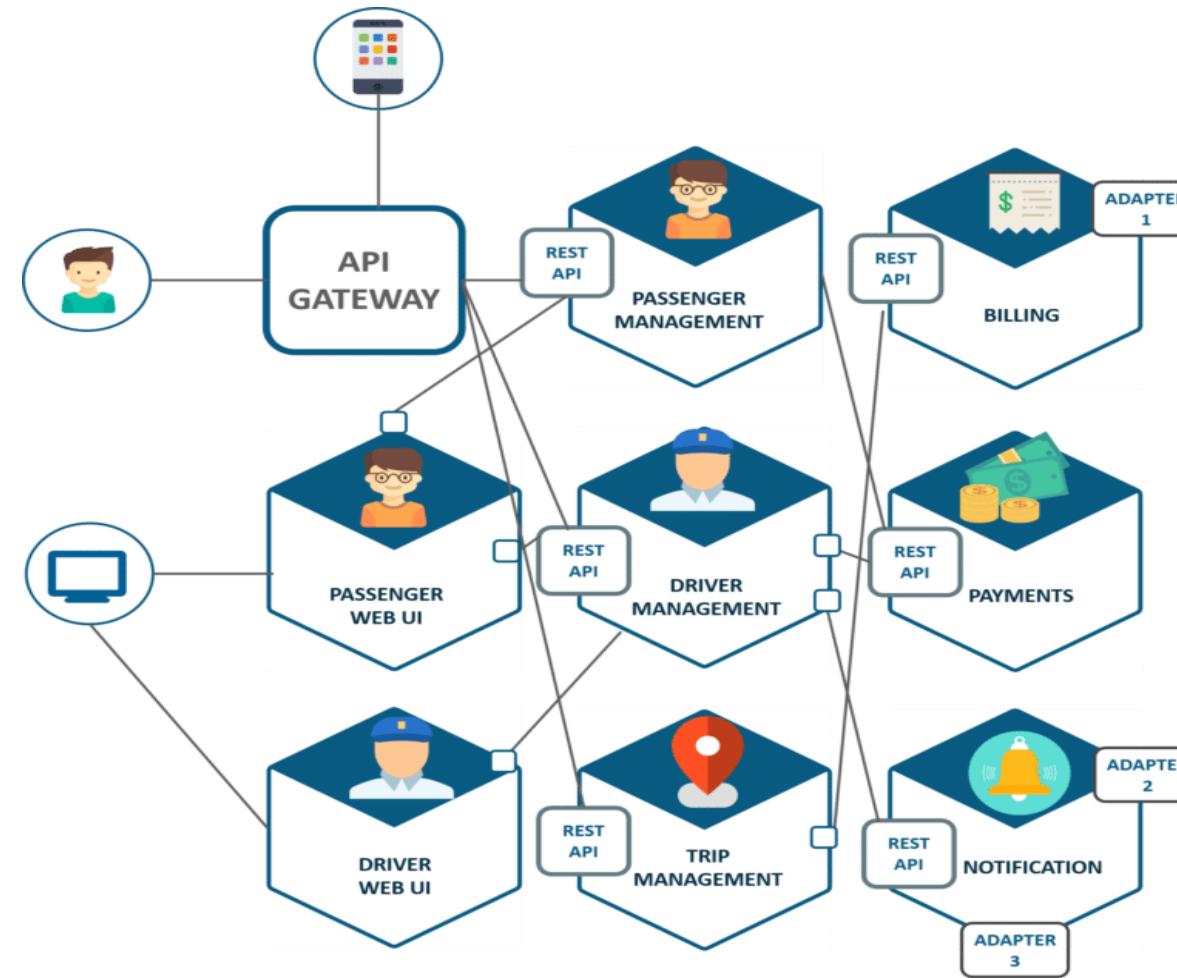
<https://medium.com/@vladikk.com/tackling-complexity-in-microservices-6253c2aad007>

MicroServices – Difficulties

- ▶ Всяка комуникация е мрежова комуникация
- ▶ Мрежата се третира като ненадежна (*design for failure*)
- ▶ Разпределението на логика в независимите системи е трудно
- ▶ По-труден анализ на логове в сравнение с монолита
- ▶ Независимо съхранение на данни (*Independent data storage*)
- ▶ Границите (*boundaries*) на микроуслугите трябва да се избират внимателно

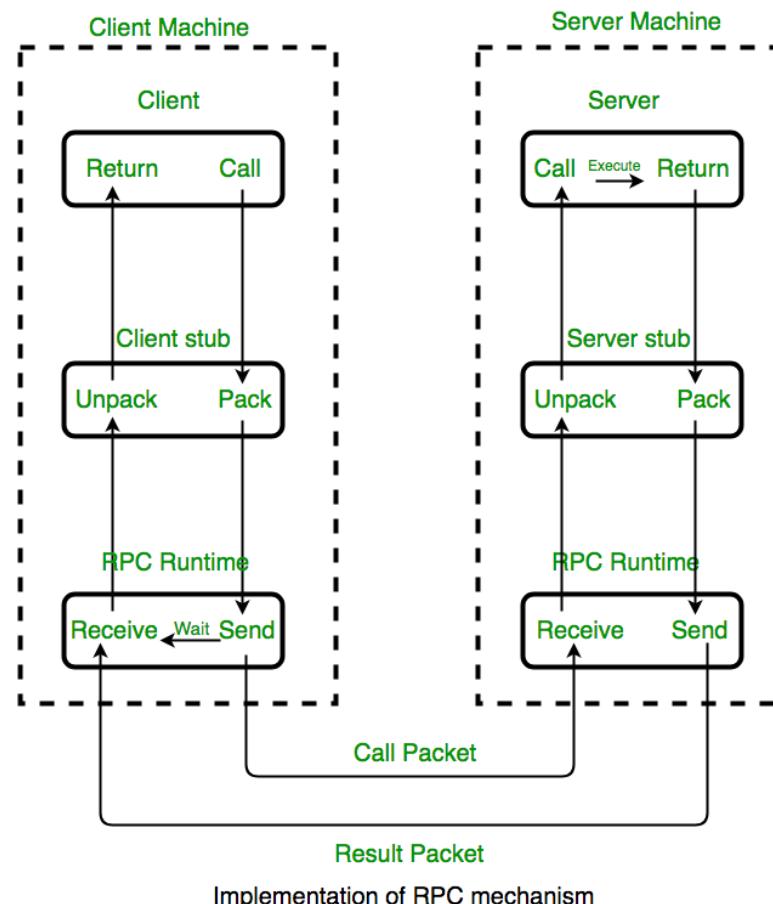


MicroServices – Communication



Synchronous Communication

Remote Procedure Call – RPC



Synchronous Communication

Service Discovery in RPC



- ▶ Има различни начини зависи от нуждите ни.
- ▶ Един от най-широко използвани и прости е DNS.
- ▶ Много различни други опции като Apache ZooKeeper (+ scalability, - complexity), Hashicorp Consul (multi-cloud deployment)

Synchronous Communication

State in RPC

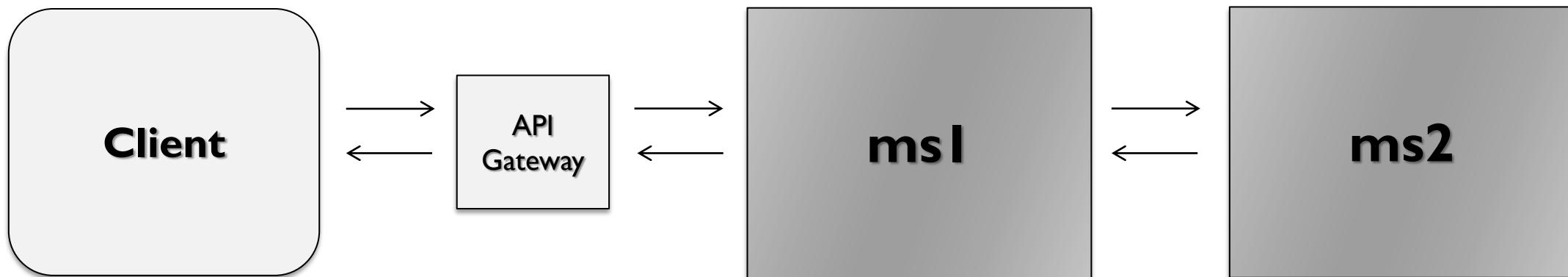
RPC могат да се разделят на 2 категории относно тяхното състояние (state):

- ▶ Stateless : в този случай просто се извършват някои изчисления и се връща резултат, без значение от състоянието на микроуслугата.

- ▶ Stateful : в този случай изчисленията се извършват с контекста на предишни транзакции и текущата транзакция може да бъде повлияна от случилото се по време на предишни транзакции.

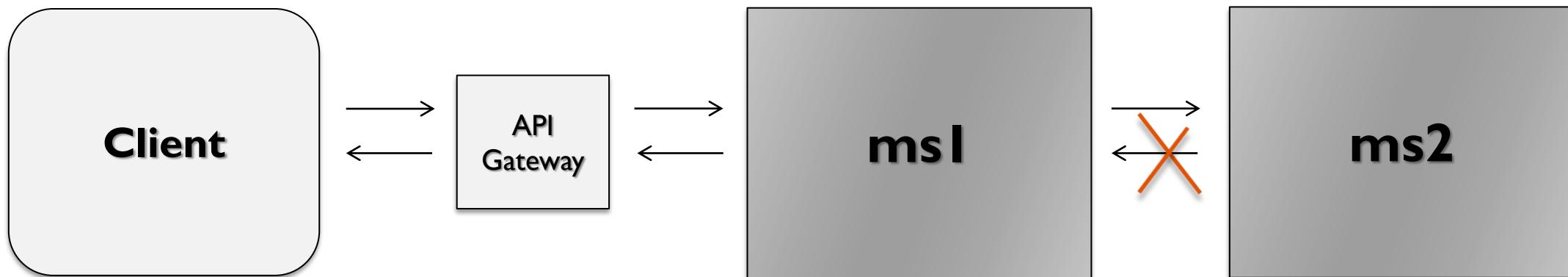
Stateful RPC

Idempotence : свойство на система, която не се променя, ако извършите една и съща операция многократно.

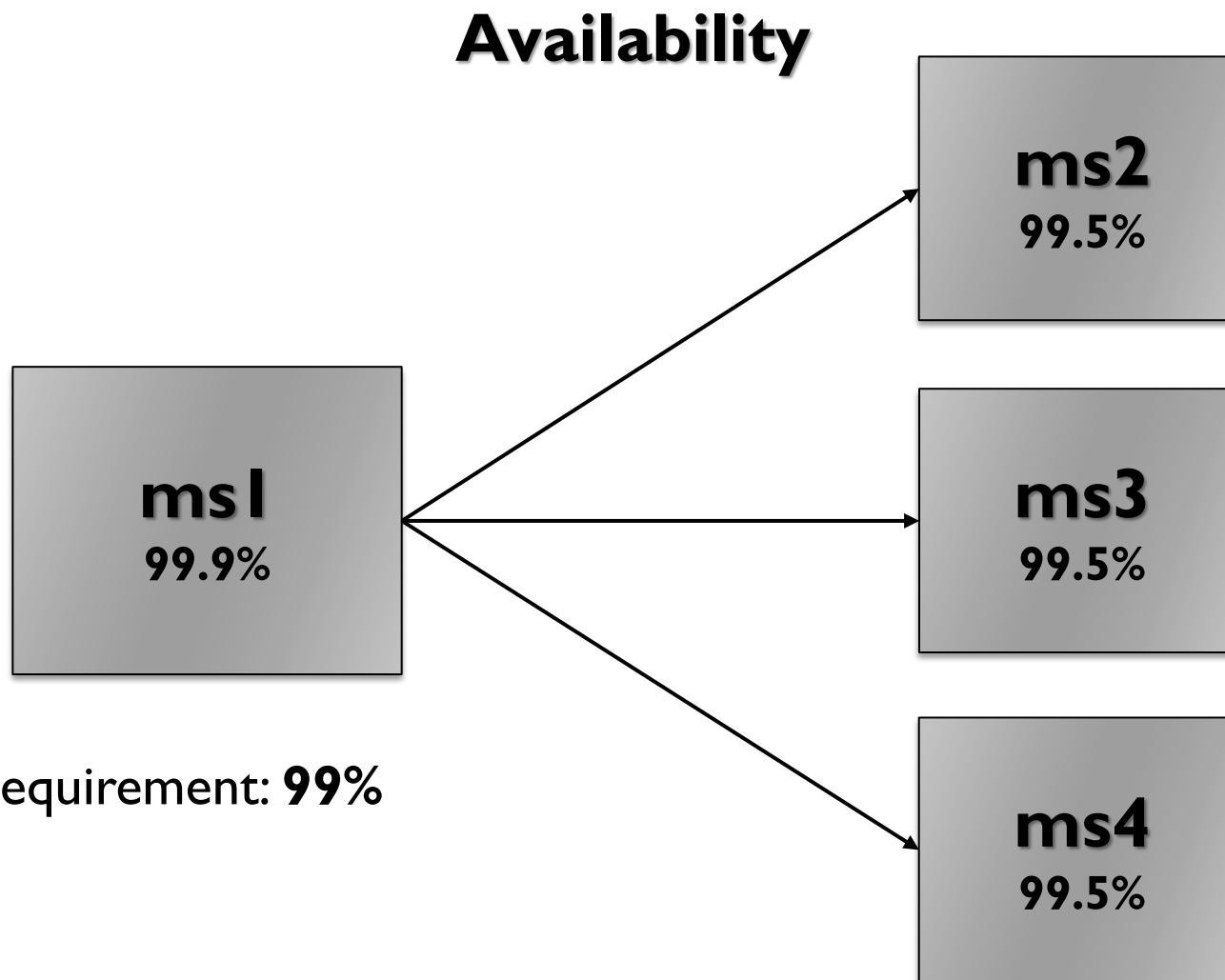


Stateful RPC

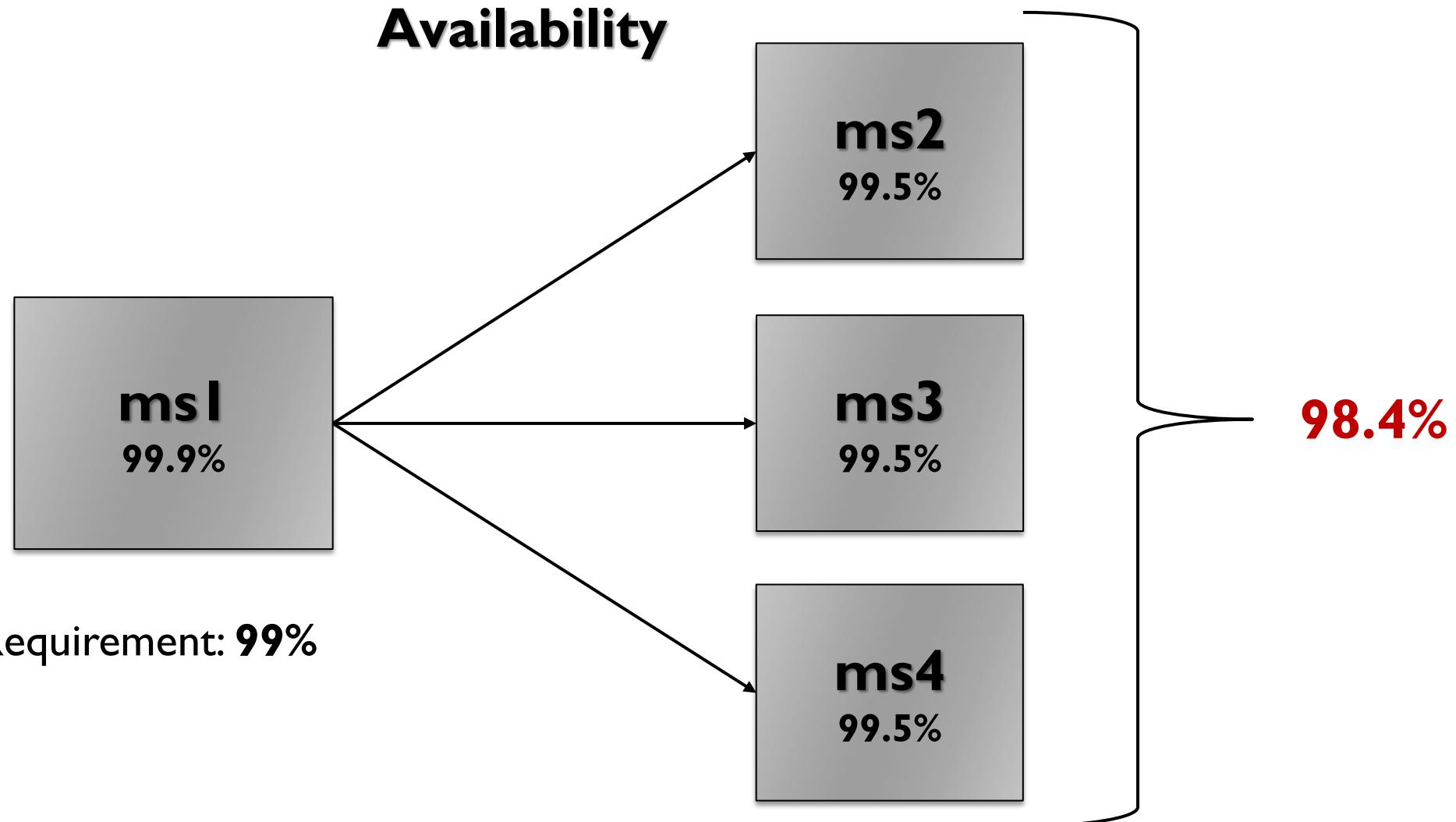
Idempotence : свойство на система, която не се променя, ако извършите една и съща операция многократно.



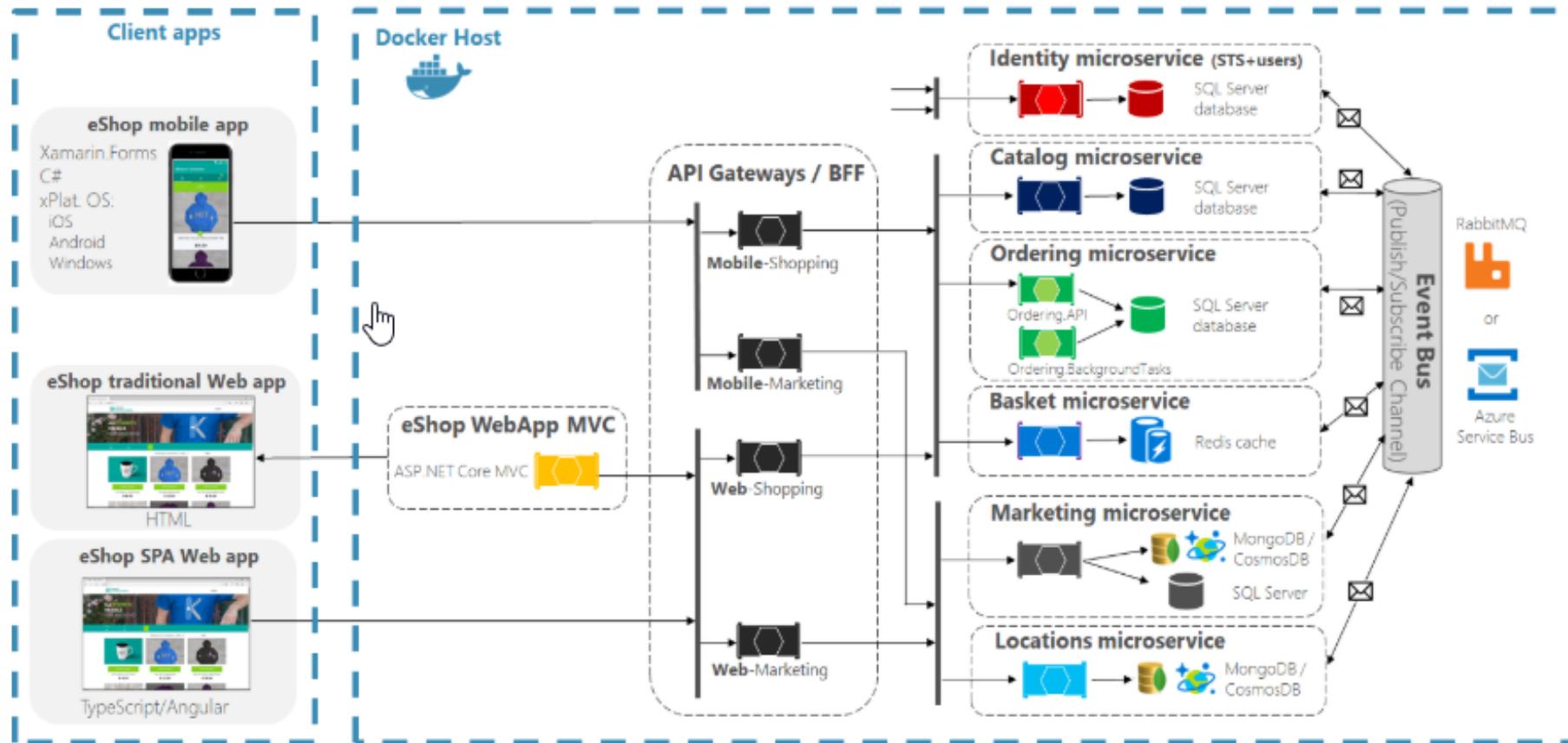
Synchronous Communication



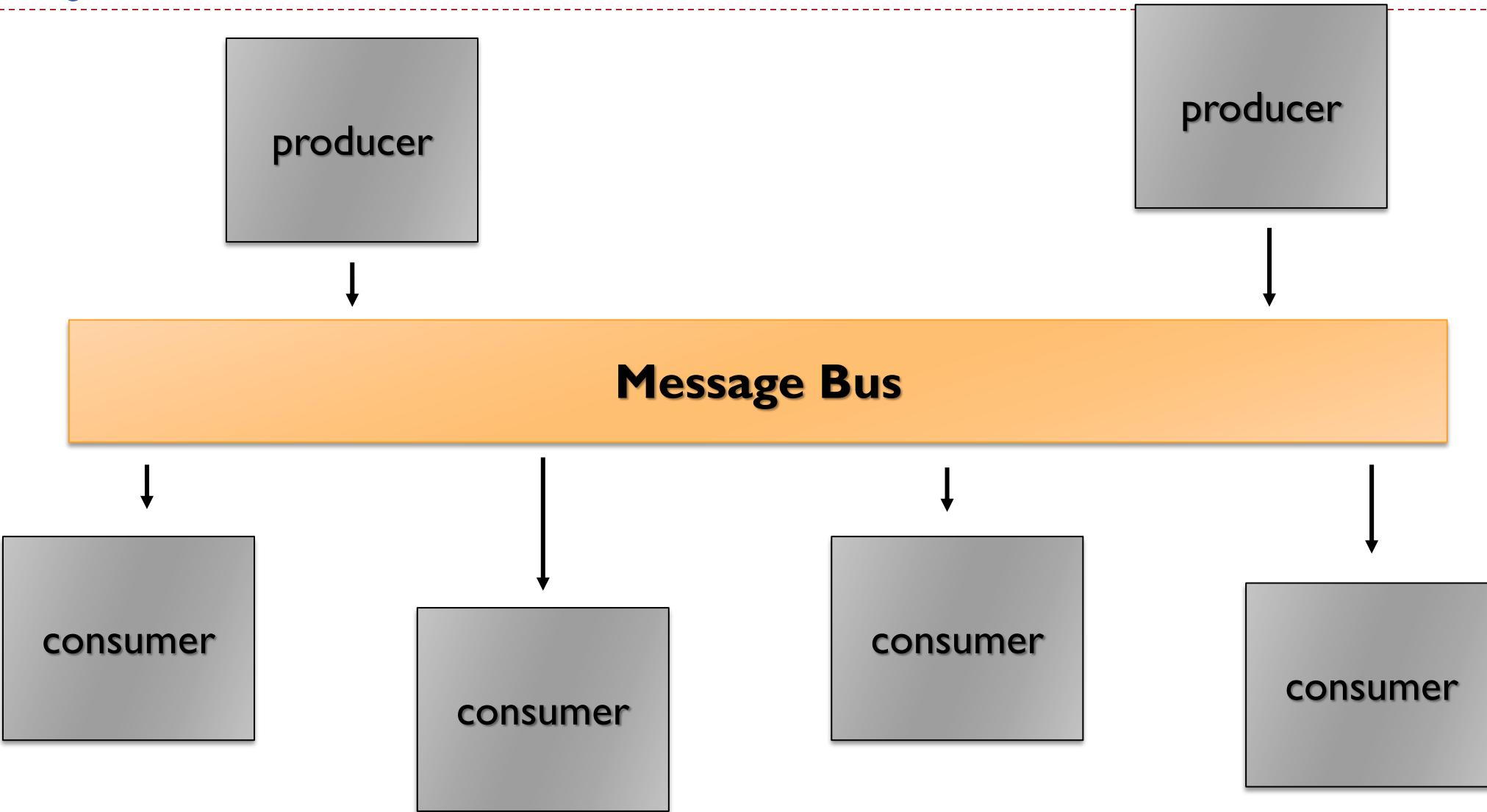
Synchronous Communication



Asynchronous Communication

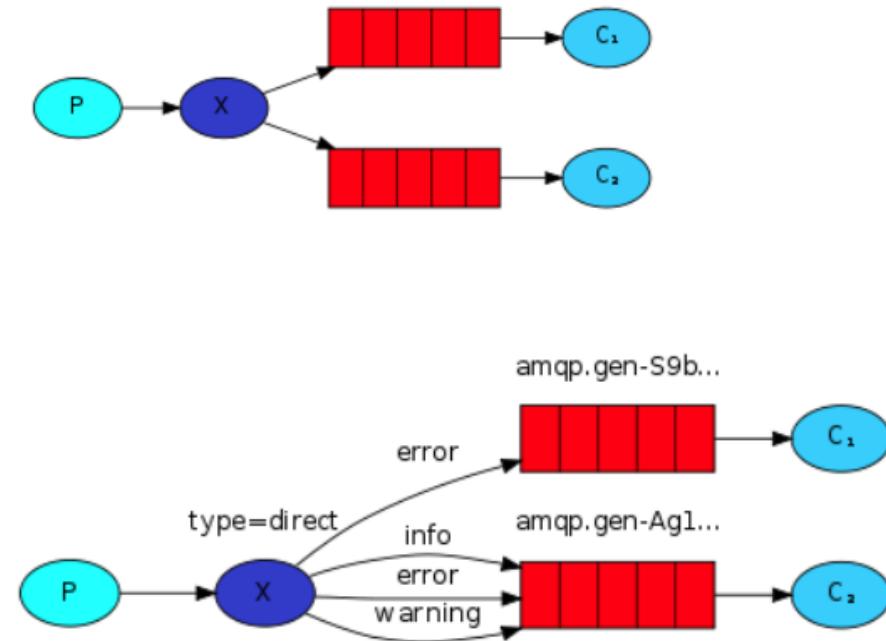
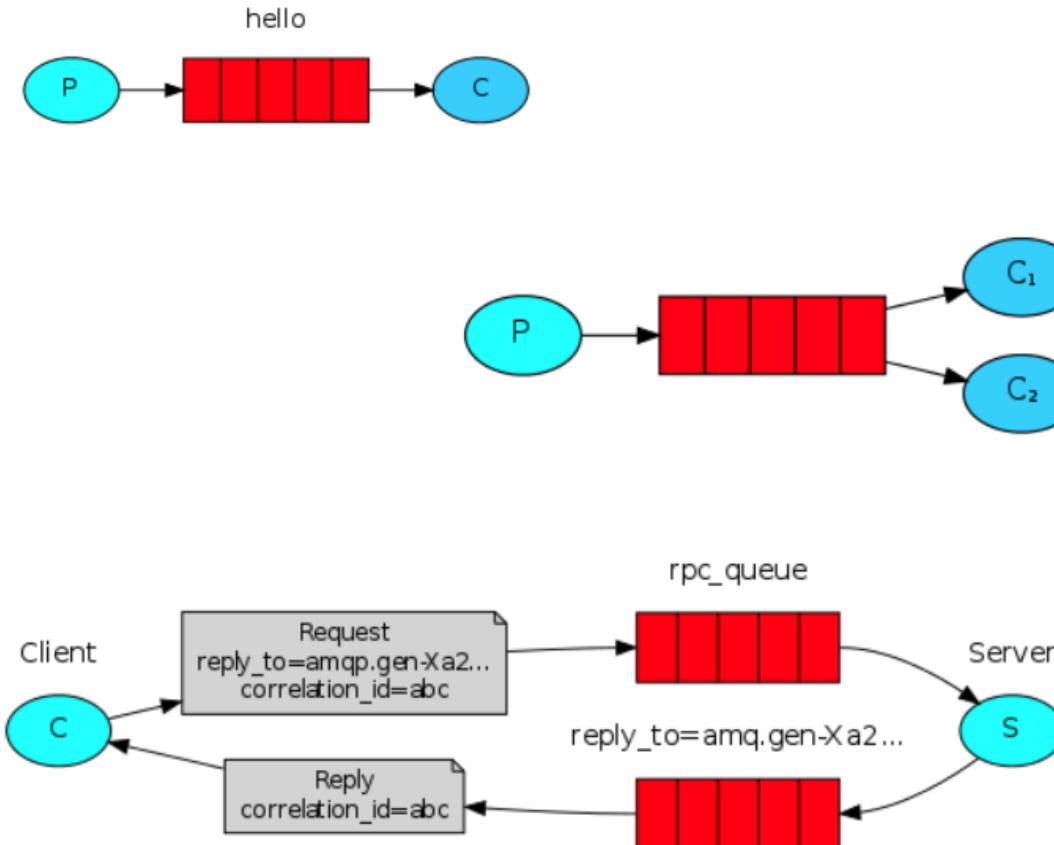


Asynchronous Communication



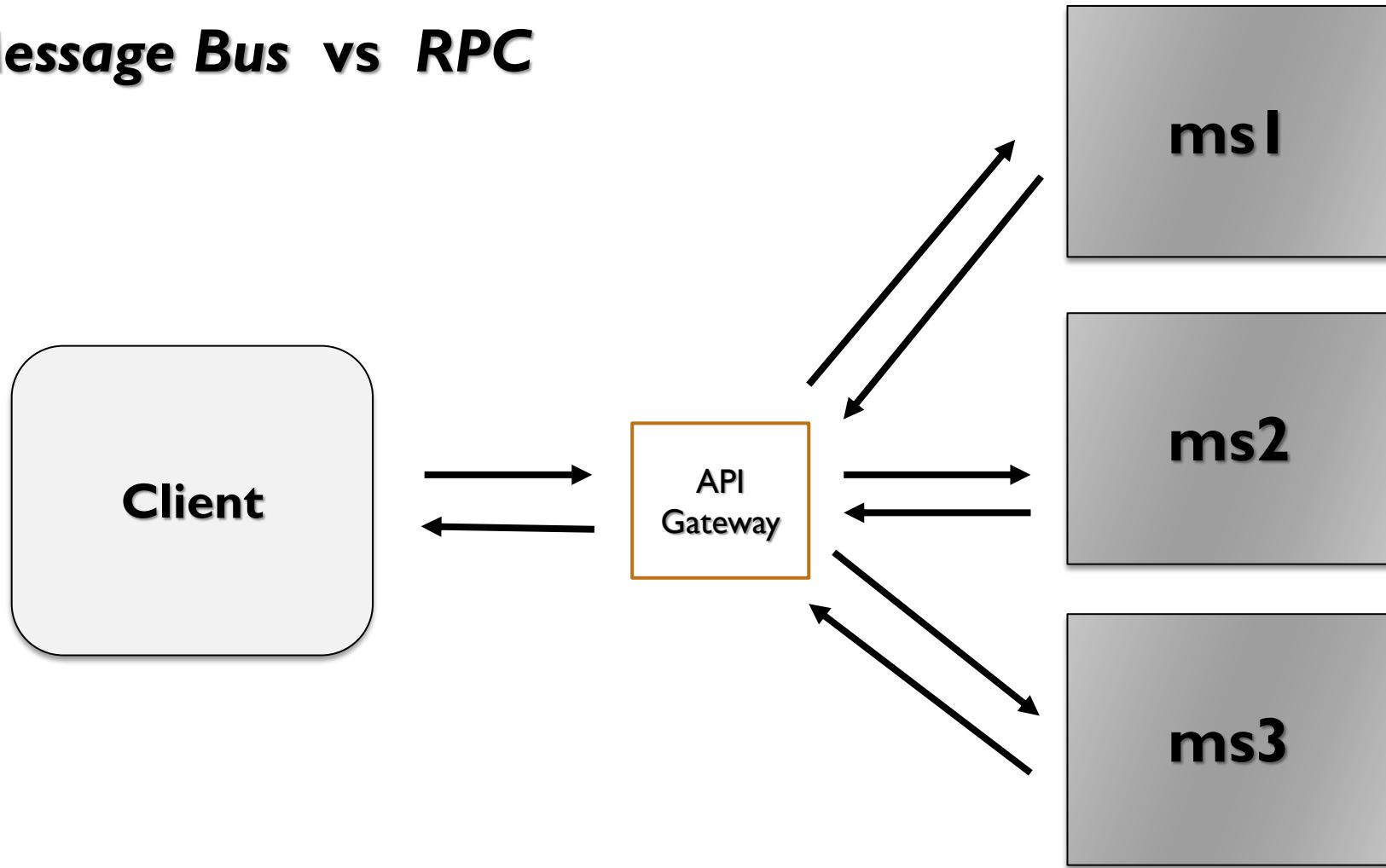
Asynchronous Communication

Queues in a Message Bus



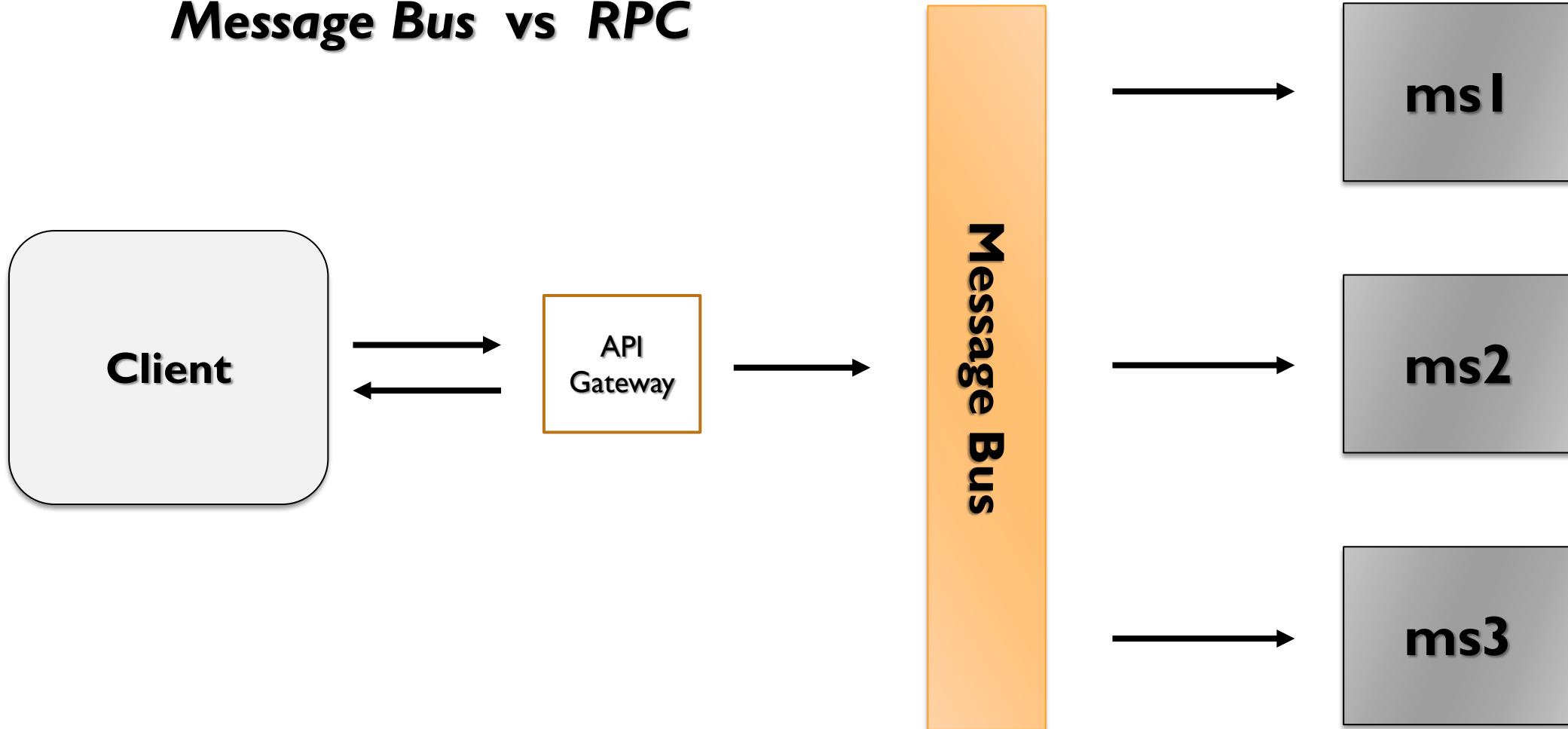
Asynchronous Communication

Message Bus vs RPC



Asynchronous Communication

Message Bus vs RPC



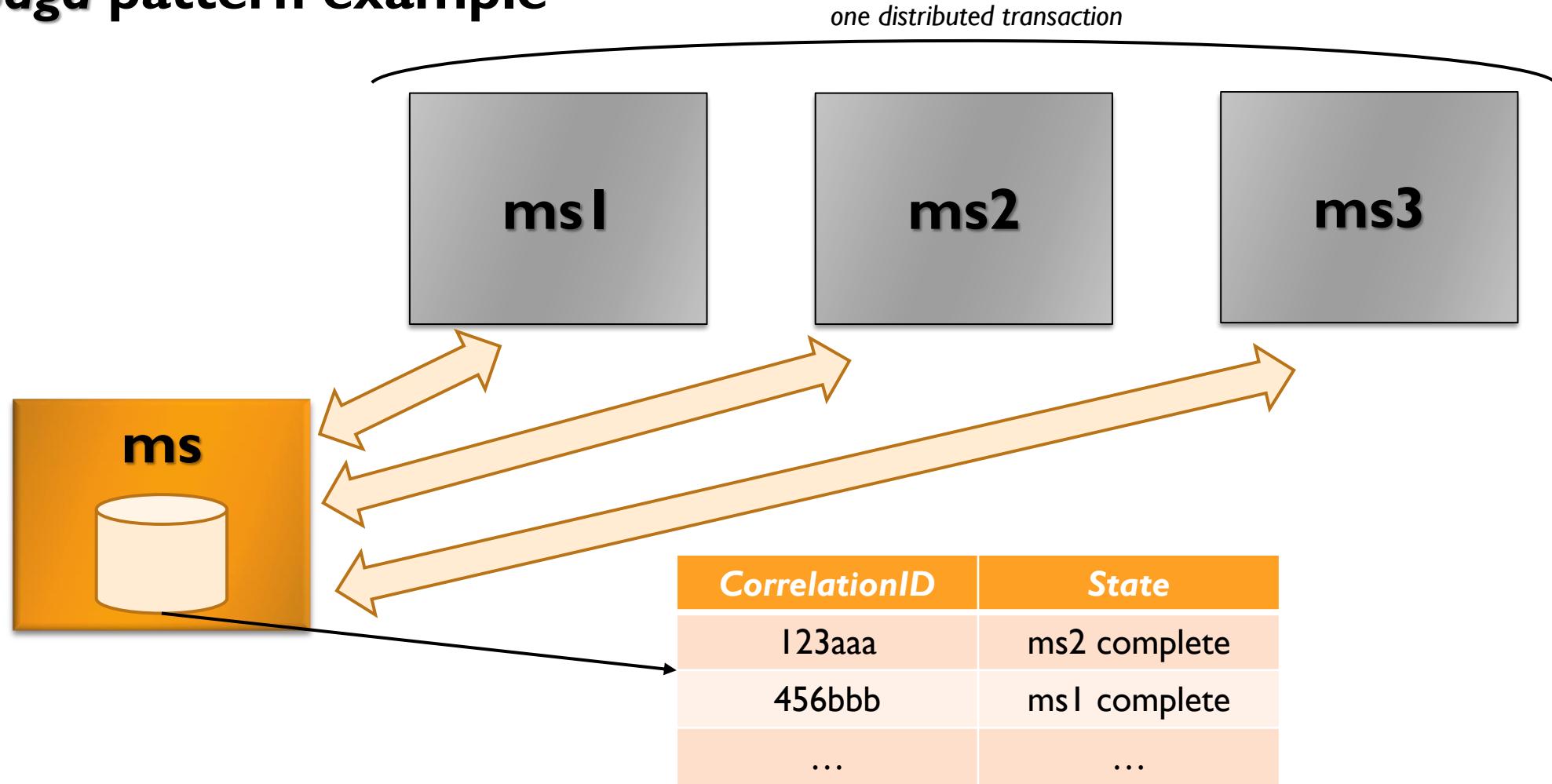
Asynchronous Communication

The *Saga* Pattern

- ▶ Всяка транзакция, която обхваща множество услуги се изпълнява като Сага (Saga). Сагата е поредица от локални (local) транзакции.
- ▶ Всяка локална транзакция актуализира базата данни и публикува съобщение за да задейства следващата локална транзакция в сагата.
- ▶ Ако една транзакция се провали, сагата изпълнява серия от компенсиращи (compensations) транзакции, които отменят промените, направени от предходните транзакции.

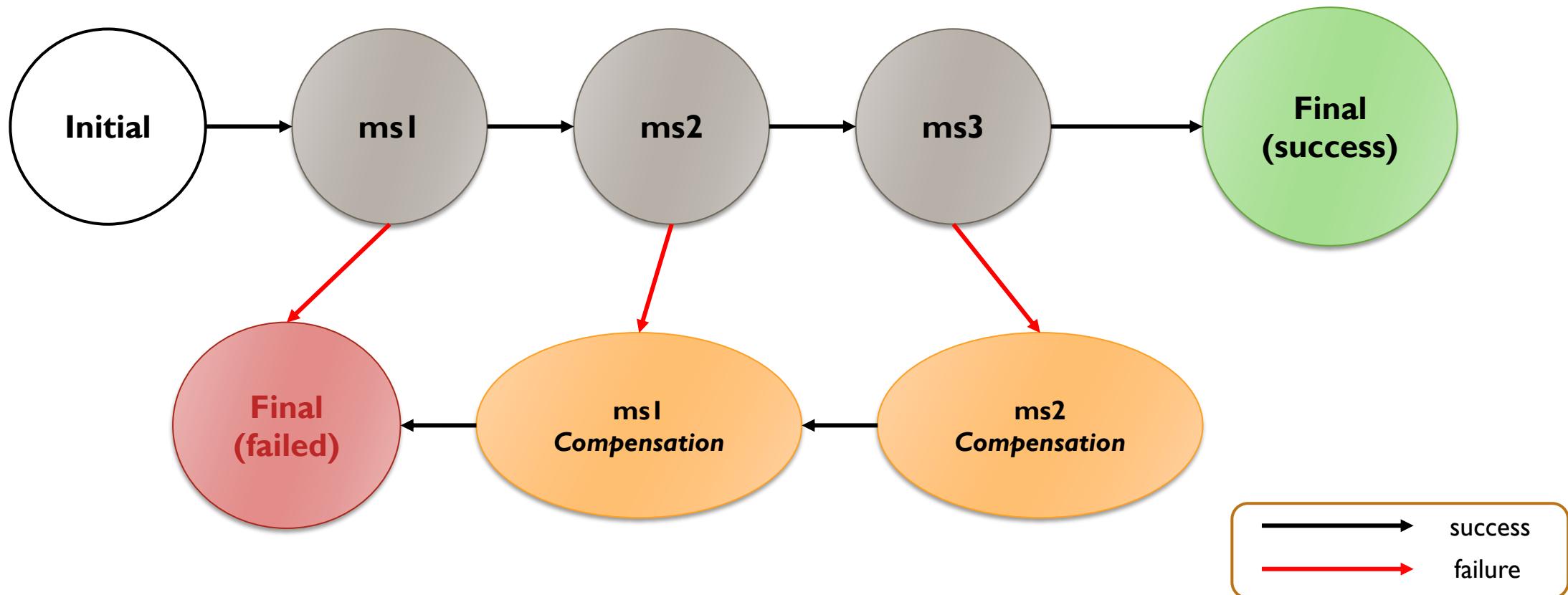
Asynchronous Communication

Saga pattern example

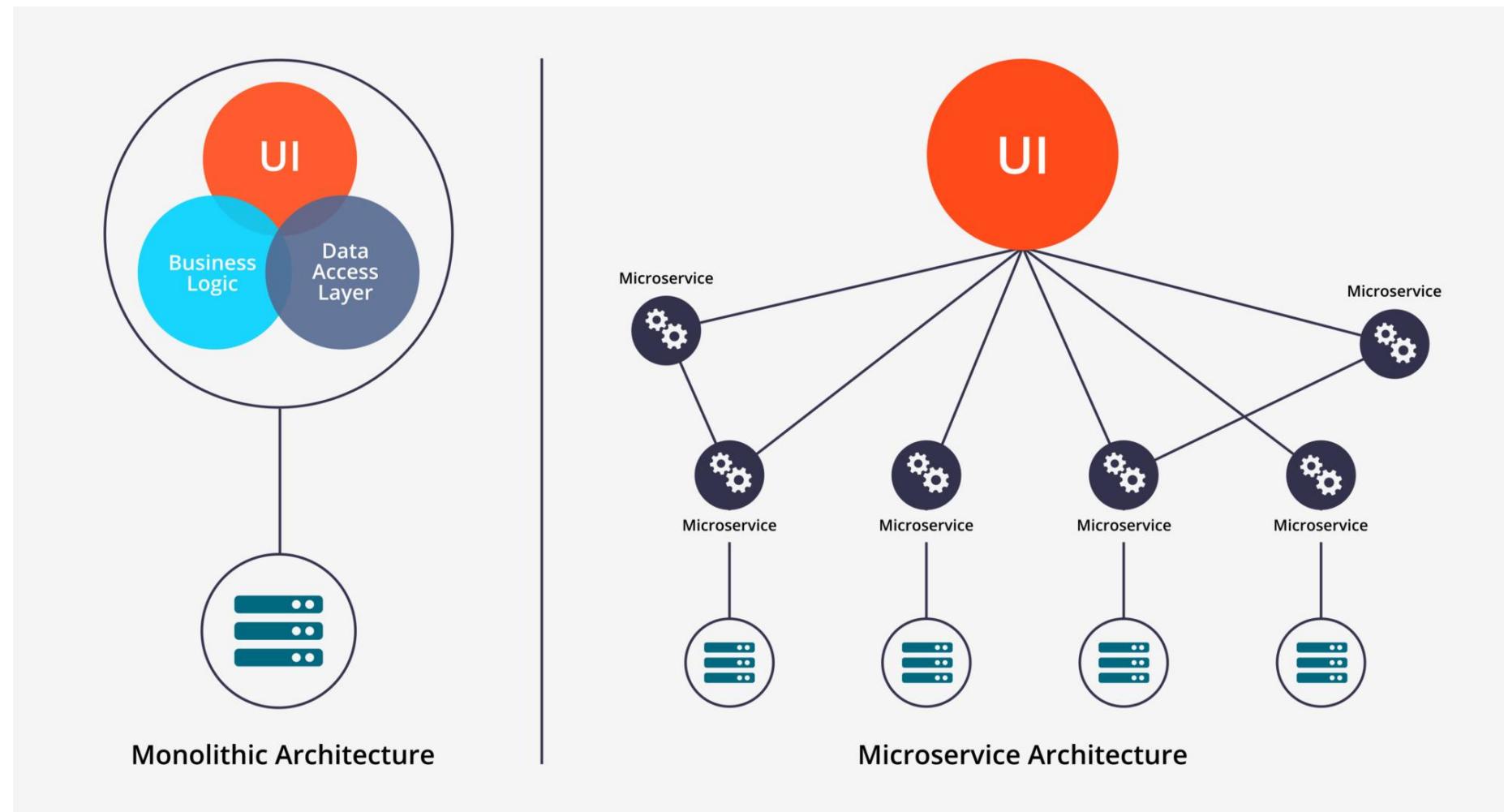


Asynchronous Communication

State Machine in Saga



Conclusion



Въпроси ?



Допълнителни материали

- ▶ I. Sommerville, *Software Engineering, 10th Ed.*, **Chapter 18**
- ▶ I. Sommerville, *Engineering Software Products*, **Chapter 6**

- ▶ M. Fowler, *Microservices* - <https://martinfowler.com/articles/microservices.html>



Проектиране на софтуерни системи. Дизайн и архитектура



Software architecture

- ▶ To create a reliable, secure and efficient product, you need to pay attention to architectural design which includes:
 - ▶ its overall organization,
 - ▶ how the software is decomposed into components,
 - ▶ the server organization
 - ▶ the technologies that you use to build the software. The architecture of a software product affects its performance, usability, security, reliability and maintainability.
- ▶ There are many different interpretations of the term ‘software architecture’.
 - ▶ Some focus on ‘architecture’ as a noun - the structure of a system and others consider ‘architecture’ to be a verb - the process of defining these structures.

The IEEE definition of software architecture

Architecture is the fundamental organization of a software system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.

Software architecture and components

- ▶ A component is an element that implements a coherent set of functionality or features.
- ▶ Software component can be considered as a collection of one or more services that may be used by other components.
- ▶ When designing software architecture, you don't have to decide how an architectural element or component is to be implemented.
- ▶ Rather, you design the component interface and leave the implementation of that interface to a later stage of the development process.

Why is architecture important?

- ▶ Architecture is important because the architecture of a system has a fundamental influence on the non-functional system properties
- ▶ Architectural design involves understanding the issues that affect the architecture of your product and creating an architectural description that shows the critical components and their relationships.
- ▶ Minimizing complexity should be an important goal for architectural designers.
 - ▶ The more complex a system, the more difficult and expensive it is to understand and change.
 - ▶ Programmers are more likely to make mistakes and introduce bugs and security vulnerabilities when they are modifying or extending a complex system..

Software Architecture

- ▶ Architecture is important because the architecture of a system *has a fundamental influence on the non-functional system properties.*
- ▶ ... *eases the communication among stakeholders.*
- ▶ ... *defines constraints on implementation.*
- ▶ ... *makes it easier to reason about and manage change.*
- ▶ ... *enables more accurate cost and schedule estimates.*
- ▶ ...

Perspectives & Views

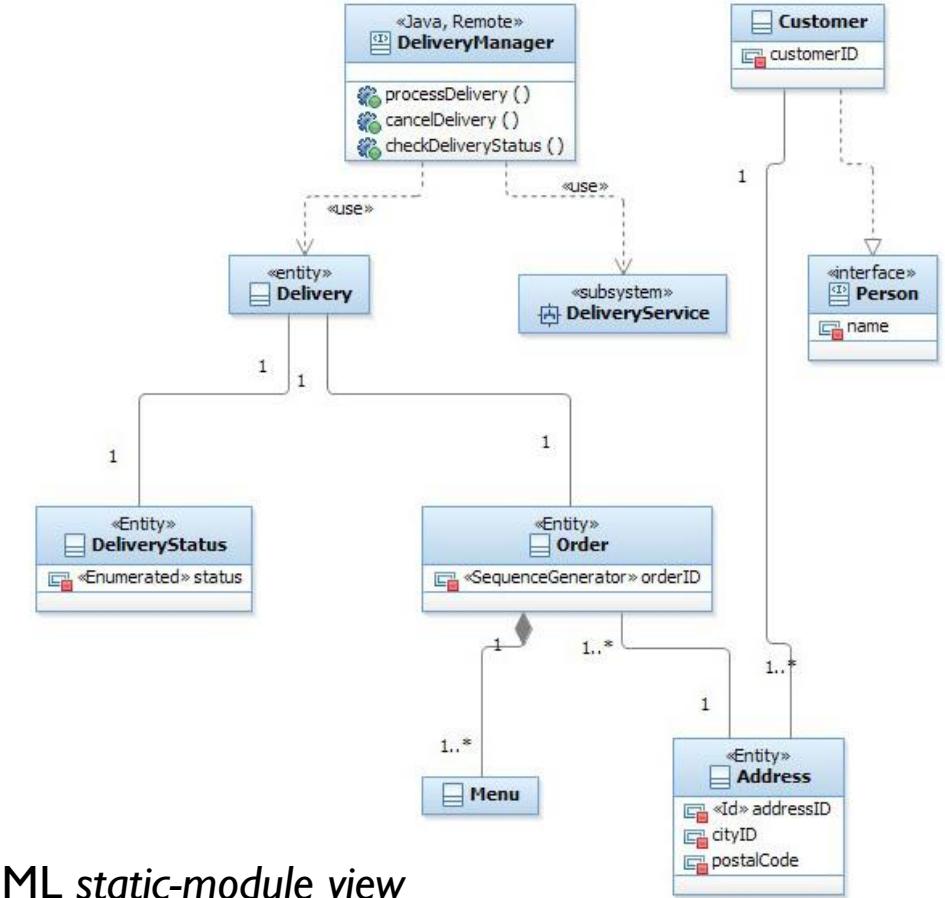
- ▶ Due to the high complexity of software systems, there are different *perspectives* defined in software architecture.
- ▶ The most important are the *static*, the *dynamic* and the *allocation* perspective, each one with a number of views.



Perspectives & Views

- ▶ How is it structured as a set of code units module views ?

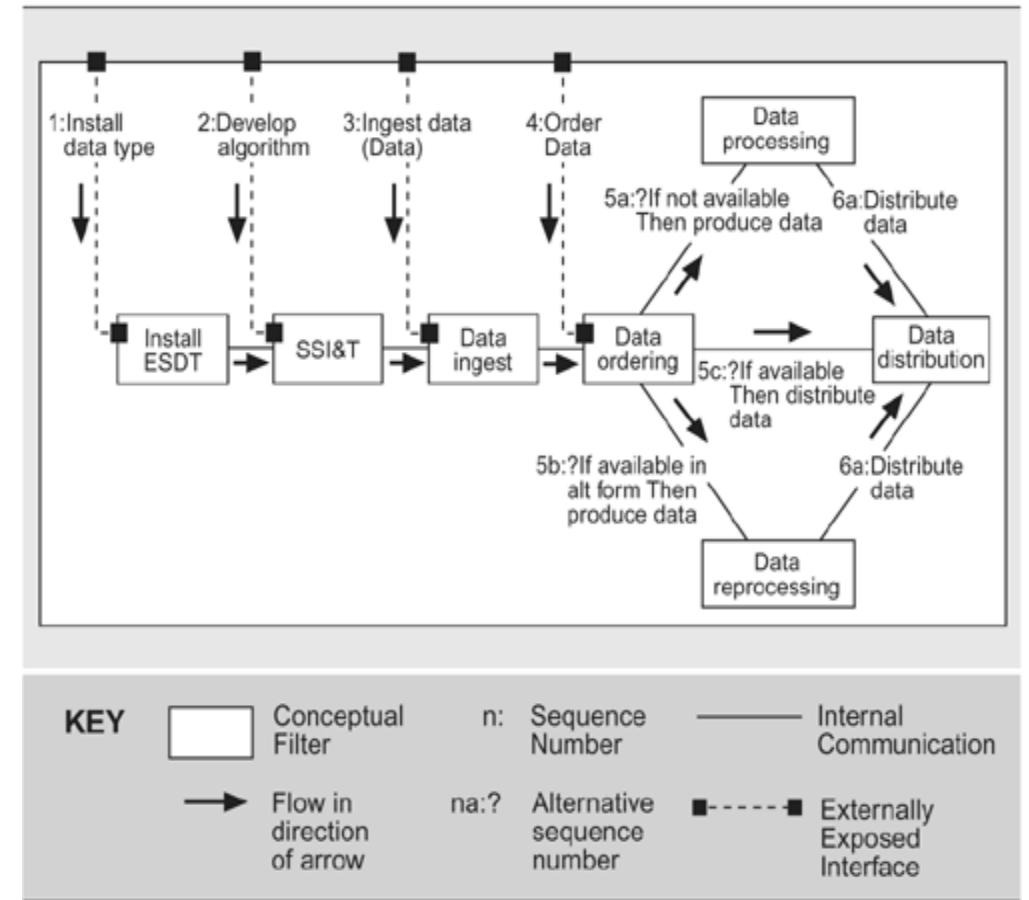
- ▶ **Module Views - Static**



Perspectives & Views

- ▶ How is it structured as a set of elements that have run-time behavior and interactions between them ?

- ▶ **Component & Connector Views**
Dynamic

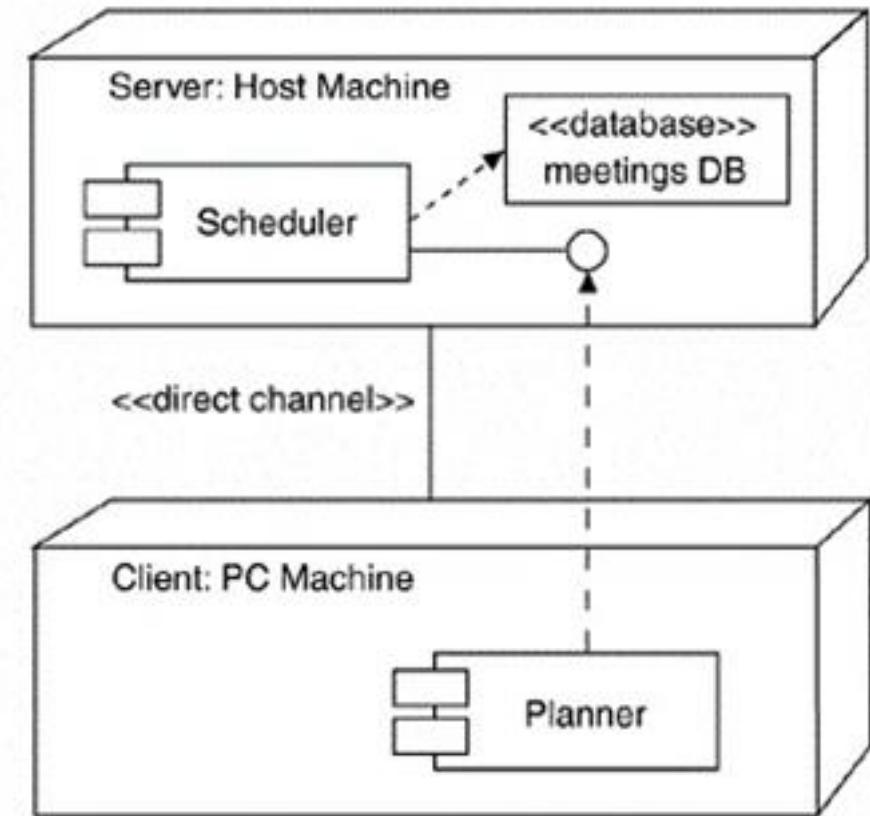


Dynamic perspective C&C view

Perspectives & Views

- ▶ How does it relate to non-software structures in its environment ?

- ▶ **Allocation Views - Deployment**



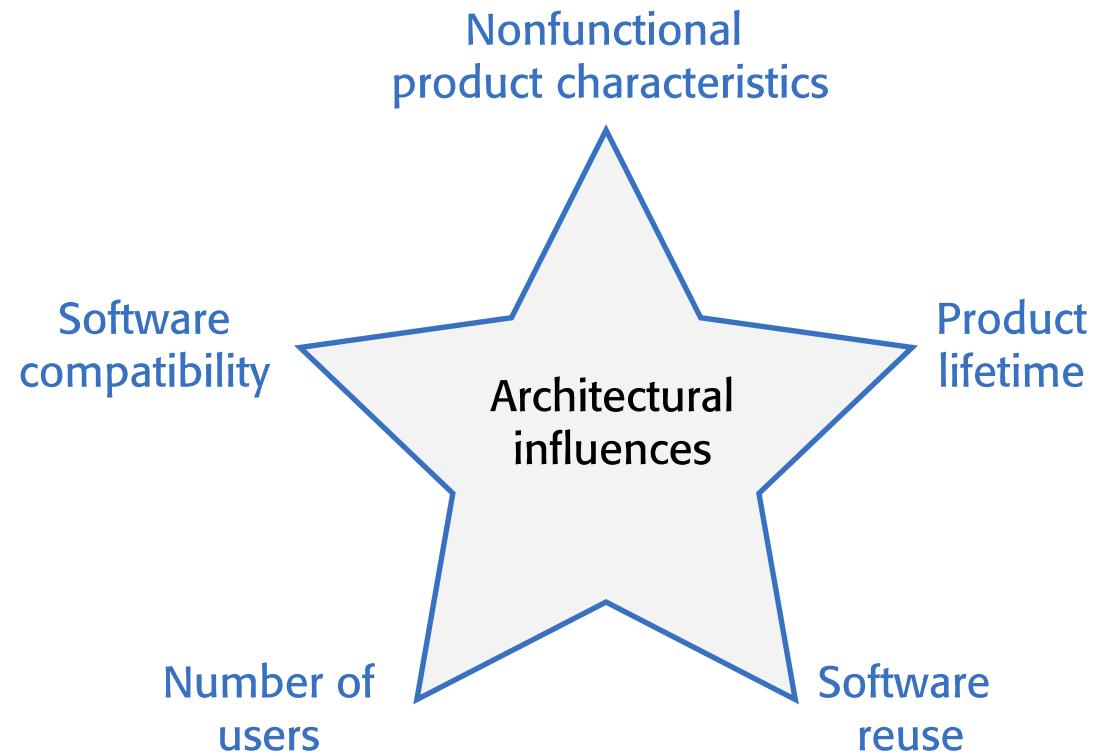
Non-functional system quality attributes

- ▶ ***Responsiveness***
Does the system return results to users in a reasonable time?
 - ▶ ***Reliability***
Do the system features behave as expected by both developers and users?
 - ▶ ***Availability***
Can the system deliver its services when requested by users?
 - ▶ ***Security***
Does the system protect itself and users' data from unauthorized attacks and intrusions?
 - ▶ ***Usability***
Can system users access the features that they need and use them quickly and without errors?
 - ▶ ***Maintainability***
Can the system be readily updated and new features added without undue costs?
 - ▶ ***Resilience***
Can the system continue to deliver user services in the event of partial failure or external attack?
-

Architecture Drivers

- ▶ Architectural drivers are the design forces that will influence the early design decisions the architects make.
- ▶ Architectural drivers are not all of the requirements for a system, but they are an early attempt to identify and capture those requirements, that are most influential.
- ▶ Architectural drivers can be one of the following :
 - ▶ **Constraints**
 - ▶ **Technical Constraints**
 - ▶ **Business Constraints**
 - ▶ **Quality Attributes**
 - ▶ **High-Level Functional Requirements**

Issues that influence architectural decisions



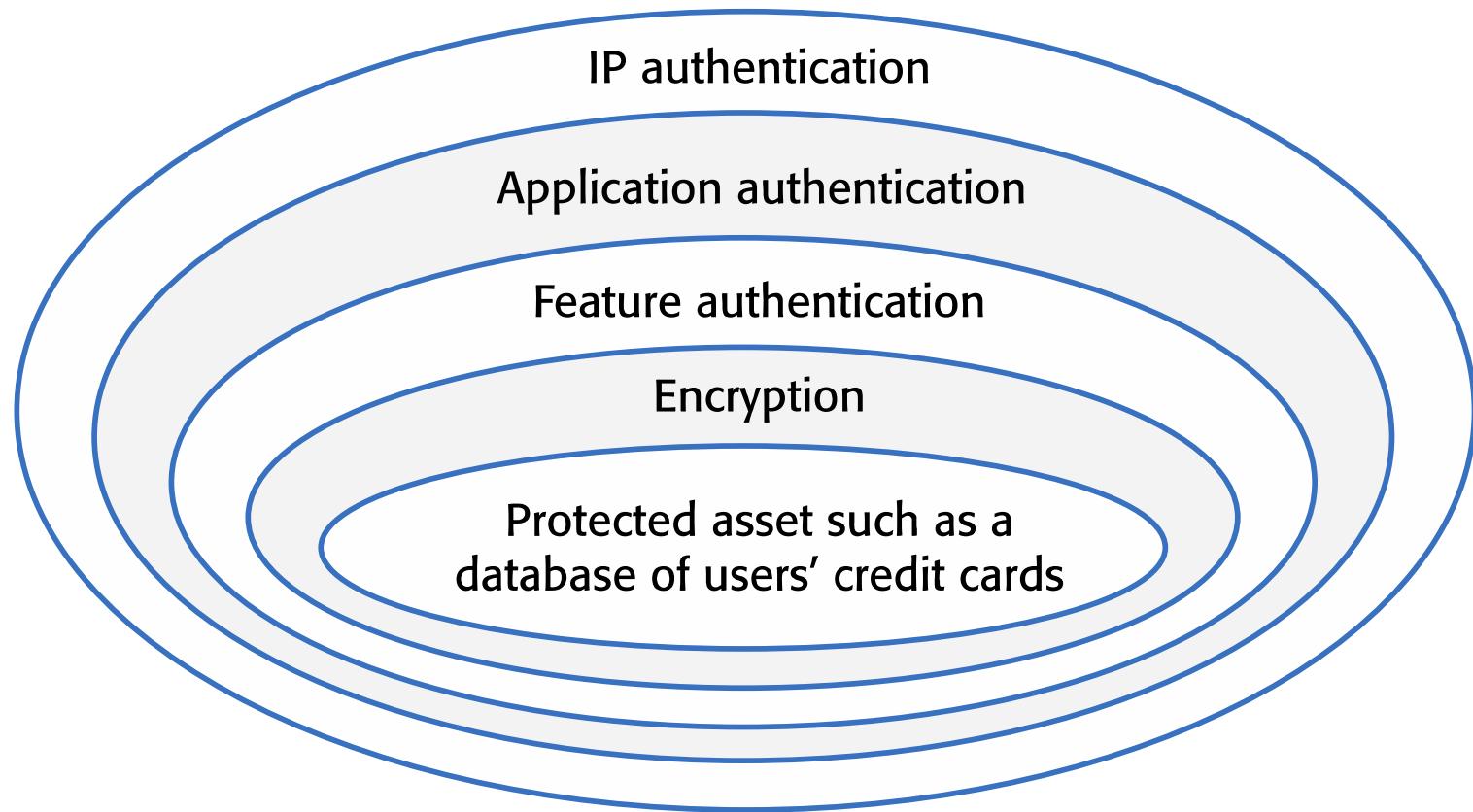
Trade off: Maintainability vs performance

- ▶ System maintainability is an attribute that reflects how difficult and expensive it is to make changes to a system after it has been released to customers.
 - ▶ You improve maintainability by building a system from small self-contained parts, each of which can be replaced or enhanced if changes are required.
- ▶ In architectural terms, this means that the system should be decomposed into fine-grain components, each of which does one thing and one thing only.
 - ▶ However, it takes time for components to communicate with each other. Consequently, if many components are involved in implementing a product feature, the software will be slower.

Trade off: Security vs usability

- ▶ You can achieve security by designing the system protection as a series of layers An attacker has to penetrate all of those layers before the system is compromised.
- ▶ Layers might include system authentication layers, a separate critical feature authentication layer, an encryption layer and so on.
- ▶ Architecturally, you can implement each of these layers as separate components so that if one of these components is compromised by an attacker, then the other layers remain intact.

Authentication layers



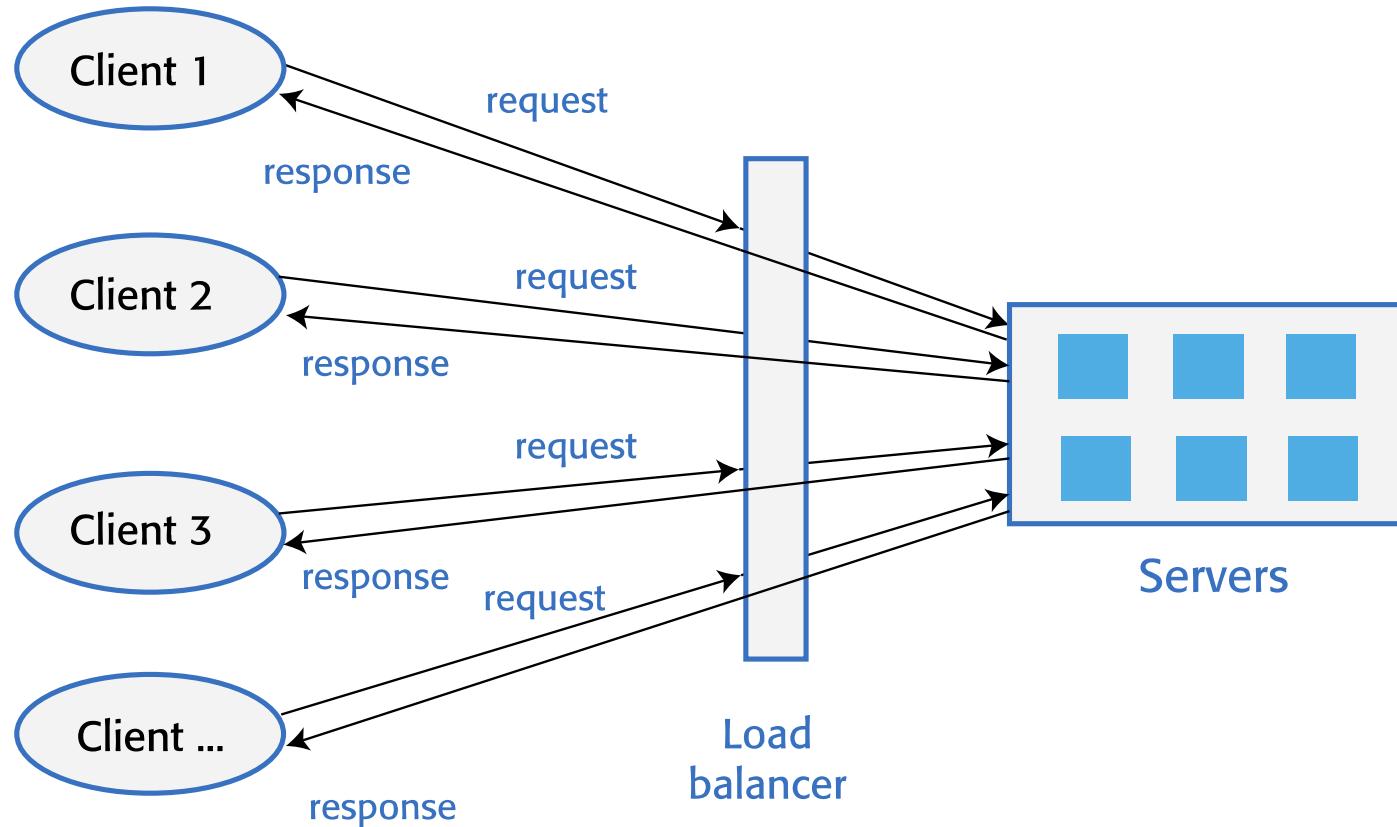
Usability issues

- ▶ A layered approach to security affects the usability of the software.
 - ▶ Users have to remember information, like passwords, that is needed to penetrate a security layer. Their interaction with the system is inevitably slowed down by its security features.
 - ▶ Many users find this irritating and often look for work-arounds so that they do not have to re-authenticate to access system features or data.
- ▶ To avoid this, you need an architecture:
 - ▶ that doesn't have too many security layers,
 - ▶ that doesn't enforce unnecessary security,
 - ▶ that provides helper components that reduce the load on users.

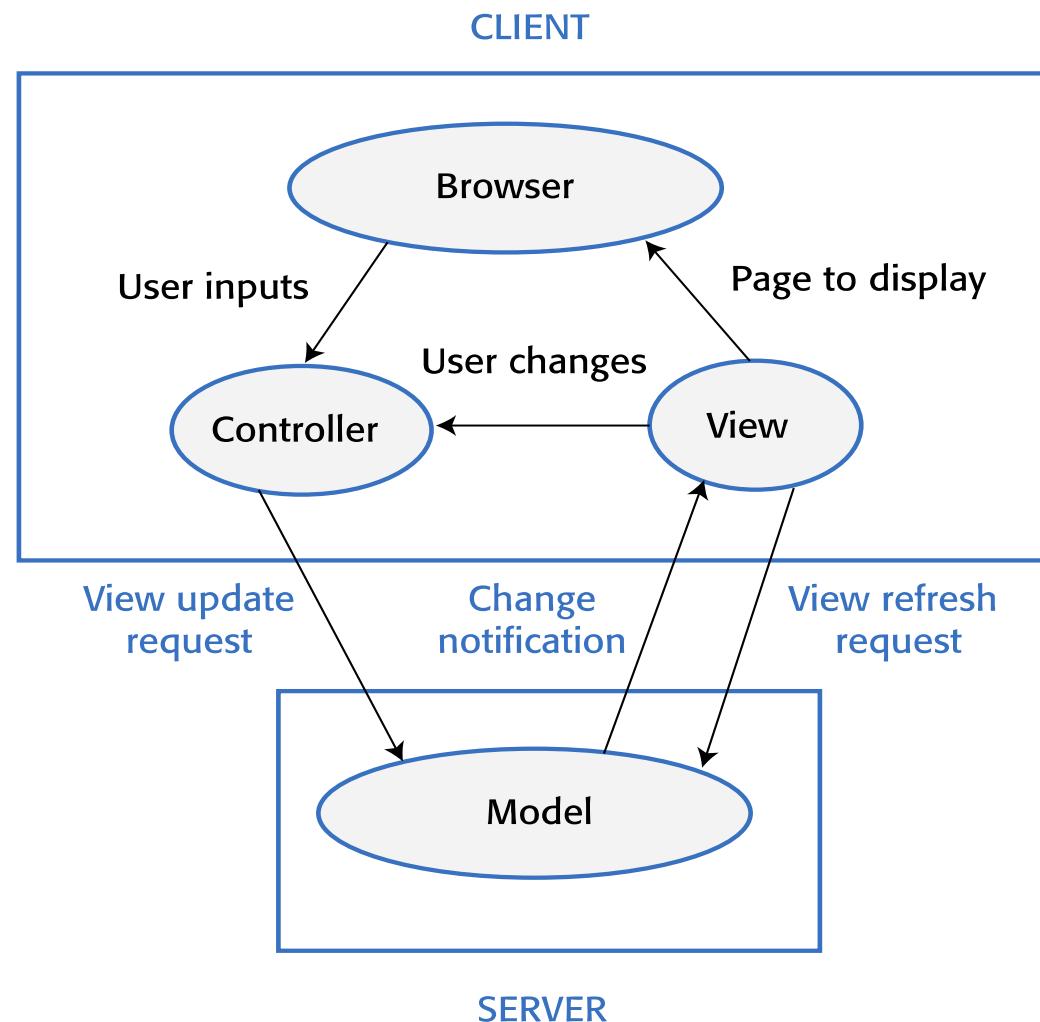
Trade off: Availability vs time-to-market

- ▶ Availability is particularly important in enterprise products, such as products for the finance industry, where 24/7 operation is expected.
- ▶ The availability of a system is a measure of the amount of ‘uptime’ of that system.
 - ▶ Availability is normally expressed as a percentage of the time that a system is available to deliver user services.
- ▶ Architecturally, you achieve availability by having redundant components in a system.
 - ▶ To make use of redundancy, you include sensor components that detect failure, and switching components that switch operation to a redundant component when a failure is detected.
- ▶ Implementing extra components takes time and increases the cost of system development. It adds complexity to the system and therefore increases the chances of introducing bugs and vulnerabilities.

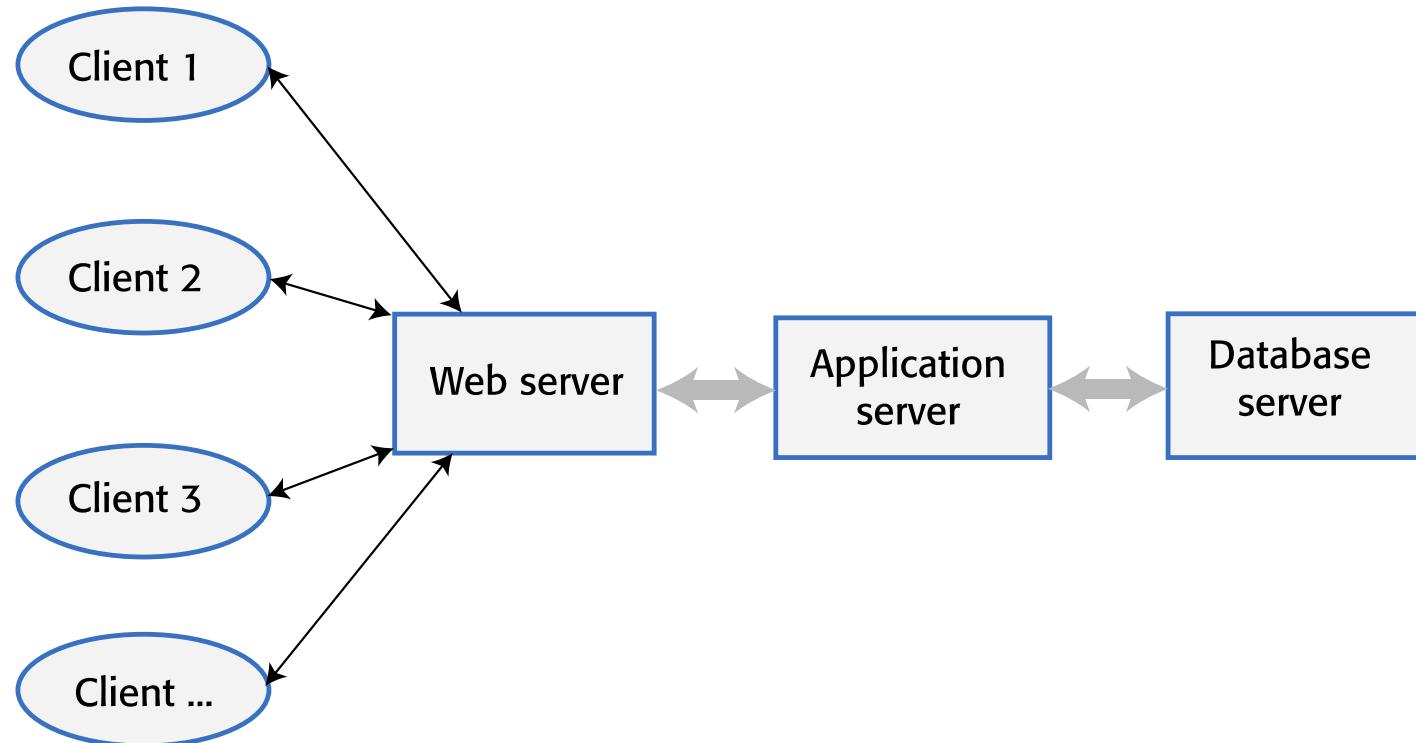
Client-server architecture



The model-view-controller pattern



Multi-tier client-server architecture



Key points

- ▶ Software architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
- ▶ The architecture of a software system has a significant influence on non-functional system properties such as reliability, efficiency and security.
- ▶ Architectural design involves understanding the issues that are critical for your product and creating system descriptions that shows components and their relationships.
- ▶ The principal role of architectural descriptions is to provide a basis for the development team to discuss the system organization. Informal architectural diagrams are effective in architectural description because they are fast and easy to draw and share.
- ▶ System decomposition involves analyzing architectural components and representing them as a set of finer-grain components.

Key points

- ▶ To minimize complexity, you should separate concerns, avoid functional duplication and focus on component interfaces.
- ▶ Web-based systems often have a common layered structure including user interface layers, application-specific layers and a database layer.
- ▶ The distribution architecture in a system defines the organization of the servers in that system and the allocation of components to these servers.
- ▶ Multi-tier client-server and service-oriented architectures are the most commonly used architectures for web-based systems.
- ▶ Making decisions on technologies such as database and cloud technologies are an important part of the architectural design process.

Процес на проектиране на софтуерната архитектура

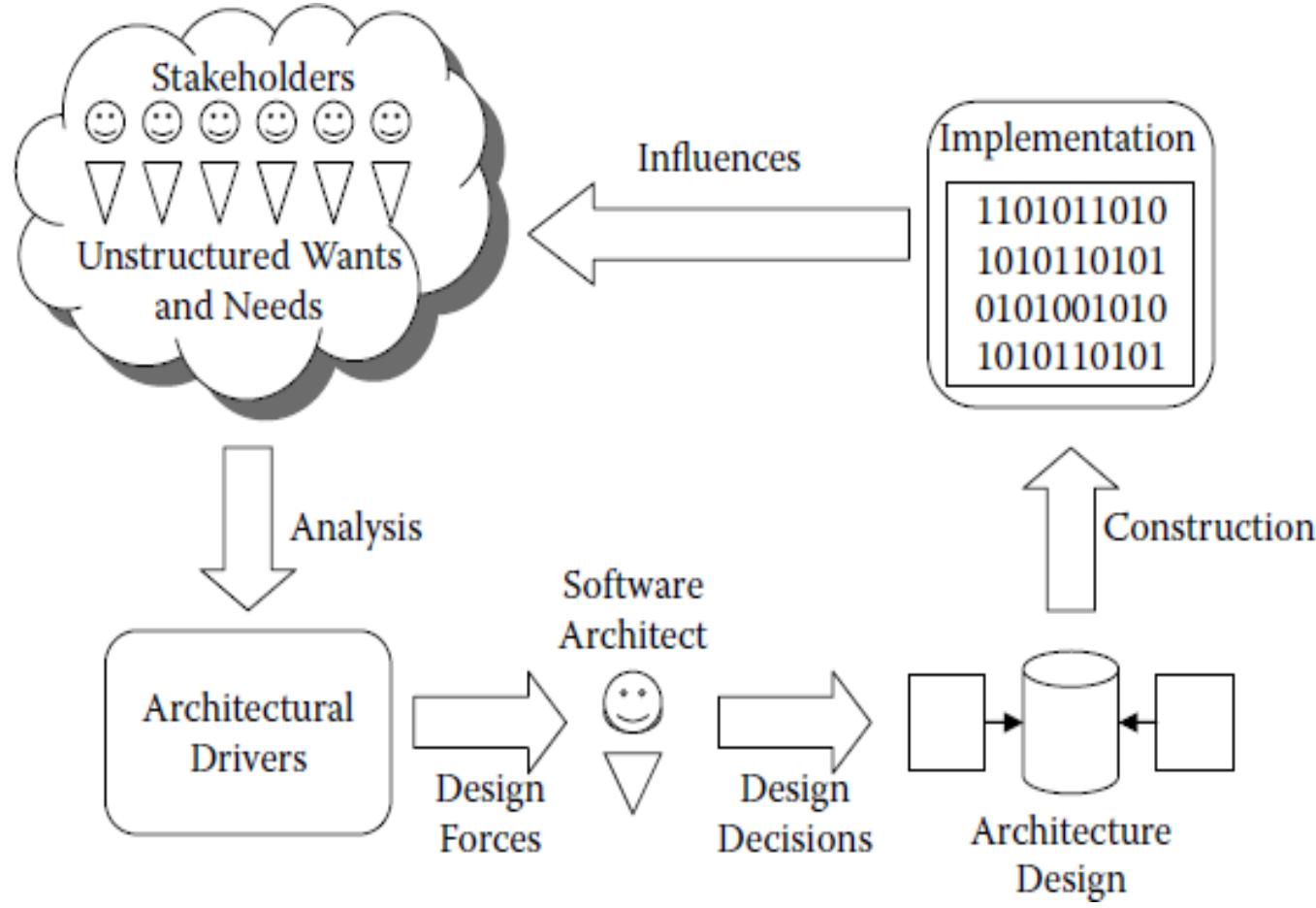
Как започва проектирането на архитектурата?

- ▶ Проектирането започва при наличието на изисквания. От друга страна, не се изисква наличието на много изисквания, за да започне проектирането

- ▶ Архитектурата се оформя от няколко (десетина) основополагащи функционални, качествени и бизнес изисквания – т.н. архитектурни драйвери

- ▶ Ще разгледаме т.нар. процес Architecture Driven Design (ADD)

Цикличен процес на създаване на архитектурата



Как се избират драйверите?

- ▶ Идентифицират се тези цели на системата, които са с най-висок приоритет
- ▶ Тези цели се превръщат в сценарии за употреба или за постигане на качествено свойство
- ▶ От тях се пробират не повече от 10 – тези, които имат най-голямо влияние върху архитектурата
- ▶ След като драйверите бъдат избрани, започва проектирането. Започва и задаването на въпроси, което може да доведе до промяна в изискванията, което пък може да доведе до промяна в драйверите и т.н.

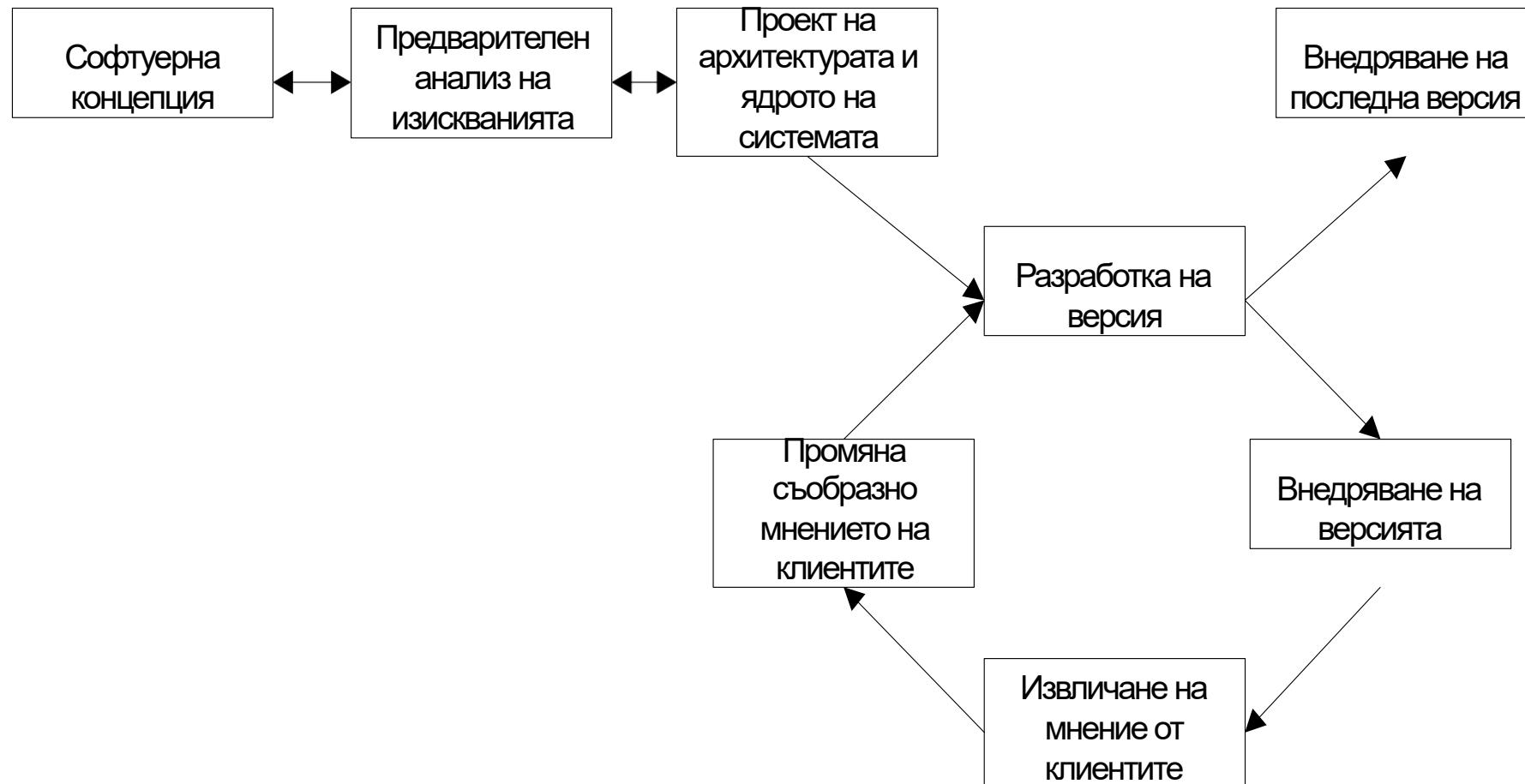
ADD – Attribute Driven Design

- ▶ ADD е подход за проектиране, в който основна роля играят качествените свойства (атрибути)
- ▶ Това е рекурсивен процес на дефиниране на архитектурата, като на всяка стъпка се използват тактики и архитектурни модели за постигане на желаните качествени свойства
- ▶ В следствие на приложението на ADD се получават първите няколко нива на модулната декомпозиция и на други структури, в зависимост от случая
- ▶ Това е първото проявление на архитектурата и като такова е на достатъчно високо ниво, без излишни детайли

Стъпки на ADD

- ▶ Избира се модул (с известни изисквания), който ще се декомпозира.
 - ▶ Модулът се детайлизира:
 - ▶ Избират се архитектурните драйвери (най-важните изисквания за този етап)
 - ▶ Избира се архитектурен модел, който удовлетворява драйверите. Модела се избира или създава въз основа на тактиките за постигане на избраните свойства. Идентифицират се типовете под-модули, необходими за постигането на тактиките
 - ▶ Създават се под-модули от идентифицираните типове и им се приписва функционалност съгласно сценариите за употреба. Създават се всички необходими структури.
 - ▶ Дефинират се интерфейсите към и от под-модулите
 - ▶ Проверяват се и се детайлизират изискванията, като същевременно се поставят ограничения върху под-модулите. На тази стъпка се проверява дали всичко съществено е налично и се подготвят под-модулите за по-нататъшна декомпозиция
 - ▶ Това се повтарят за всички модули, които се нуждаят от по-нататъшна декомпозиция
-

Цикличен процес на създаване на архитектурата



Формиране на екипи

- ▶ След като се идентифицират първите няколко нива на декомпозицията, могат да се формират екипи, които да работят по съответните модули
- ▶ Структурата на екипите обикновено отговаря на структурата на декомпозицията
- ▶ Екипа представлява обособена екосистема, със собствени правила и експертиза
- ▶ Комуникацията в рамките на екипа е различна от комуникацията между екипите

Създаване на скелетна система

- ▶ Когато архитектурата е готова донякъде и има сформирани екипи може да се започне работа по системата
- ▶ По време на разработката обикновено се използват стъбове, за да могат модулите да се разработват и тестват поотделно
- ▶ Но с кои модули да започне създаването на скелетна система?

Последователност на реализацията

- ▶ Първо се реализират компонентите, свързани с изпълнението на и взаимодействието между архитектурните компоненти (middleware)
- ▶ След това се реализират някои прости функционалности
- ▶ Последователността по нататък може да се диктува от:
 - ▶ Намаляване на риска – първо най-проблематичните;
 - ▶ В зависимост от наличния персонал и квалификацията му;
 - ▶ Бързото създаване на нещо, което се продава;
- ▶ След това на база структурата на употребата се определят следващите функционалности

Въпроси ?



Допълнителни материали

- ▶ I. Sommerville, *Engineering Software Products*, **Chapter 4**
- ▶ *Attribute-Driven Design Method Collection*, Software Engineering Institute, CMU -
<https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=484077>



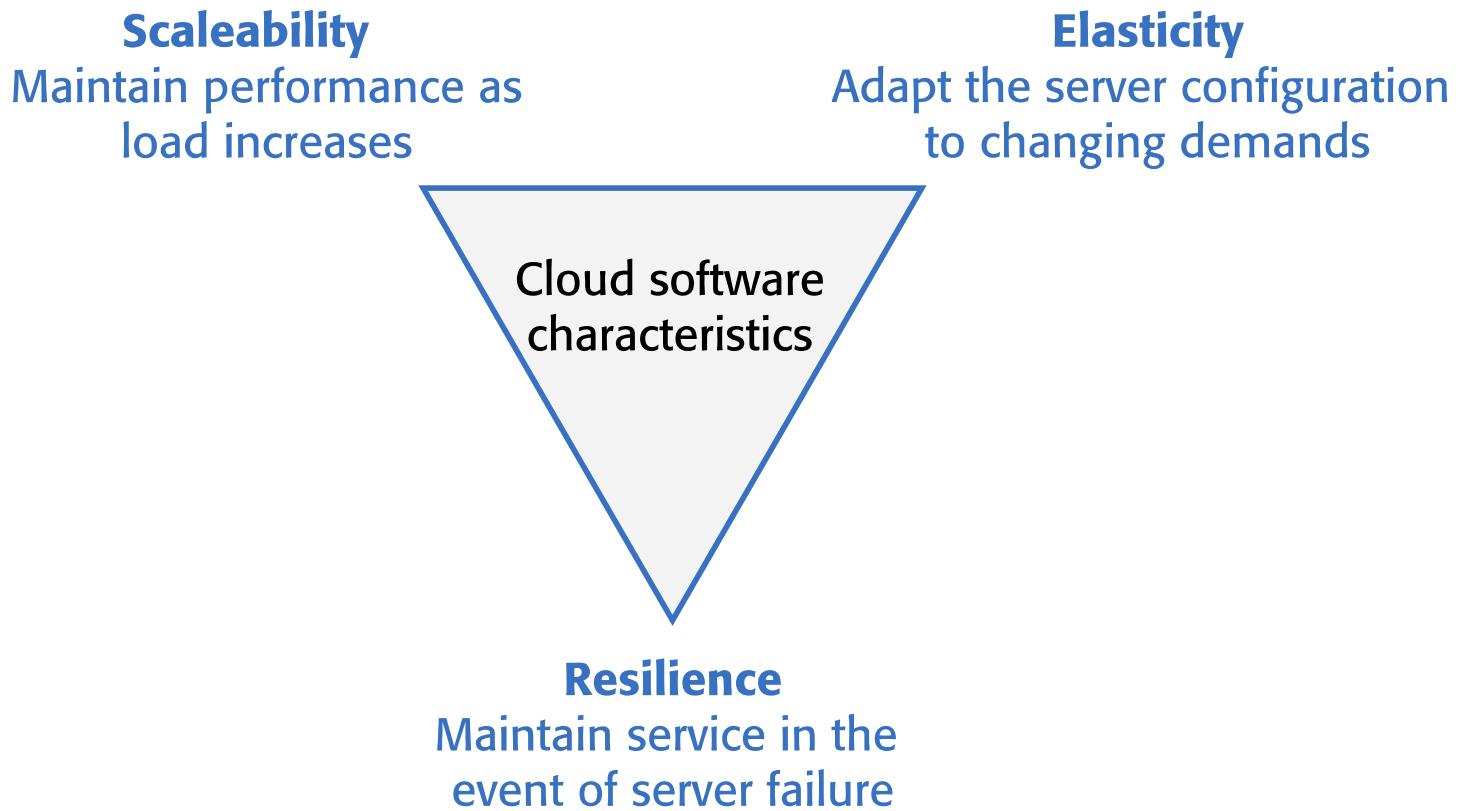
Облачни (Cloud) технологии и виртуализация



The cloud

- ▶ The cloud is made up of very large number of remote servers that are offered for rent by companies that own these servers.
 - ▶ Cloud-based servers are ‘virtual servers’, which means that they are implemented in software rather than hardware.
- ▶ You can rent as many servers as you need, run your software on these servers and make them available to your customers.
 - ▶ Your customers can access these servers from their own computers or other networked devices such as a tablet or a TV.
 - ▶ Cloud servers can be started up and shut down as demand changes.
- ▶ You may rent a server and install your own software, or you may pay for access to software products that are available on the cloud.

Scaleability, elasticity and resilience



Scaleability, elasticity and resilience

- ▶ **Scaleability** reflects the ability of your software to cope with increasing numbers of users.
 - ▶ As the load on your software increases, your software automatically adapts so that the system performance and response time is maintained.
- ▶ **Elasticity** is related to scalability but also allows for scaling-down as well as scaling-up.
 - ▶ That is, you can monitor the demand on your application and add or remove servers dynamically as the number of users change.
- ▶ **Resilience** means that you can design your software architecture to tolerate server failures.
 - ▶ You can make several copies of your software concurrently available. If one of these fails, the others continue to provide a service.

Benefits of using the cloud for software development

▶ **Cost**

You avoid the initial capital costs of hardware procurement

▶ **Startup time**

You don't have to wait for hardware to be delivered before you can start work. Using the cloud, you can have servers up and running in a few minutes.

▶ **Server choice**

If you find that the servers you are renting are not powerful enough, you can upgrade to more powerful systems. You can add servers for short-term requirements, such as load testing.

▶ **Distributed development**

If you have a distributed development team, working from different locations, all team members have the same development environment and can seamlessly share all information.

Virtualization

- ▶ “**Virtualization** is a process that allows for more efficient utilization of physical computer hardware and is the foundation of cloud computing.”
- ▶ “**Virtualization** uses software to create an abstraction layer over computer hardware that allows the hardware elements of a single computer—processors, memory, storage and more—to be divided into multiple virtual computers, commonly called virtual machines (VMs). Each VM runs its own operating system (OS) and behaves like an independent computer, even though it is running on just a portion of the actual underlying computer hardware.”

Virtual cloud servers

- ▶ A **virtual server** runs on an underlying physical computer and is made up of an operating system plus a set of software packages that provide the server functionality required.
- ▶ A **virtual server** is a stand-alone system that can run on any hardware in the cloud.
 - ▶ This ‘run anywhere’ characteristic is possible because the virtual server has no external dependencies.
- ▶ **Virtual machines (VMs)**, running on physical server hardware, can be used to implement virtual servers.
 - ▶ A hypervisor provides hardware emulation that simulates the operation of the underlying hardware.
- ▶ If you use a virtual machine to implement virtual servers, you have exactly the same hardware platform as a physical server.

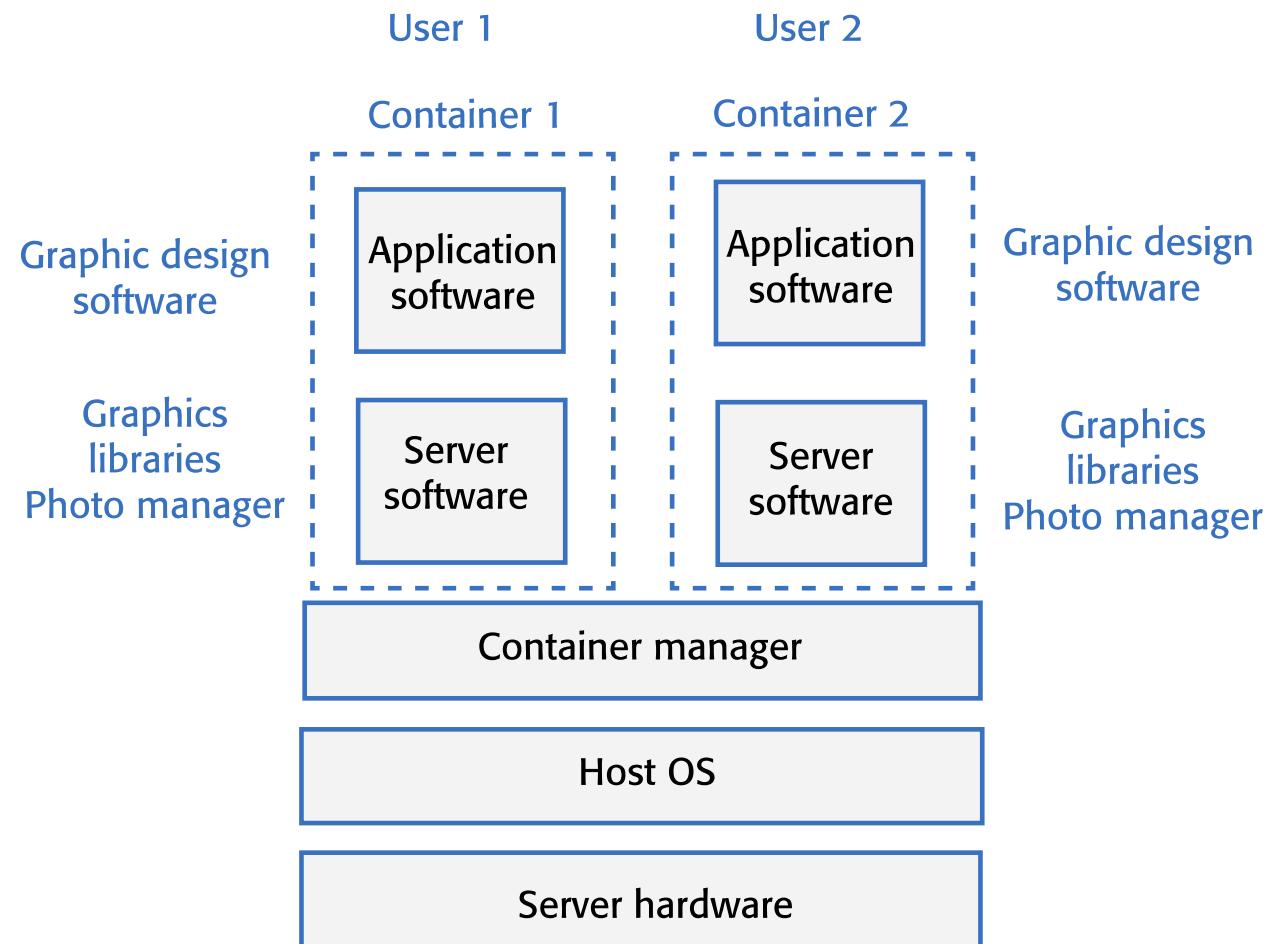
Types of Virtualization

- ▶ Desktop virtualization
- ▶ Network virtualization
- ▶ Storage virtualization
- ▶ Data virtualization
- ▶ Application virtualization
- ▶ Data center virtualization
- ▶ CPU virtualization
- ▶ GPU virtualization
- ▶ Linux virtualization
- ▶ Cloud virtualization (*IaaS, PaaS, SaaS*)

Container-based virtualization

- ▶ If you are running a cloud-based system with many instances of applications or services, these all use the same operating system, you can use a simpler virtualization technology called ‘containers’.
- ▶ Using containers accelerates the process of deploying virtual servers on the cloud.
 - ▶ Containers are usually megabytes in size whereas VMs are gigabytes.
 - ▶ Containers can be started and shut down in a few seconds rather than the few minutes required for a VM.
- ▶ **Containers are an operating system virtualization technology that allows independent servers to share a single operating system.**
 - ▶ They are particularly useful for providing isolated application services where each user sees their own version of an application.

Using containers to provide isolated services



Everything as a service

► The idea of a service that is rented rather than owned is fundamental to cloud computing.

► **Infrastructure as a service (IaaS)**

- Cloud providers offer different kinds of infrastructure service such as a compute service, a network service and a storage service that you can use to implement virtual servers.

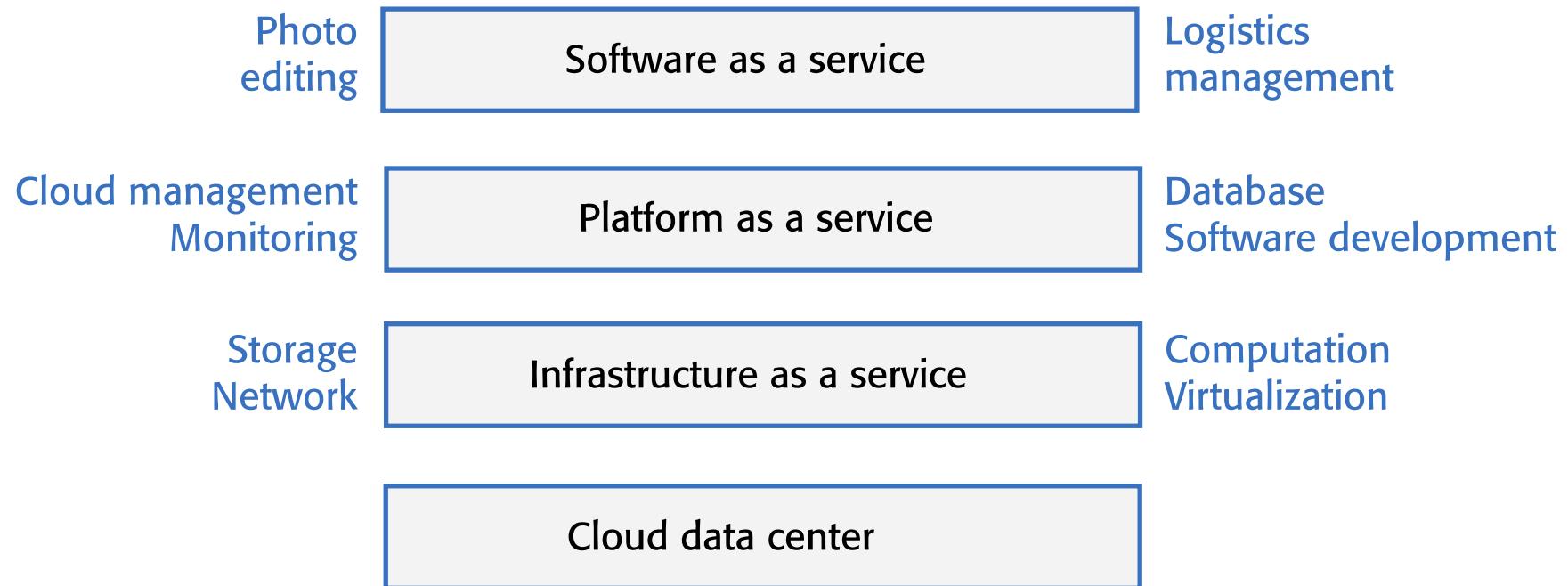
► **Platform as a service (PaaS)**

- This is an intermediate level where you use libraries and frameworks provided by the cloud provider to implement your software. These provide access to a range of functions, including SQL and NoSQL databases.

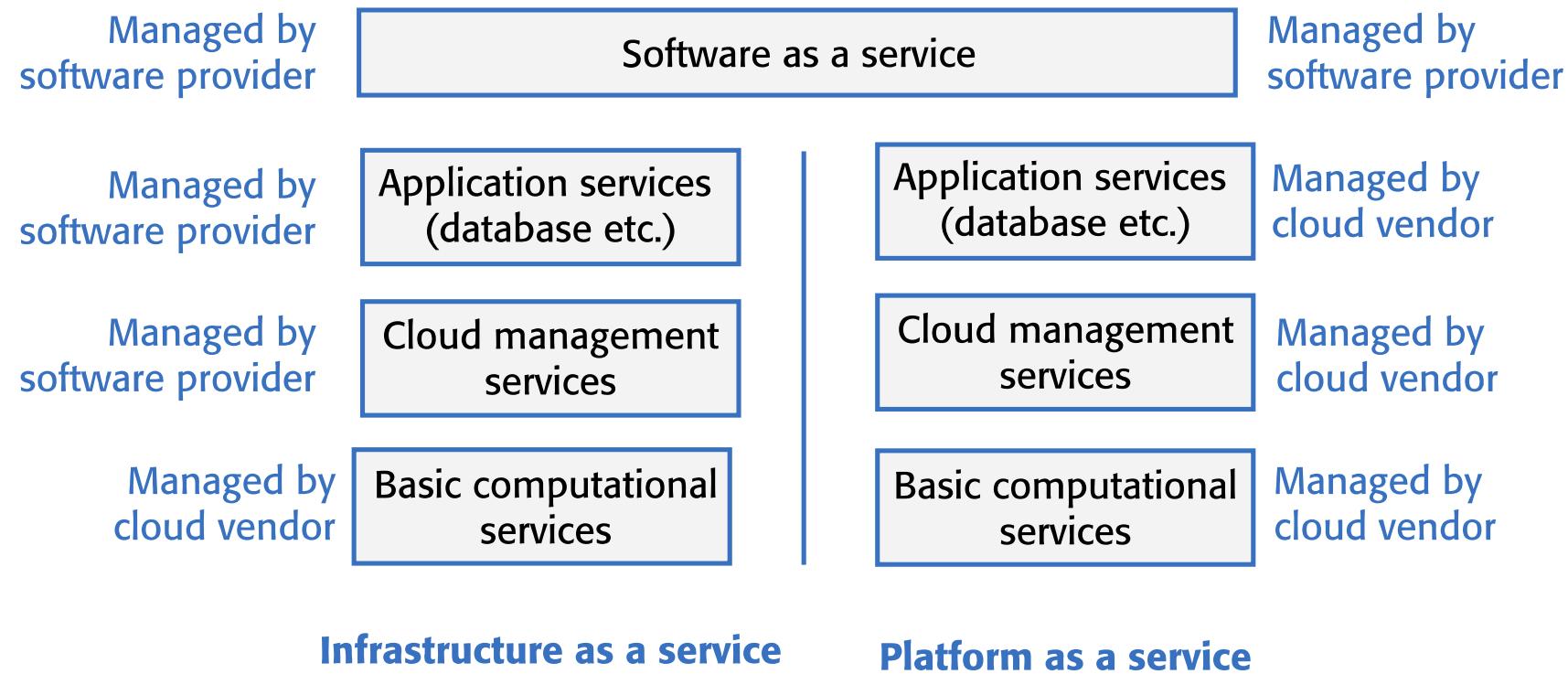
► **Software as a service (SaaS)**

- Your software product runs on the cloud and is accessed by users through a web browser or mobile app.

Everything as a service



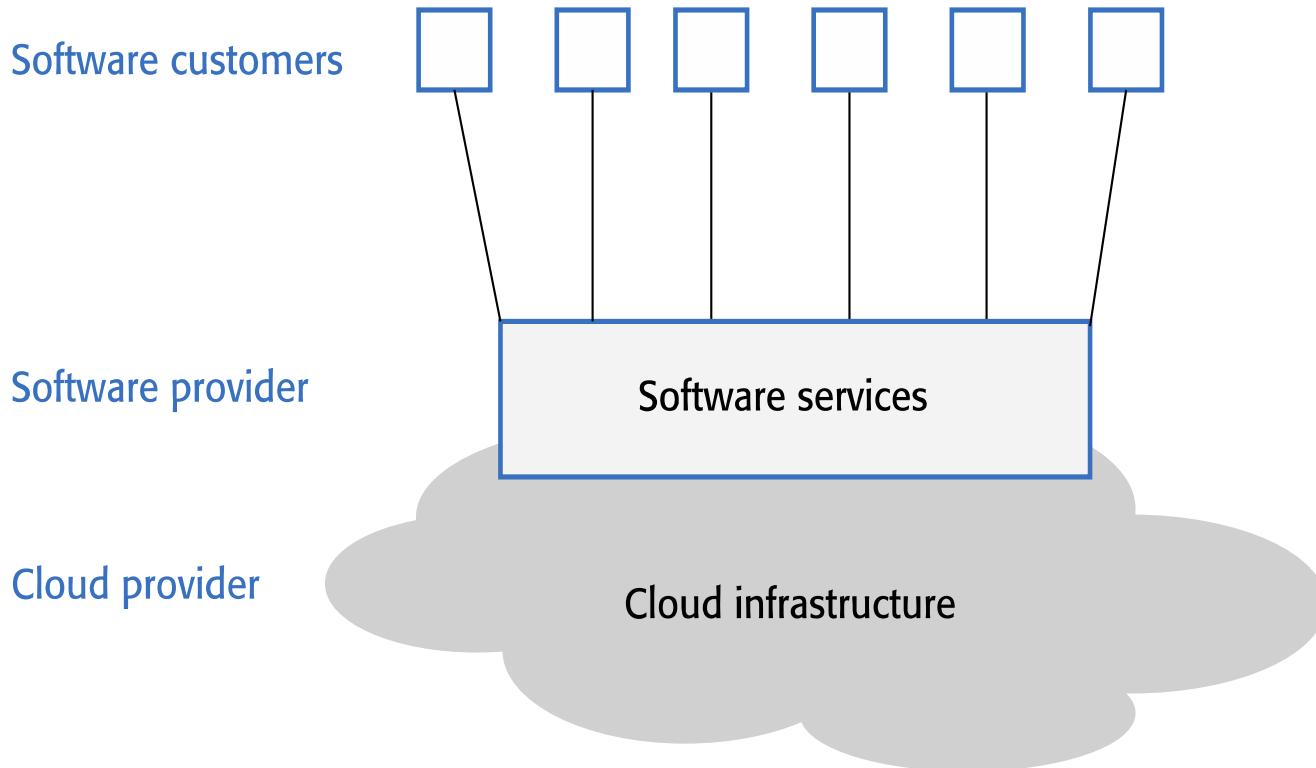
Management responsibilities for IaaS and PaaS



Software as a service

- ▶ Increasingly, software products are being delivered as a service, rather than installed on the buyer's computers.
- ▶ If you deliver your software product as a service, you run the software on your servers, which you may rent from a cloud provider.
- ▶ Customers don't have to install software and they access the remote system through a web browser or dedicated mobile app.
- ▶ The payment model for software as a service is usually a subscription model.
 - ▶ Users pay a monthly fee to use the software rather than buy it outright.

Software as a service



Benefits of SaaS for software product providers

► ***Cash flow***

Customers either pay a regular subscription or pay as they use the software. This means you have a regular cash flow, with payments throughout the year. You don't have a situation where you have a large cash injection when products are purchased but very little income between product releases.

► ***Update management***

You are in control of updates to your product and all customers receive the update at the same time. You avoid the issue of several versions being simultaneously used and maintained. This reduces your costs and makes it easier to maintain a consistent software code base.

► ***Continuous deployment***

You can deploy new versions of your software as soon as changes have been made and tested. This means you can fix bugs quickly so that your software reliability can continuously improve.

Benefits of SaaS for software product providers

► ***Payment flexibility***

You can have several different payment options so that you can attract a wider range of customers. Small companies or individuals need not be discouraged by having to pay large upfront software costs.

► ***Try before you buy***

You can make early free or low-cost versions of the software available quickly with the aim of getting customer feedback on bugs and how the product could be approved.

► ***Data collection***

You can easily collect data on how the product is used and so identify areas for improvement. You may also be able to collect customer data that allows you to market other products to these customers.

Advantages and disadvantages of SaaS for customers

Advantages

Mobile, laptop and desktop access

No upfront costs for software or servers

Immediate software updates

Reduced software management costs

Software customer

Disadvantages

Privacy regulation conformance

Network constraints
Security concerns

Loss of control over updates

Service lock-in
Data exchange

Data storage and management issues for SaaS

► **Regulation**

Some countries, such as EU countries, have strict laws on the storage of personal information. These may be incompatible with the laws and regulations of the country where the SaaS provider is based. If a SaaS provider cannot guarantee that their storage locations conform to the laws of the customer's country, businesses may be reluctant to use their product.

► **Data transfer**

If software use involves a lot of data transfer, the software response time may be limited by the network speed. This is a problem for individuals and smaller companies who can't afford to pay for very high speed network connections.

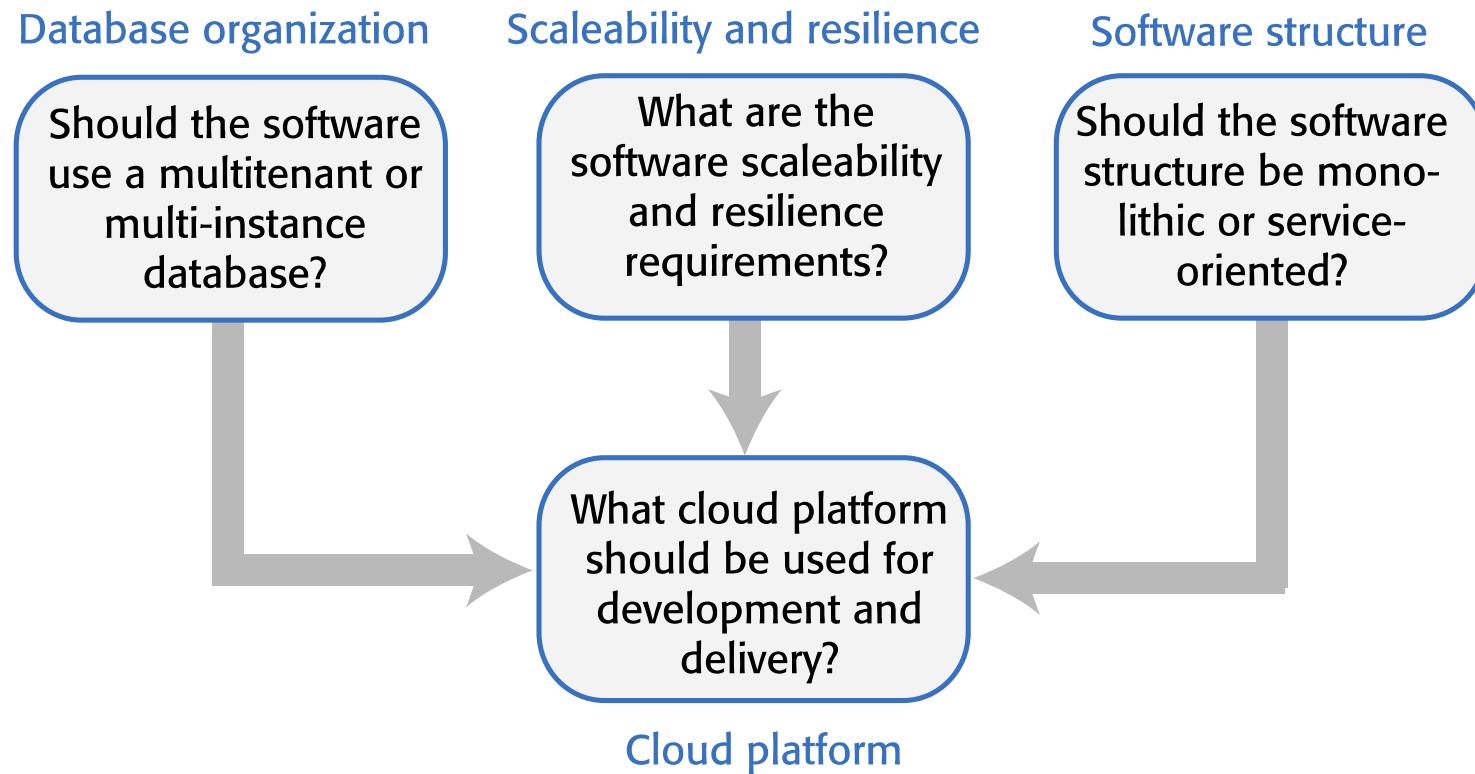
► **Data security**

Companies dealing with sensitive information may be unwilling to hand over the control of their data to an external software provider. As we have seen from a number of high profile cases, even large cloud providers have had security breaches. You can't assume that they always provide better security than the customer's own servers.

► **Data exchange**

If you need to exchange data between a cloud service and other services or local software applications, this can be difficult unless the cloud service provides an API that is accessible for external use.

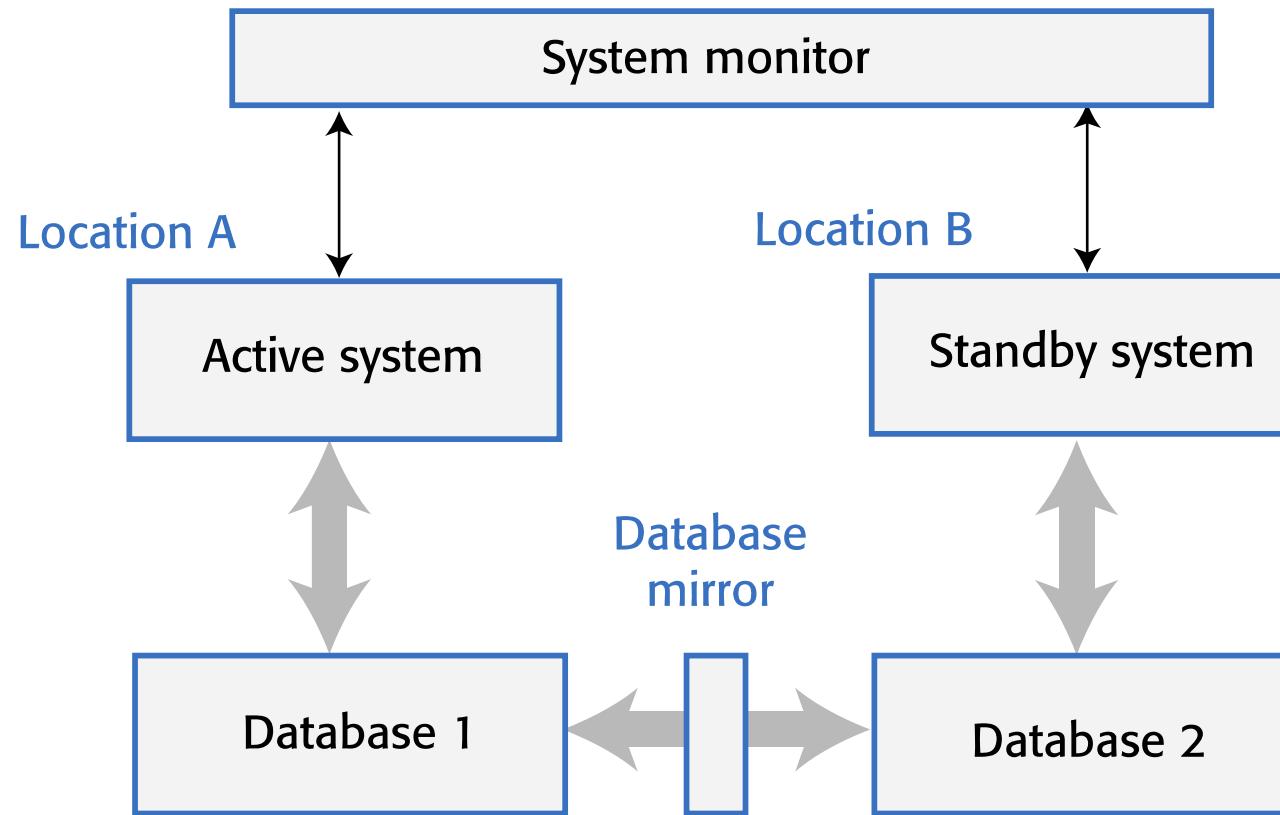
Architectural decisions for cloud software engineering



Scalability and resilience

- ▶ The **scalability** of a system reflects its ability to adapt automatically to changes in the load on that system.
- ▶ The **resilience** of a system reflects its ability to continue to deliver critical services in the event of system failure or malicious system use.
- ▶ You achieve **scalability** in a system by making it possible to **add new virtual servers (scaling-out)** or **increase the power of a system server (scaling-up)** in response to increasing load.
 - ▶ In cloud-based systems, scaling-out rather than scaling-up is the normal approach used. Your software has to be organized so that individual software components can be replicated and run in parallel.
 - ▶ To achieve **resilience**, you need to be able to restart your software quickly after a hardware or software failure.

Using a standby system to provide resilience



Resilience

- ▶ **Resilience relies on redundancy:**
 - ▶ Replicas of the software and data are maintained in different locations.
 - ▶ Database updates are mirrored so that the standby database is a working copy of the operational database.
 - ▶ A system monitor continually checks the system status. It can switch to the standby system automatically if the operational system fails.
- ▶ **You should use redundant virtual servers that are not hosted on the same physical computer and locate servers in different locations.**
 - ▶ Ideally, these servers should be located in different data centers.
 - ▶ If a physical server fails or if there is a wider data center failure, then operation can be switched automatically to the software copies elsewhere.

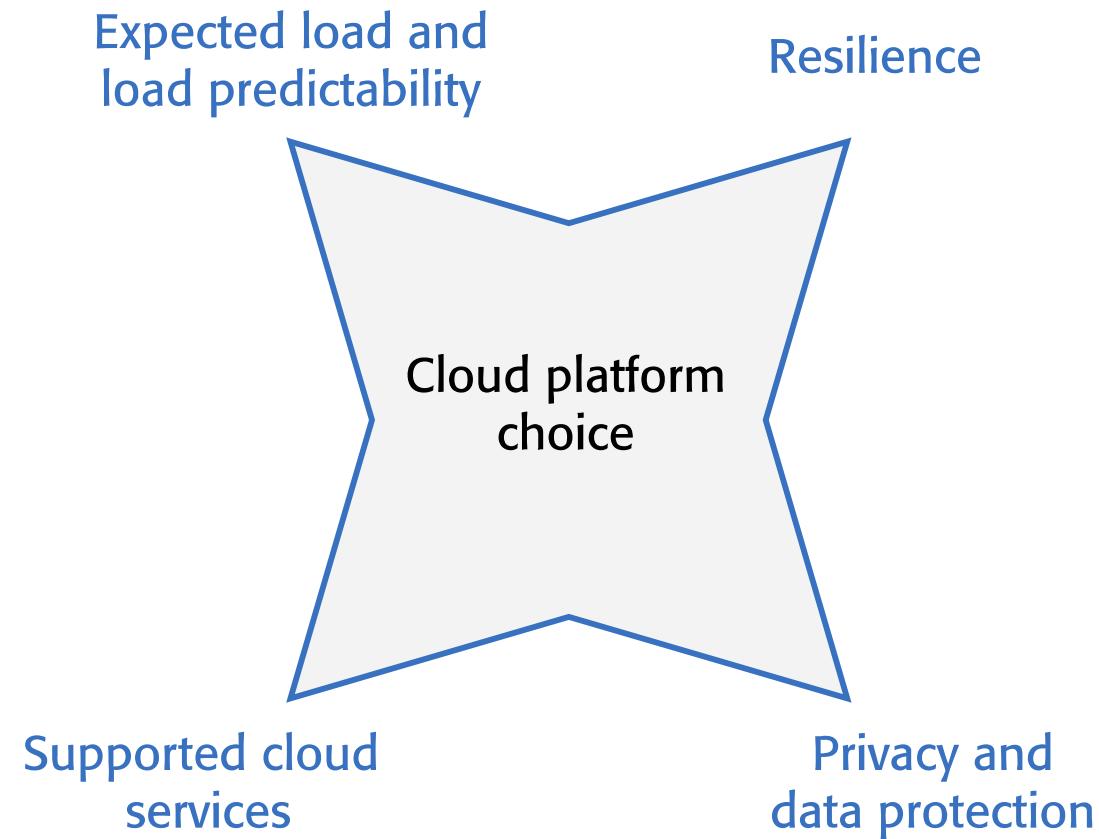
System structure

- ▶ An object-oriented approach to software engineering has been that been extensively used for the development of client-server systems built around a shared database.
- ▶ The system itself is, logically, a monolithic system with distribution across multiple servers running large software components. The traditional multi-tier client server architecture is based on this distributed system model.
- ▶ The alternative to a monolithic approach to software organization is a service-oriented approach where the system is decomposed into fine-grain, stateless services.
 - ▶ Because it is stateless, each service is independent and can be replicated, distributed and migrated from one server to another.
 - ▶ The service-oriented approach is particularly suitable for cloud-based software, with services deployed in containers.

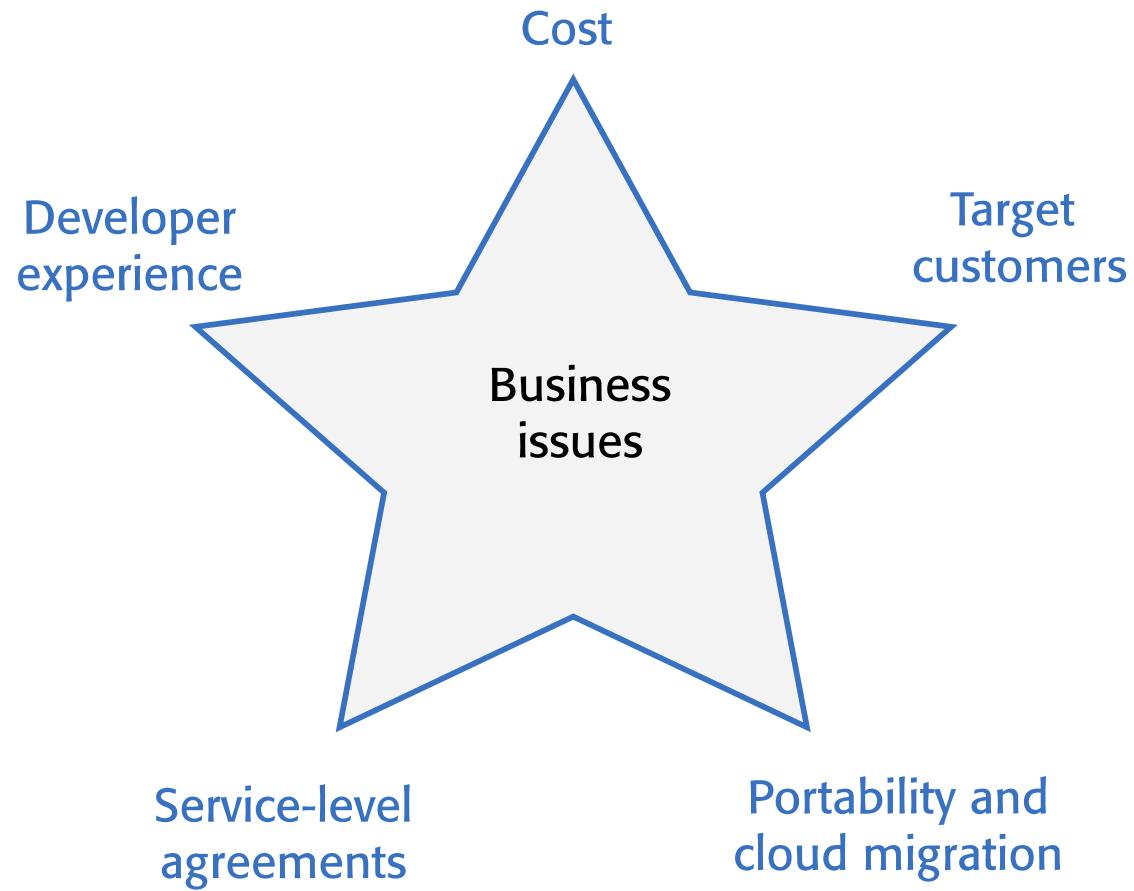
Cloud platform

- ▶ Cloud platforms include general-purpose clouds such as Amazon Web Services or lesser known platforms oriented around a specific application, such as the SAP Cloud Platform. There are also smaller national providers that provide more limited services but who may be more willing to adapt their services to the needs of different customers.
- ▶ There is no ‘best’ platform and you should choose a cloud provider based on your background and experience, the type of product that you are developing and the expectations of your customers.
- ▶ You need to consider both technical issues and business issues when choosing a cloud platform for your product.

Technical issues in cloud platform choice



Business issues in cloud platform choice



So technically Moses was the first man to download files from the cloud using a tablet.



Conclusion: Key points 1

- ▶ The cloud is made up of a large number of virtual servers that you can rent for your own use. You and your customers access these servers remotely over the internet and pay for the amount of server time used.
- ▶ Virtualization is a technology that allows multiple server instances to be run on the same physical computer. This means that you can create isolated instances of your software for deployment on the cloud.
- ▶ Virtual machines are physical server replicas on which you run your own operating system, technology stack and applications.
- ▶ Containers are a lightweight virtualization technology that allow rapid replication and deployment of virtual servers. All containers run the same operating system.
- ▶ A fundamental feature of the cloud is that ‘everything’ can be delivered as a service and accessed over the internet. A service is rented rather than owned and is shared with other users.

Conclusion: Key points 2

- ▶ Infrastructure as a service (IaaS) means computing, storage and other services are available over the cloud. There is no need to run your own physical servers.
- ▶ Platform as a service (PaaS) means using services provided by a cloud platform vendor to make it possible to auto-scale your software in response to demand.
- ▶ Software as a service (SaaS) means that application software is delivered as a service to users. This has important benefits for users, such as lower capital costs, and software vendors, such as simpler deployment of new software releases.

References

- ▶ Sommerville, I. *Software Engineering*. 10th edition, Published by Pearson Education, ISBN: 978-1-292-09613-1 (2016)
- ▶ Pressman, R., Maxim, B. *Software Engineering: A Practitioner's Approach*. 9th edition, Published by McGraw-Hill Education, ISBN: 9781260548006, (2019)
- ▶ *Virtualization Overview* - <https://www.ibm.com/topics/virtualization>



Моделиране на софтуерни системи. Унифициран език за моделиране (UML).



Първа Част

Models and modelling

- ▶ **Model:** (mathematical) presentation of structure and processes of a given system (used for analysis and planning)
- ▶ **Modeling:** process of describing of the system by means of its model (physical, conceptual, mathematical or based on imitation) and simulation of system activities by means of applying the model on a data set. Models represent real phenomenon that are difficult to observe directly. System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.

System modelling

- ▶ System modeling is the process of developing abstract models of a system – existing or being under development, whereupon each model represents a different perspective (so called view) of the system.
- ▶ System models describe the system structure, behavior and functionality (external perspective of the system) and are used to communicate them with customers:
 - ▶ Static models – describe system structure, e.g. entity/relation data models.
 - ▶ Dynamic models – describe system behavior.

Models of existing and planned system

- ▶ Models of an existing system are used for collecting and describing the requirements. They model how the existing system functions and are applied for revealing its strengths and weaknesses. They provide a base for the requirements for a new version of the system.
- ▶ Models of a new system are used for design of the requirements in order to explain them to other stakeholders in the system. They help system engineers to select design proposals and document the system.
- ▶ Model engineering process – allows to generate a complete or partial implementation of the system from a system model.

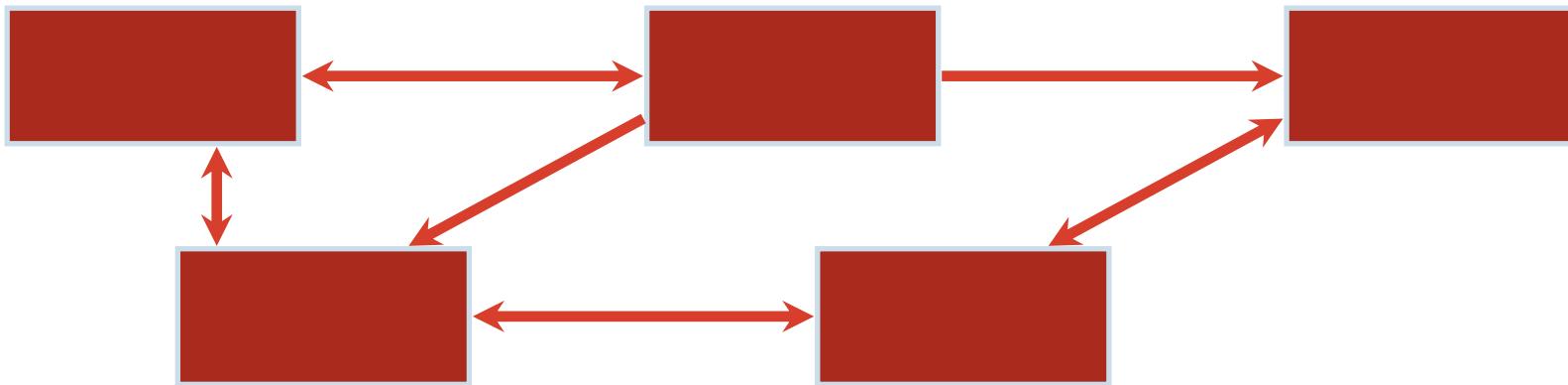
Appliance of system models

Practical system models should be:

- Describing the system in a correct way
- Formal – provide notations and techniques for unambiguous system specification
- Consistent – different views should not describe things being in conflict each other
- Easy to be explained to and understood by other people – as simple as possible but not oversimplified
- Easy for updates and maintenance
- In a form suitable for transfer to other people
- Balanced between visual and textual description

Object-oriented (OO) system modelling

- ▶ **OO system modelling** represents a system as a group of **objects** which co-operate, having a structure and constituting an organic, coherent set.
- ▶ A **system** is a collection of units connected and organized in order to accomplish a specific goal.

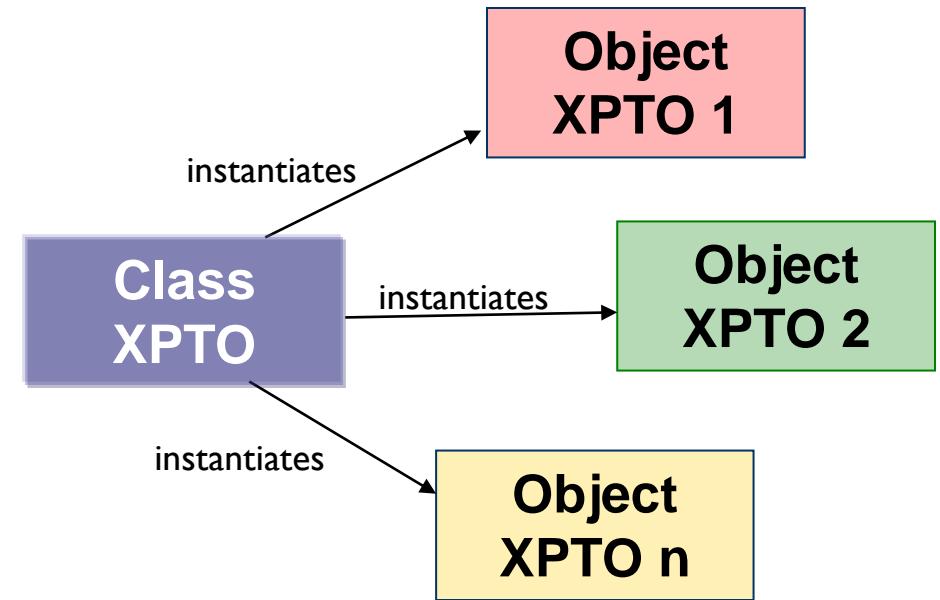


Objects

- ▶ An **object** is an atomic unit having:
 - ▶ an identity
 - ▶ a state (represented by the values of its properties) and
 - ▶ a behavior (operations with possible parameters and, in many cases, returning a result – realized as program methods to be executed)
- ▶ Class properties may be:
 - ▶ Data attributes of specific type (such as String, Integer, Float, ...)
 - ▶ Structural relationships referencing other objects (like pointers to given objects)
- ▶ **Information hiding** – the internal representation, or state, of an object is hidden from the outside

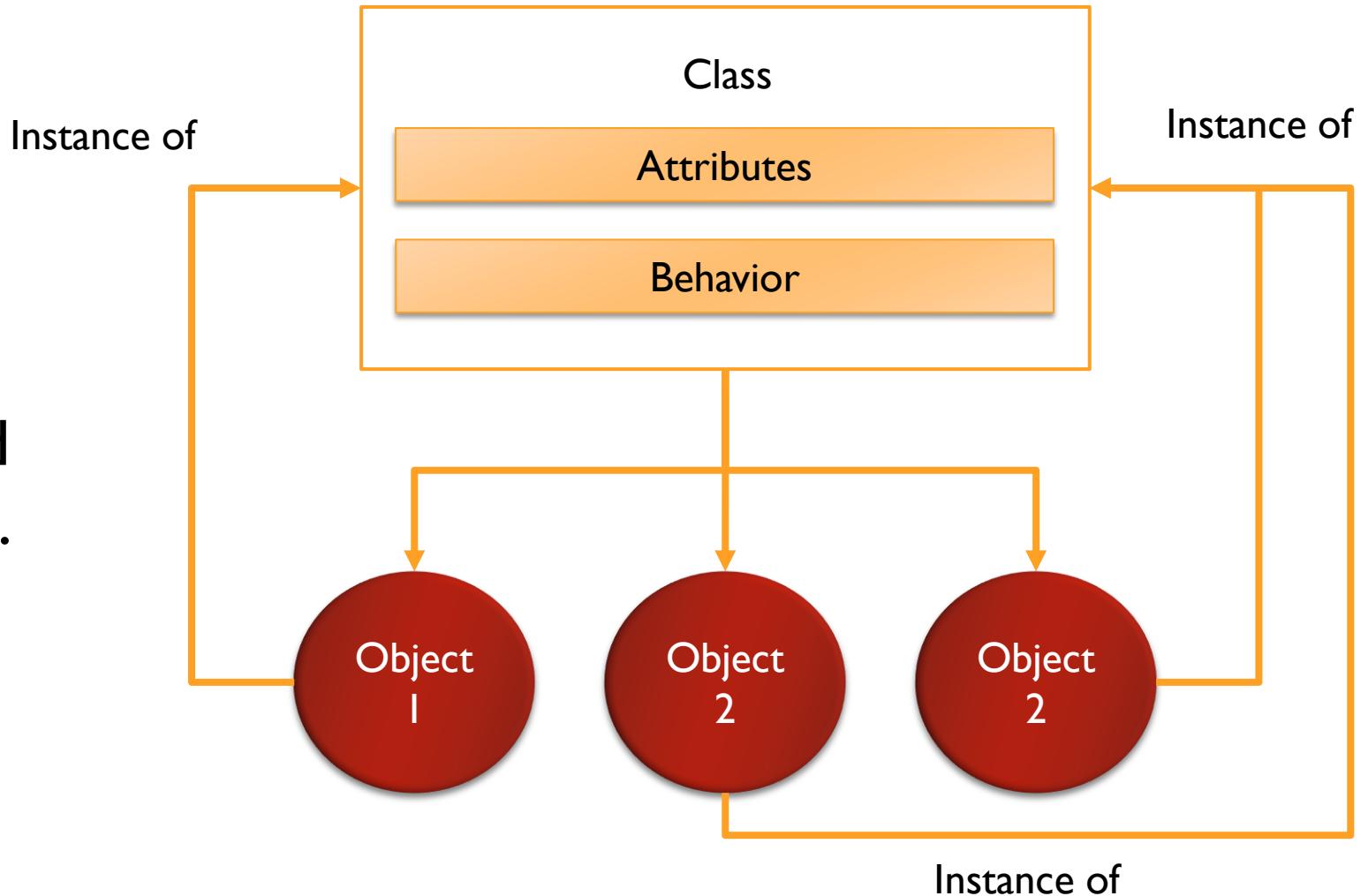
Object and classes

- A **class** is a template to create objects
 - A **class** can be viewed as a container of objects with the same data and structural attributes, operations, and semantics
 - Each object is an instance of a specific class
 - The object cannot be instance (exemplar) of more than one class
- **Encapsulation** – bundling data and methods that work on that data within one unit



Object instance

An **object instance** (i.e., **exemplar**) is a specific **object** created from a particular **class**.



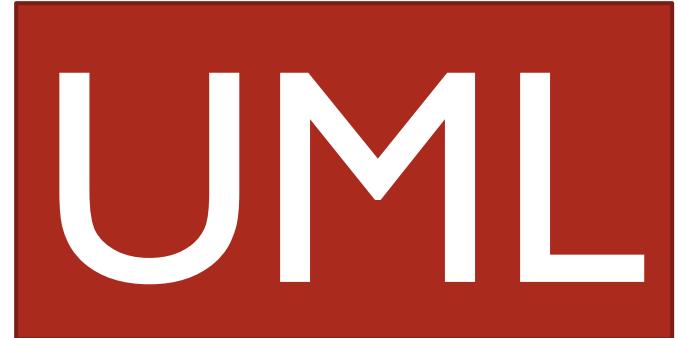
Object-oriented analysis and design

- ▶ OO analysis is a process of defining the problem in terms of objects:
 - ▶ real-world objects with which the system must interact, and
 - ▶ candidate software objects used to explore various solution alternatives.
- ▶ You can define all of your real-world objects in terms of their classes, attributes, and operations.
- ▶ OO design means defining the software solution of the problem by components, interfaces, objects, classes, attributes, and operations that will satisfy the requirements.
- ▶ You typically start with the candidate objects defined during analysis, and add or change objects as needed to refine a design solution.

Unified Modelling Language (UML)

UML is an OO modelling language for:

- ▶ **specifying,**
- ▶ **visualizing,**
- ▶ **constructing, and**
- ▶ **documenting**

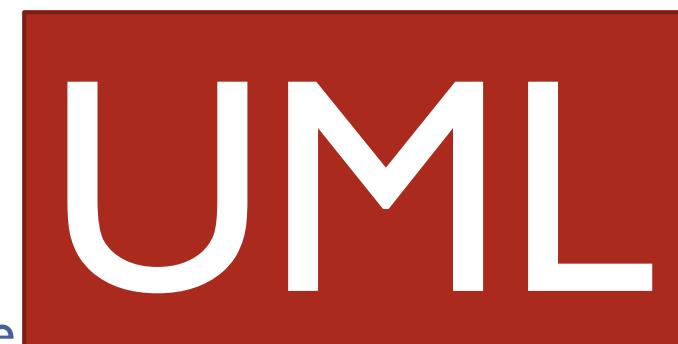


the artifacts of software systems, as well as for business modeling and other non-software systems. As a modeling language UML includes:

- **Model elements** — fundamental modeling concepts and semantics
- **Notation** — visual rendering of model elements
- **Guidelines** — idioms of usage

Goals of UML 1/2

- ▶ Provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models.
- ▶ Furnish extensibility and specialization mechanisms to extend the core concepts:
 - ▶ build models using core concepts without using extension mechanisms for most normal applications,
 - ▶ add new concepts and notations for issues not covered by the core,
 - ▶ choose among variant interpretations of existing concepts, when there is no clear consensus,
 - ▶ specialize the concepts, notations, and constraints for particular application domains.



Goals of UML 2/2

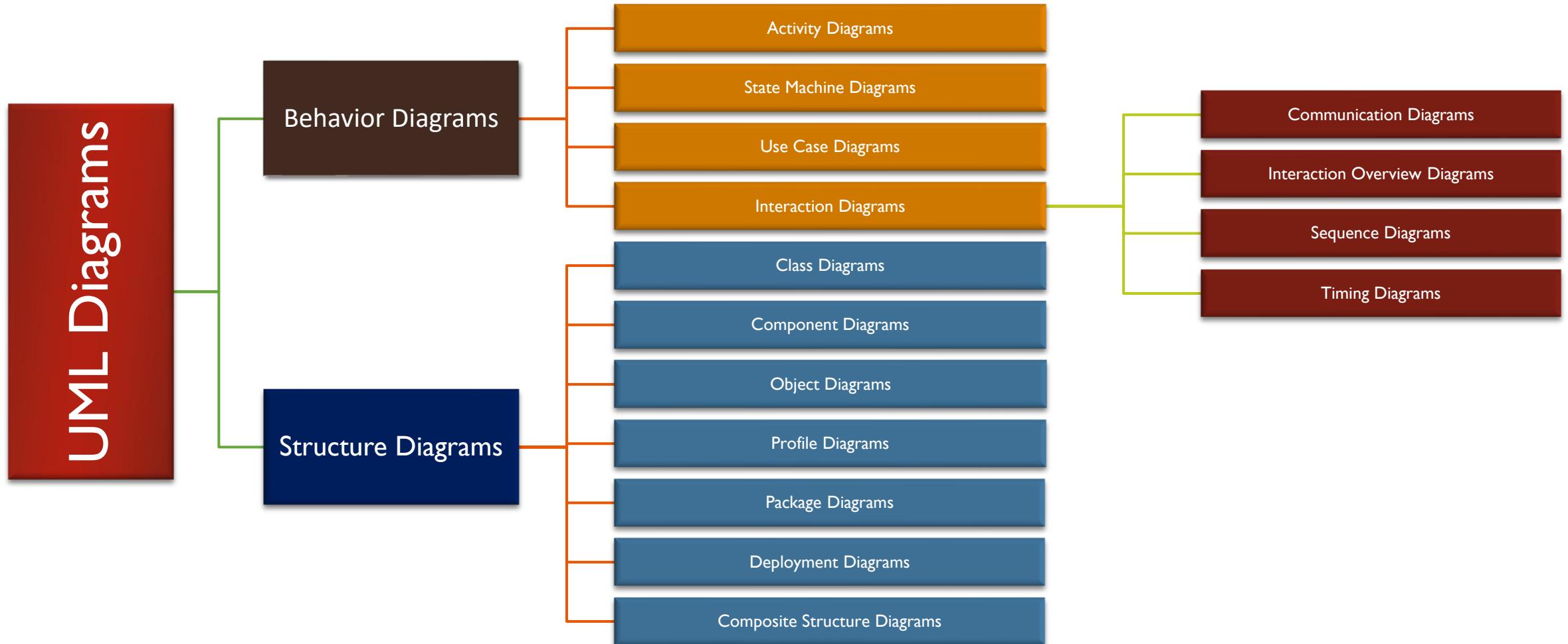
- ▶ Support specifications that are independent of particular programming languages and development processes.
- ▶ Provide a formal basis for understanding the *process-independent* modeling language.
- ▶ Encourage the growth of the object tools market.
- ▶ Support higher-level development concepts such as components, collaborations, frameworks and patterns.
- ▶ Integrate best practices – UML fuses the concepts of Booch, OMT, and OOSE, in a single, common, and widely usable modeling language.



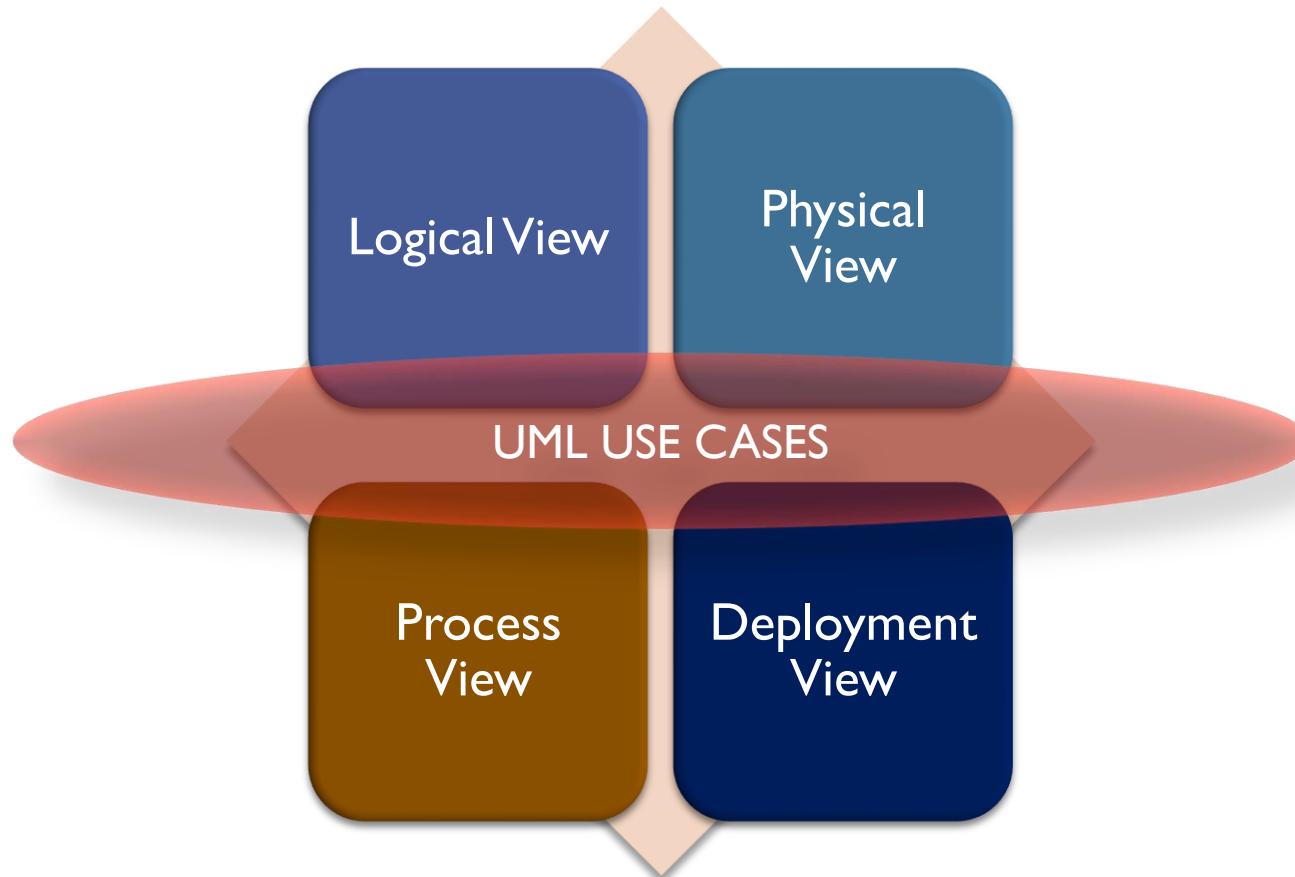
UML diagrams

- ▶ UML is the modern, general-purpose approach to modeling and documenting software systems and business processes
- ▶ UML facilitates it by diagrams of various types
- ▶ UML diagrams represent the OO analysis and design solutions
- ▶ You can draw UML diagrams by hand or by using CASE (Computer Aided Software Engineering) tools
- ▶ Using CASE tools requires some expertise, training, and commitment by the project management

Types of UML diagrams



The 4+1 views of the software architecture



The 4+1 views of the software architecture

Scenarios (UML Use Cases)

Logical View
(Object-oriented
Decomposition)

Development View
(Subsystem
Decomposition)

Physical View (Mapping
the Software to
Hardware)

Process View
(Process
Decomposition)

So-called conceptual view -
describes the object model
of the design

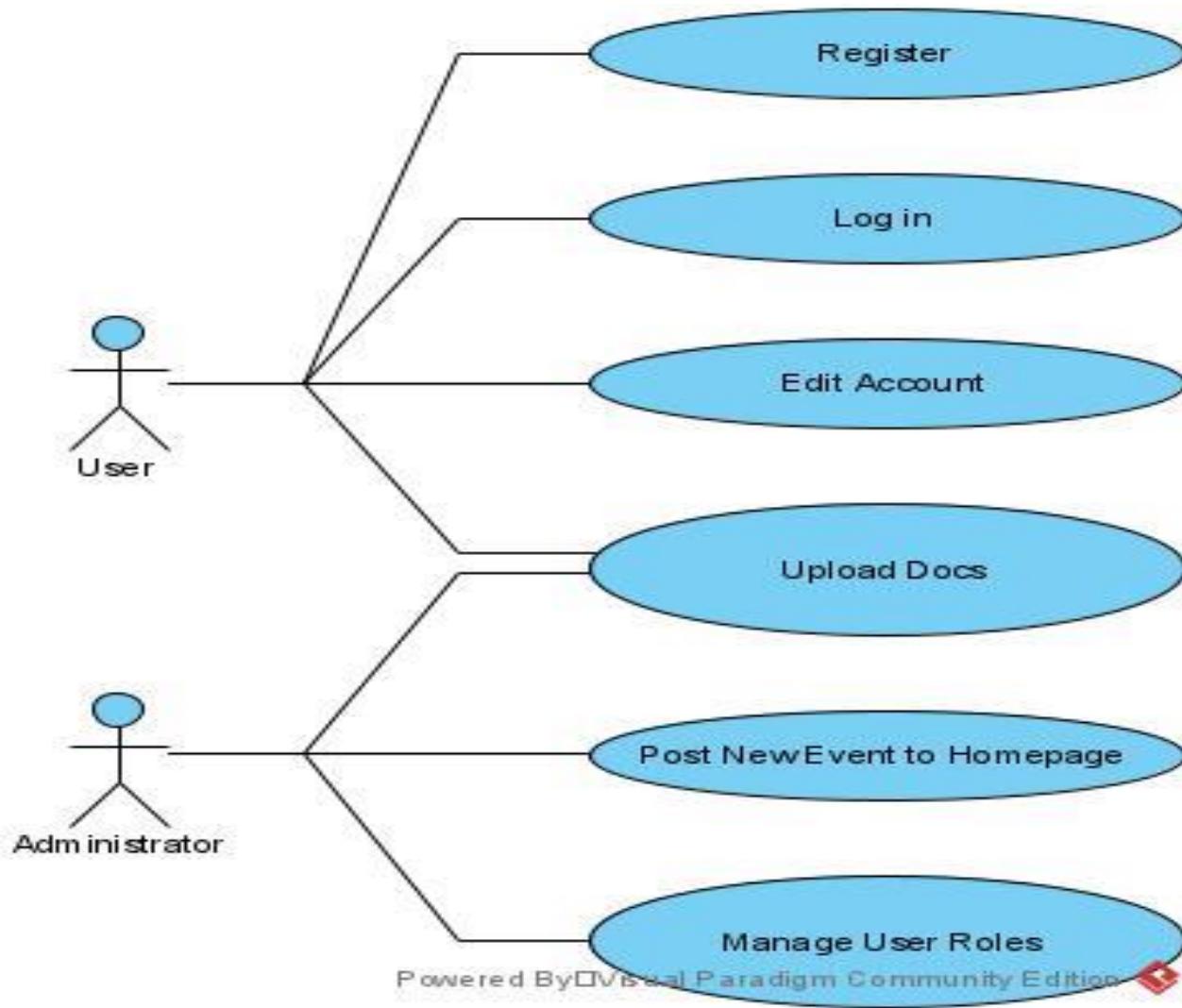
Describes the static
organization or structure
of the code in the
development environment

Describes the deployment
of the software on the
hardware

Describes the aspects of
competitiveness and
synchronization

UML use case diagrams

- ▶ UML use case diagrams provide overview of usage requirements for a system.
- ▶ For actual system or software engineering use case diagrams describe actual system/software requirements
- ▶ Useful also for simple presentations to management and/or project stakeholders



- ▶ A use case diagram shows user's interaction with the system
- ▶ The use case diagrams are used mainly in specification and analysis of requirements

Used Visual Paradigm Community Edition tool: <https://www.visual-paradigm.com/> last accessed 10.03.2023

Elements of use case

Actors - a person, organization, or external system that plays a role in one or more interactions with your system

Use cases - describe a sequence of actions that provide something of measurable value to an actor

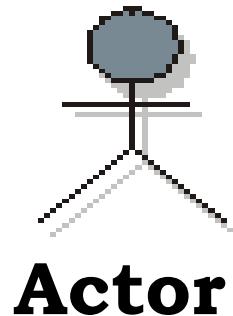
Associations - exist whenever an actor is involved with an interaction described by a use case

Other relations – *include, extend, generalize and depend*

System boundary boxes (optional) - rectangles around the use cases to indicates the scope of your system

Packages (optional) - UML constructs that enable you to organize model elements (such as use cases) into groups

Defining Actors



An **actor instance** is *someone or something* outside the system that interacts with the system.

An **actor class** defines a set of actor instances, in which each actor instance plays the same role in relation to the system.

To fully understand the system's purpose you must know **who** the system is for, or who will use the system. Different user types are represented as actors.

An actor is **anything** that exchanges data with the system. An actor can be a user, external hardware, or another system

How to find actors

Who will supply/use/remove information?

Who will use this functionality?

Who is interested in any requirement?

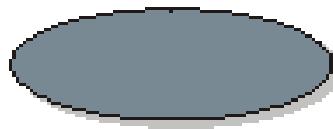
Where in the organization is the system used?

Who will support/maintain the system?

What are the system's external resources?

What other systems will need to interact with this one?

Defining use cases



A **use case *instance* (*scenario*)** is a sequence of actions a system performs that yields an observable result of value for one or more particular actors or other stakeholders of the system.

A **use case (*class*)** defines a set of use-case instances.

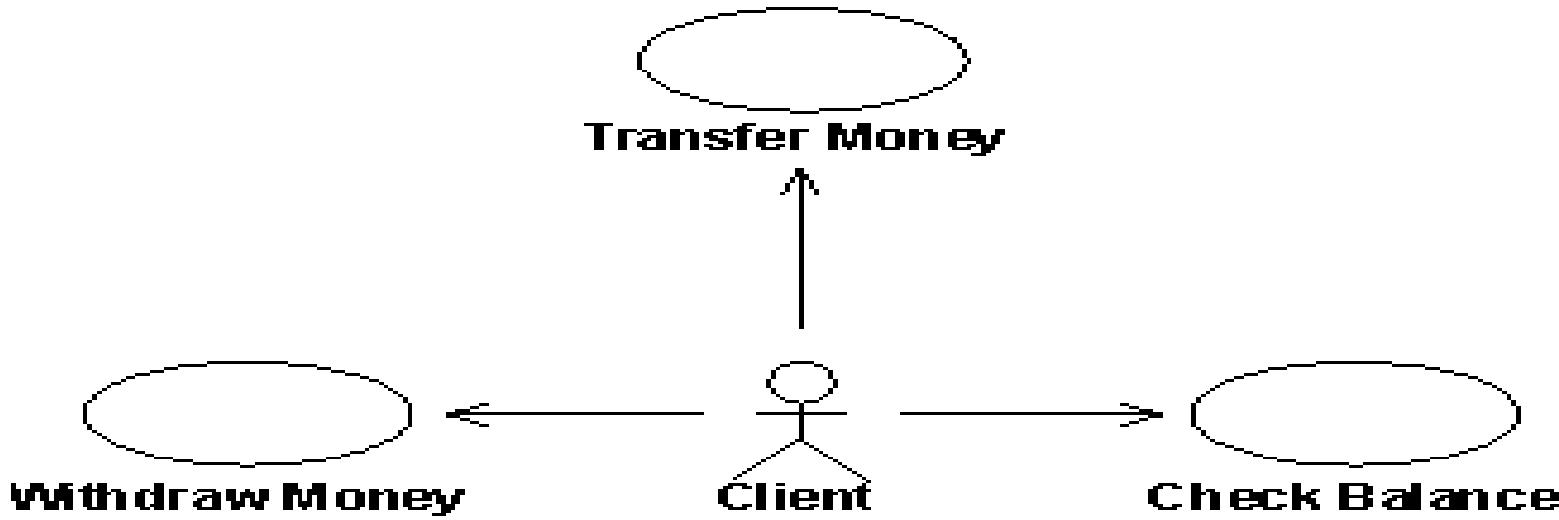
How to find use cases

- What are the system tasks for each actor you have identified?
- Does the actor need to be informed about certain occurrences in the system?
- What information must be modified or created in the system?
- Does the system supply the business with the correct behavior?
- What use cases will support and maintain the system?

Associations (relationships) in use case diagrams

- ▶ Associations between actors and/or use cases are indicated in use case diagrams by solid lines.
- ▶ An association exists whenever an actor is involved with an interaction described by a use case.
- ▶ Associations are modeled as lines connecting use cases and actors to one another
- ▶ The arrowhead is often used to indicate the direction of the initial invocation of the relationship (but not the direction of information exchange)

A sample use case diagram



An ATM example - the system functionality is defined by different use cases, each of which represents a specific flow of events, defines what happens in the system when the use case is performed, and has a task of its own to perform.

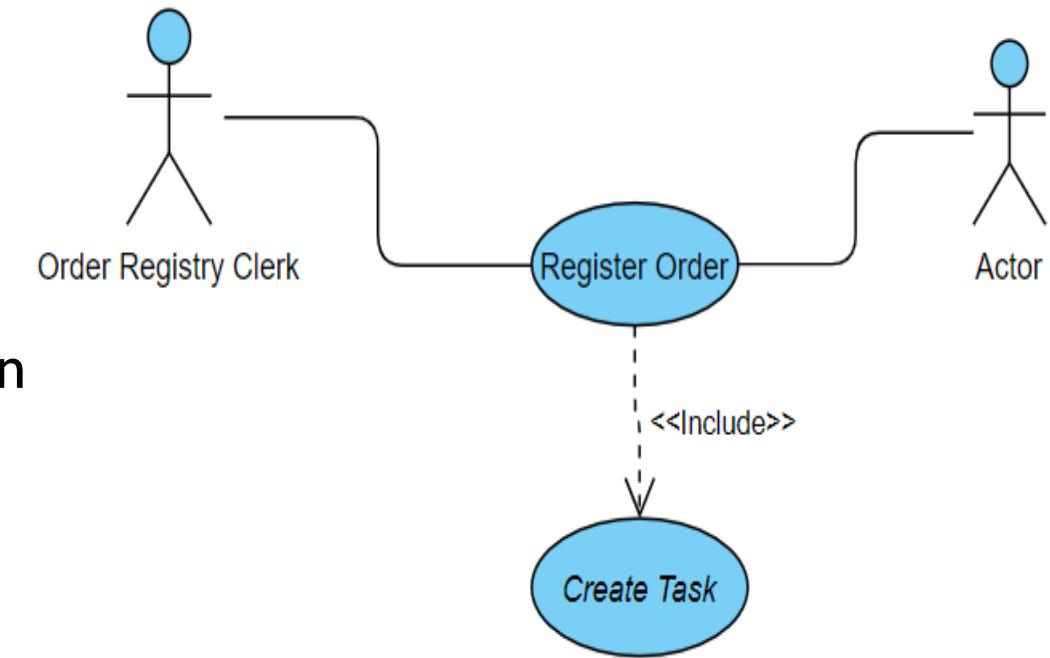
Use case documenting – flow of events

The **Flow of Events** of a use case contains the most important information derived from use-case modeling work. Its contents

- Describe how the use case starts and ends
- Describe what data is exchanged between the actor and the use case
- Do not describe the details of the user interface, unless it is necessary to understand the behavior of the system
- Describe the flow of events, not only the functionality. To enforce this, start every action with "When the actor ... "
- Describe only the events that belong to the use case, and not what happens in other use cases or outside of the system
- Avoid vague terminology such as "for example", "etc." and "information"
- Detail the flow of events - all "whats" should be answered.

Concrete and abstract use cases

- ▶ A **concrete** use case is initiated by an actor and constitutes a complete flow of events (instance of the use case performs the entire operation called for by the actor).
- ▶ An *abstract* use case (written in *italics*) is never instantiated in itself. Abstract use cases are included in, extended into, or generalizing other use cases. When a concrete use case is initiated, an instance of the use case is created. This instance also exhibits the behavior specified by its associated abstract use cases. Thus, no separate instances are created from abstract use cases.



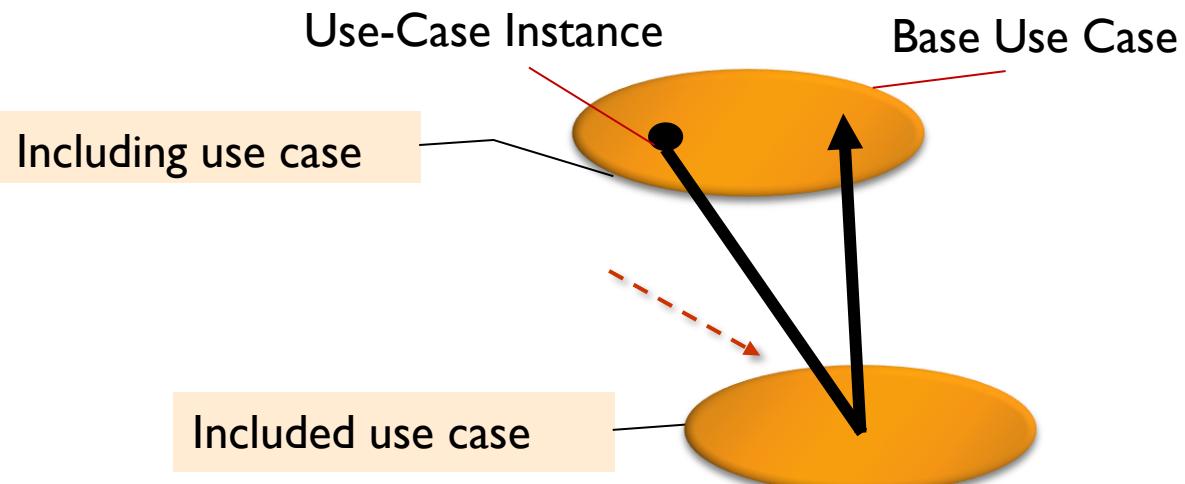
Used Visual Paradigm Community Edition tool: <https://www.visual-paradigm.com/> last accessed 10.03.2023

<<Include>> relationship

An **include-relationship** is a directed relationship from a base use case to an inclusion use case, specifying how the behavior defined for the inclusion use case is non-optionally, explicitly inserted into the behavior defined for the base use case.



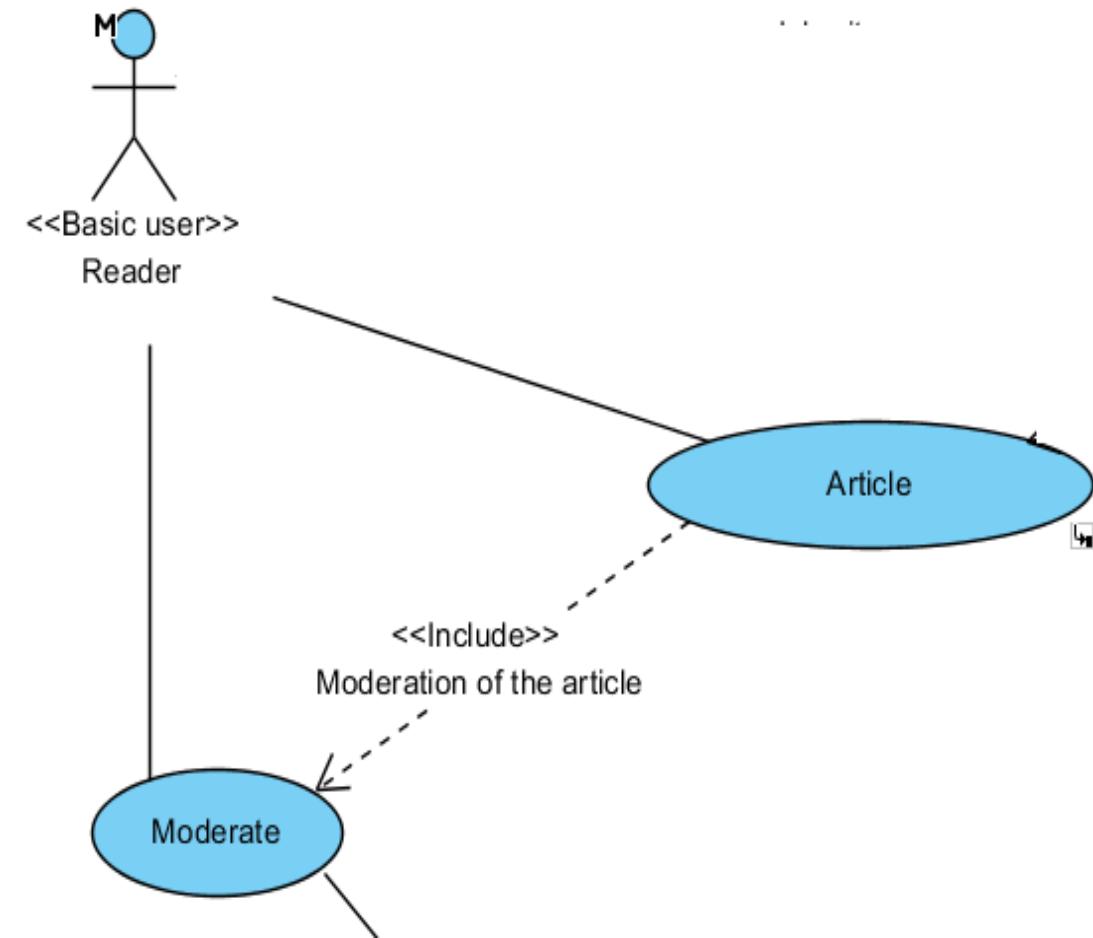
«include»



Executing a use-case instance following the description of a base use case including its inclusion.

More about <<Include>> relationship

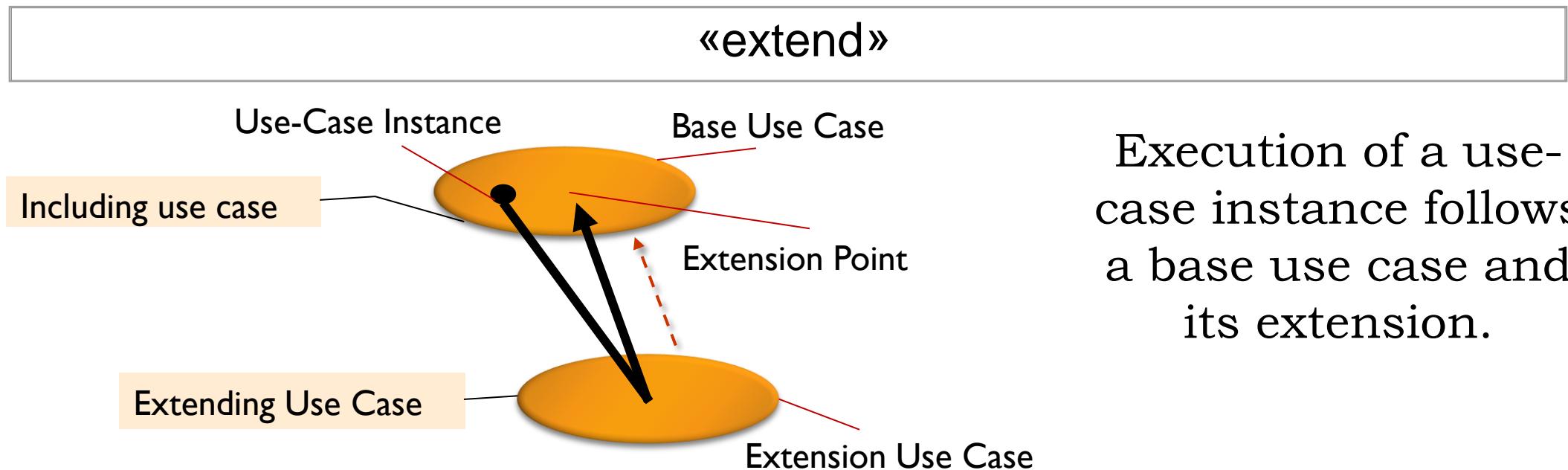
- ▶ Including use case includes the “addition” and owns the include relationship.
- ▶ Addition is use case that is to be included.
- ▶ The including use case may only depend on the result (value) of the included use case.
- ▶ This value is obtained as a result of the execution of the included use case.



Used Visual Paradigm Community Edition tool: <https://www.visual-paradigm.com/> last accessed 10.03.2023

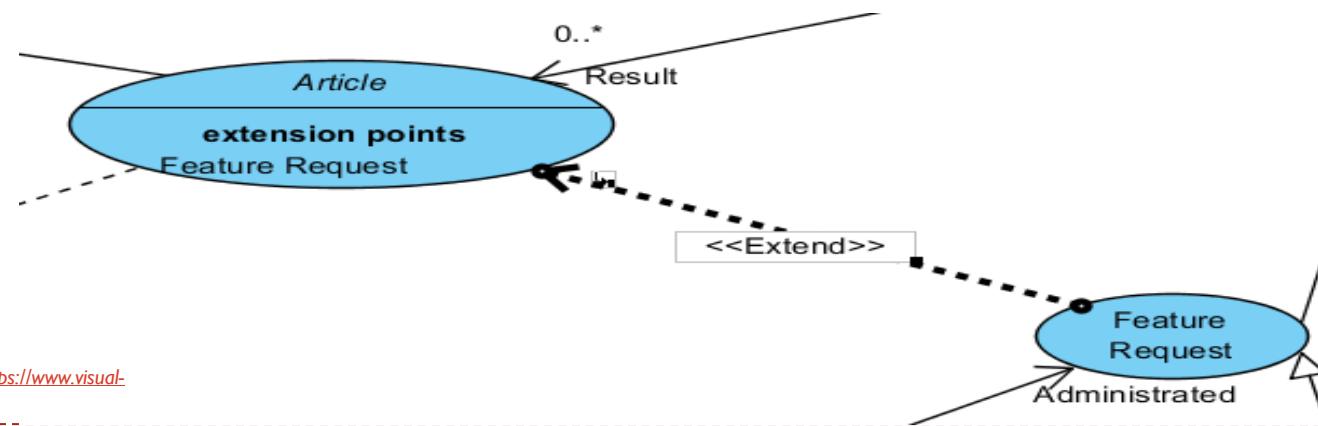
<<Extend>> relationship

An **extend-relationship** goes from an extension use case to a base use case, specifying how the behavior defined for the extension use case can be inserted into the behavior of the base use case. It is implicitly inserted in the sense that the extension is not shown in the base use case.



More about <<Extend>> relationship

- ▶ This relationship specifies that the behavior of a use case may be extended by the behavior of another (supplementary) use case.
- ▶ The extended use case is defined independently of the extending use case and is meaningful independently of the extending use case.
- ▶ On the other hand, the extending use case typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending use case defines a set of modular behavior increments that augment an execution of the extended use case **under specific conditions**.



Used Visual Paradigm Community Edition tool: <https://www.visual-paradigm.com/> last accessed 10.03.2023

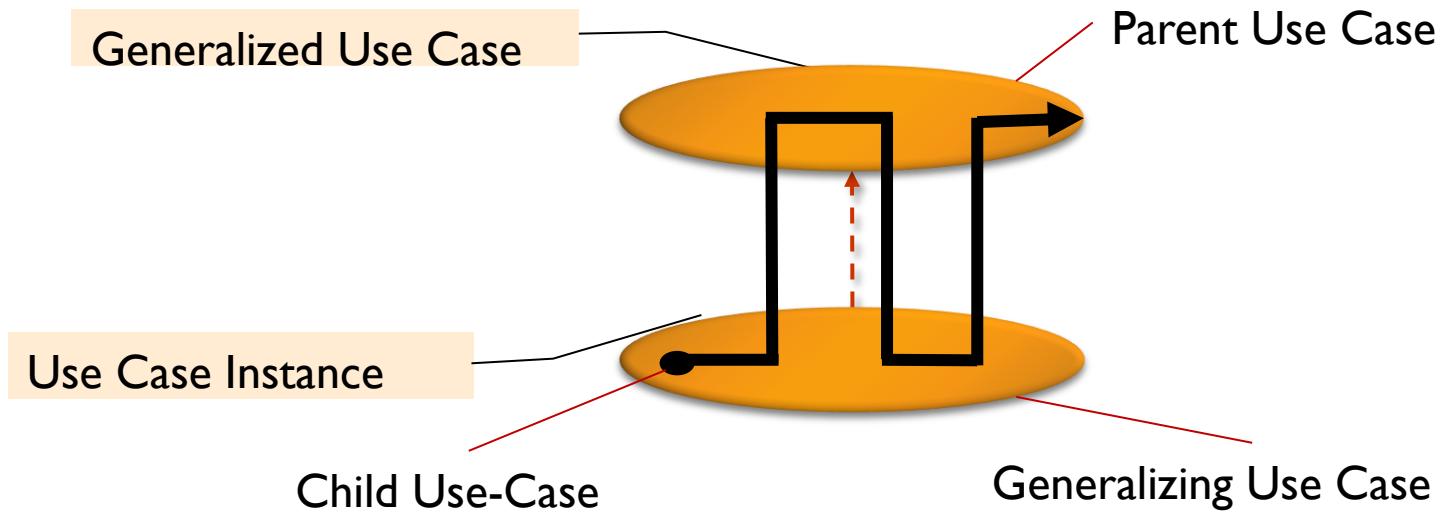
Extension points

- ▶ Extension points (since UML 2.0) show the actual logic necessary for one use case to extend another.
- ▶ An extension point identifies the point in the base use case where the behavior of an extension use case can be inserted.
- ▶ The extension point is specified for a base use case and is referenced by an extend relationship between the base use case and the extension use case.

Use case generalization

A **use-case-generalization** is a taxonomic relationship from a child use case to a more general, parent use case, specifying how a child can specialize all behavior and characteristics described for the parent.

Use case generalization

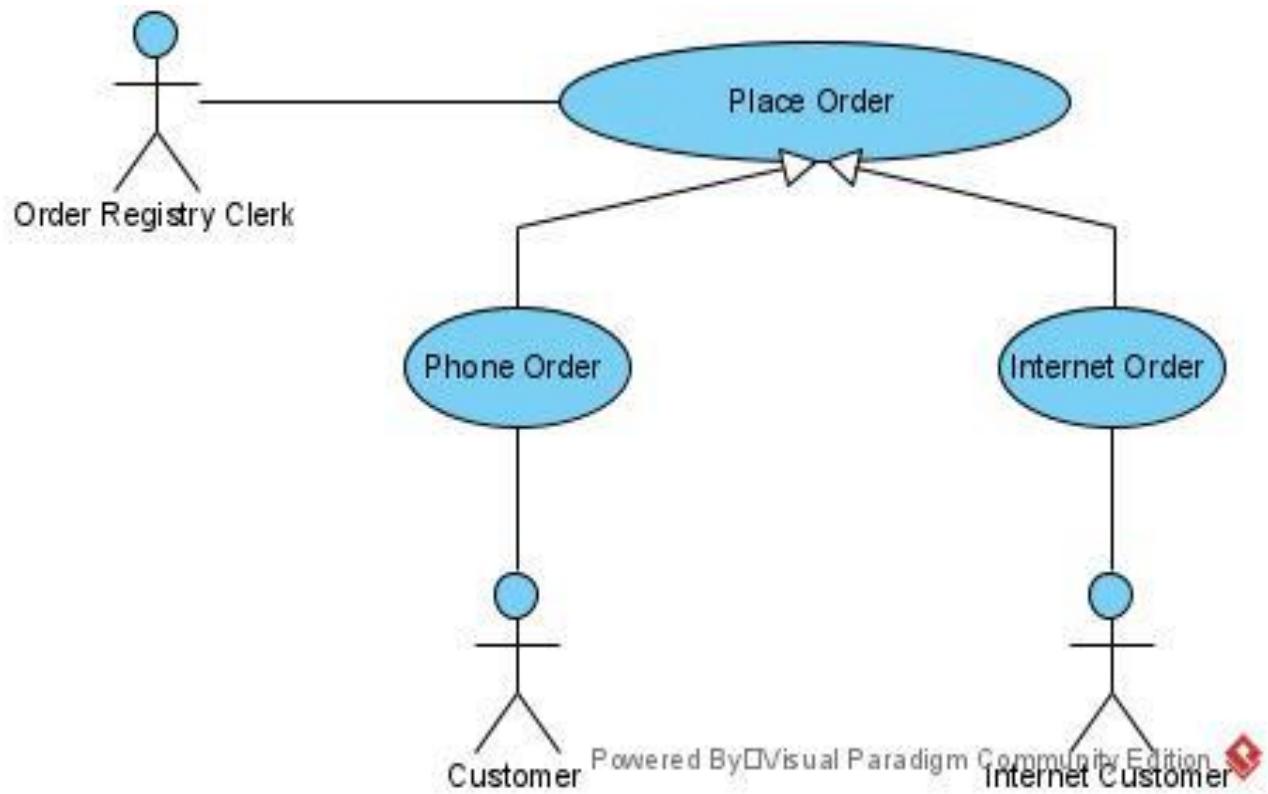


Execution: the use-case instance follows the parent use case, with behavior inserted or modified as described in the child use case

More about generalization

| | |
|-------------------|---|
| General Use Case | References the general classifier in the Generalization relationship. |
| Specific Use Case | References the specializing classifier in the Generalization relationship. |
| Substitutable | Indicates whether the specific classifier can be used wherever the general classifier can be used. If true, the execution traces of the specific classifier will be a superset of the execution traces of the general classifier. |

Example of use case generalization

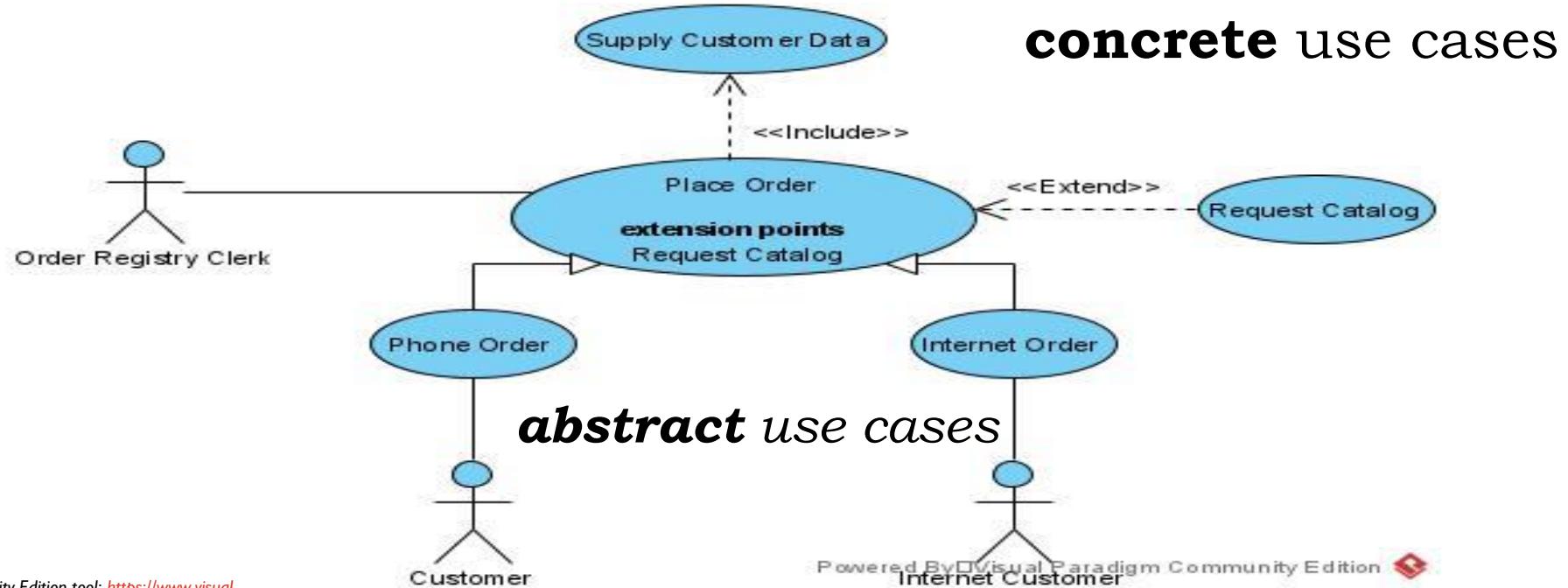
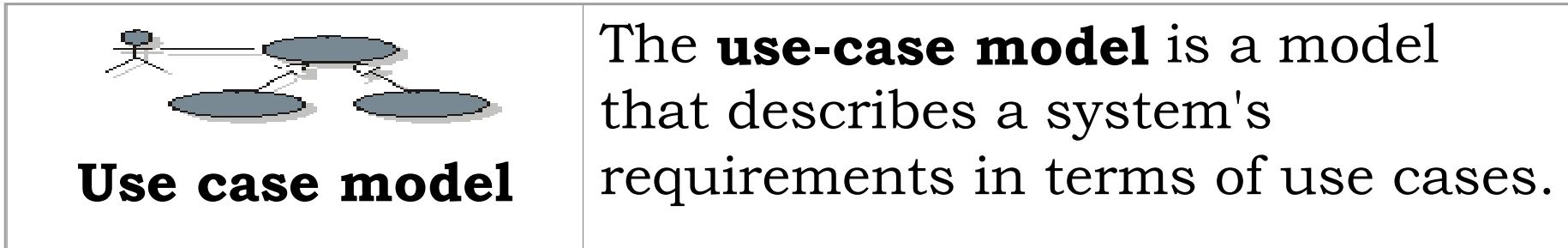


The actor **Order Registry Clerk** can instantiate the general use case **Place Order**. **Place Order** can also be specialized by the use cases **Phone Order** or **Internet Order**.

The child may modify behavior segments inherited from the parent. The structure of the parent use case is preserved by the child. Both use-case-generalization and include can be used to reuse behavior among use cases.

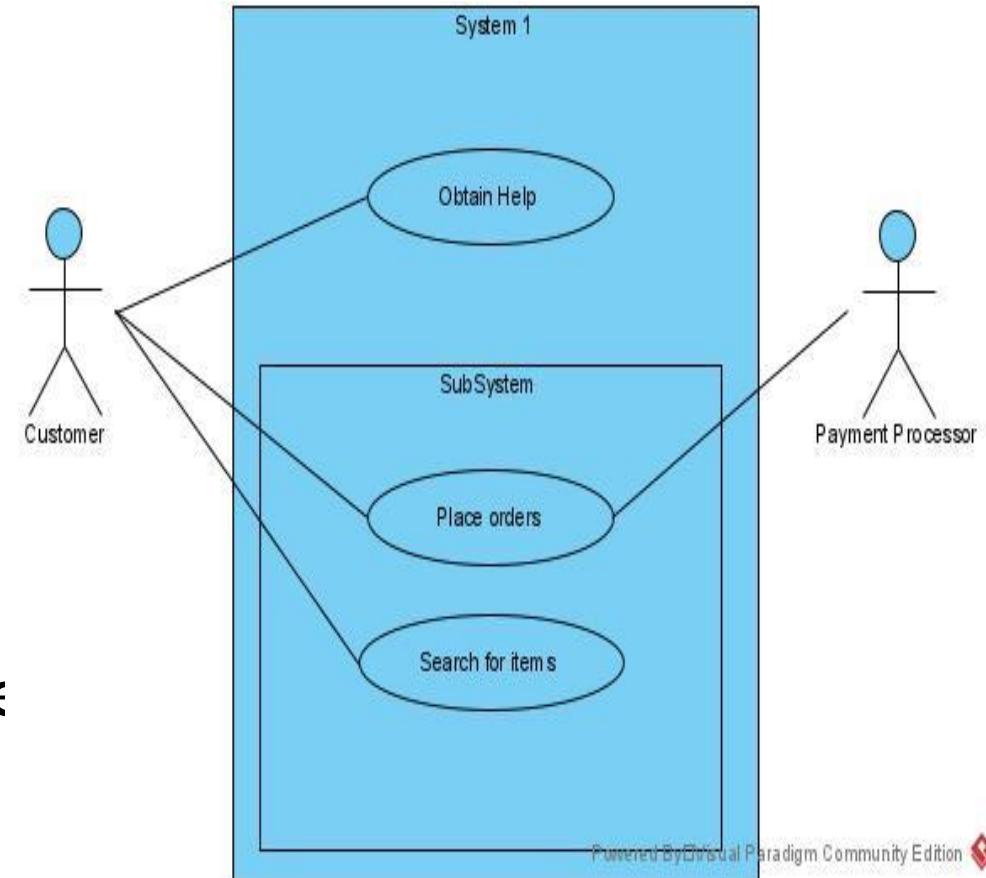
Used Visual Paradigm Community Edition tool: <https://www.visual-paradigm.com> last accessed 10.03.2023

Use case model of an Order Management System



System boundary boxes

- ▶ System boundary box (optional) - a rectangle around the use cases to indicates the scope of your sub-system
- ▶ Anything within the box represents functionality that is in scope and anything outside the box is not
- ▶ Rarely used – i.e., to identify which use cases will be delivered in each major release of a system



Used Visual Paradigm Community Edition tool: <https://www.visual-paradigm.com/> last accessed 10.03.2023

References

- ▶ These slides are for educational purposes and used in the FMI Course “Software Technologies” and are part of the ICT-TEX Project 2022.
- ▶ Sommerville, I. *Software Engineering*. 10th edition, Published by Pearson Education, ISBN: 978-1-292-09613-1 (2016)
- ▶ Pressman, R., Maxim, B. *Software Engineering: A Practitioner's Approach*. 9th edition, Published by McGraw-Hill Education, ISBN: 9781260548006, (2019)
- ▶ Page-Jones, M., Constantine, L. *Fundamentals of object-oriented design in UML*, Addison-Wesley, ISBN: 0-201-69946-X (2000)



Моделиране на софтуерни системи. Унифициран език за моделиране (UML).

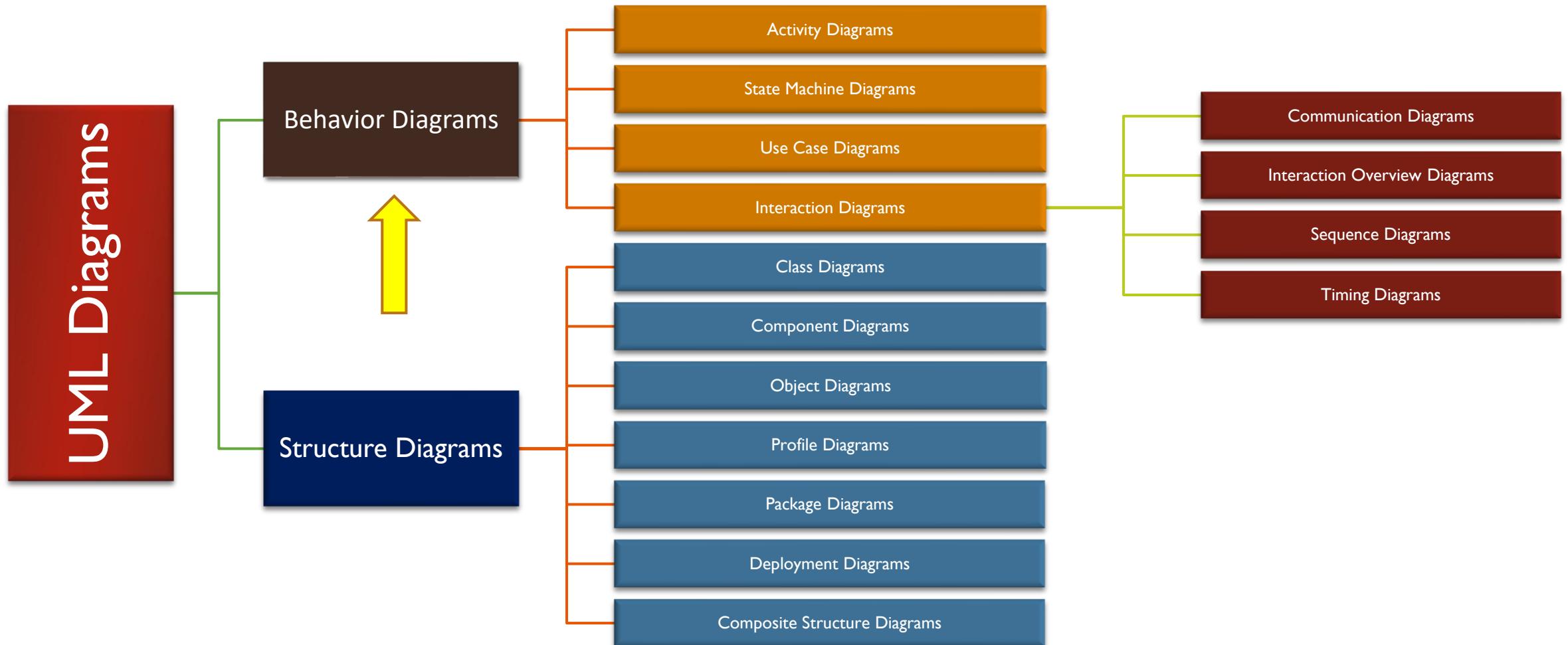


Втора Част

UML diagrams

- ▶ **UML** is the modern, general-purpose modelling approach to specifying, visualizing, constructing, and documenting the artifacts of software systems, business modeling and other non-software systems
- ▶ **UML** facilitates it by diagrams of various types
- ▶ **UML** diagrams represent the OO analysis and design solutions
- ▶ You can draw **UML diagrams** by hand or by using CASE (Computer Aided Software Engineering) tools
- ▶ Using CASE tools requires some expertise, training, and commitment by the project management

Types of UML diagrams



The 4+1 views of the software architecture

Scenarios (UML Use Cases)

Logical View
(Object-oriented
Decomposition)

Development View
(Subsystem
Decomposition)

Physical View (Mapping
the Software to
Hardware)

Process View
(Process
Decomposition)

So-called conceptual view -
describes the object model
of the design

Describes the static
organization or structure
of the code in the
development environment

Describes the deployment
of the software on the
hardware

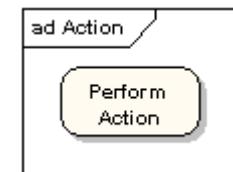
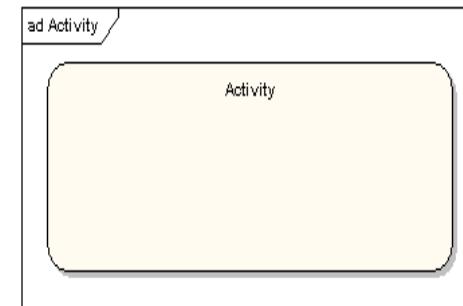
Describes the aspects of
competitiveness and
synchronization

UML Activity diagrams

- ▶ While UML use cases diagrams describe the behavior of the target system from an external point of view and capture user requirements, the UML activity diagrams provide a way to model the workflow (sequence of activities producing observable value) of a business process.
- ▶ An activity diagram is basically a special case of a state machine in which most of the states are activities and most of the transitions are implicitly triggered by completion of the actions in the source activities.

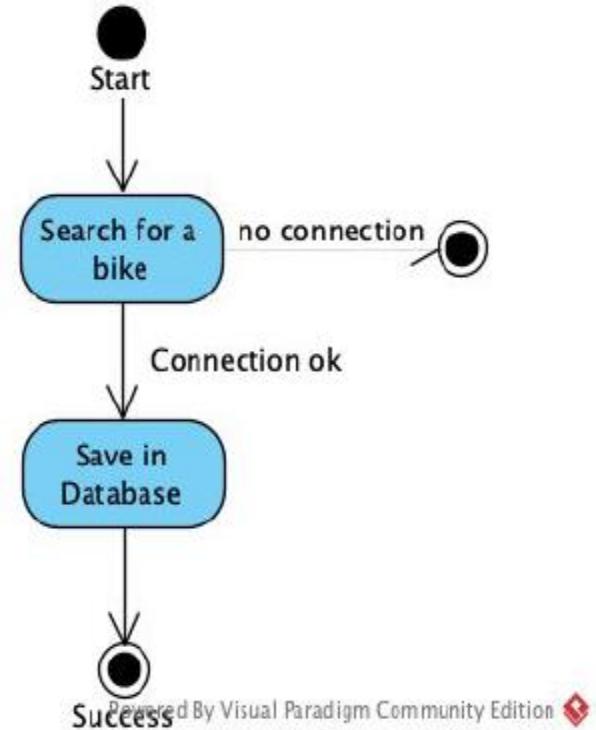
Activities and actions

- ▶ An **activity diagram** may describe *a use case, an operation or a message*
 - the purpose is to describe implementation-oriented details
- ▶ An activity diagram is composed by mainly by *activities, actions, transitions, pins, decisions, synchronizations, and swimlanes*.
- ▶ An *activity* is a sequence of actions that take finite time and can be interrupted; the specification of a parameterized sequence of behavior. An activity is shown as a round-cornered rectangle enclosing all the actions, control flows and other elements that make up the activity
- ▶ An *action* is an atomic task that cannot be interrupted (at least from user's perspective). An action represents a single step within an activity.



Transitions

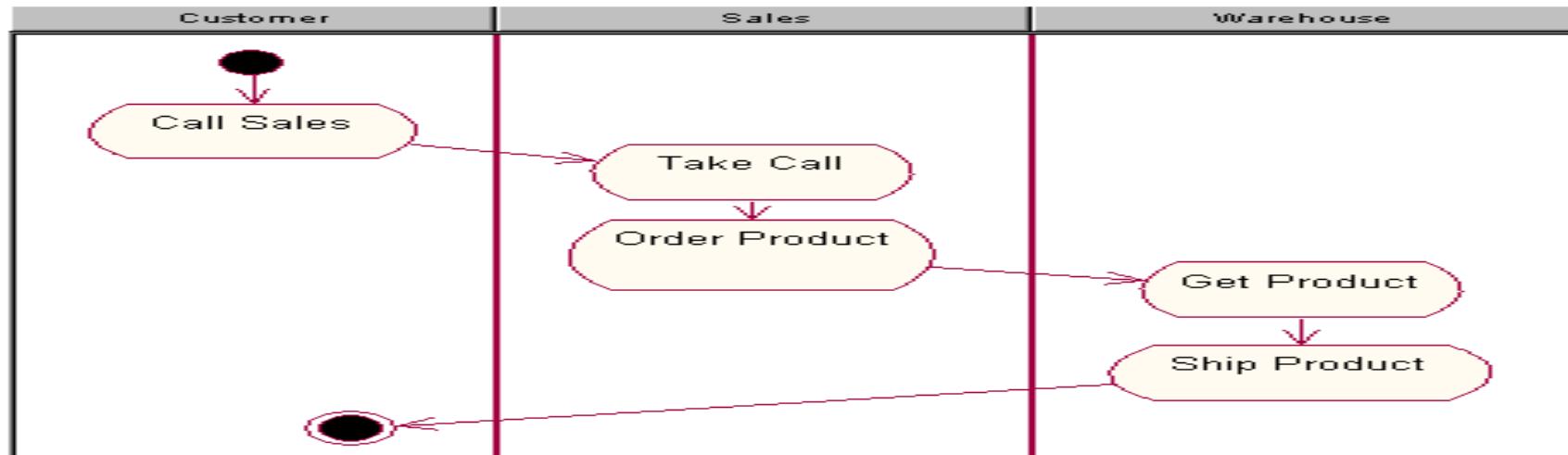
- ▶ Transitions in an activity diagram:
 - ▶ Are presented by arrows
 - ▶ Indicate the completion of an action or sub-activity and show the sequence of actions or sub-activities
 - ▶ Consequently, these transitions are not based on external events
- ▶ In the left figure, the ovals represent activities, the arrows show transitions, and ● and ○ present initial and final states, resp.



Used Visual Paradigm Community Edition tool:
<https://www.visual-paradigm.com/> last accessed 10.03.2023

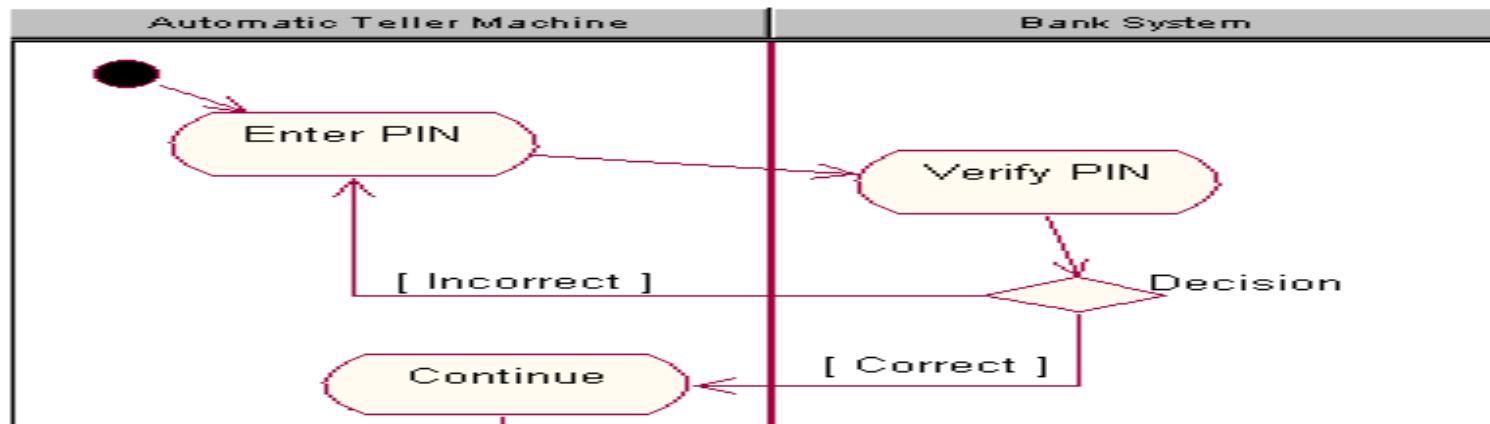
Swimlanes (partitions)

- ▶ Swimlanes only appear on activity diagrams and determine which unit is responsible for carrying out the specific activity.
- ▶ Example: Get Product and Ship Product activities reside within the Warehouse swimlane indicating that the warehouse is responsible for getting the correct product and then shipping the product to the customer. The workflow ends when the customer (noted through the Customer swimlane) receives the product.



Decision nodes

- ▶ A **decision** represents a specific location where the workflow may branch based upon guard conditions. There may be more than two outgoing transitions with different guard conditions, but for the most part, a decision will have only two outgoing transitions determined by a Boolean expression.
- ▶ The following figure displays a decision with [Correct] and [Incorrect] as the guard conditions. If the personal identification number (PIN) is incorrect, the flow of control goes back to the Enter PIN activity. If it is correct, the flow of control moves to the Continue activity.

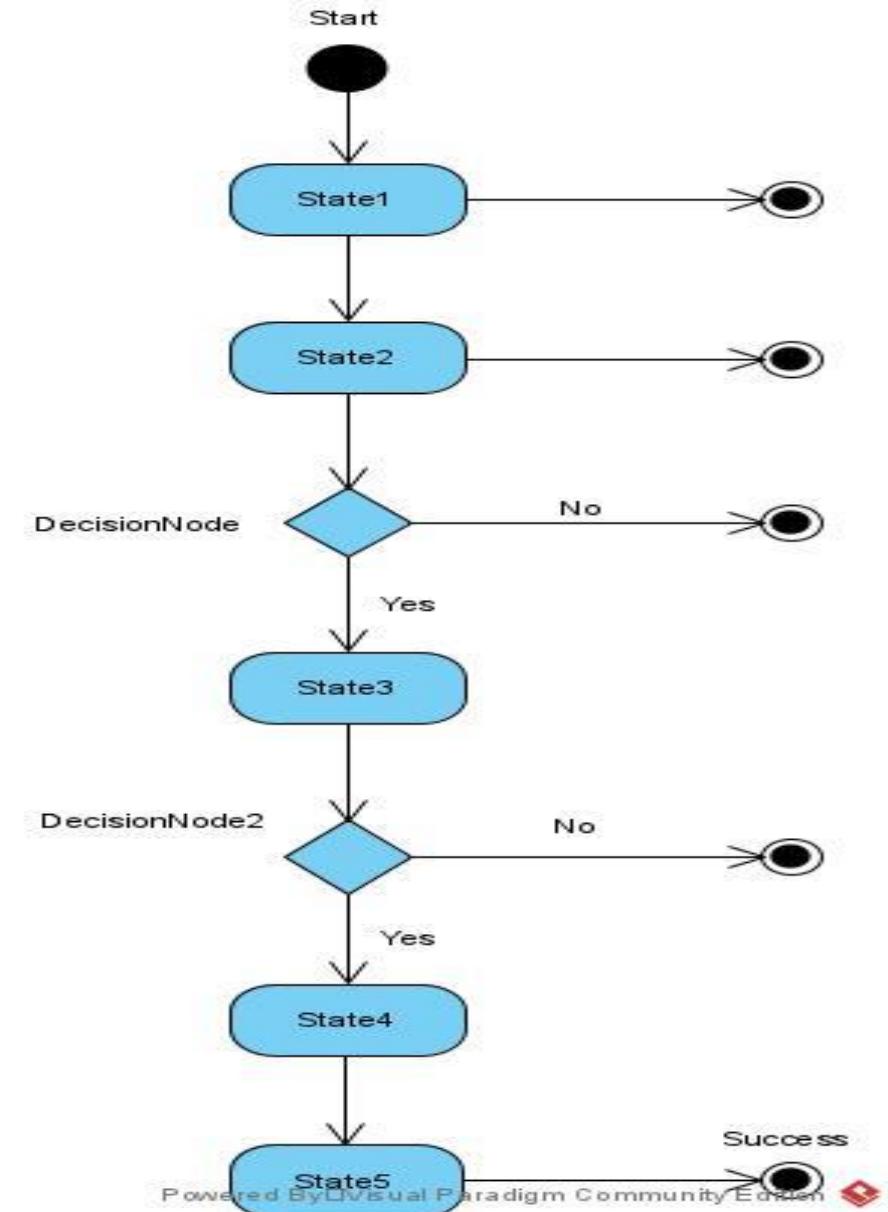


Used Visual Paradigm Community Edition tool:
<https://www.visual-paradigm.com/> last accessed 10.03.2023

A Sample state diagram

The diagram on the right has:

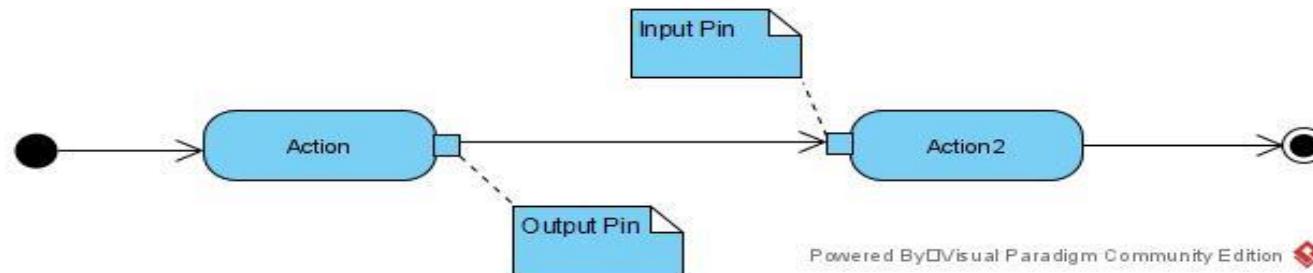
- ▶ One start and five final states,
- ▶ Four activities
- ▶ Two decision blocks
- ▶ 12 transitions



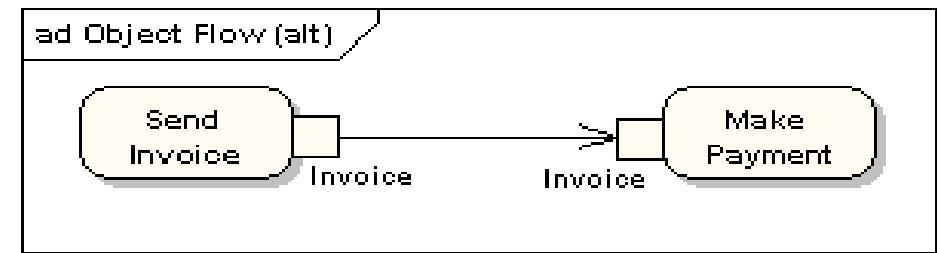
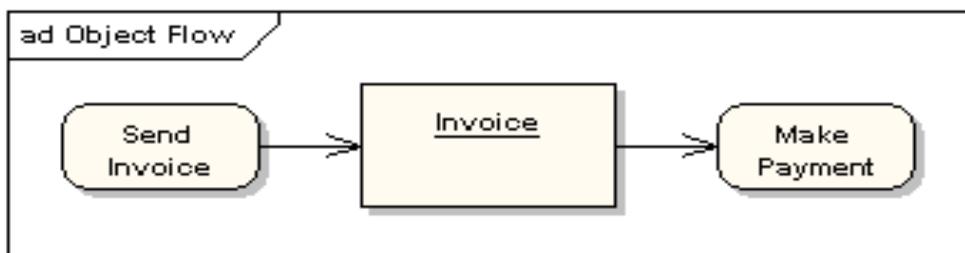
Used Visual Paradigm Community Edition tool:
<https://www.visual-paradigm.com/> last accessed 10.03.2023

Pins (since UML 2.0)

- ▶ An *input pin* means that the specified object is input to an action.
- ▶ An *output pin* means that the specified object is output from an action.



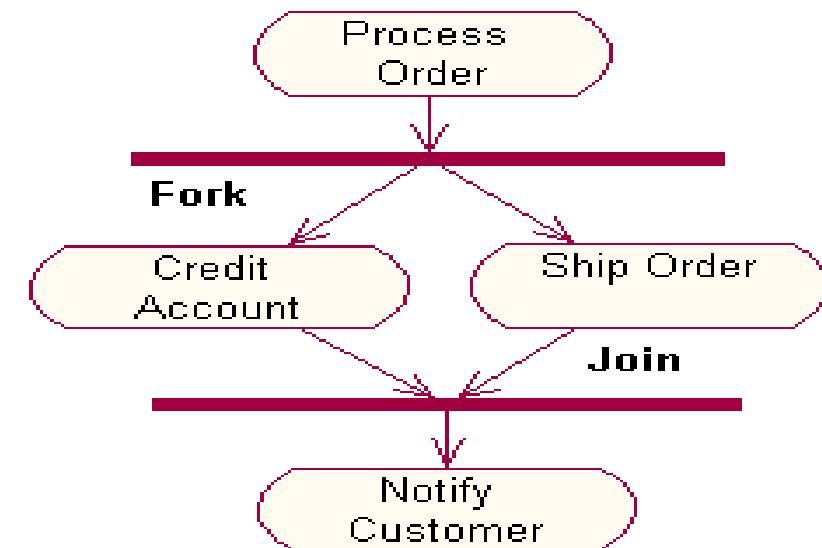
- ▶ An object flow must have an object on at least one of its ends (left figure below). A shorthand notation for the above diagram would be to use input and output pins (right figure below).



Used Visual Paradigm Community Edition tool:
<https://www.visual-paradigm.com/> last accessed 10.03.2023

Synchronizations

- ▶ **Synchronizations** enable you to see a simultaneous workflow. Synchronizations visually define forks and joins representing parallel workflow.
- ▶ A **fork** construct is used to model a single flow of control that divides into two or more separate, but simultaneous flows.
- ▶ A **join** consists of two or more flows of control that unite into a single flow of control. All activities and states that appear between a fork and join must complete before the flow of controls can unite into one.



Used Visual Paradigm Community Edition tool:
<https://www.visual-paradigm.com/> last accessed 10.03.2023

State diagrams

- ▶ A **state (statechart) diagram** shows a state machine, a dynamic behavior that specifies the sequences of states that an object goes through during its life in response to events, together with its responses and actions.
- ▶ A **state machine diagram** models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.
- ▶ It shows the sequences of states that an object goes through, the events that cause a transition from one state to another, and the actions that result from a state change.

States

- ▶ **State** - represents a condition or situation during the life of an object during which it:
 - ▶ Satisfies some condition
 - ▶ Performs some action
 - ▶ Waits for some event
- ▶ May be of type:
 - ▶ Simple or composite state
 - ▶ Real or pseudo-state (like history or junction pseudo-states)
- ▶ Each state represents the cumulative history of its behavior. The state icon appears as a rectangle with rounded corners and a name. It also contains a compartment for actions.

State transitions

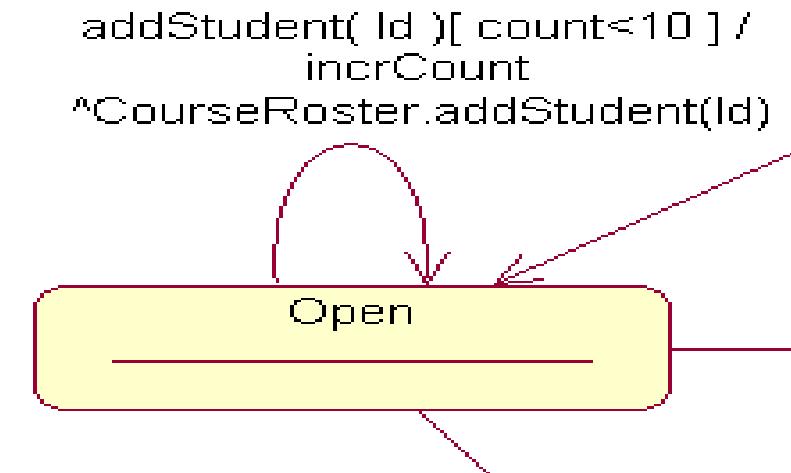
- ▶ A **state transition** indicates that an object in the source state will perform certain specified actions and enter the destination state when:
 - ▶ Non-automatic - a specified event occurs
 - ▶ Automatic - when certain conditions are satisfied.
- ▶ It takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type

State transitions details

- ▶ *Naming:* transitions are labeled with the following syntax:

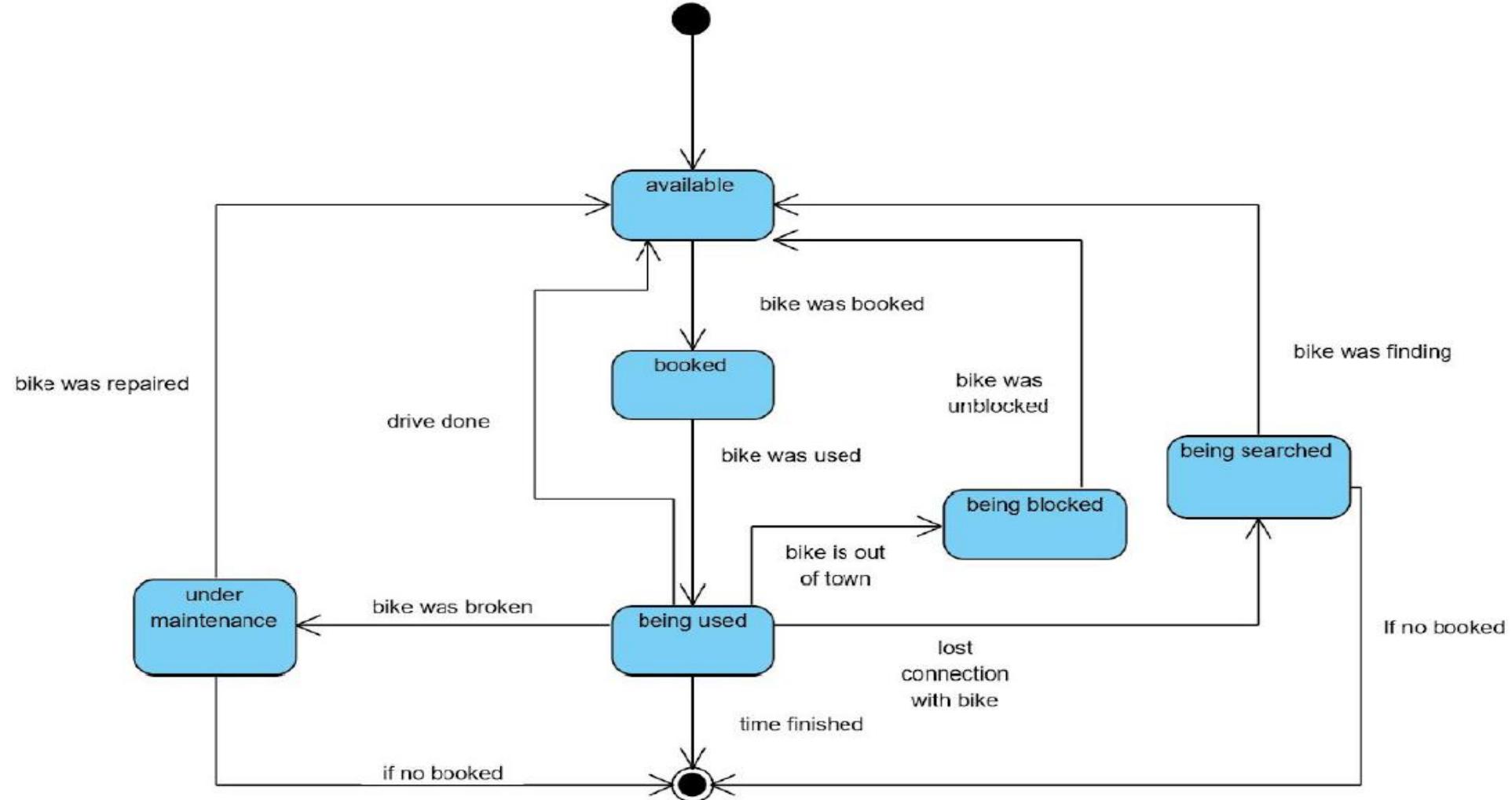
```
event (arguments) [condition] /  
action ^target.sendEvent (args)
```

- ▶ *Triggering event:* behavior (operation) that occurs in the state transition, like addStudent(Id)
- ▶ *Guard condition:* a boolean expression over the attributes that allows the transition, like [count<10]
- ▶ *Action:* an action over the attributes, like incrCount
- ▶ *Send event:* an event invoked in a target object, like ^CourseRoster.addStudent(Id)



Used Visual Paradigm Community Edition tool:
<https://www.visual-paradigm.com/> last accessed 10.03.2023

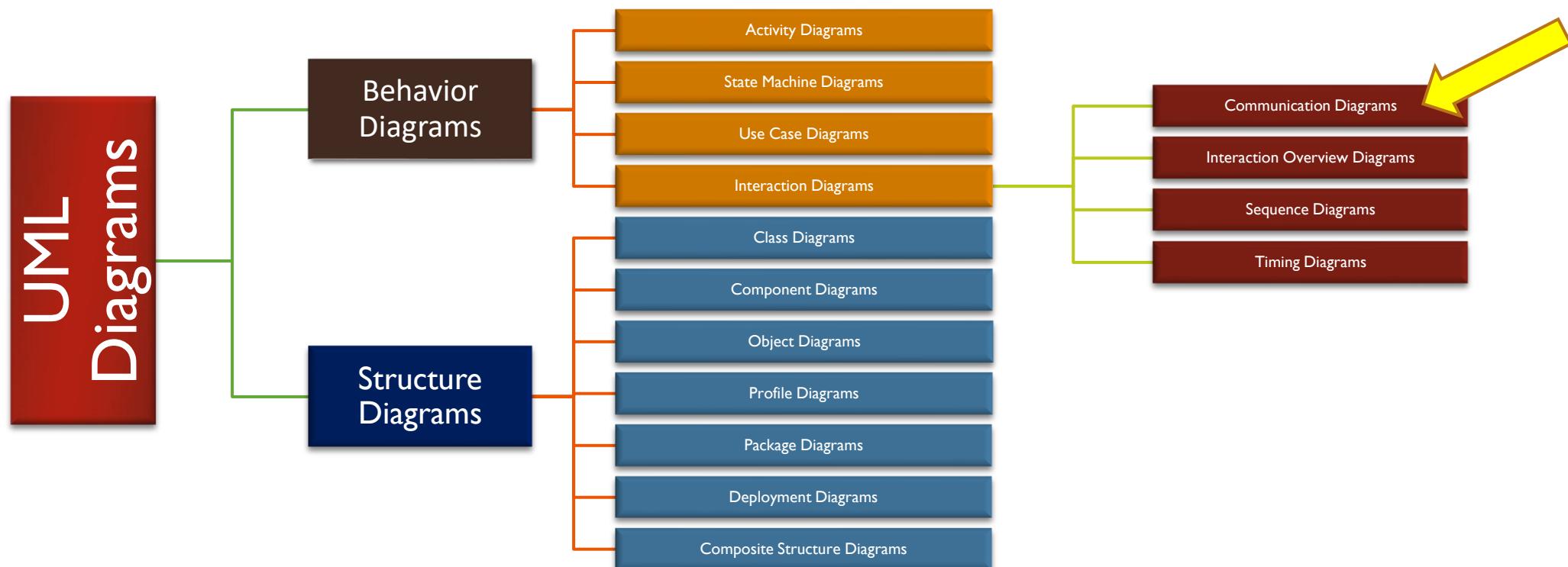
Sample state diagram



Used Visual Paradigm Community Edition tool:
<https://www.visual-paradigm.com/> last accessed 10.03.2023

Interaction diagrams in UML 1/2

For each use-case realization there is one or more interaction diagrams depicting its participating objects and their interactions.
There are four types of interaction diagrams in UML 2*:



Interaction diagrams in UML 2/2

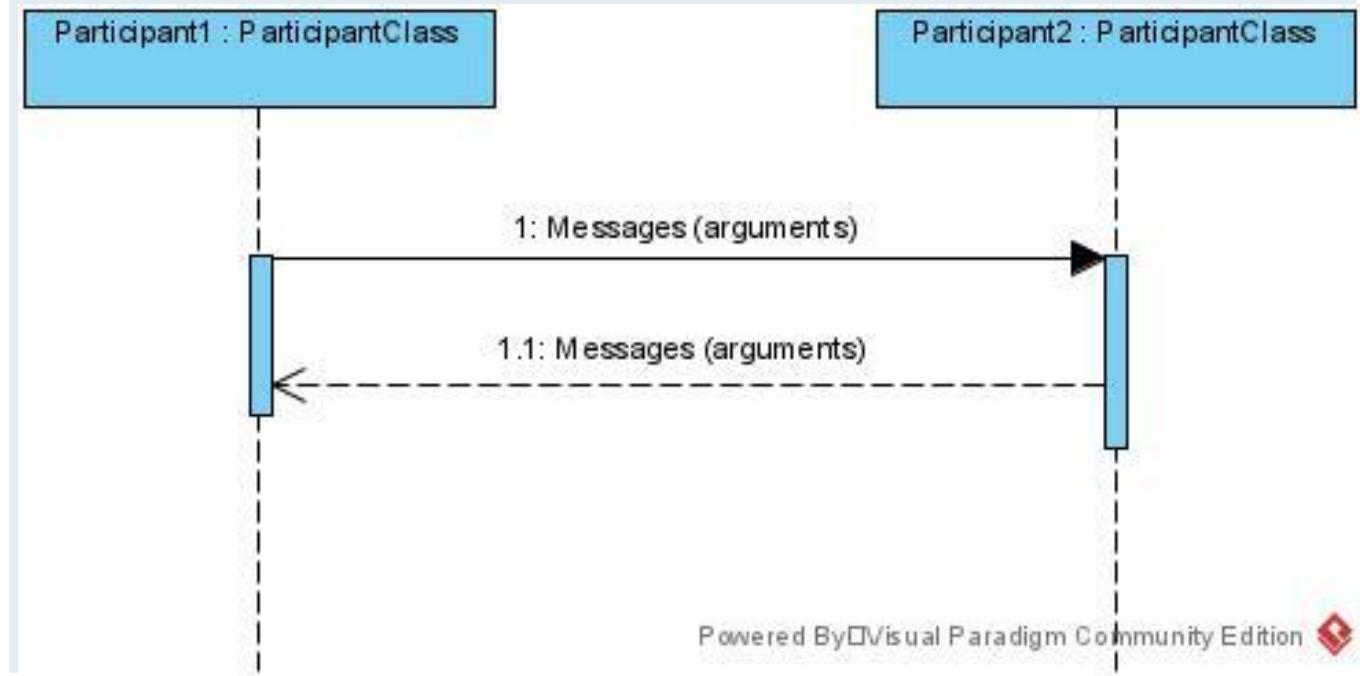
1. **Sequence diagrams** - show the explicit sequence of messages and are better for real-time specifications and for complex scenarios;
2. **Communications** (prior UML 2.* - collaboration) diagrams - show the communication links between objects and are better for understanding all of the effects on a given object and for algorithm design.
3. **Interaction overview diagrams** - show a control flow with nodes that can contain interaction diagrams which show how a set of fragments might be initiated in various scenarios. Interaction overview diagrams focus on the overview of the flow of control where the nodes are interactions (sequence diagrams) or interaction use (a reference);
4. **Timing diagrams** - focus on conditions changing within and among lifelines along a linear time axis; paying attention on time of events causing changes in the modeled conditions of the lifelines.

Sequence diagrams

- ▶ A **sequence diagram** describes a pattern of interaction among objects, arranged in a chronological order.
- ▶ Sequence diagrams show the objects participating in the interaction by their "lifelines" and the messages that they send to each other, i.e. how objects interact to perform the behavior of a use case.
- ▶ Sequence diagrams contain:
 - ▶ objects (i.e. class instances), and
 - ▶ actor instances
- ▶ The diagram describes what takes place in the participating objects, in terms of activations of operations, and how the objects communicate by sending messages.

Messages between objects in sequence diagrams

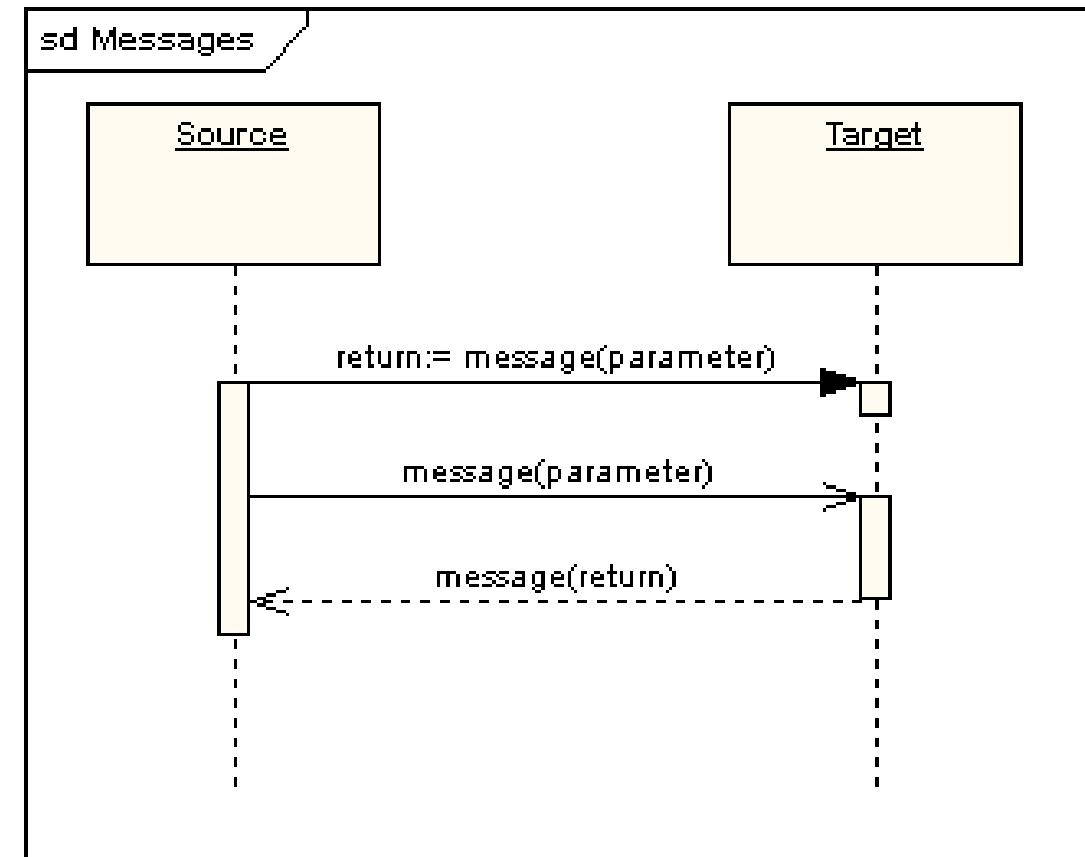
- ▶ **Objects** - shown as a vertical dashed line called the "lifeline".
- ▶ An object participant shows the name of the object and its class:
object name : class name
- ▶ **Interactions** on a sequence diagram are shown as messages between participants



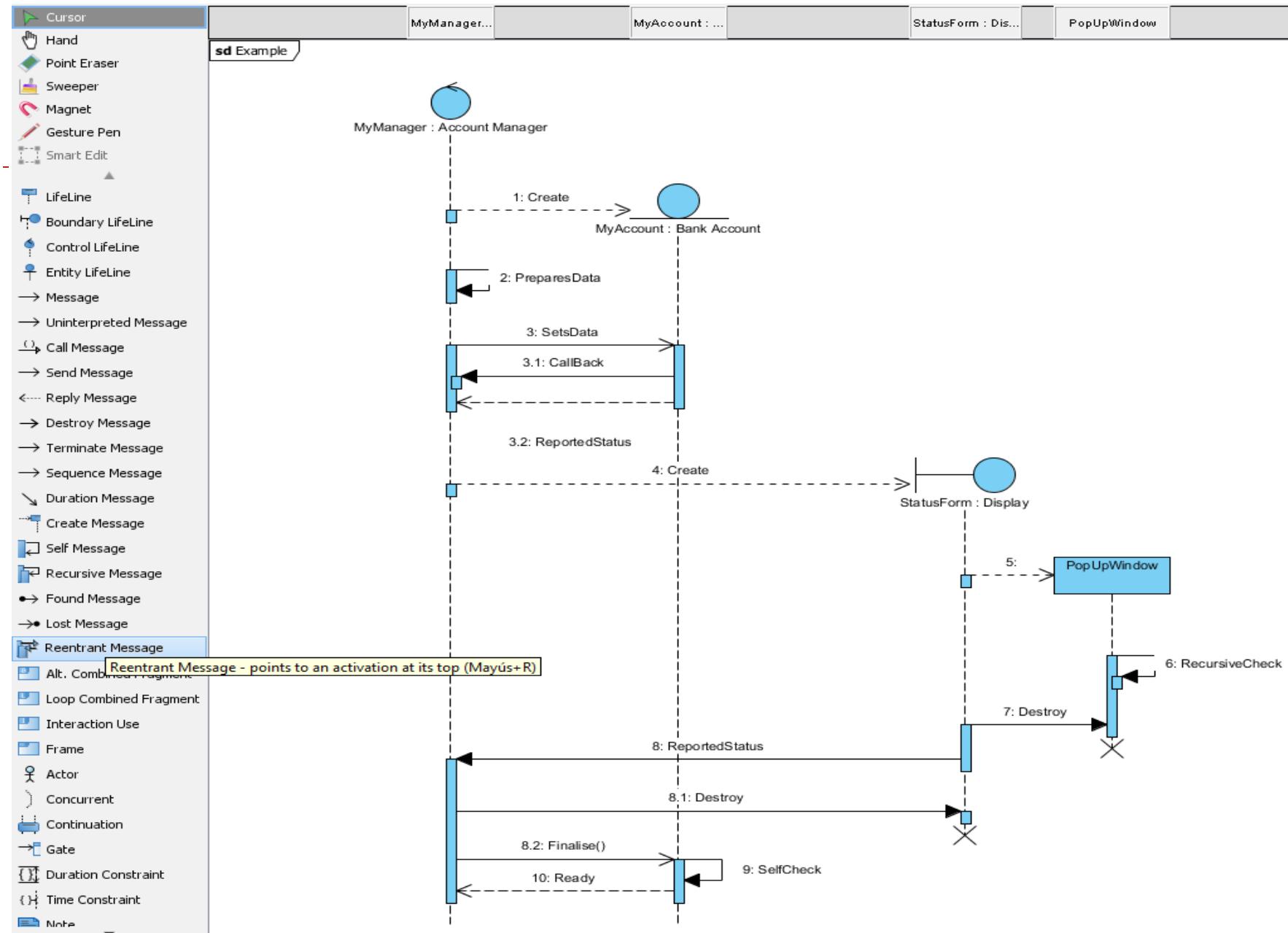
Powered By Visual Paradigm Community Edition

Synchronous and asynchronous messages

- ▶ The first message is a **synchronous** message (denoted by the solid arrowhead) complete with an *implicit return message*;
- ▶ The second message is **asynchronous** (denoted by line arrowhead), and the third is the *asynchronous return message* (denoted by the dashed line).



Sample sequence diagram in the editor of Visual Paradigm™

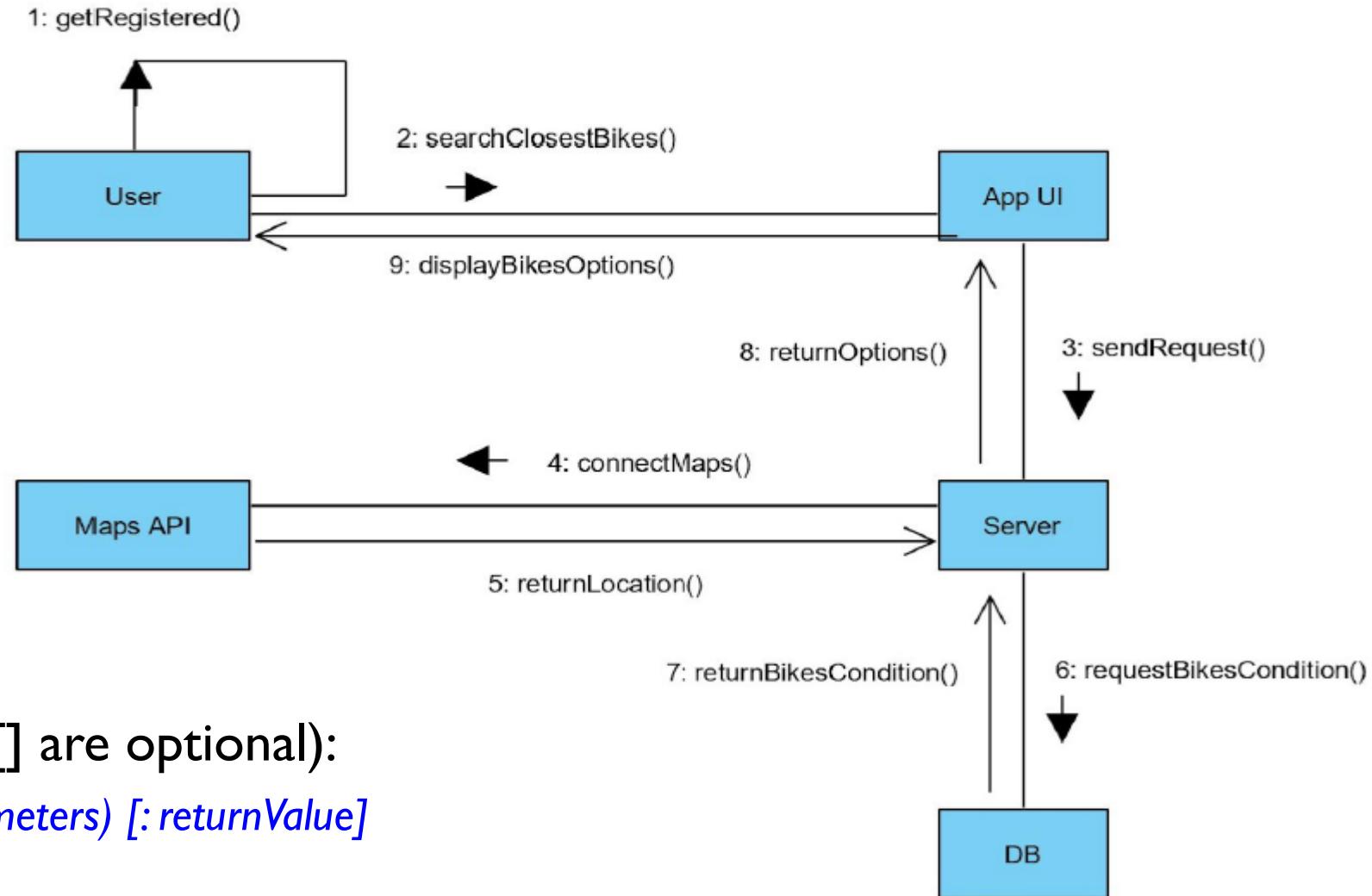


Visual Paradigm Tool Interface <https://www.visual-paradigm.com/>

Communication diagrams

- ▶ A **communication** (prior UML 2.0 called collaboration) diagram describes a pattern of objects interaction; it shows the objects participating in the interaction by their links to each other and the messages sent.
- ▶ Unlike a sequence diagram, a collaboration diagram shows the relationships among the objects. Sequence and communication diagrams are so similar that most UML tools can automatically convert from one diagram type to the other.
- ▶ You can have objects and actor instances in communication (collaboration) diagrams, together with links and messages describing how they are related and how they interact.

A sample communication diagram



Message notation (the parts in [] are optional):

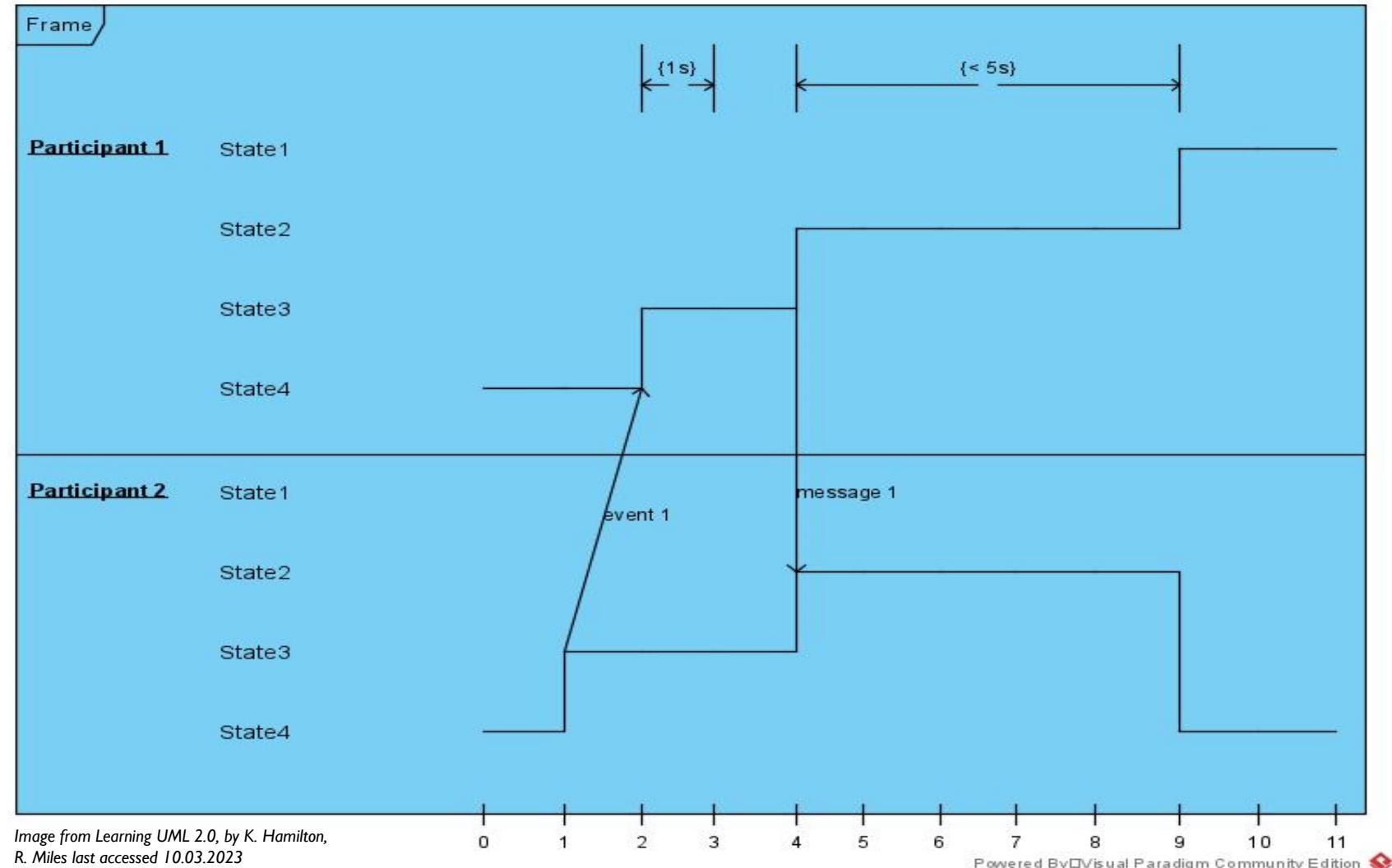
[sequenceNumber:] methodName(parameters) [:returnValue]

Time diagrams

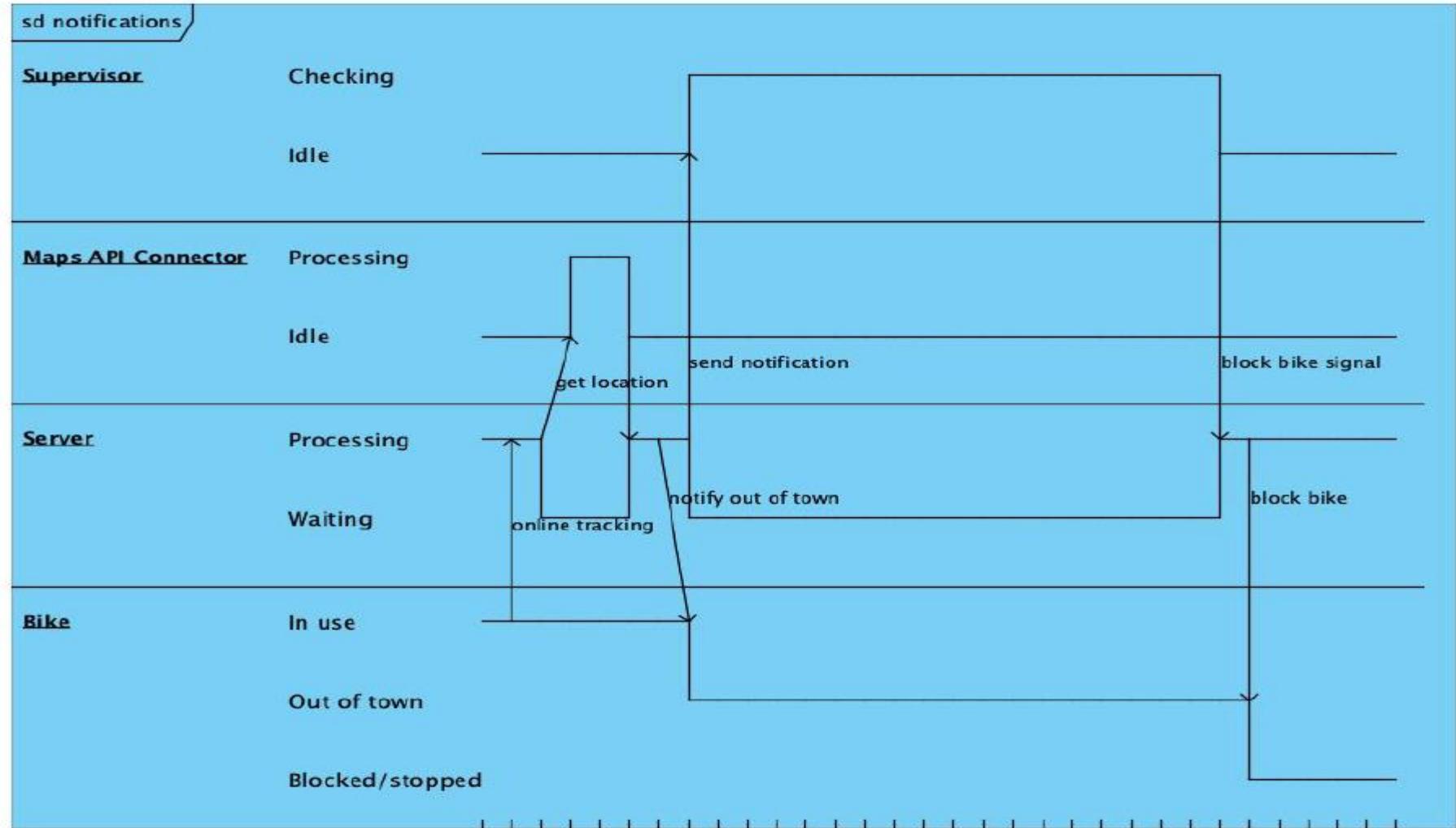
- ▶ The UML interaction diagrams shown over do not model detailed timing information
- ▶ In UML timing diagrams each event has timing information associated with it
- ▶ Timing diagrams describe:
 - ▶ when the event is invoked,
 - ▶ how long it takes for another participant to receive the event, and
 - ▶ how long the receiving participant is expected to be in a particular state.

Events

Events on a timing diagram can even have their own durations, as shown by event1 taking 1 unit of time from invocation by p1:Participant1 and reception by p2:Participant2



Sample timing diagram



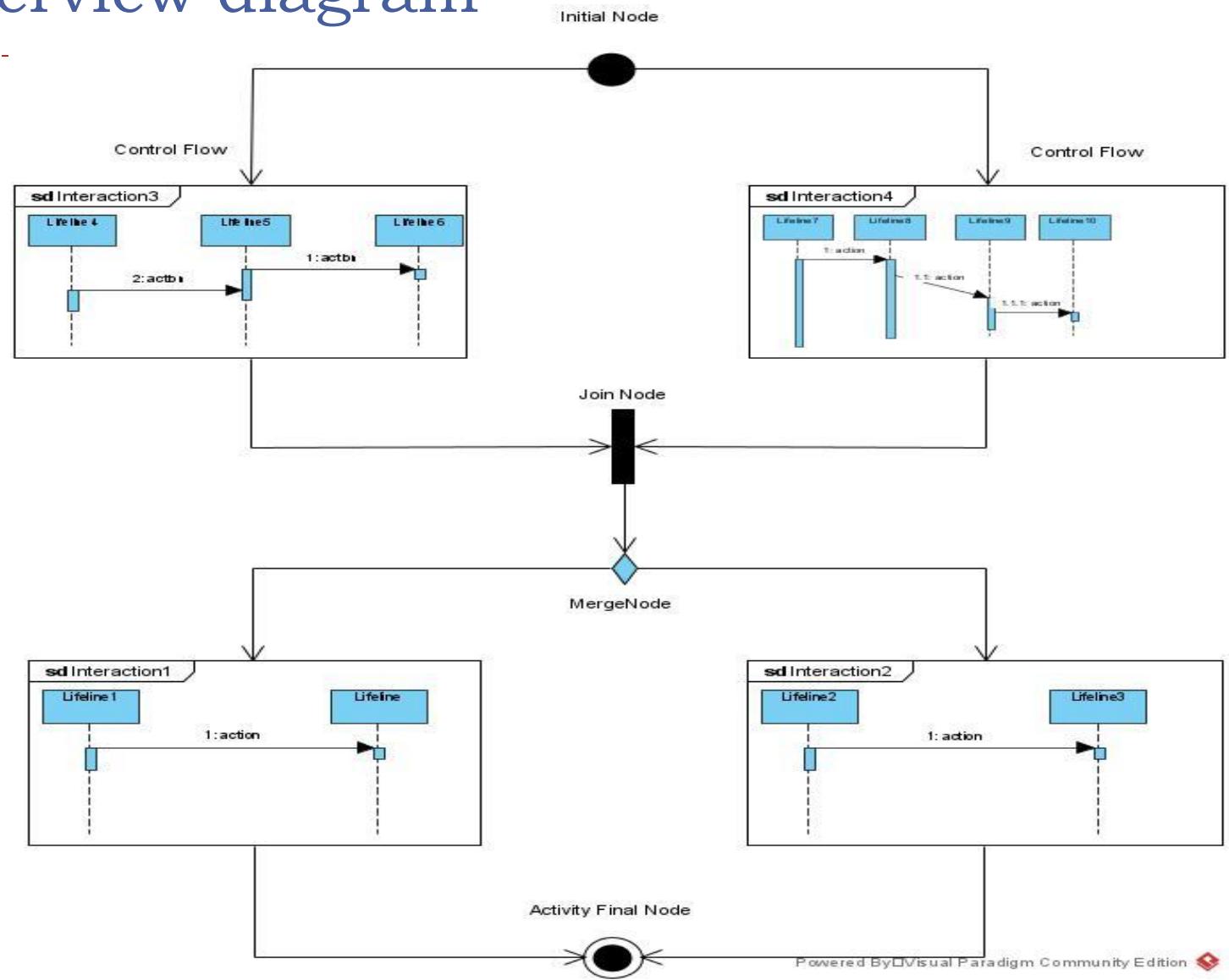
Used Visual Paradigm Community Edition tool:
<https://www.visual-paradigm.com/> last accessed 10.03.2023

Powered By Visual Paradigm Community Edition

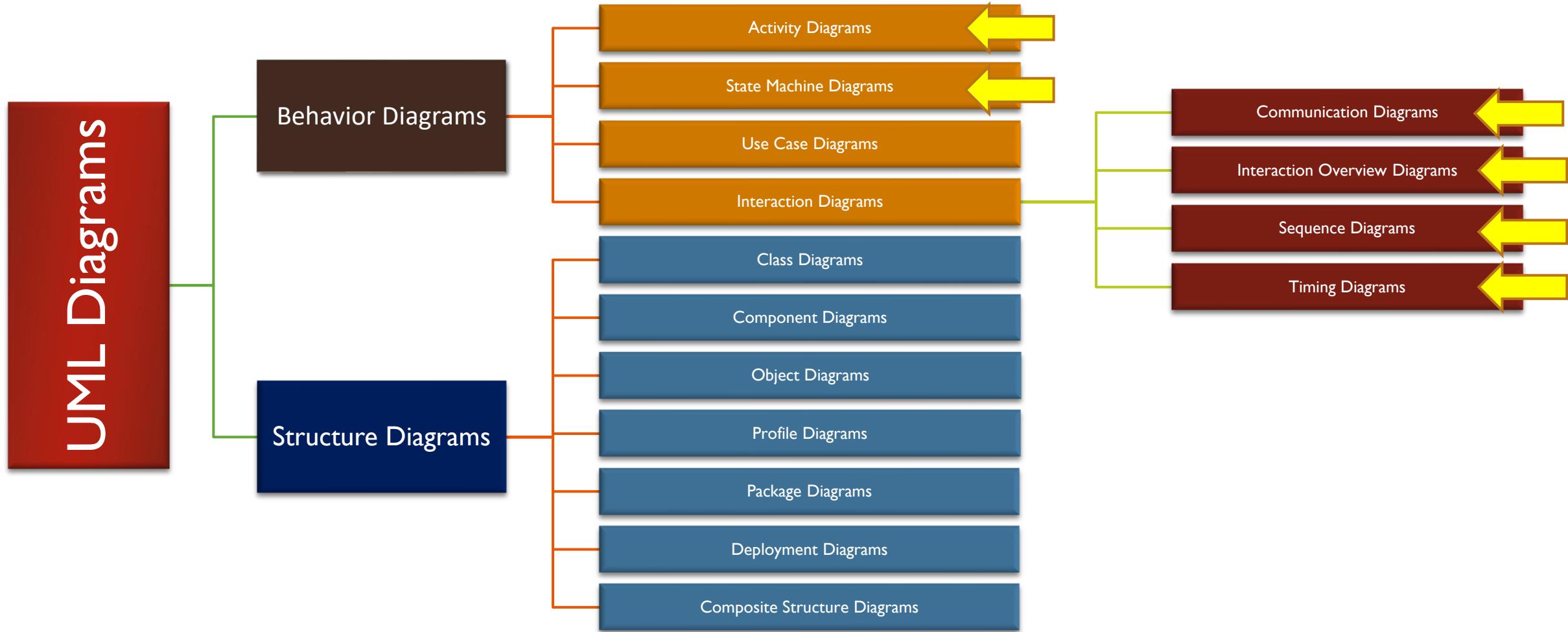
Interaction overview diagrams

- ▶ Interaction overview diagrams (since UML 2.0) overview control flow between *interaction diagrams* and show how a set of fragments might be initiated in various scenarios.
- ▶ They are variants on UML activity diagrams which represents at a higher level of abstraction the control flow between diagrams
- ▶ The nodes within these diagrams are frames of two types:
 - ▶ interaction frames depicting any type of UML interaction diagram (sequence diagram: sd, communication diagram: cd, timing diagram: td, etc.)
 - ▶ interaction occurrence frames (ref; typically anonymous) which indicate an activity or operation to invoke.

Sample interaction overview diagram

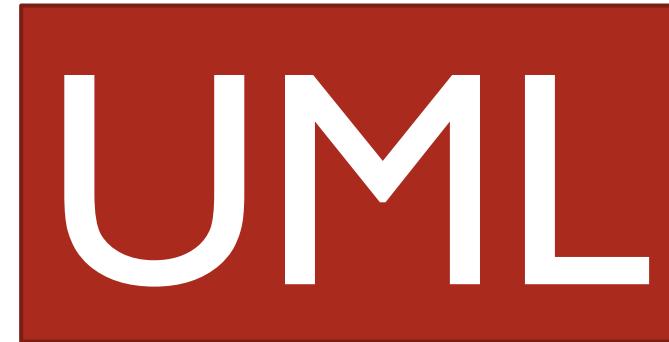


Types of UML diagrams



UML process diagrams - conclusions

- ▶ UML process diagrams describe the system aspects of competitiveness and synchronization
- ▶ The illustration of the flow of events of UML use cases, performed by the set of system object and subsystem instances.



References

- ▶ These slides are for educational purposes and used in the FMI Course “Software Technologies” and are part of the ICT-TEX Project 2022.
 - ▶ Sommerville, I. *Software Engineering*. 10th edition, Published by Pearson Education, ISBN: 978-1-292-09613-1 (2016)
 - ▶ Pressman, R., Maxim, B. *Software Engineering: A Practitioner's Approach*. 9th edition, Published by McGraw-Hill Education, ISBN: 9781260548006, (2019)
 - ▶ Page-Jones, M., Constantine, L. *Fundamentals of object-oriented design in UML*, Addison-Wesley, ISBN: 0-201-69946-X (2000)
 - ▶ Dan Pilone, Neil Pitman. *UML 2.0 in a Nutshell*, O'Reilly Media, Inc., June 2005.
 - ▶ Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley Object Technology Series, September 2003.
 - ▶ Russ Miles, Kim Hamilton. *Learning UML 2.0: A Pragmatic Introduction to UML*, O'Reilly Media, 1st edition, May 2006.
-



Въведение в управление на качеството.
Верификация и валидация на софтуерни системи.



Software quality management

- ▶ Concerned with ensuring that the **required level of quality** is achieved in a software product.
- ▶ Three principal concerns:
 - ▶ At the **organizational level**, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software.
 - ▶ At the **project level**, quality management involves the application of specific quality processes and checking that these planned processes have been followed.
 - ▶ At the **project level**, quality management is also concerned with establishing a quality plan for a project. The **quality plan** should set out the **quality goals** for the project and define what processes and standards are to be used.

Quality management activities

- ▶ **Quality management** provides an independent check on the software development process.
- ▶ The **quality management process** checks the project deliverables to ensure that they are consistent with organizational standards and goals
- ▶ The **quality team** should be independent from the development team so that they can take an objective view of the software. This allows them to report on **software quality** without being influenced by software development issues.

Quality management and software development

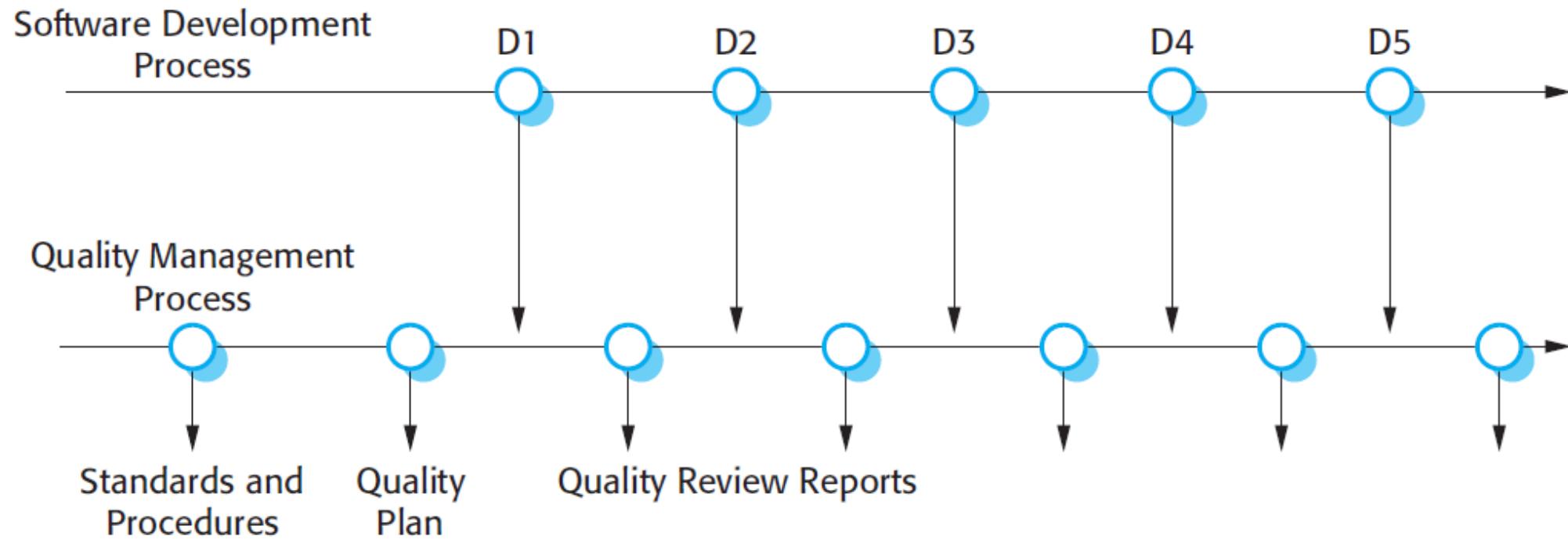


Image from: Sommerville, I. *Engineering Software Products: An Introduction to Modern Software Engineering 1st Edition*, Published by Pearson, ISBN: 978-0135210642, (2019)

Quality planning

- ▶ A **quality plan** sets out the desired product qualities and how these are assessed and defines the most significant quality attributes.
- ▶ The **quality plan** should define the quality assessment process.
- ▶ It should set out which organisational standards should be applied and, where necessary, define new standards to be used.

Quality plans

- ▶ **Quality plan structure**
 - ▶ Product introduction;
 - ▶ Product plans;
 - ▶ Process descriptions;
 - ▶ Quality goals;
 - ▶ Risks and risk management.
- ▶ **Quality plans should be short, succinct documents**
 - ▶ If they are too long, no-one will read them.

Scope of quality management

- ▶ **Quality management** is particularly important for large, complex systems. The quality documentation is a record of progress and supports continuity of development as the development team changes.
- ▶ For smaller systems, quality management needs less documentation and should focus on establishing a **quality culture**.

Software quality

- ▶ **Quality**, simplistically, means that a product should meet its specification.
- ▶ This is problematical for software systems
 - ▶ There is a tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.);
 - ▶ Some quality requirements are difficult to specify in an unambiguous way;
 - ▶ Software specifications are usually incomplete and often inconsistent.
- ▶ The focus may be '**fitness for purpose**' rather than specification conformance or 'The software is suitable for its purpose?'

Software fitness for purpose

- ▶ Have programming and documentation standards been followed in the development process?
- ▶ Has the software been properly tested?
- ▶ Is the software sufficiently dependable to be put into use?
- ▶ Is the performance of the software acceptable for normal use?
- ▶ Is the software usable?
- ▶ Is the software well-structured and understandable?

Software quality attributes

| | | |
|-------------|-------------------|--------------|
| Safety | Understandability | Portability |
| Security | Testability | Usability |
| Reliability | Adaptability | Reusability |
| Resilience | Modularity | Efficiency |
| Robustness | Complexity | Learnability |

Quality conflicts

- ▶ It is not possible for any system to be optimized for all of these attributes – for example, improving robustness may lead to loss of performance.
- ▶ The quality plan should therefore define the most important quality attributes for the software that is being developed.
- ▶ The plan should also include a definition of the quality assessment process, an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

Process and product quality

- ▶ The quality of a developed product is **influenced** by the quality of the production process.
- ▶ This is important in software development as some product quality attributes are hard to assess.
- ▶ However, there is a very complex and poorly understood relationship between software processes and product quality.
 - ▶ The application of individual skills and experience is particularly important in software development;
 - ▶ External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality.

Process-based quality

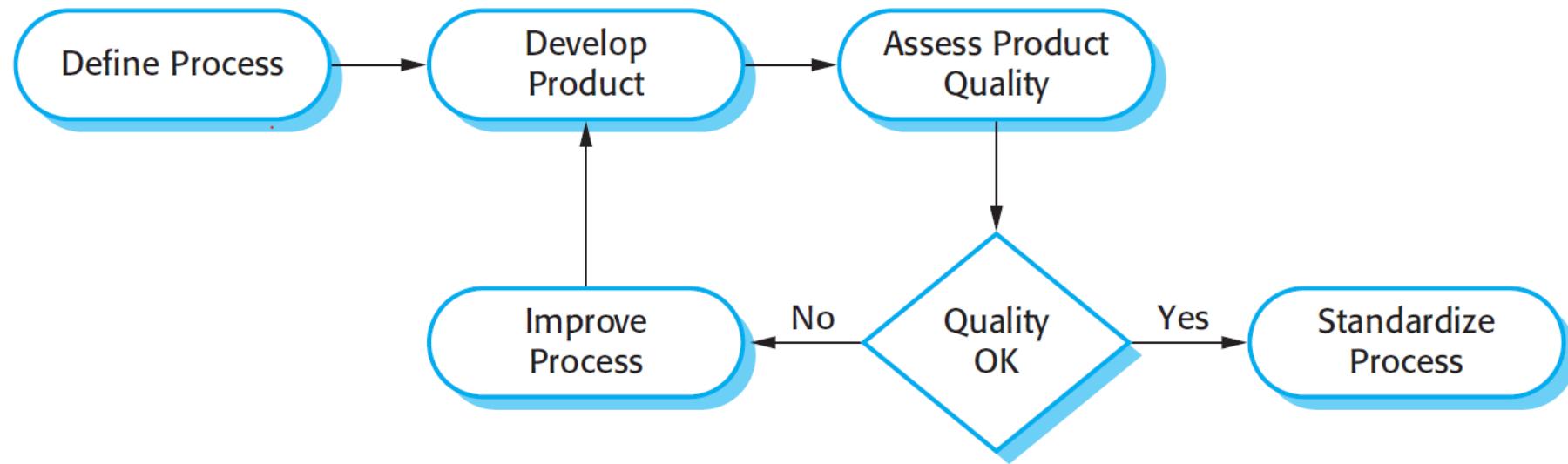


Image from: Sommerville, I. *Engineering Software Products: An Introduction to Modern Software Engineering 1st Edition*, Published by Pearson, ISBN: 978-0135210642, (2019)

Software standards

- ▶ **Standards** define the required attributes of a product or process. They play an important role in **quality management**.
- ▶ Standards may be **international, national, organizational or project standards**.
- ▶ **Product standards** define characteristics that all software components should exhibit e.g. a common programming style.
- ▶ **Process standards** define how the software process should be enacted.

Importance of standards

- ▶ **Encapsulation of best practice**- avoids repetition of past mistakes.
- ▶ They are a **framework for defining what quality means in a particular setting** i.e. that **organization's view of quality**.
- ▶ **They provide continuity** - new staff can understand the organisation by understanding the standards that are used.

Product and process standards

| Product standards | Process standards |
|---------------------------------|--|
| Design review form | Design review conduct |
| Requirements document structure | Submission of new code for system building |
| Method header format | Version release process |
| Java programming style | Project plan approval process |
| Project plan format | Change control process |
| Change request form | Test recording process |

Software testing

Software testing

- ▶ **Software testing is a process in which you execute your program using data that simulates user inputs.**
- ▶ You observe its behaviour to see whether or not your program is doing what it is supposed to do.
 - ▶ Tests pass if the behaviour is what you expect.
 - ▶ Tests fail if the behaviour differs from that expected.
 - ▶ If your program does what you expect, this shows that for the inputs used, the program behaves correctly.
- ▶ If these inputs are representative of a larger set of inputs, you can infer that the program will behave correctly for all members of this larger input set.

Program bugs

- ▶ If the behaviour of the program does not match the behaviour that you expect, then this means that there are bugs in your program that need to be fixed.
- ▶ There are two causes of program bugs:
 - ▶ **Programming errors** You have accidentally included faults in your program code. For example, a common programming error is an ‘off-by-1’ error where you make a mistake with the upper bound of a sequence and fail to process the last element in that sequence.
 - ▶ **Understanding errors** You have misunderstood or have been unaware of some of the details of what the program is supposed to do. For example, if your program processes data from a file, you may not be aware that some of this data is in the wrong format, so your program doesn’t include code to handle this.

Types of testing

► **Functional testing**

Test the functionality of the overall system. The goals of functional testing are to discover as many bugs as possible in the implementation of the system and to provide convincing evidence that the system is fit for its intended purpose.

► **User testing**

Test that the software product is useful to and usable by end-users. You need to show that the features of the system help users do what they want to do with the software. You should also show that users understand how to access the software's features and can use these features effectively.

► **Performance and load testing**

Test that the software works quickly and can handle the expected load placed on the system by its users. You need to show that the response and processing time of your system is acceptable to end-users. You also need to demonstrate that your system can handle different loads and scales gracefully as the load on the software increases.

► **Security testing**

Test that the software maintains its integrity and can protect user information from theft and damage.

Functional testing

- ▶ **Functional testing** involves developing a large set of program tests so that, ideally, all of a program's code is executed at least once.
- ▶ The number of tests needed obviously depends on the size and the functionality of the application.
- ▶ For a business-focused web application, you may have to develop thousands of tests to convince yourself that your product is ready for release to customers.
- ▶ **Functional testing is a staged activity in which you initially test individual units of code.** You integrate code units with other units to create larger units then do more testing.
- ▶ The process continues until you have created a complete system ready for release.

Functional testing

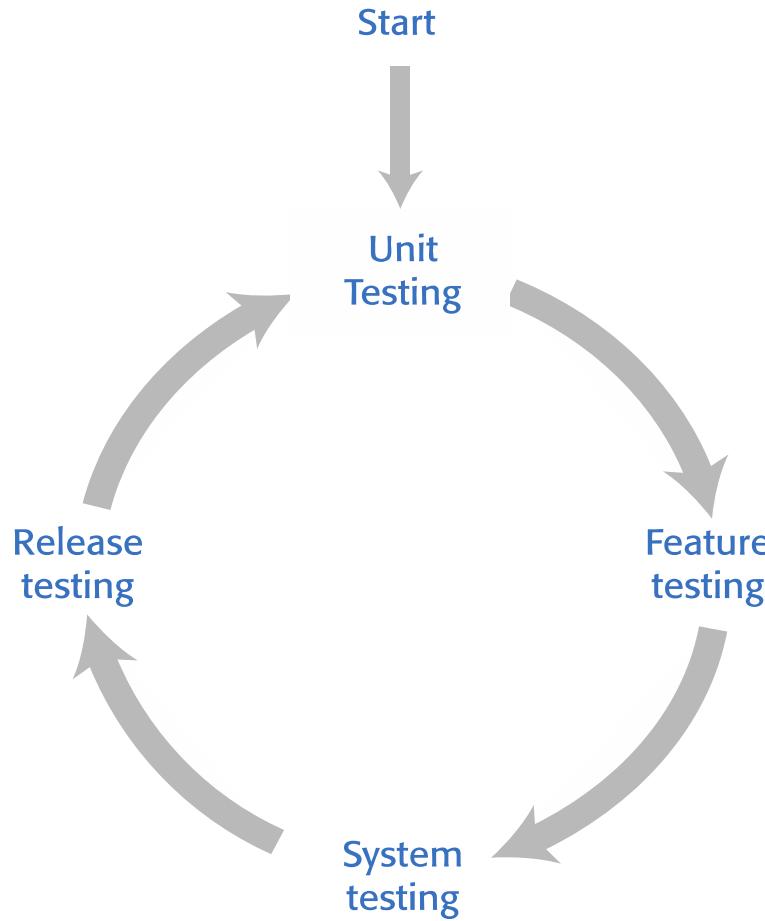


Image from: Sommerville, I. Engineering Software Products: An Introduction to Modern Software Engineering 1st Edition, Published by Pearson, ISBN: 978-0135210642, (2019)

Functional testing processes

► **Unit testing**

The aim of unit testing is to test program units in isolation. Tests should be designed to execute all of the code in a unit at least once. Individual code units are tested by the programmer as they are developed.

► **Feature testing**

Code units are integrated to create features. Feature tests should test all aspects of a feature. All of the programmers who contribute code units to a feature should be involved in its testing.

► **System testing**

Code units are integrated to create a working (perhaps incomplete) version of a system. The aim of system testing is to check that there are no unexpected interactions between the features in the system. System testing may also involve checking the responsiveness, reliability and security of the system. In large companies, a dedicated testing team may be responsible for system testing. In small companies, this is impractical, so product developers are also involved in system testing.

► **Release testing**

The system is packaged for release to customers and the release is tested to check that it operates as expected. The software may be released as a cloud service or as a download to be installed on a customer's computer or mobile device. If DevOps is used, then the development team are responsible for release testing otherwise a separate team has that responsibility.

Unit testing

- ▶ As you develop a code unit, you should also develop tests for that code.
- ▶ A code unit is anything that has a clearly defined responsibility. It is usually a function or class method but could be a module that includes a small number of other functions.
- ▶ Unit testing is based on a simple general principle:
 - ▶ If a program unit behaves as expected for a set of inputs that have some shared characteristics, it will behave in the same way for a larger set whose members share these characteristics.

Feature testing

- ▶ Features have to be tested to show that the functionality is implemented as expected and that the functionality meets the real needs of users.
 - ▶ For example, if your product has a feature that allows users to login using their Google account, then you have to check that this registers the user correctly and informs them of what information will be shared with Google.
 - ▶ You may want to check that it gives users the option to sign up for email information about your product.
- ▶ Normally, a feature that does several things is implemented by multiple, interacting, program units.
- ▶ These units may be implemented by different developers and all of these developers should be involved in the feature testing process.

Types of feature test

► Interaction tests

- ▶ These test the interactions between the units that implement the feature. The developers of the units that are combined to make up the feature may have different understandings of what is required of that feature.
- ▶ These misunderstandings will not show up in unit tests but may only come to light when the units are integrated.
- ▶ The integration may also reveal bugs in program units, which were not exposed by unit testing.

► Usefulness tests

- ▶ These test that the feature implements what users are likely to want.
- ▶ For example, the developers of a login with Google feature may have implemented an opt-out default on registration so that users receive all emails from a company. They must expressly choose what type of emails that they don't want.
- ▶ What might be preferred is an opt-in default so that users choose what types of email they do want to receive.

System and release testing

- ▶ System testing involves testing the system as a whole, rather than the individual system features.
- ▶ **System testing should focus on four things:**
 - ▶ Testing to discover if there are unexpected and unwanted interactions between the features in a system.
 - ▶ Testing to discover if the system features work together effectively to support what users really want to do with the system.
 - ▶ Testing the system to make sure it operates in the expected way in the different environments where it will be used.
 - ▶ Testing the responsiveness, throughput, security and other quality attributes of the system.

Scenario-based testing

- ▶ The best way to **systematically test a system is to start with a set of scenarios that describe possible uses of the system** and then work through these scenarios each time a new version of the system is created.
- ▶ Using the **scenario**, you **identify a set of end-to-end pathways that users might follow when using the system**.
- ▶ An **end-to-end pathway** is a **sequence of actions from starting to use the system for the task, through to completion of the task**.

Release testing

- ▶ Release testing is a type of system testing where a system that's intended for release to customers is tested.
- ▶ **The fundamental differences between release testing and system testing are:**
 - ▶ Release testing tests the system in its real operational environment rather than in a test environment. Problems commonly arise with real user data, which is sometimes more complex and less reliable than test data.
 - ▶ The aim of release testing is to decide if the system is good enough to release, not to detect bugs in the system. Therefore, some tests that 'fail' may be ignored if these have minimal consequences for most users.
- ▶ Preparing a system for release involves packaging that system for deployment (e.g. in a container if it is a cloud service) and installing software and libraries that are used by your product. You must define configuration parameters such as the name of a root directory, the database size limit per user and so on.

Test automation

- ▶ Automated testing is based on the idea that tests should be executable.
- ▶ An executable test includes the input data to the unit that is being tested, the expected result and a check that the unit returns the expected result.
- ▶ You run the test and the test passes if the unit returns the expected result.
- ▶ Normally, you should develop hundreds or thousands of executable tests for a software product.

Automated testing

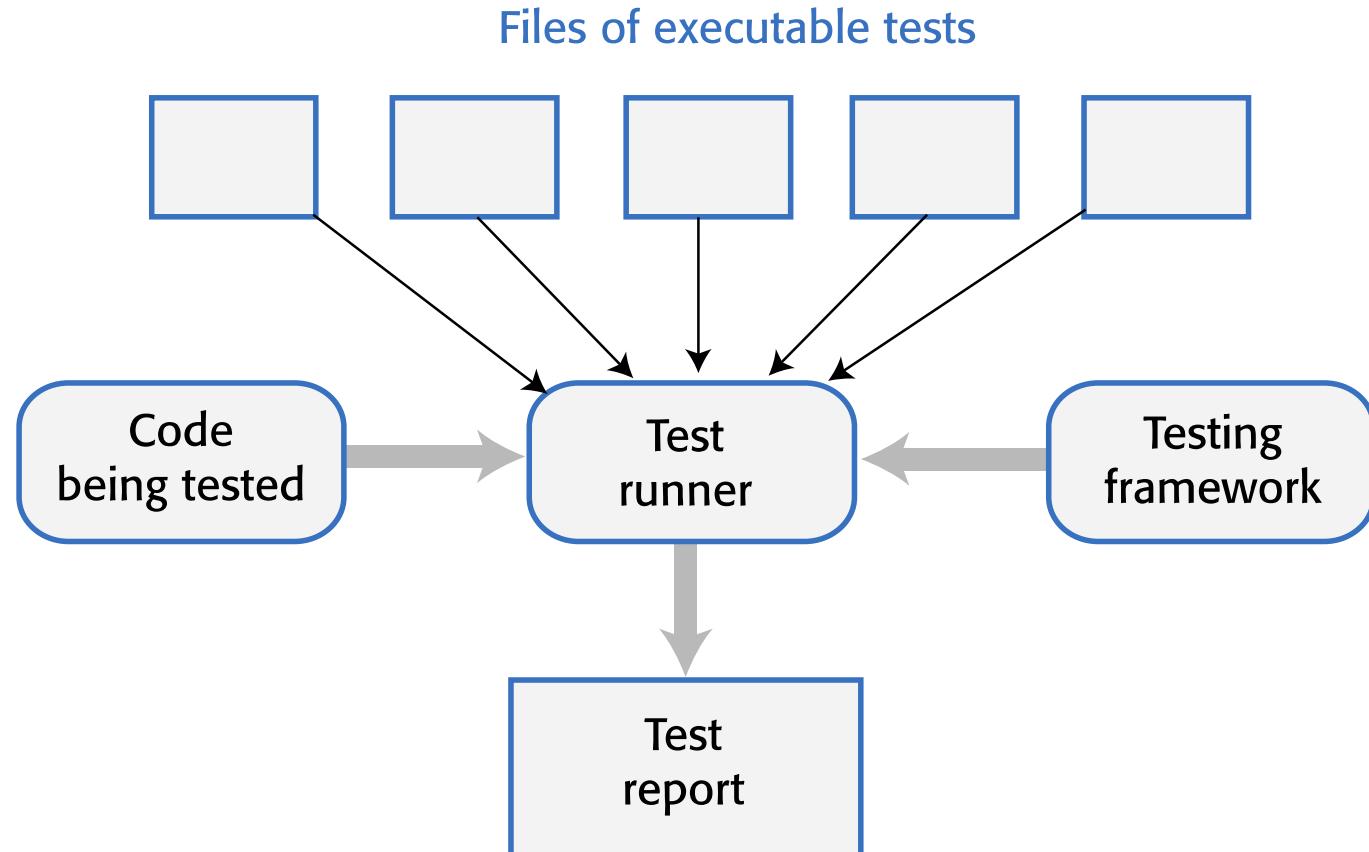


Image from: Sommerville, I. *Engineering Software Products: An Introduction to Modern Software Engineering 1st Edition*, Published by Pearson, ISBN: 978-0135210642, (2019)

The test pyramid

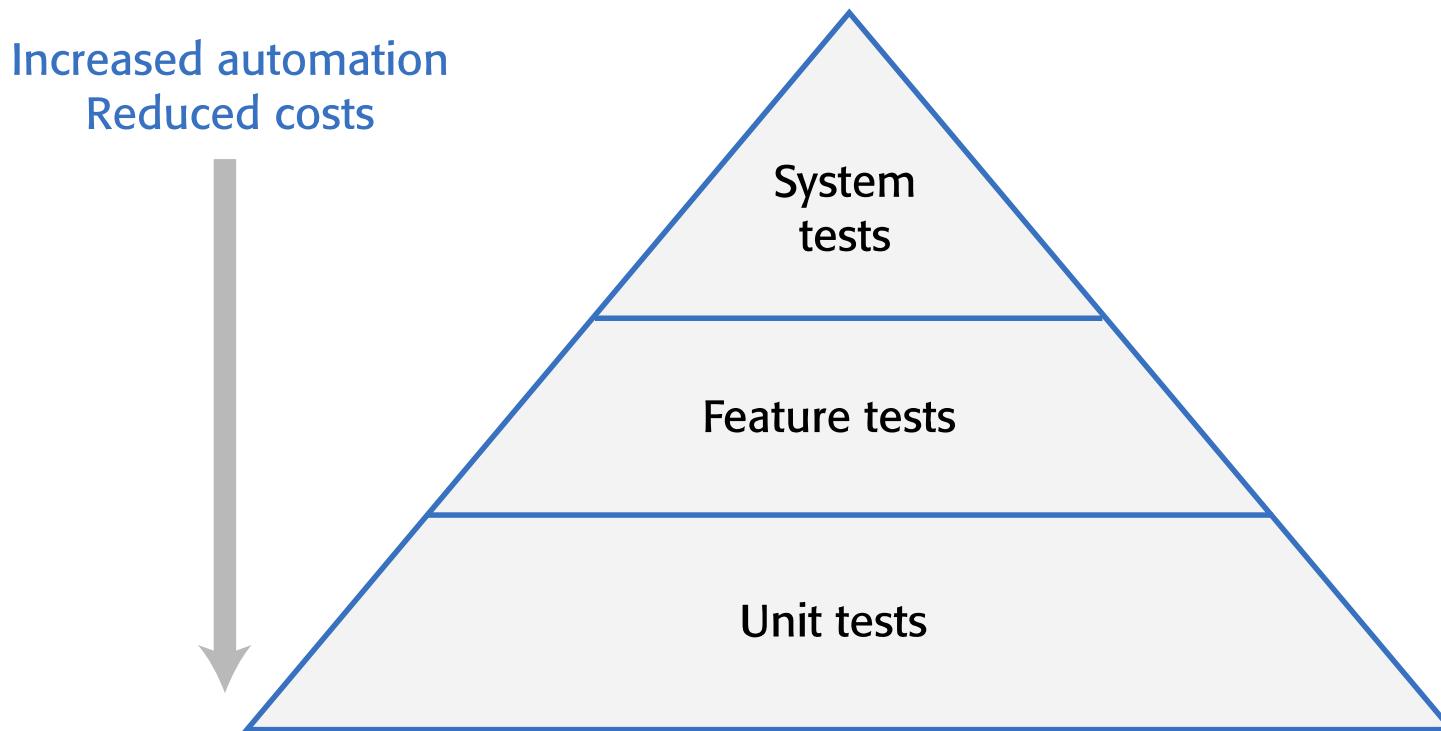
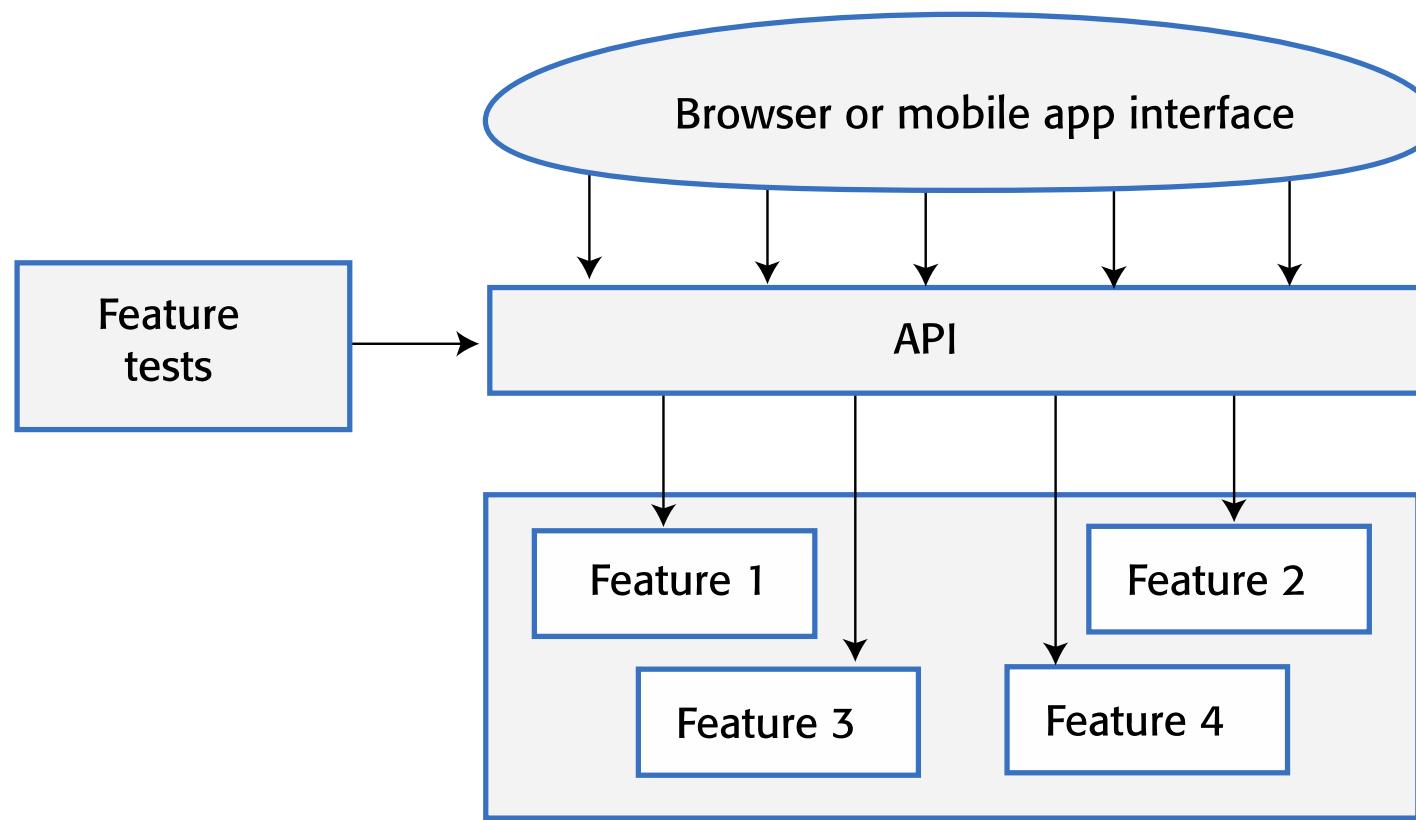


Image from: Sommerville, I. Engineering Software Products: An Introduction to Modern Software Engineering 1st Edition, Published by Pearson, ISBN: 978-0135210642, (2019)

Automated feature testing

- ▶ Generally, users access features through the product's graphical user interface (GUI).
- ▶ However, GUI-based testing is expensive to automate so it is best to design your product so that its features can be directly accessed through an API and not just from the user interface.
- ▶ The feature tests can then access features directly through the API without the need for direct user interaction through the system's GUI.
- ▶ Accessing features through an API has the additional benefit that it is possible to re-implement the GUI without changing the functional components of the software.

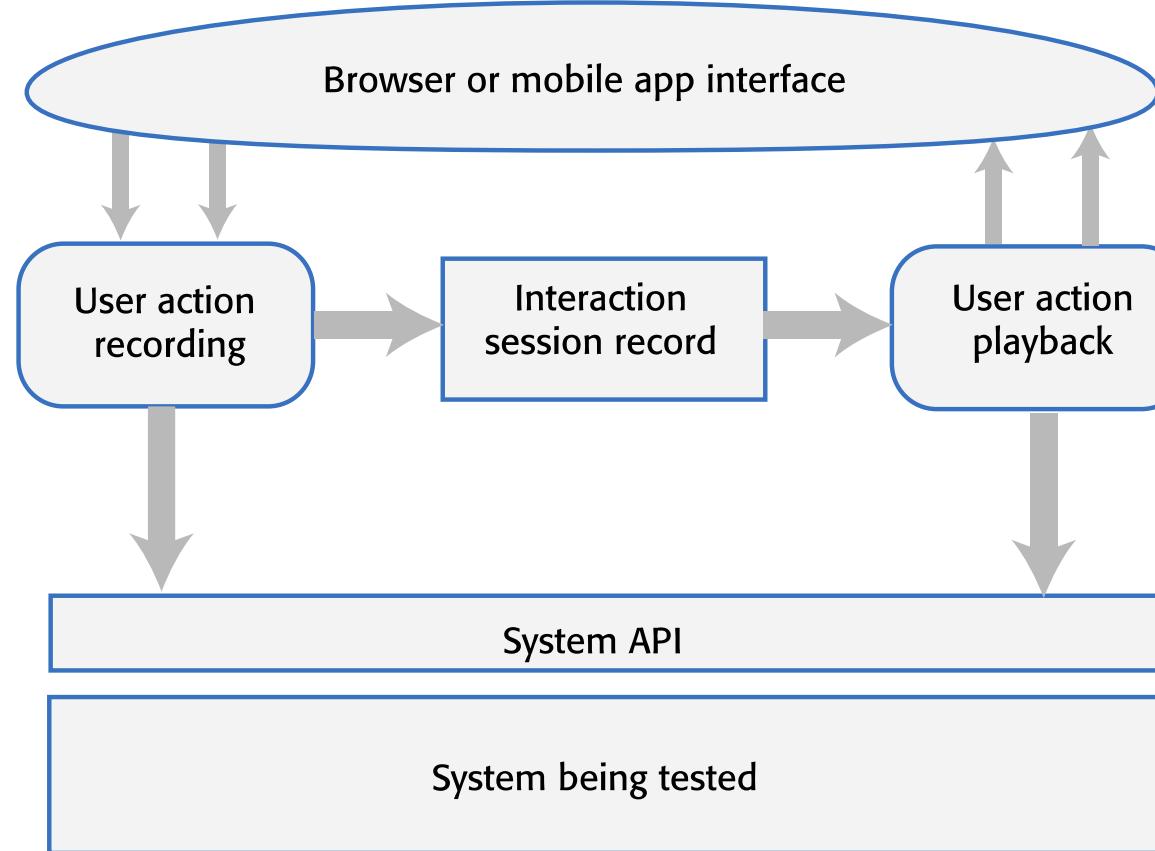
Feature editing through an API



System testing

- ▶ System testing, which should follow feature testing, involves testing the system as a surrogate user.
- ▶ As a system tester, you go through a process of selecting items from menus, making screen selections, inputting information from the keyboard and so on.
- ▶ You are looking for interactions between features that cause problems, sequences of actions that lead to system crashes and so on.
- ▶ Manual system testing, when testers have to repeat sequences of actions, is boring and error-prone. In some cases, the timing of actions is important and is practically impossible to repeat consistently.
 - ▶ To avoid these problems, testing tools have been developed that can record a series of actions and automatically replay these when a system is retested

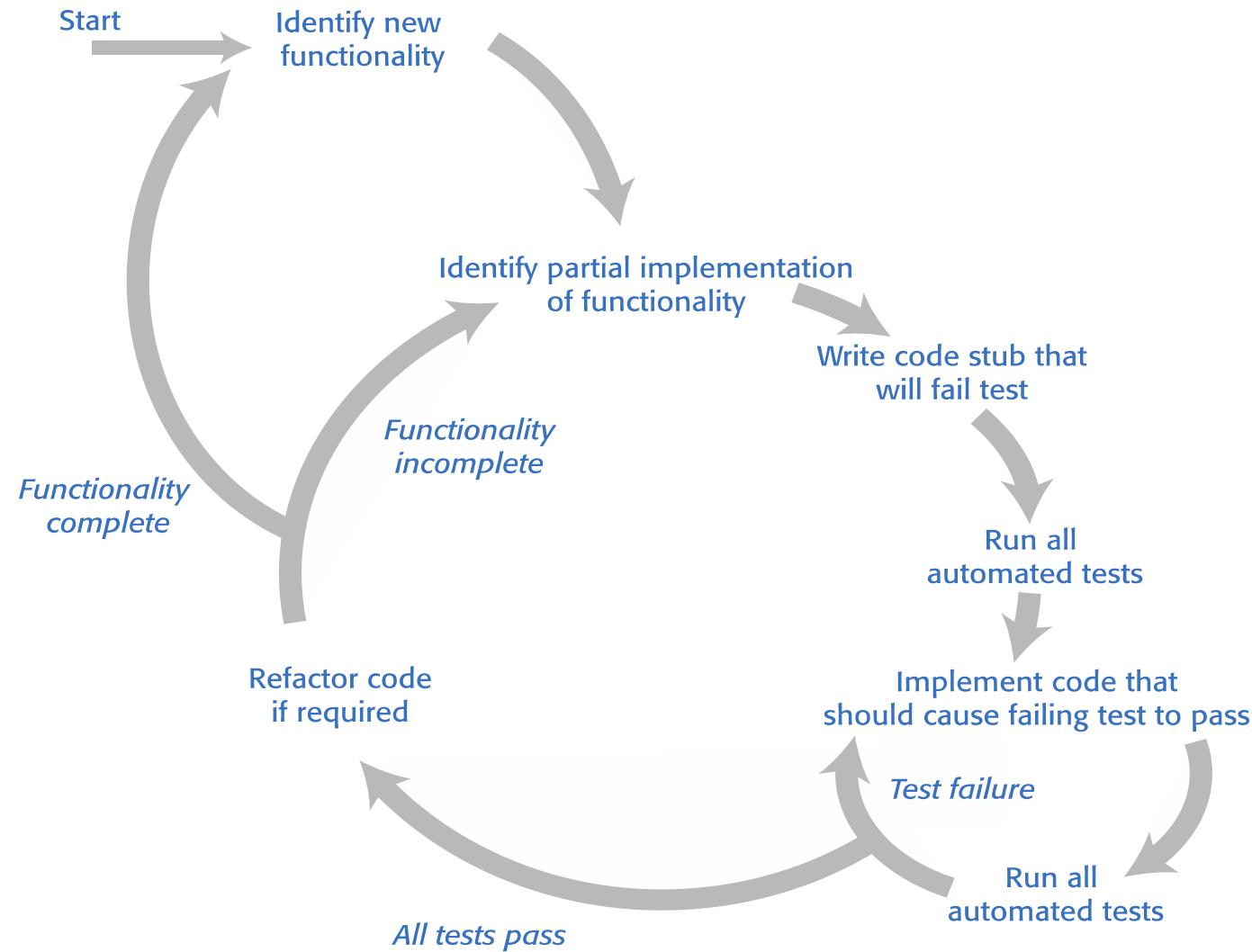
Interaction recording and playback



Test-driven development

- ▶ Test-driven development (TDD) is an approach to program development that is based around the general idea that **you should write an executable test or tests for code that you are writing before you write the code.**
- ▶ It was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach.
- ▶ Test-driven development works best for the development of individual program units and it is more difficult to apply to system testing.
- ▶ Even the strongest advocates of TDD accept that it is challenging to use this approach when you are developing and testing systems with graphical user interfaces.

Test-driven development



Stages of test-driven development (1)

- ▶ ***Identify partial implementation***

Break down the implementation of the functionality required into smaller mini-units.
Choose one of these mini-units for implementation.

- ▶ ***Write mini-unit tests***

Write one or more automated tests for the mini-unit that you have chosen for implementation. The mini-unit should pass these tests if it is properly implemented.

- ▶ ***Write a code stub that will fail test***

Write incomplete code that will be called to implement the mini-unit. You know this will fail.

- ▶ ***Run all existing automated tests***

All previous tests should pass. The test for the incomplete code should fail.

Stages of test-driven development (2)

- ▶ **Implement code that should cause the failing test to pass**

Write code to implement the mini-unit, which should cause it to operate correctly

- ▶ **Rerun all automated tests**

If any tests fail, your code is probably incorrect. Keep working on it until all tests pass.

- ▶ **Refactor code if necessary**

If all tests pass, you can move on to implementing the next mini-unit. If you see ways of improving your code, you should do this before the next stage of implementation.

Benefits of test-driven development

- ▶ It is a systematic approach to testing in which tests are clearly linked to sections of the program code.
 - ▶ This means you can be confident that your tests cover all of the code that has been developed and that there are no untested code sections in the delivered code - this is the most significant benefit of TDD.
- ▶ The tests act as a written specification for the program code. In principle at least, it should be possible to understand what the program does by reading the tests.
- ▶ Debugging is simplified because, when a program failure is observed, you can immediately link this to the last increment of code that you added to the system.
- ▶ It is argued that TDD leads to simpler code as programmers only write code that's necessary to pass tests. They don't over-engineer their code with complex features that aren't needed.

Security testing

- ▶ Security testing aims to find vulnerabilities that may be exploited by an attacker and to provide convincing evidence that the system is sufficiently secure.
- ▶ The tests should demonstrate that the system can resist attacks on its availability, attacks that try to inject malware and attacks that try to corrupt or steal users' data and identity.
- ▶ Comprehensive security testing requires specialist knowledge of software vulnerabilities and approaches to testing that can find these vulnerabilities.

Risk-based security testing

- ▶ A risk-based approach to security testing involves identifying common risks and developing tests to demonstrate that the system protects itself from these risks.
- ▶ You may also use automated tools that scan your system to check for known vulnerabilities, such as unused HTTP ports being left open.
- ▶ Based on the risks that have been identified, you then design tests and checks to see if the system is vulnerable.
- ▶ It may be possible to construct automated tests for some of these checks, but others inevitably involve manual checking of the system's behaviour and its files.

Examples of security risks

- ▶ Unauthorized attacker gains access to a system using authorized credentials
- ▶ Authorized individual accesses resources that are forbidden to them
- ▶ Authentication system fails to detect unauthorized attacker
- ▶ Attacker gains access to database using SQL poisoning attack
- ▶ Improper management of HTTP session
- ▶ HTTP session cookies revealed to attacker
- ▶ Confidential data are unencrypted
- ▶ Encryption keys are leaked to potential attackers

Code reviews

- ▶ Code reviews involve one or more people examining the code to check for errors and anomalies and discussing issues with the developer.
- ▶ If problems are identified, it is the developer's responsibility to change the code to fix the problems.
- ▶ Code reviews complement testing. They are effective in finding bugs that arise through misunderstandings and bugs that may only arise when unusual sequences of code are executed.
- ▶ Many software companies insist that all code has to go through a process of code review before it is integrated into the product codebase.

Code reviews

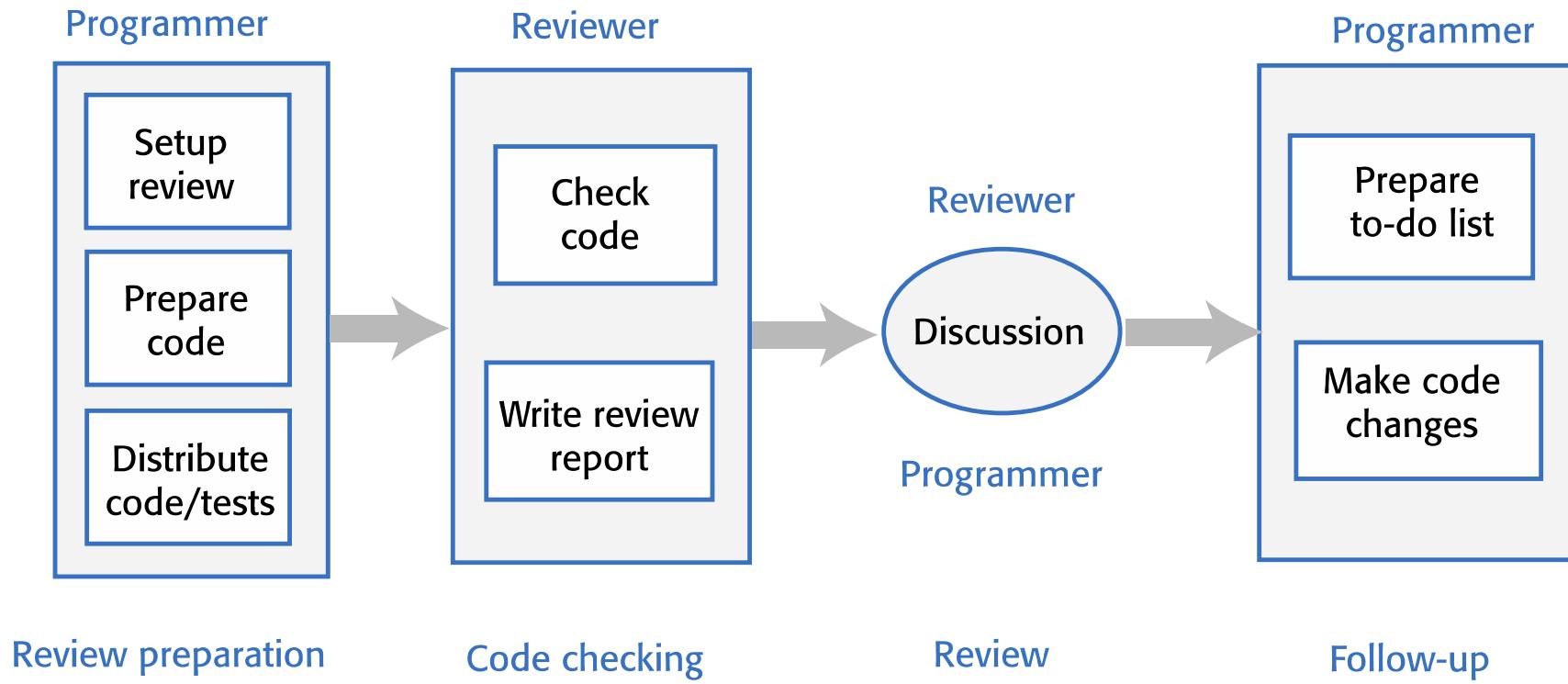


Image from: Sommerville, I. *Engineering Software Products: An Introduction to Modern Software Engineering 1st Edition*, Published by Pearson, ISBN: 978-0135210642, (2019)

Code review activities (1)

- ▶ **Setup review**

The programmer contacts a reviewer and arranges a review date.

- ▶ **Prepare code**

The programmer collects the code and tests for review and annotates them with information for the reviewer about the intended purpose of the code and tests.

- ▶ **Distribute code/tests**

The programmer sends code and tests to the reviewer.

- ▶ **Check code**

The reviewer systematically checks the code and tests against their understanding of what they are supposed to do.

- ▶ **Write review report**

The reviewer annotates the code and tests with a report of the issues to be discussed at the review meeting.

Code review activities (2)

► ***Discussion***

The reviewer and programmer discuss the issues and agree on the actions to resolve these.

► ***Make to-do list***

The programmer documents the outcome of the review as a to-do list and shares this with the reviewer.

► ***Make code changes***

The programmer modifies their code and tests to address the issues raised in the review.

Conclusion: Key points 1

- ▶ Software quality management is concerned with ensuring that software has a low number of defects and that it reaches the required standards of maintainability, reliability, portability and so on.
- ▶ SQM includes defining standards for processes and products and establishing processes to check that these standards have been followed.
- ▶ Software standards are important for quality assurance as they represent an identification of ‘best practice’.
- ▶ The aim of program testing is to find bugs and to show that a program does what its developers expect it to do.
- ▶ Four types of testing that are relevant to software products are functional testing, user testing, load and performance testing and security testing.

Conclusion: Key points 2

- ▶ Unit testing involves testing program units such as functions or class methods that have a single responsibility. Feature testing focuses on testing individual system features. System testing tests the system as a whole to check for unwanted interactions between features and between the system and its environment.
- ▶ User stories may be used as a basis for deriving feature tests.
- ▶ Test automation is based on the idea that tests should be executable. You develop a set of executable tests and run these each time you make a change to a system.

Conclusion: Key points 3

- ▶ Test-driven development is an approach to development where executable tests are written before the code. Code is then developed to pass the tests.
- ▶ A disadvantage of test-driven development is that programmers focus on the detail of passing tests rather than considering the broader structure of their code and algorithms used.
- ▶ Security testing may be risk driven where a list of security risks is used to identify tests that may identify system vulnerabilities.
- ▶ Code reviews are an effective supplement to testing. They involve people checking the code to comment on the code quality and to look for bugs.

References

- ▶ Sommerville, I. *Software Engineering*. 9th edition, Published by Pearson Education, ISBN: 978-0-13-703515-1 (2011)
- ▶ Sommerville, I. *Software Engineering*. 10th edition, Published by Pearson Education, ISBN: 978-1-292-09613-1 (2016)
- ▶ Pressman, R., Maxim, B. *Software Engineering: A Practitioner's Approach*. 9th edition, Published by McGraw-Hill Education, ISBN: 9781260548006, (2019)
- ▶ Sommerville, I. *Engineering Software Products: An Introduction to Modern Software Engineering* 1st Edition, Published by Pearson, ISBN: 978-0135210642, (2019)

Надежност и сигурност на софтуерни системи

Reliability and Security of Software systems

Съдържание

- ▶ Понятие за надежност/изправност на софтуерни системи
 - ▶ Reliability/Dependability
- ▶ Dependability
 - ▶ Threats
 - ▶ Attributes
 - ▶ Means to attain
- ▶ Software security

70s: Mainframes

Systems/users:

$\sim 10^4$

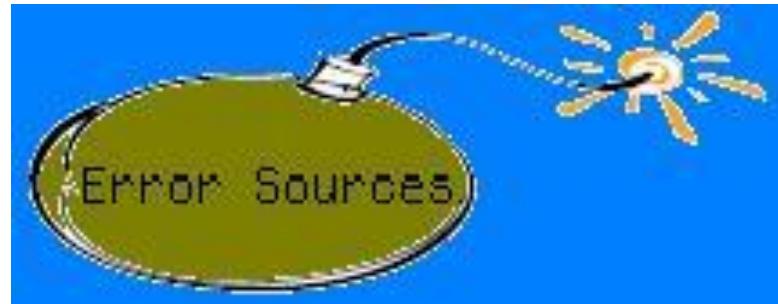
User competency:

Engineers



Integration complexity

- ▶ Close systems
- ▶ Highly custom designs
- ▶ Hardware and software fully controlled by vendors



- Hardware

80s: Workstations

Systems/users:

$\sim 10^6$

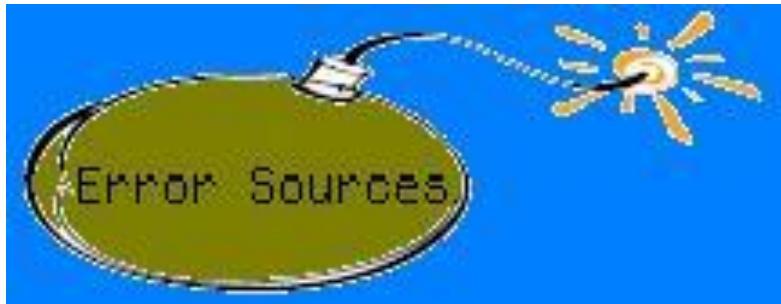
User competency:

Basic knowledge



Integration complexity

- ▶ Mostly close systems
- ▶ Network connectivity
- ▶ Standard interfaces exported for users



- Hardware
- Network

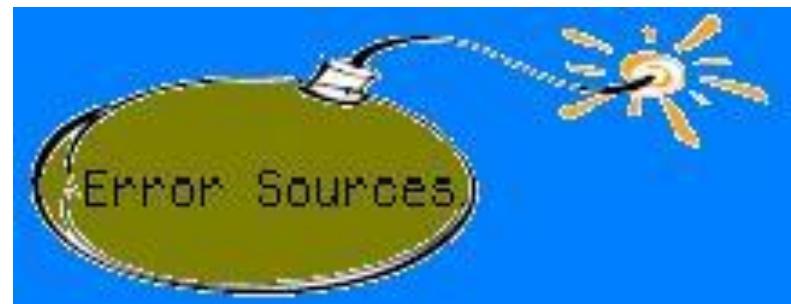
90s: Personal Computers

Systems/users:

$\sim 10^7$

User competency:

Computer Literacy



Integration complexity

- ▶ Open systems
- ▶ Wide network access
- ▶ Commercial OS
- ▶ Third party software and hardware

- Hardware, Software
- Network
- Human mistakes

2000s: Mobile Devices

Systems/users:

$\sim 10^8$

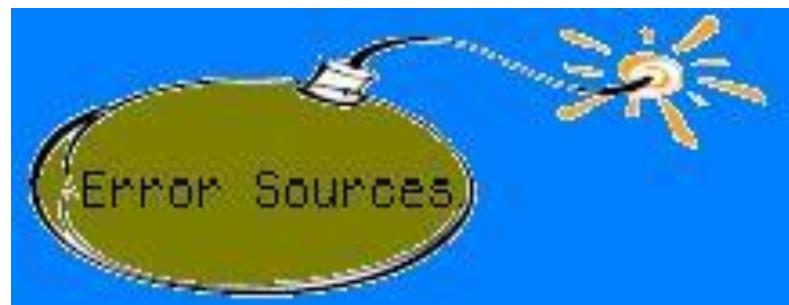
User competency:

Undefined



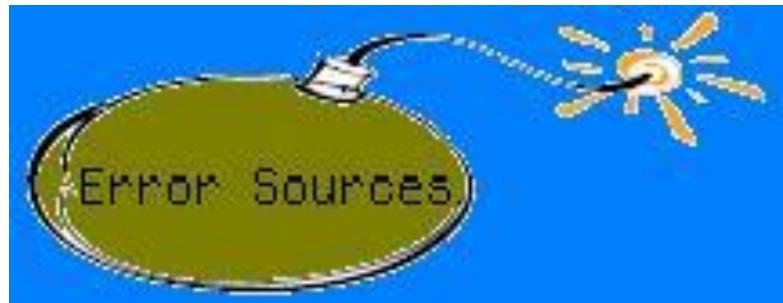
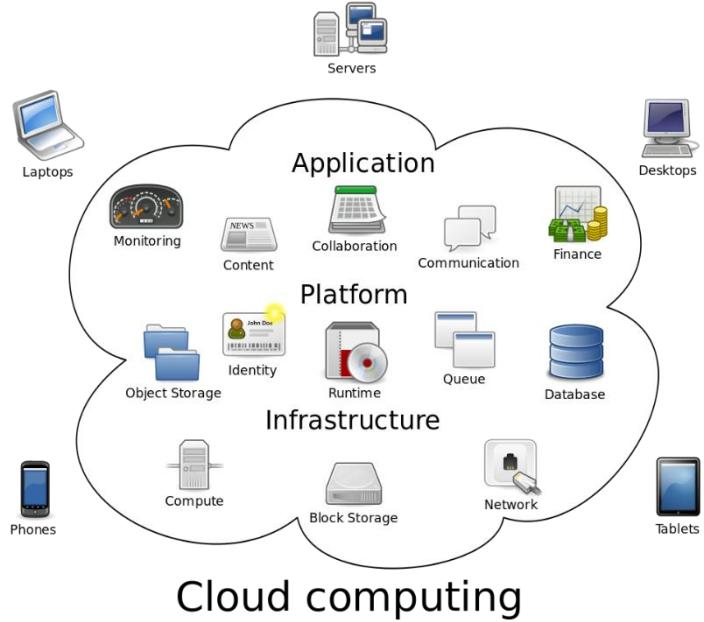
Integration complexity

- ▶ Open systems
- ▶ COTS/proprietary OS
- ▶ Highly integrated computer systems
- ▶ Wider range of networks



- Hardware, Software
- Network (wired/wireless)
- Human mistakes
- Malicious faults

2010s: Cloud computing



Systems/users:

~ 10^9 and more

User competency:

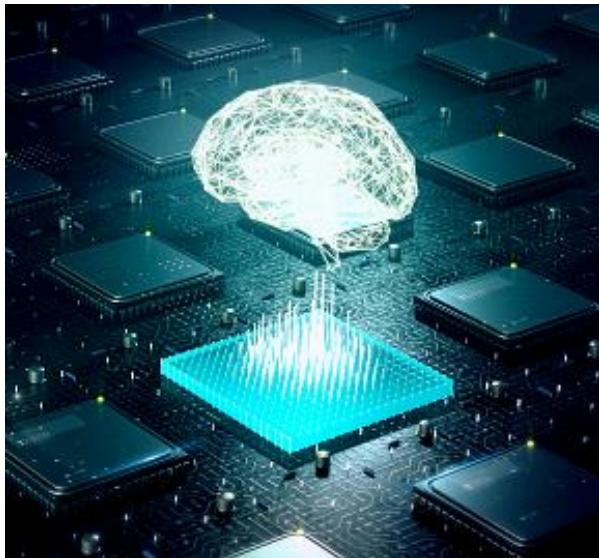
Undefined

Human mistakes

Malicious faults

Software itself

2020s: Artificial Intelligence

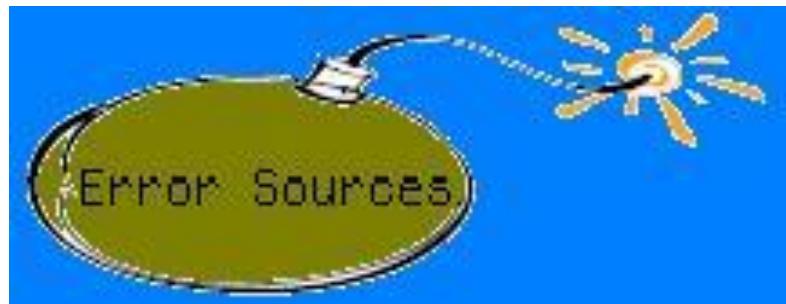


Systems/users:

???

User competency:

???



???

Industrial trends

- ▶ Newer application domains
- ▶ Increase in complexity of systems
- ▶ Increase in interactions among them
- ▶ Increase in volume of units
- ▶ Shift in error sources
- ▶ Reduced user tolerance levels

Underline the growing importance of
building dependable systems

What does this mean?

- ▶ Dependable systems
- ▶ Dependability
- ▶ Resilience, robustness, survivability....

Definition of dependability

“Dependability of a computing system is the ability to deliver service that can justifiably be trusted”

The service delivered by a system is its behaviour as it is perceived by its users

A user is another system (physical,human)



HW Reliability vs. SW Dependability

Hardware

- ▶ Deterioration over time
- ▶ Design faults removed before manufacture
- ▶ No new faults enter during life-cycle
- ▶ Initial use & end of life failures common
- ▶ No need of time to correct fault

Software

- ▶ No deterioration over time
- ▶ Faults removed after build
- ▶ Faults possibly enter during fault correction
- ▶ Failures during Initial test period, early use common, then stable
- ▶ Time needed to correct faults

WARRANTY

DISCLAIMER

Classes of dependable systems



Classes of Dependable Systems

► Safety-critical

- ▶ Airplanes
- ▶ Cars
- ▶ Nuclear power stations
- ▶ Traffic control systems
- ▶ Energy supply and distribution systems
- ▶ Telecommunication systems
- ▶ Etc...

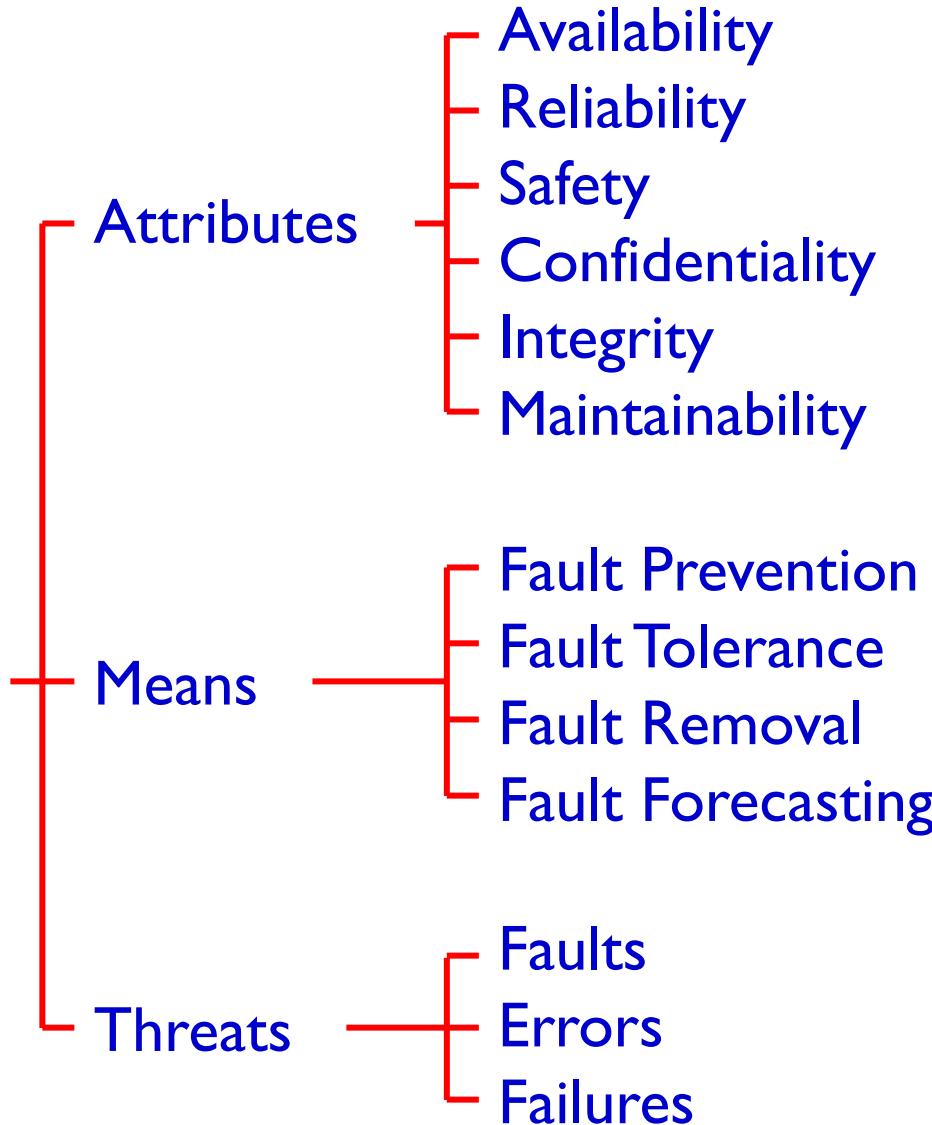
► Mission-critical

- ▶ Shuttles
- ▶ Airplanes

► Business-critical

- ▶ Industrial systems
- ▶ Information systems
- ▶ Traffic control systems

Dependability
Ability to deliver service that can justifiably be trusted



Causes of failure

- ▶ **Hardware failure**
 - ▶ Hardware fails because of design and manufacturing errors or because components have reached the end of their natural life.
- ▶ **Software failure**
 - ▶ Software fails due to errors in its specification, design or implementation.
- ▶ **Operational failure**
 - ▶ Human operators make mistakes. Now perhaps the largest single cause of system failures in socio-technical systems.



Adjudged or hypothesized cause of an error

Part of system state that may cause a subsequent service failure

Deviation of the delivered service from correct service, i.e., implementing the system function

System does not comply with specification

Specification does not adequately describe function

Fault, Error, Failure - Example

► A Fault:

- ▶

```
int increment (int x) {  
    x = x++;// should be x = x + 1;  
}
```

► An Error – fault activated

- ▶ `Y = increment(2);`
- ▶ Can be propagated.

► A Failure – Error exposed to interface

- ▶ `Print(Y);`

► Masked error

- ▶ `Print(Y%10);`

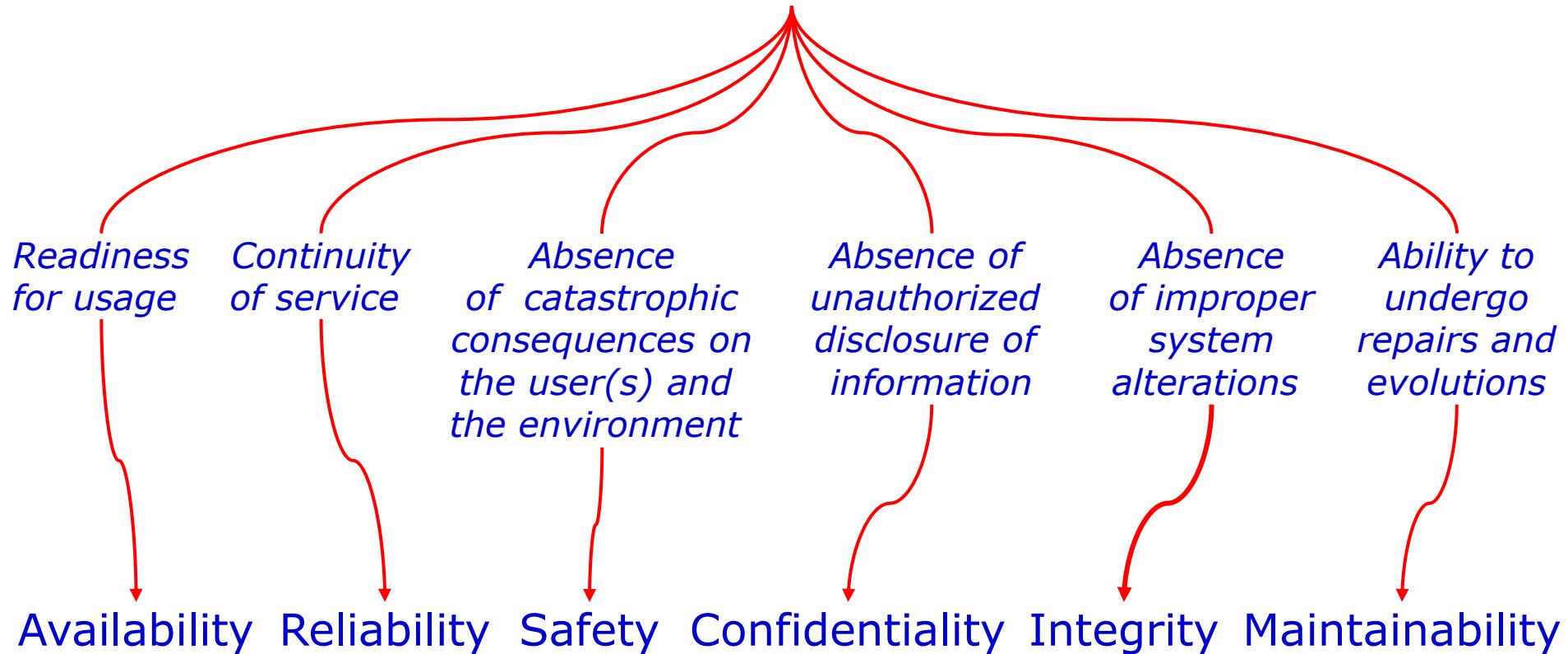
Faults nature

- ▶ Not all code in a program is executed. The code that includes a fault (e.g., the failure to initialize a variable) may never be executed because of the way that the software is used.
- ▶ Errors may be transient. A state variable may have an incorrect value caused by the execution of faulty code. However, before this is accessed and causes a system failure, some other system input may be processed that resets the state to a valid value.
- ▶ The system may include fault detection and protection mechanisms. These ensure that the erroneous behavior is discovered and corrected before the system services are affected.

Faults and failures

- ▶ Failures are usually a result of system errors that are derived from faults in the system
- ▶ However, faults do not necessarily result in system errors
 - ▶ The erroneous system state resulting from the fault may be transient and ‘corrected’ before an error arises.
 - ▶ The faulty code may never be executed.
- ▶ Errors do not necessarily lead to system failures
 - ▶ The error can be corrected by built-in error detection and recovery
 - ▶ The failure can be protected against by built-in protection facilities. These may, for example, protect system resources from system errors

Dependability



Dependability: Ability of the system to provide service that can justifiably trusted

Strict definitions

- ▶ **Reliability:** The probability of failure-free (as per specification) operation over a specified time, in a given environment, for a specific purpose.
 - ▶ Depends on the environment

- ▶ **Availability:** The probability that a system, at a point in time, will be operational and able to deliver the requested services.
 - ▶ Does not just depend on the number of system crashes, but also on the time needed to repair the faults that have caused the failure.

Reliability/availability terminology

| Term | Description |
|------------------------|---|
| Human error or mistake | Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight (midnight is 00.00 in the 24-hour clock). |
| System fault | A characteristic of a software system that can lead to a system error. The fault is the inclusion of the code to add 1 hour to the time of the last transmission, without a check if the time is greater than or equal to 23.00. |
| System error | An erroneous system state that can lead to system behavior that is unexpected by system users. The value of transmission time is set incorrectly (to 24.XX rather than 00.XX) when the faulty code is executed. |
| System failure | An event that occurs at some point in time when the system does not deliver a service as expected by its users. No weather data is transmitted because the time is invalid. |

Safety

- ▶ Safety is a property of a system that reflects the system's ability to operate, normally or abnormally, without danger of causing human injury or death and without damage to the system's environment.
- ▶ It is important to consider software safety as most devices whose failure is critical now incorporate software-based control systems.
- ▶ Safety requirements are often exclusive requirements i.e. they exclude undesirable situations rather than specify required system services. These generate functional safety requirements.

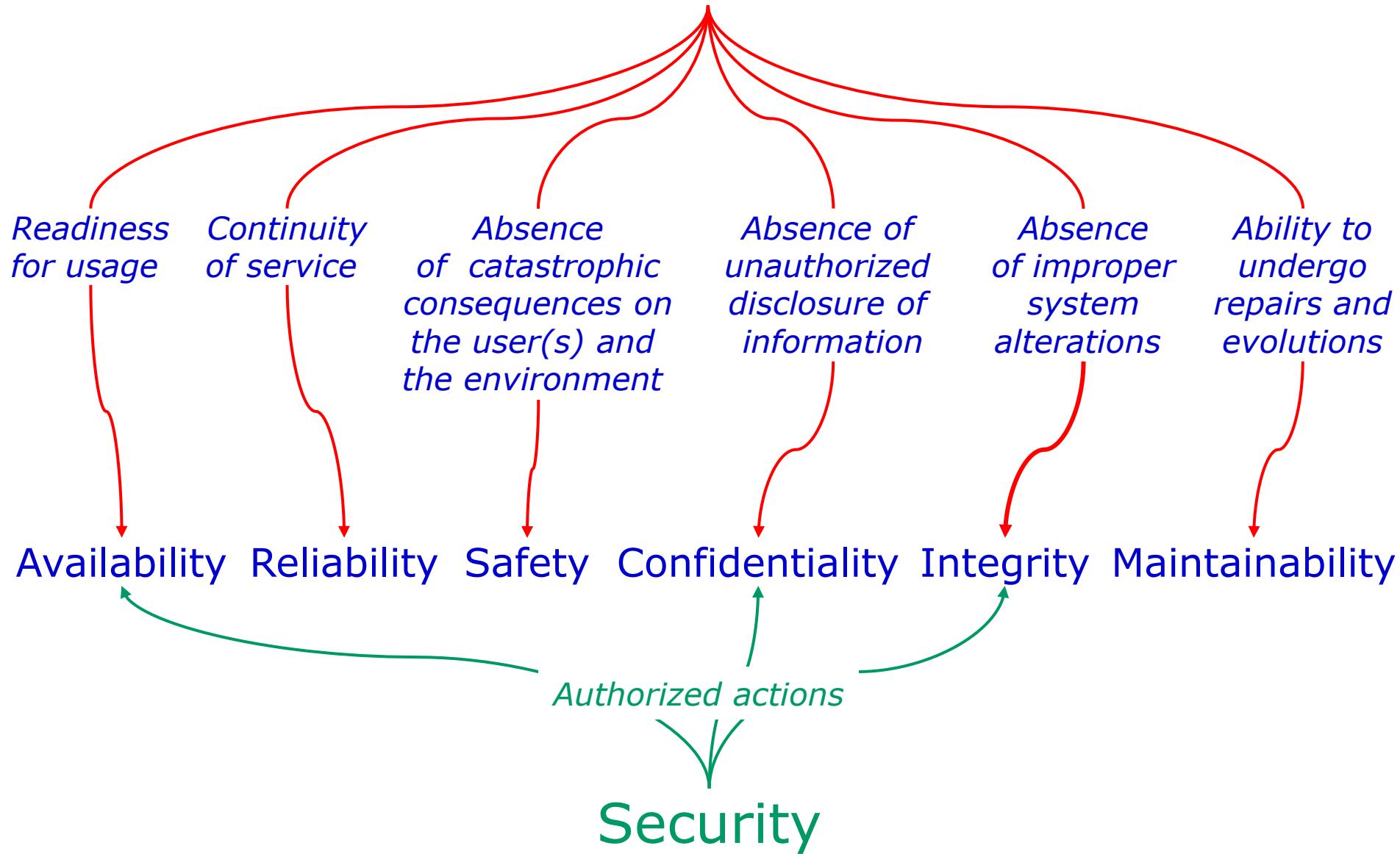
Safety and reliability

- ▶ Safety and reliability are related but distinct
 - ▶ In general, reliability and availability are necessary but not sufficient conditions for system safety
- ▶ Reliability is concerned with conformance to a given specification and delivery of service
- ▶ Safety is concerned with ensuring system cannot cause damage irrespective of whether or not it conforms to its specification

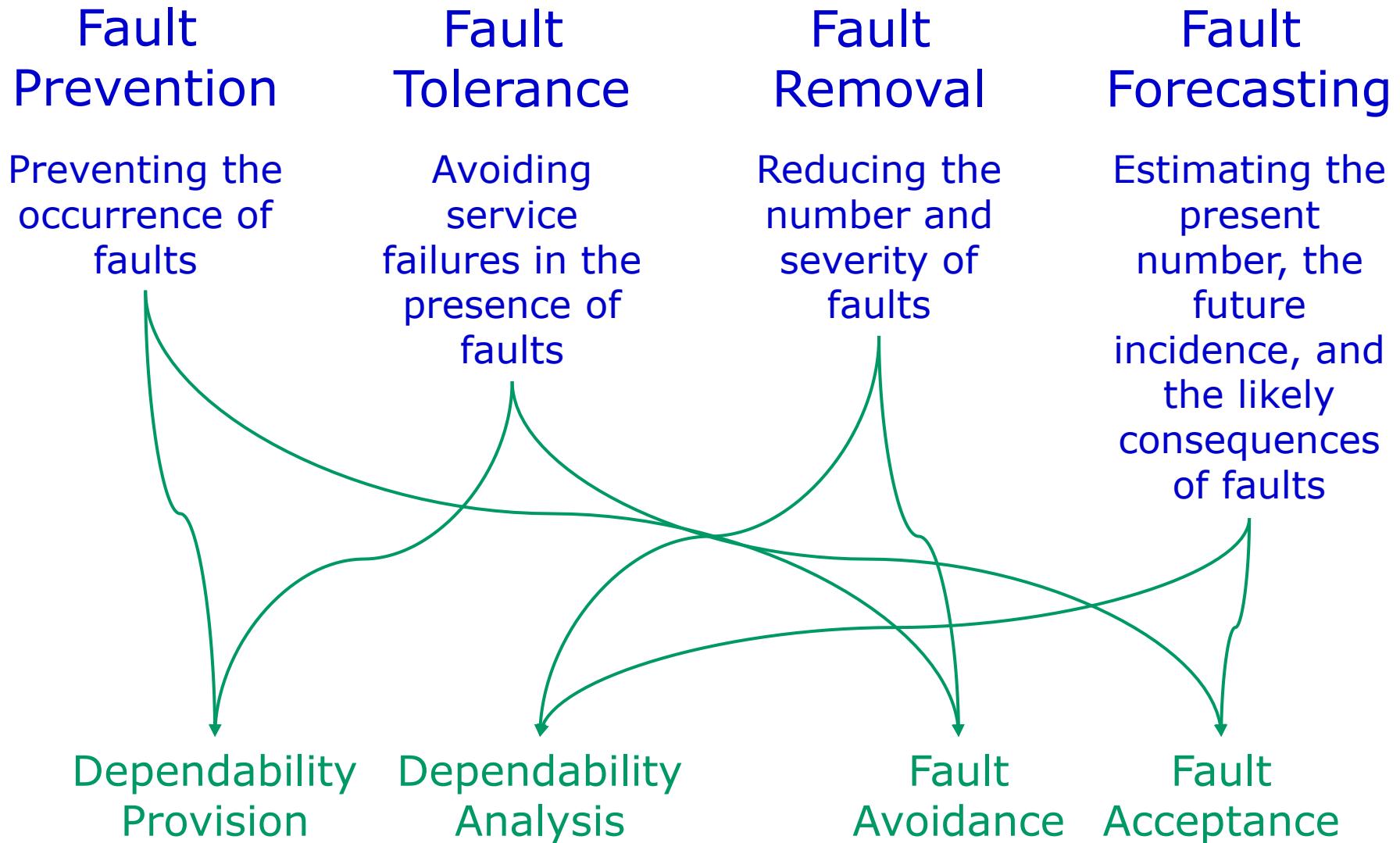
Safety terminology

| Term | Definition |
|----------------------|---|
| Accident (or mishap) | An unplanned event or sequence of events which results in human death or injury, damage to property, or to the environment. An overdose of insulin is an example of an accident. |
| Hazard | A condition with the potential for causing or contributing to an accident. A failure of the sensor that measures blood glucose is an example of a hazard. |
| Damage | A measure of the loss resulting from a mishap. Damage can range from many people being killed as a result of an accident to minor injury or property damage. Damage resulting from an overdose of insulin could be serious injury or the death of the user of the insulin pump. |
| Hazard severity | An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic, where many people are killed, to minor, where only minor damage results. When an individual death is a possibility, a reasonable assessment of hazard severity is 'very high.' |
| Hazard probability | The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from 'probable' (say 1/100 chance of a hazard occurring) to 'implausible' (no conceivable situations are likely in which the hazard could occur). The probability of a sensor failure in the insulin pump that results in an overdose is probably low. |
| Risk | This is a measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity, and the probability that the hazard will lead to an accident. The risk of an insulin overdose is probably medium to low. |

Dependability & Security



Means to attain dependability



Fault Prevention

▶ Fault avoidance

- ▶ Techniques that prevent introduction of faults during development
 - ▶ Hardware: use reliable components, packaging
 - ▶ Software: formal specs, use of proven design methods.

▶ Fault removal

- ▶ System testing is most important
- ▶ Reviews, verifications, code inspections
- ▶ 'A test can only show presence of faults, but never prove its absence'

[Dijkstra]

Fault Tolerance

- ▶ The ability of the system to continue functioning irrespective of the presence of faults and failures
- ▶ Levels of fault tolerance
 - ▶ Fail operational (Full fault tolerance)
 - ▶ Fail soft (Graceful degradation)
 - ▶ Failsafe
- ▶ **Redundancy is the key for fault-tolerance**
 - ▶ Physical (Space), Information (data), Time, Analytical...

Fault Avoidance and Detection Techniques

- ▶ **Timeouts**
 - ▶ Retry
 - ▶ Abort
- ▶ **Audits**
 - ▶ Overcome data inconsistency.
 - ▶ Applicable in distributed systems
- ▶ **Exception Handling**
- ▶ **Task Rollback**

Fault forecasting

- ▶ **Evaluation of system behavior**
 - ▶ How to estimate the present number, the future incidents, the probability of different consequences
 - ▶ Qualitative (identify, classify, rank the failure modes, the event combinations, environmental conditions that would lead to system failures)
 - ▶ Quantitative (probabilistic)
- ▶ **Data analysis**



Software security

Security

- ▶ **Information security**
 - ▶ “The protection of information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide confidentiality, integrity, and availability.”
- ▶ **Cyber security**
 - ▶ The ability to protect or defend the use of cyberspace from cyber attacks.

Types of security attacks

- ▶ **Injection attacks**
 - ▶ Buffer overflow
 - ▶ SQL injection (poisoning)
 - ▶ Cross-site scripting attacks
- ▶ Session hijacking attacks
- ▶ Denial of service attacks
- ▶ Bruteforce attacks
- ▶ Social engineering

Injection attacks

- ▶ Injection attacks are a type of attack where a malicious user uses a valid input field to input malicious code or database commands.
- ▶ These malicious instructions are then executed, causing some damage to the system. Code can be injected that leaks system data to the attackers.
- ▶ Common types of injection attack include buffer overflow attacks and SQL poisoning attacks.

SQL poisoning attacks

- ▶ SQL poisoning attacks are attacks on software products that use an SQL database.
- ▶ They take advantage of a situation where a user input is used as part of an SQL command.
- ▶ A malicious user uses a form input field to input a fragment of SQL that allows access to the database.
- ▶ The form field is added to the SQL query, which is executed and returns the information to the attacker.

SQL Injection

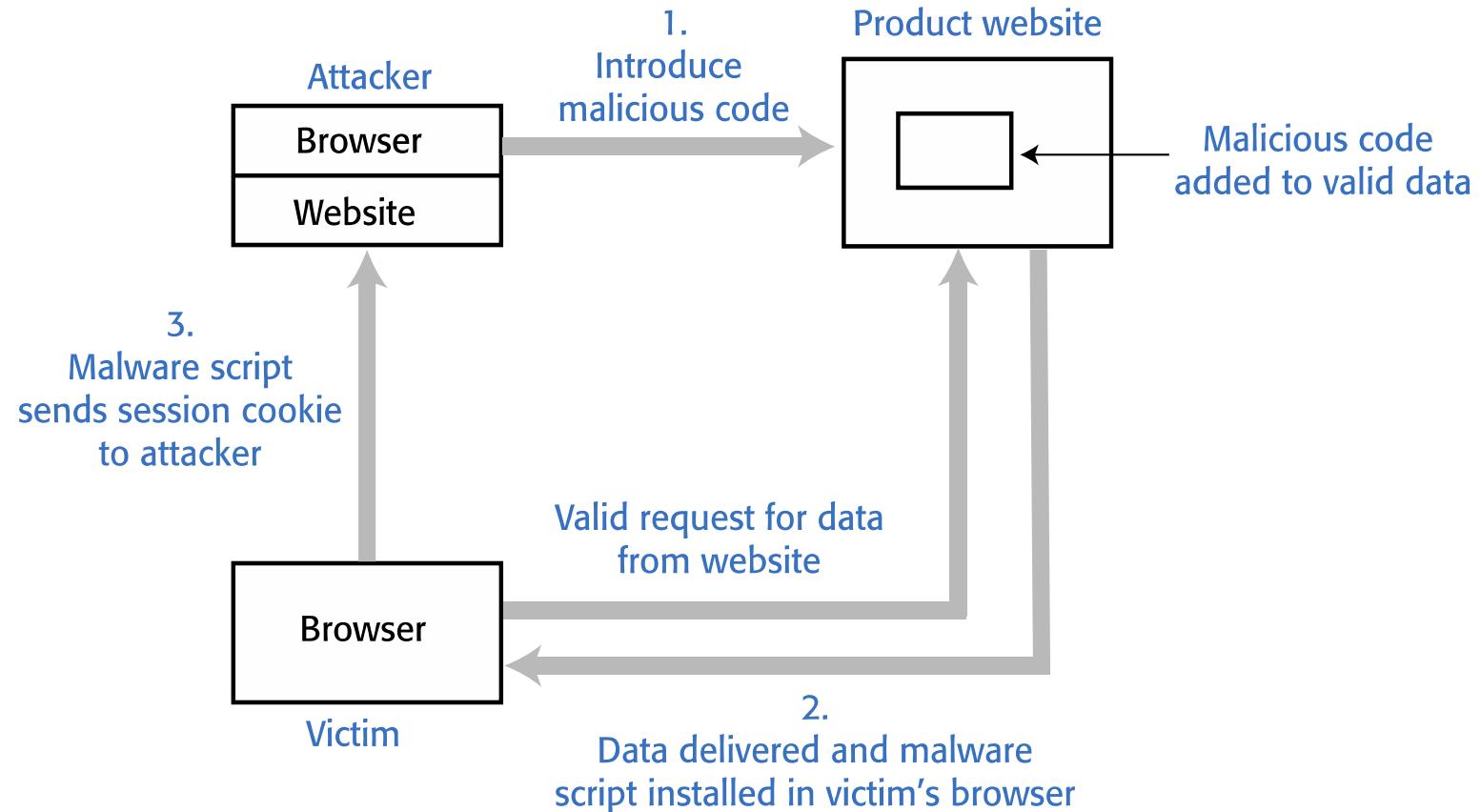
```
Select * FROM AccountHolders  
WHERE accountnumber = '2324324325'
```

```
Select * FROM AccountHolders  
WHERE accountnumber = '11000111' OR '1'='1'
```

Cross-site scripting attacks

- ▶ Cross-site scripting attacks are another form of injection attack.
- ▶ An attacker adds malicious Javascript code to the web page that is returned from a server to a client and this script is executed when the page is displayed in the user's browser.
- ▶ The malicious script may steal customer information or direct them to another website.
 - ▶ This may try to capture personal data or display advertisements.
 - ▶ Cookies may be stolen, which makes a session hijacking attack possible.
- ▶ As with other types of injection attack, cross-site scripting attacks may be avoided by input validation.

Cross-site scripting attack



Session hijacking attacks

- ▶ When a user authenticates themselves with a web application, a session is created.
 - ▶ A session is a time period during which the user's authentication is valid. They don't have to re-authenticate for each interaction with the system.
 - ▶ The authentication process involves placing a session cookie on the user's device
- ▶ Session hijacking is a type of attack where an attacker gets hold of a session cookie and uses this to impersonate a legitimate user.
- ▶ There are several ways that an attacker can find out the session cookie value including cross-site scripting attacks and traffic monitoring.
 - ▶ In a cross-site scripting attack, the installed malware sends session cookies to the attackers.
 - ▶ Traffic monitoring involves attackers capturing the traffic between the client and server. The session cookie can then be identified by analysing the data exchanged.

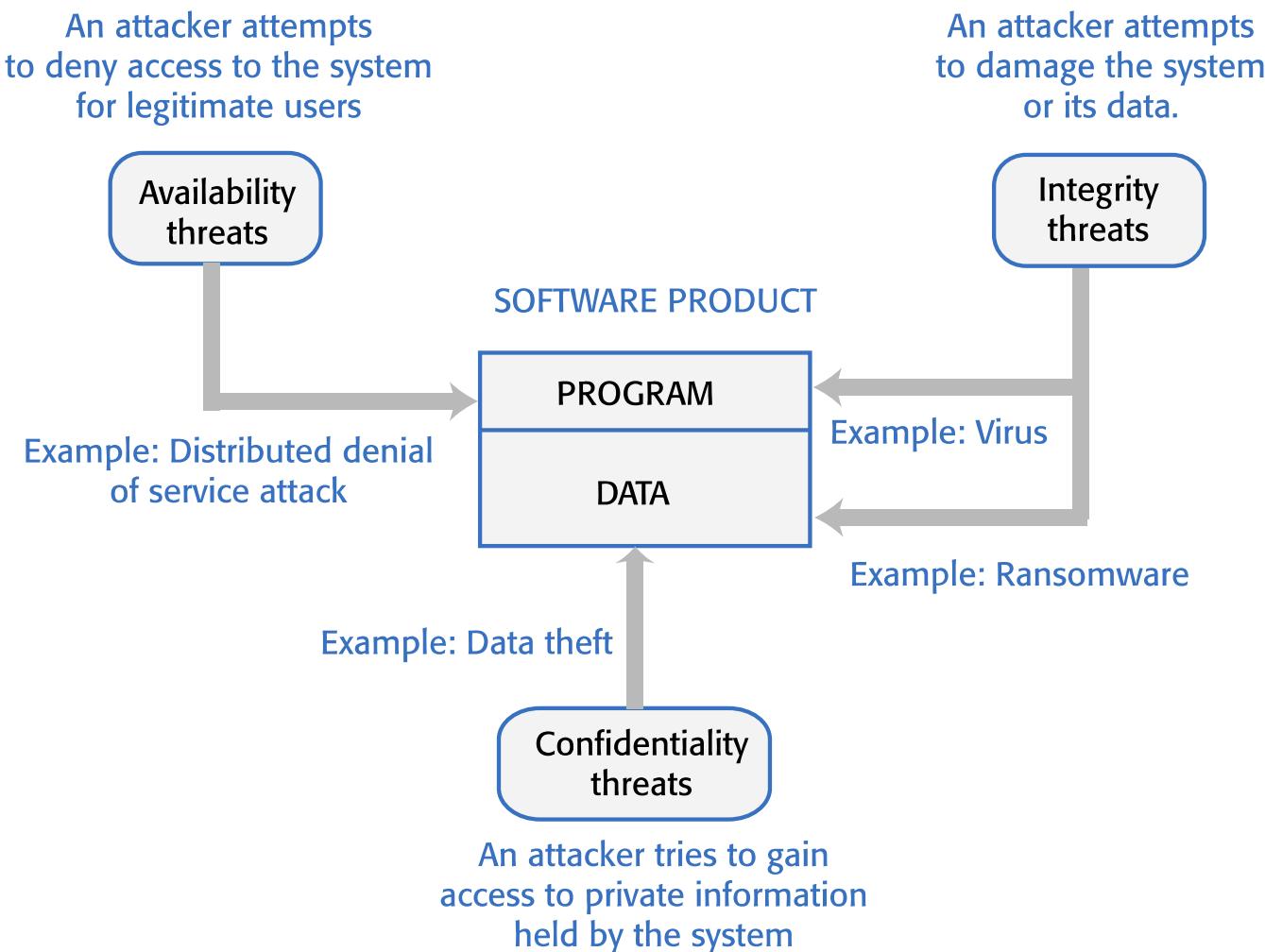
Denial of service attacks

- ▶ Denial of service attacks are attacks on a software system that are intended to make that system unavailable for normal use.
- ▶ Distributed denial of service attacks (DDOS) are the most common type of denial of service attacks.
 - ▶ These involve distributed computers, that have usually been hijacked as part of a botnet, sending hundreds of thousands of requests for service to a web application. There are so many service requests that legitimate users are denied access.
- ▶ Other types of denial of service attacks target application users.
 - ▶ User lockout attacks take advantage of a common authentication policy that locks out a user after a number of failed authentication attempts. Their aim is to lock users out rather than gain access and so deny the service to these users.
 - ▶ Users often use their email address as their login name so if an attacker has access to a database of email addresses, he or she can try to login using these addresses.
- ▶ If you don't lock accounts after failed validation, then attackers can use brute-force attacks on your system. If you do, you may deny access to legitimate users.

Brute force attacks

- ▶ Brute force attacks are attacks on a web application where the attacker has some information, such as a valid login name, but does not have the password for the site.
- ▶ The attacker creates different passwords and tries to login with each of these. If the login fails, they then try again with a different password.
 - ▶ Attackers may use a string generator that generates every possible combination of letters and numbers and use these as passwords.
 - ▶ To speed up the process of password discovery, attackers take advantage of the fact that many users choose easy-to-remember passwords. They start by trying passwords from the published lists of the most common passwords.
- ▶ Brute force attacks rely on users setting weak passwords, so you can circumvent them by insisting that users set long passwords that are not in a dictionary or are common words.

Types of security threats



Actions to reduce the likelihood of hacking

► **Traffic encryption**

Always encrypt the network traffic between clients and your server. This means setting up sessions using https rather than http. If traffic is encrypted it is harder to monitor to find session cookies.

► **Multi-factor authentication**

Always use multi-factor authentication and require confirmation of new actions that may be damaging. For example, before a new payee request is accepted, you could ask the user to confirm their identity by inputting a code sent to their phone. You could also ask for password characters to be input before every potentially damaging action, such as transferring funds.

► **Short timeouts**

Use relatively short timeouts on sessions. If there has been no activity in a session for a few minutes, the session should be ended and future requests directed to an authentication page. This reduces the likelihood that an attacker can access an account if a legitimate user forgets to log off when they have finished their transactions.

Authentication and authorization

▶ Authentication

- ▶ Authentication is the process of ensuring that a user of your system is who they claim to be.

▶ Authorization

- ▶ Authorization is a process in which user identity is used to control access to software system resources.

Authentication

- ▶ Authentication is the process of ensuring that a user of your system is who they claim to be.
- ▶ You need authentication in all software products that maintain user information, so that only the providers of that information can access and change it.
- ▶ You also use authentication to learn about your users so that you can personalize their experience of using your product.

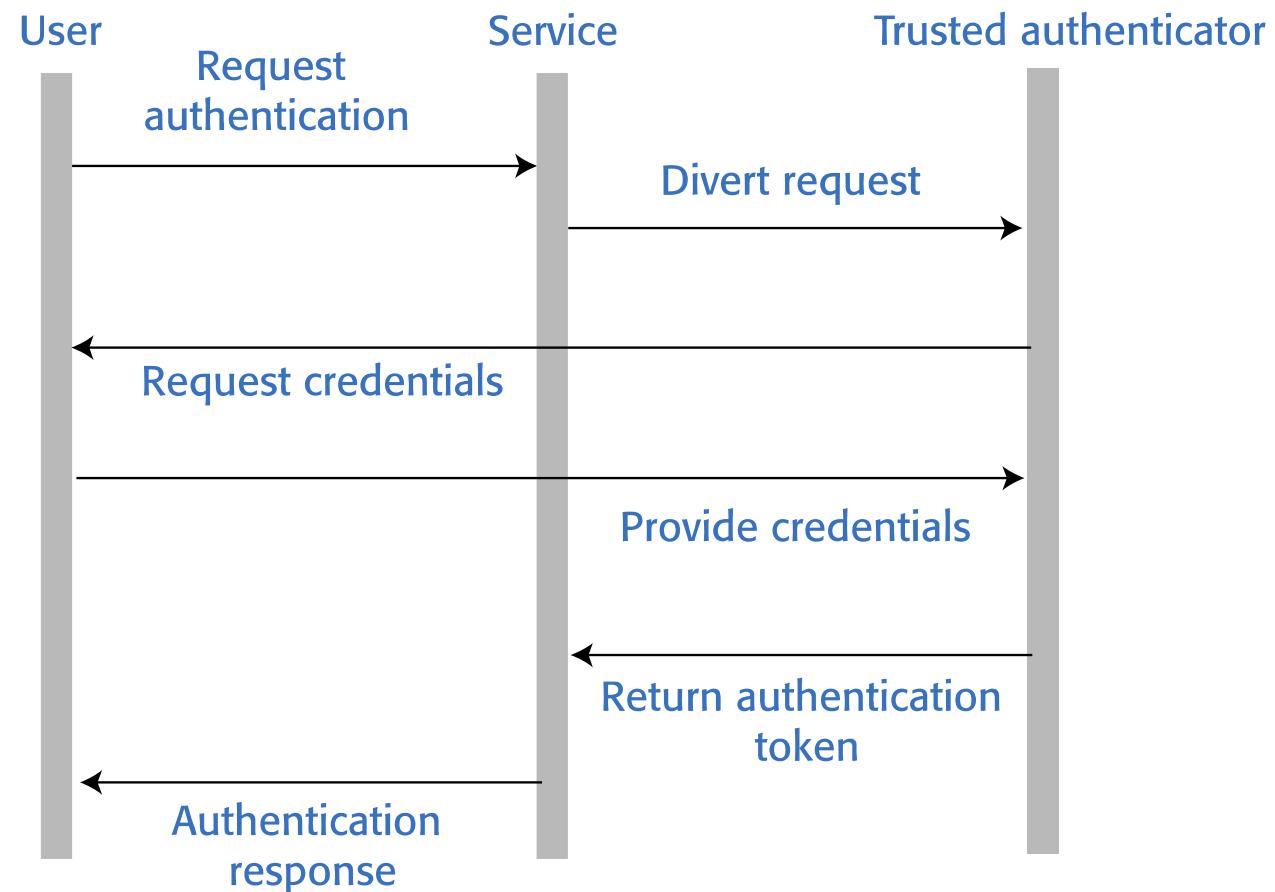
Authentication methods

- ▶ **Knowledge-based authentication**
 - ▶ The user provides secret, personal information when they register with the system. Each time they log on, the system asks them for this information.
- ▶ **Possession-based authentication**
 - ▶ This relies on the user having a physical device (such as a mobile phone) that can generate or display information that is known to the authenticating system. The user inputs this information to confirm that they possess the authenticating device.
- ▶ **Attribute-based authentication** is based on a unique biometric attribute of the user, such as a fingerprint, which is registered with the system.
- ▶ **Multi-factor authentication** combines these approaches and requires users to use more than one authentication method.

Federated identity

- ▶ Federated identity is an approach to authentication where you use an external authentication service.
- ▶ ‘Login with Google’ and ‘Login with Facebook’ are widely used examples of authentication using federated identity.
- ▶ The advantage of federated identity for a user is that they have a single set of credentials that are stored by a trusted identity service.
- ▶ Instead of logging into a service directly, a user provides their credentials to a known service who confirms their identity to the authenticating service.
- ▶ They don’t have to keep track of different user ids and passwords. Because their credentials are stored in fewer places, the chances of a security breach where these are revealed is reduced.

Federated identity



Authorization

- ▶ Authentication involves a user proving their identity to a software system.
- ▶ Authorization is a complementary process in which that identity is used to control access to software system resources.
 - ▶ For example, if you use a shared folder on Dropbox, the folder's owner may authorize you to read the contents of that folder, but not to add new files or overwrite files in the folder.
- ▶ When a business wants to define the type of access that users get to resources, this is based on an access control policy.
- ▶ This policy is a set of rules that define what information (data and programs) is controlled, who has access to that information and the type of access that is allowed

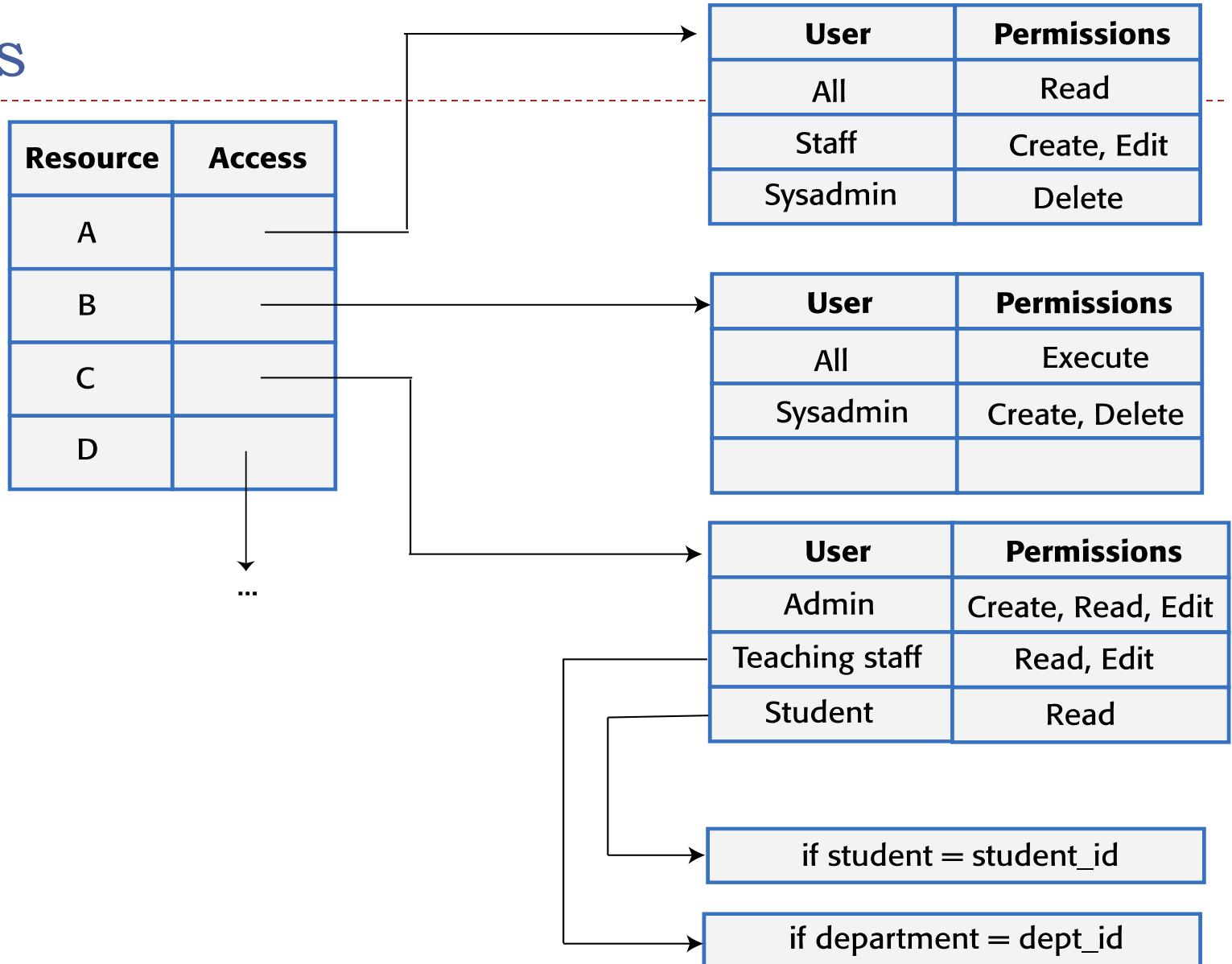
Access control policies

- ▶ Explicit access control policies are important for both legal and technical reasons.
- ▶ Data protection rules limit the access the personal data and this must be reflected in the defined access control policy. If this policy is incomplete or does not conform to the data protection rules, then there may be subsequent legal action in the event of a data breach.
- ▶ Technically, an access control policy can be a starting point for setting up the access control scheme for a system.
- ▶ For example, if the access control policy defines the access rights of students, then when new students are registered, they all get these rights by default.

Access control lists

- ▶ Access control lists (ACLs) are used in most file and database systems to implement access control policies.
- ▶ Access control lists are tables that link users with resources and specify what those users are permitted to do.
 - ▶ For example, for a given book it is required to be able to set up an access control list to a book file that allows reviewers to read that file and annotate it with comments. However, they are not allowed to edit the text or to delete the file.
- ▶ If access control lists are based on individual permissions, then these can become very large. However, you can dramatically cut their size by allocating users to groups and then assigning permissions to the group

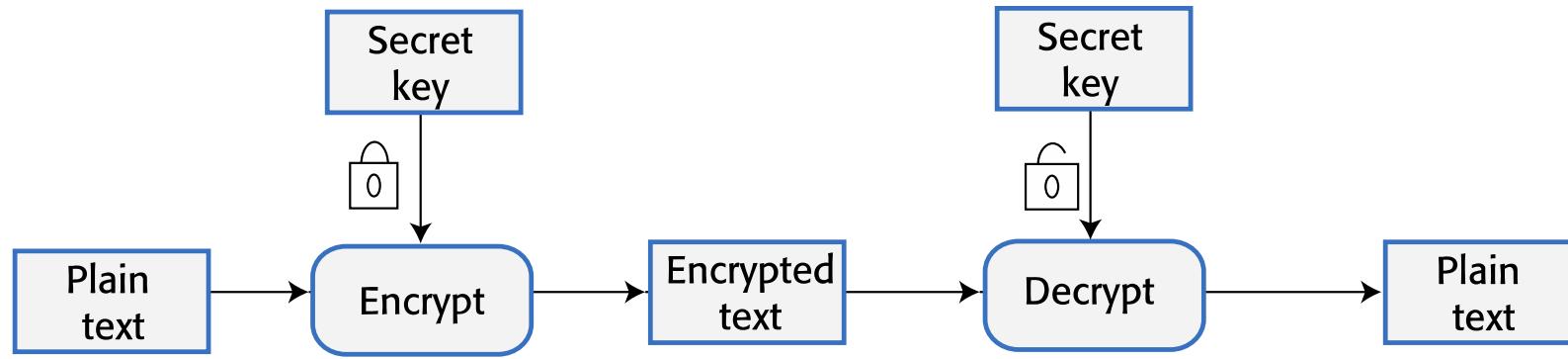
Access control lists



Encryption

- ▶ Encryption is the process of making a document unreadable by applying an algorithmic transformation to it.
- ▶ A secret key is used by the encryption algorithm as the basis of this transformation. You can decode the encrypted text by applying the reverse transformation.
- ▶ Modern encryption techniques are such that you can encrypt data so that it is practically uncrackable using currently available technology.
- ▶ However, history has demonstrated that apparently strong encryption may be crackable when new technology becomes available.
- ▶ If commercial quantum systems become available, we will have to use a completely different approach to encryption on the Internet.

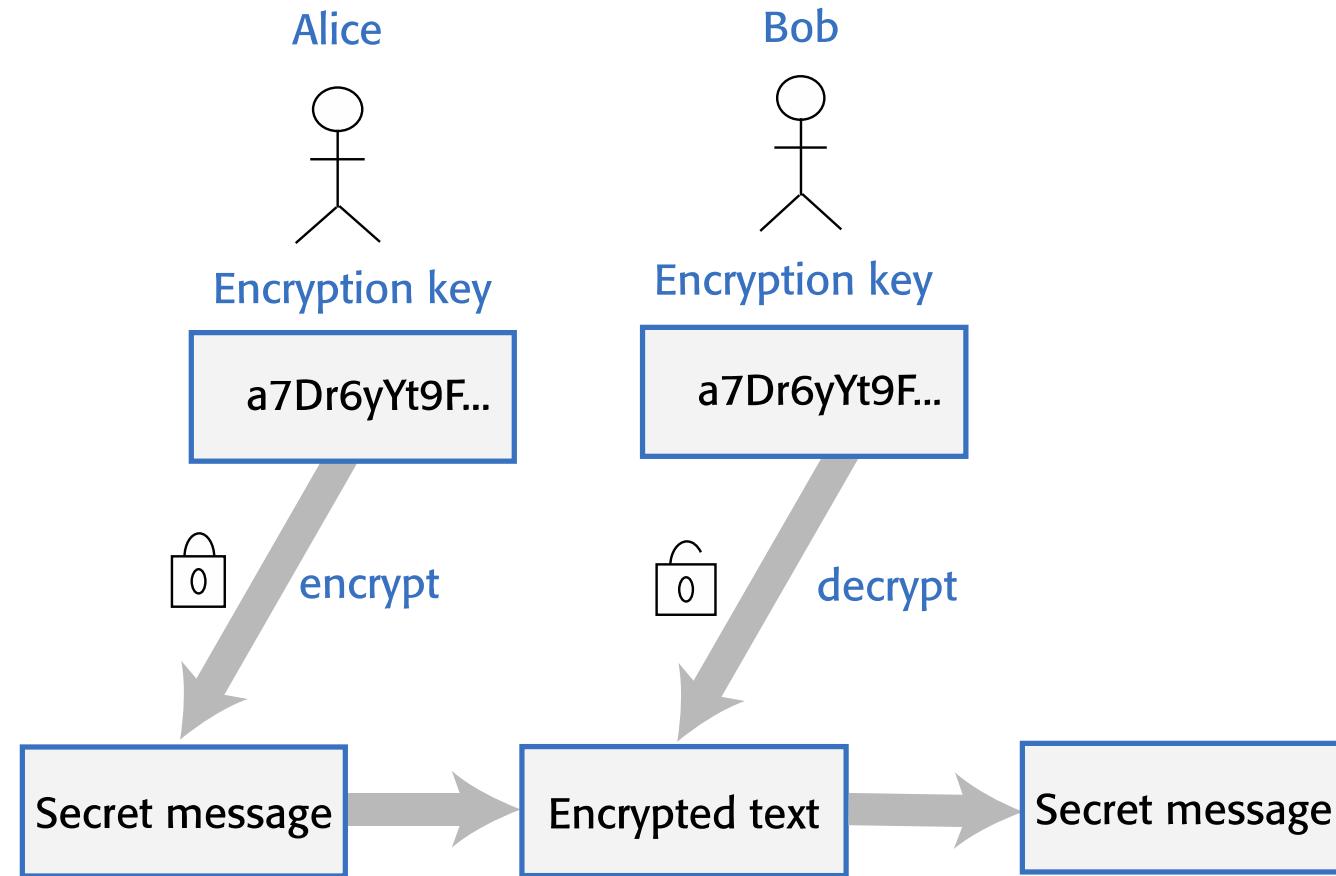
Figure 7.9 Encryption and decryption



Symmetric encryption

- ▶ In a symmetric encryption scheme, the same encryption key is used for encoding and decoding the information that is to be kept secret.
- ▶ If Alice and Bob wish to exchange a secret message, both must have a copy of the encryption key. Alice encrypts the message with this key. When Bob receives the message, he decodes it using the same key to read its contents.
- ▶ The fundamental problem with a symmetric encryption scheme is securely sharing the encryption key.
- ▶ If Alice simply sends the key to Bob, an attacker may intercept the message and gain access to the key. The attacker can then decode all future secret communications.

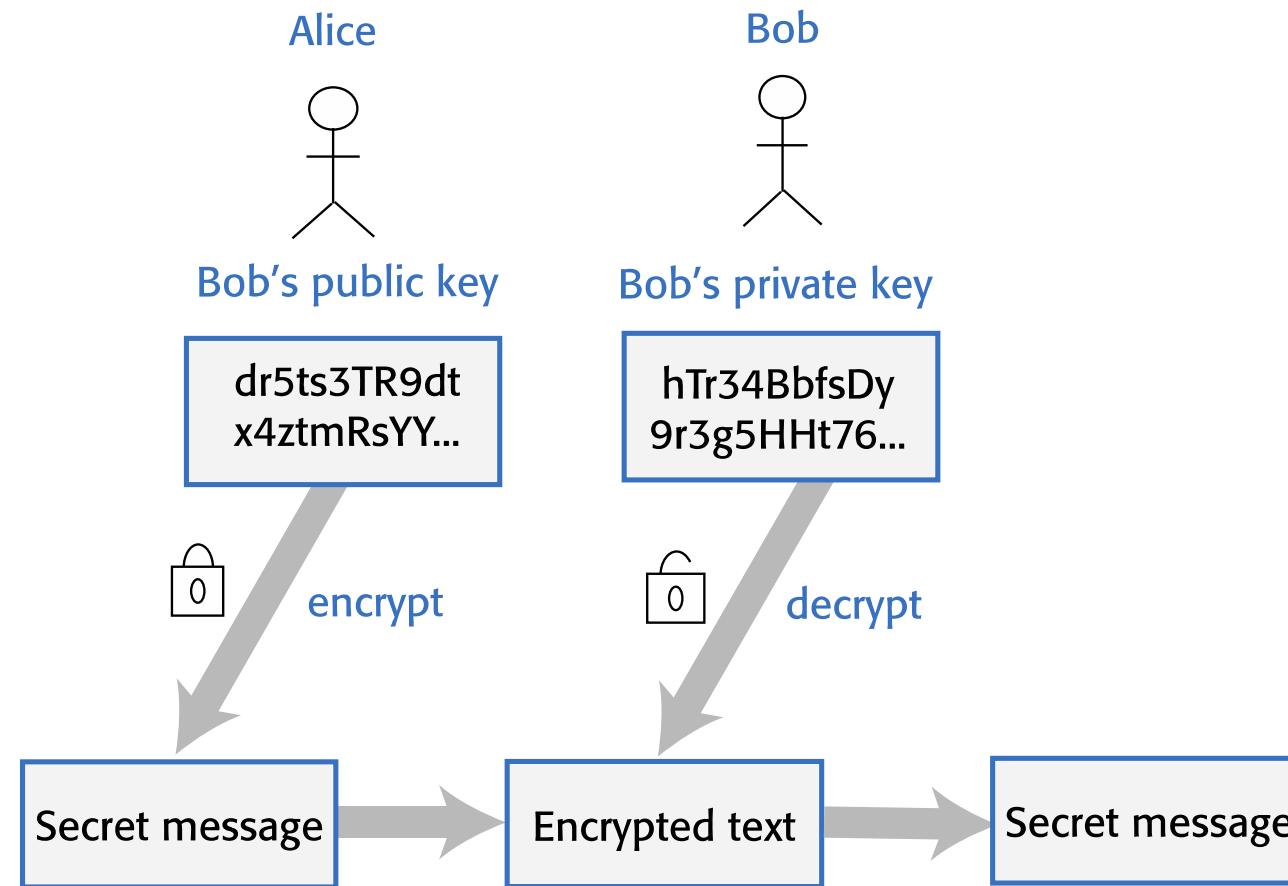
Symmetric encryption



Asymmetric encryption

- ▶ Asymmetric encryption, does not require secret keys to be shared.
- ▶ An asymmetric encryption scheme uses different keys for encrypting and decrypting messages.
- ▶ Each user has a public and a private key. Messages may be encrypted using either key but can only be decrypted using the other key.
- ▶ Public keys may be published and shared by the key owner. Anyone can access and use a published public key.
- ▶ However, messages can only be decrypted by the user's private key so is only readable by the intended recipient

Asymmetric encryption



Additional reading

- ▶ A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. In Proceedings of the 3rd Information Survivability Workshop, 2000.
 - ▶ https://www.cs.cmu.edu/~garlan/17811/Readings/avizienis01_fund_concp_depend.pdf
- ▶ Sommerville, Ian, Engineering Software Products: An Introduction to Modern Software Engineering, 2021;
 - ▶ Chapters 7 & 8

Лекция 4

Анализ на изискванията:

- А. Изисквания към софтуера
- Б. Процес на инженеринг на изискванията

А. Изисквания към софтуера

Съдържание

- Потребителски и системни изисквания
- Функционални и нефункционални изисквания
- Как софтуерните изисквания могат да бъдат описани в документ

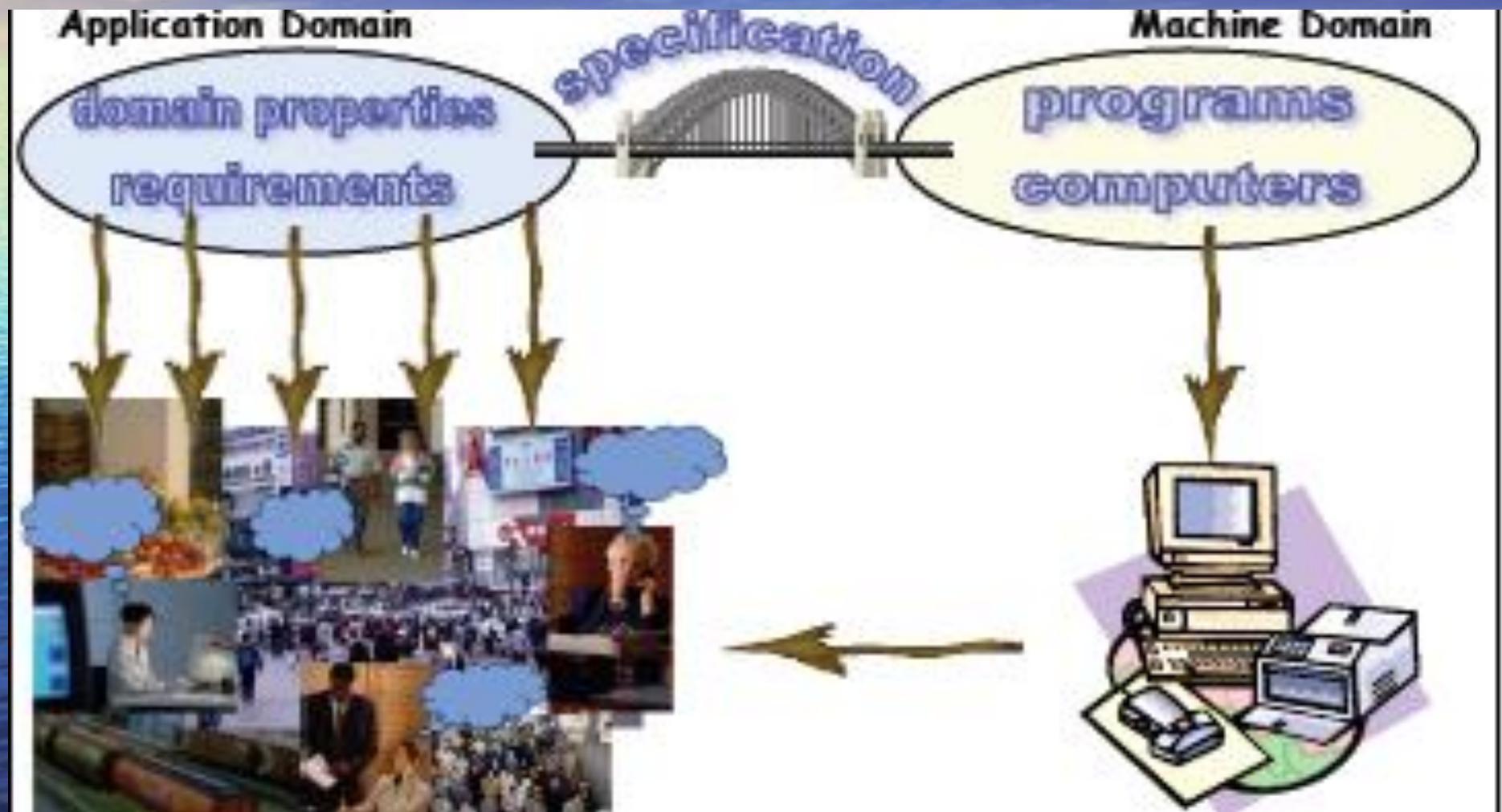
Какво е изискване?

- Изискванията са спецификации на това, което трябва да бъде разработено.
- Изискванията са описание на:
 - **начина на поведение (взаимодействие с потребителя) на системата,**
 - **на системни характеристики или атрибути.**
 - **ограничения** върху процеса на разработване.
- Изискванията дефинират:
 - Общо разбиране за **системната функционалност(и);**
 - **Необходимостта от системата и стойността на разработката;**
 - **Какво трябва да прави дадена система, но не и как да го прави!**

Какво е изискване?

- Изискванията са зависими от гледните точки на различните участници в процеса на разработване
 - ... най-честата причина за трудности и проблеми при разработване
- Изискванията формират база за планиране на проекта, управление на рисковете, тестването, отстъпките, които могат да се направят, както и промените.
- *Добрите* изисквания дават много преимущества
- *Лошите* изисквания водят до трудности на проекта и провал.

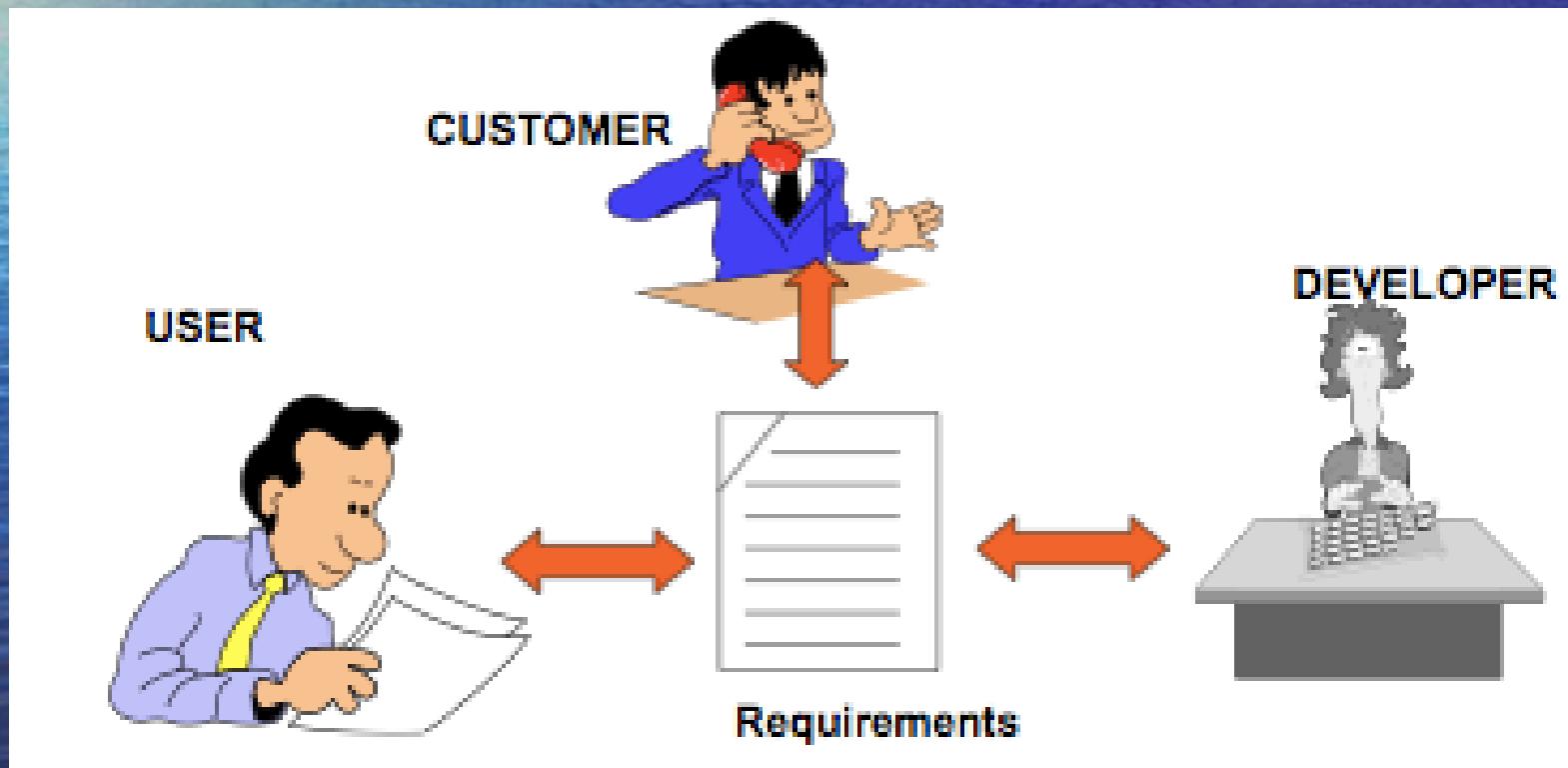
Предизвикателства



Защо са важни изискванията?

Изискванията са как общуваме.

Те са единствената част, която всички разбират.
Основа за бъдещата разработка.



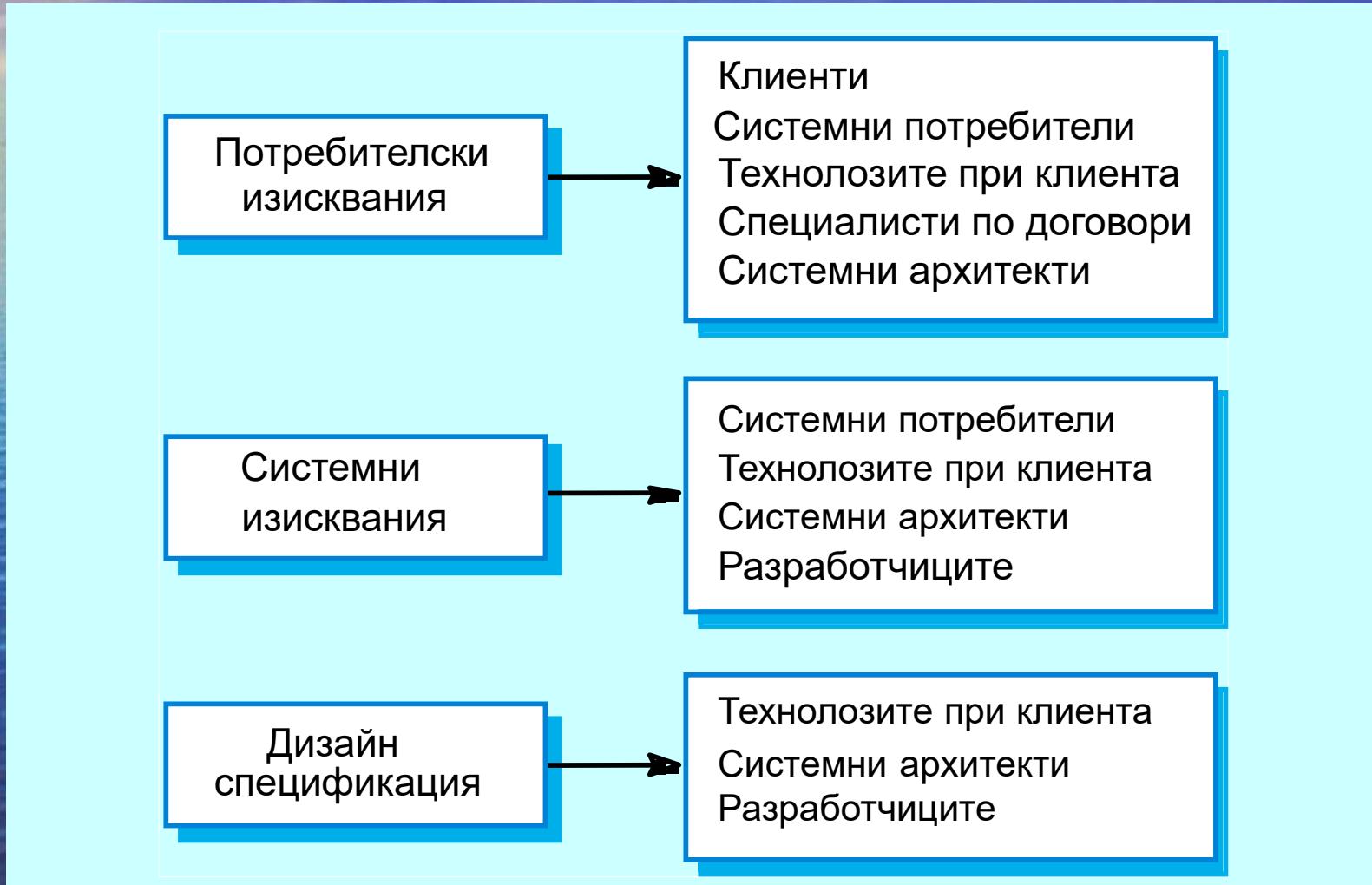
Видове изисквания - 1

- Изискванията могат да варират от много абстрактни описания на дадена системна функционалност или ограничения до много точни и детайлни математически функционални спецификации
- Изискванията могат да имат и друга роля:
 - Могат да послужат като заявка за търг – следователно трябва да са отворени за интерпретация
 - Могат да бъдат и база за договор – следователно трябва да бъдат детайлно дефинирани.

Видове изисквания - 2

- **Потребителски изисквания**
 - Указания на естествен език и диаграми на услугите, които системата ще предоставя, както и съответните оперативни ограничения. Написани са за клиентите.
- **Системни изисквания**
 - Структуриран документ с детайлно описание на системните функции, услуги и оперативни ограничения. Определя какво трябва да се имплементира и може да бъде част от договора между клиента и разработчика

За кого са предназначени изискванията



Системни изисквания

- По-детайлни спецификации на системните функции, услуги и ограничения от потребителските изисквания.
- Предназначени са да послужат като основа на дизайна на системата.
- Могат да бъдат включени в договора за разработване на системата.
- Системните изисквания могат да бъдат дефинирани и илюстрирани като се използват и различни **системни модели**

Функционални и нефункционални изисквания

- **Функционални изисквания**
 - Описание на услугите, които системата трябва да предоставя, начинът по който системата трябва да реагира на конкретни входни данни и поведението и в конкретни ситуации.
- **Нефункционални изисквания**
 - Ограничения върху услугите или функционалността на системата като времеви ограничения, ограничения върху процеса на разработване, използваните стандарти и др.
- **Изисквания на приложната област**
 - Изисквания произлизящи от приложната област на системата, които отразяват спецификите и.

ФУНКЦИОНАЛНИ ИЗИСКВАНИЯ

- Описват функции или услуги на системата.
- Зависят от типа на софтуера, потенциалните потребители и типа на системата, в която ще бъде използван софтуера.
- Функционалните потребителски изисквания могат да бъдат описания на високо ниво на детайлност, на това какво трябва да прави системата. Но функционалните системни изисквания трябва да описват системната функционалност в детайли – **нужната входна и получаваната изходна информация**.

Пример – Библиотечна система

Потребителски изисквания:

- Библиотечна система, която предоставя единен интерфейс към множество бази данни от статии в различни библиотеки.
- Потребителите могат да търсят, четат, свалят и разпечатват статии.

Примери за функционални изисквания

- Потребителите трябва да могат да търсят във всички бази или да изберат подмножество (**точно кои?**).
- Системата трябва да предостави подходящата (**каква?**) среда за потребителите, за да могат да разглеждат и четат съответните документи.
- На всяка заявка трябва да бъде даден уникален индентификатор (ORDER_ID).

Неточности при изискванията

- Поради неточното описание на изискванията могат да възникнат проблеми:
 - Двусмислените изисквания могат да бъдат интерпретирани по различен начин от разработчиците и потребителите.
- Пример: за Библиотечната система
Ако разгледаме изискването за ‘подходящата среда’
 - Според потребителя – различна среда за всеки различен тип документ.
 - Според разработчиците – да се предостави текстова среда, която да покаже съдържанието на документа.

Пълнота и консистентност на изискванията

- Изискванията трябва да са **пълни** и **консистентни**.
- **Пълни**
 - Трябва да включват пълно описание на всички изисквани функции.
- Пример: „...търсят във всички бази или да изберат подмножество (**точно кои?**).“
- **Консистентни**
 - Не трябва да има конфликти или противоречия при описанието на системните функции.
- **Note:** На практика е невъзможно да се направи съвсем пълна и консистентна спецификация на изискванията.

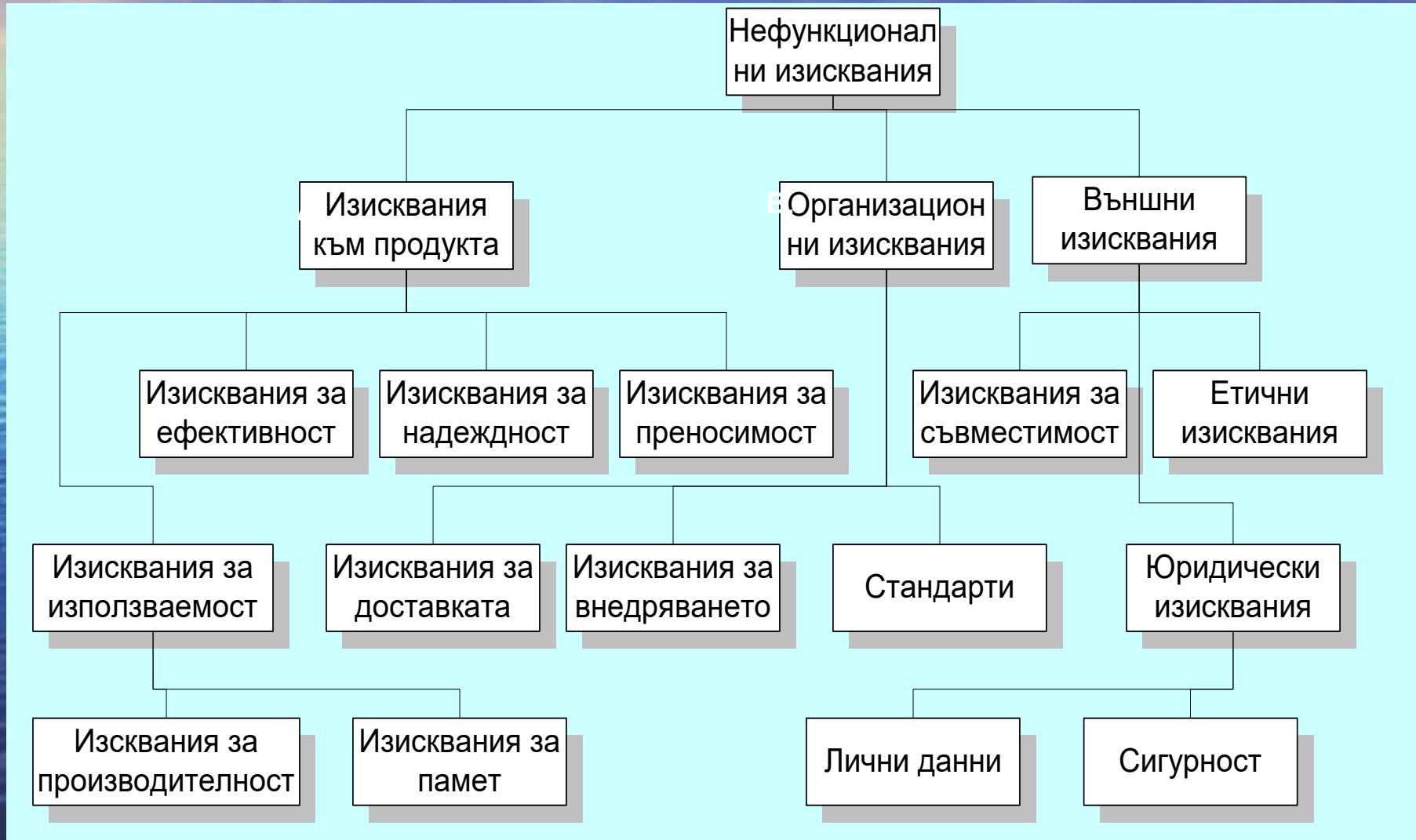
Нефункционални изисквания

- Определят системни характеристики и ограничения като надежност, време за отговор и др. – **качество на софтуера**
- **Изисквания към процеса** могат също да бъдат специфицирани – например конкретна CASE система, програмен език или метод на разработване.
- Нефункционалните изисквания **могат да бъдат покритични** от функционалните и от тях може да зависи пригодността на цялата система.

Типове нефункционални изисквания

класификация на НФИ

/по Sommerville/



Класифициране на нефункционалните изисквания

- **Продуктови изисквания**
 - Изисквания, които определят поведението/качеството на продукта, като време за изпълнение, надеждност и др.
- **Организационни изисквания**
 - Изисквания, които са следствие на практиката и процедурите в дадена организация, като прилаганите стандарти при процеса на разработване, изисквания към имплементацията и др.
- **Външни изисквания**
 - Изисквания породени от външни за системата и процеса на разработване фактори, като изисквания за оперативна съвместимост и нормативни (законодателни) изисквания и др.

Примери за нефункционални изисквания

- **Продуктови изисквания**
- - Потребителят трябва да може лесно (с 3 клика) да намира търсения документ. (*usability*)
- **Организационни изисквания**
 - Потребителският интерфейс на библиотечната система трябва да бъде имплементиран като HTML без фреймове Java аплети.
 - Процесът на разработване на системата, както и документите трябва да съответстват на стандарт XYZCo-SP-STAN-95.
- **Външни изисквания**

Системата не трябва да включва никаква персонална информация за клиентите, освен тяхното име.

Цели и изисквания

- Трудно е да се специфицират прецизно нефункционалните изисквания. От друга страна непрецизните изисквания трудно се верифицират.
- Целите са полезни на разработчиците, защото представлят виждането на системните потребители.
- *Пример: Цел*
 - Основно виждане на потребителя - лесно ползване.
- *Верифицираме нефункционални изисквания*
 - Основават се на мерки, които могат да бъдат обективно тествани.

Мерки (метрики) за оценка на изискванията

| | |
|--------------------------------------|--|
| Харakterистика | Мярка |
| Скорост | Обработени транзакции за секунда Време за отговор на потребител/събитие Време за обновяване на екрана |
| Размер | M Bytes Брой ROM чипове |
| Леснота на ползване | Време за обучение |
| Надеждност | Най-малкото време до грешка Вероятност за не наличност Ниво на проявяване на грешки Наличност |
| Възможност за преодоляване на грешки | Време за рестартиране след грешка Процент на събитията, водещи до грешка Вероятност за нарушаване на данните при грешка. |

Взаимодействие на изискванията

- Конфликти между различните нефункционални изисквания са често срещани в големите комплексни системи.
- Пример: Самолетна система
 - За да се минимизира теглото, трябва да се намали броят на отделните чипове.
 - За да се минимизира консумацията на енергия, трябва да се използват чипове, които консумират по-малко енергия.
 - Обаче...използването на чипове, които консумират по-малко енергия води до използването на повече чипове

Кое е по-важното изискване?

Изисквания на приложната област

- Произлизат от приложната област и описват характеристики и особености на системата отразяващи спецификите на областта.
- Могат да бъдат: 1) нови функционални изисквания, 2) ограничения върху вече съществуващи изисквания или 3) да дефинират специфични изчисления.
- Ако изисквания на приложната област не са удовлетворени, това може да доведе до непригодност на системата.

Примери: Библиотечна система (авторско право)

Медицинска система (лични данни на пациент)

Проблеми при изисквания на приложната област

- Разбираемост
 - Изискванията са изразени на езика на приложната област (пример?)
 - Това често ги прави неразбираеми за софтуерните инженери, разработващи системата.
- Имплицитност
 - Специалистите от приложната област познават областта толкова добре, че те не могат да определят изискванията експлицитно/явно.

Проблеми с описание на изискванията на естествен език

- Липса на яснота
 - Трудно е да се постигне прецизност, без това да се отрази на четимостта на документа
- Объркване при изискванията
 - Могат да се смесят функционалните и нефункционални изисквания.
- Сливане на изисквания
 - Няколко различни изисквания могат да бъдат описани заедно.

Пример: Изисквания към библиотечната система

Библиотечната система трябва да предостави финансова счетоводна система, която да отчита всички плащания, направени от потребителите на системата. Системните менажери трябва да могат да конфигурират системата така, че редовните потребители да могат да ползват намаления.

Пример: Проблеми при изискванията

- По-горните изисквания включват и концептуална и много детайлна информация
 - Описват концепция за финансово счетоводна система
 - Описват и детайли – менажерите могат да конфигурират системата – това е излишно на това ниво

Насоки за описание на изисквания

- Добре е да се използва **единен стандартен** формат за описание, който да се използва за всички изисквания
- Езикът трябва да се използва консистентно. Да се използва “трябва” за задължителни изисквания и “би могло” за желателни.
- Може да се използва удебеляване на текста за да се подчертаят ключови части от изискванията.
- Да се избягва употребата на (компютърен) жаргон.

Изисквания и дизайн

- Изискванията определят какво тряба да прави системата, а дизайнът описва как ще го прави.
-
- На практика, изискванията и дизайна са неразделни
 - Системната архитектура и дизайн структурират изискванията
 - Системата може да работи съвместно с други системи, което да определи нови изисквания към дизайна
 - Използването на специфичен дизайн може да е вследствие на изисквания на приложната област.

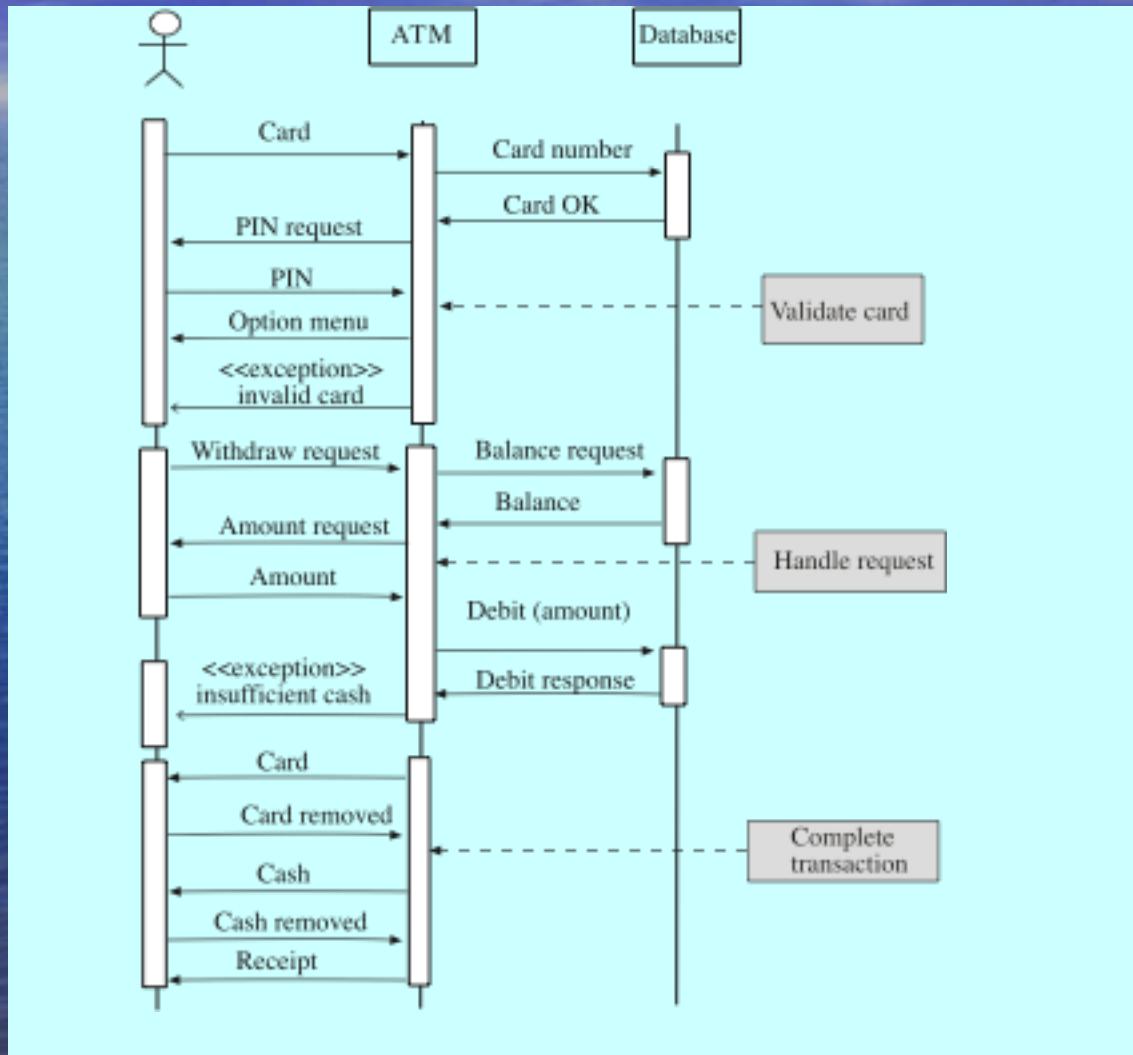
Модели на системата

- Графичните модели са най-полезни, когато се налага да се опише как се променят състояния или конкретна последователност от действия.
- Видове модели:
 - Структурни модели
 - Формални модели

Диаграми на последователност

- Показват последователност от събития, които се случват докато потребителят си взаимодейства със системата.
- Разглеждат се отгоре надолу за да се проследи реда на действие на събитията.
- Теглене на пари от банкомат
 - Валидиране на картата;
 - Обработване на заявката;
 - Завършване на трансакцията.

Диаграмма на последовательност – теглене на пари от банкомат



IEEE стандарт за изисквания

- Дефинира основна структура за спецификацията на софтуерните изисквания, която да бъде инстанцирана за всяка отделна система.
 - Въведение
 - Основно описание
 - Специфични изисквания
 - Приложения
 - Индекс

Структура на документа

- Предисловие
- Въведение
- Речник
- Дефиниране на потребителските изисквания
- Системна архитектура
- Спецификация на системните изисквания
- Модели на системата
- Развитие на системата
- Приложения
- Индекс

Обобщение

- Изискванията определят какво трябва да прави системата и дефинират ограниченията върху нейната работа и имплементация.
- Функционалните изисквания определят услугите, които системата трябва да предоставя.
- Нефункционалните изисквания ограничават разработваната система и процеса на разработване.
- Потребителските изисквания са твърдения на високо ниво, за това какво трябва да прави системата. Те трябва да бъдат описани на естествен език и чрез таблици и диаграми.

Обобщение 2

- Системните изисквания са предназначени да комуникират функциите, които системата трябва да предостави.
- Спецификацията на софтуерните изисквания е съгласувано споразумение за системните изисквания.
- IEEE стандартът е полезен като отправна точка за дефиниране на по-детайлни специфични стандарти за изисквания.

Б. Процес на инженеринг на изискванията

Съдържание

- Основни дейности по специфициране на изискванията и връзките между тях.
- Техники за идентифициране и анализ на изискванията
- Валидиране на изискванията и ролята на ревютата на изисквания.
- Ролята на управление на изискванията в подкрепа на останалите дейности при специфицирането на изискванията.

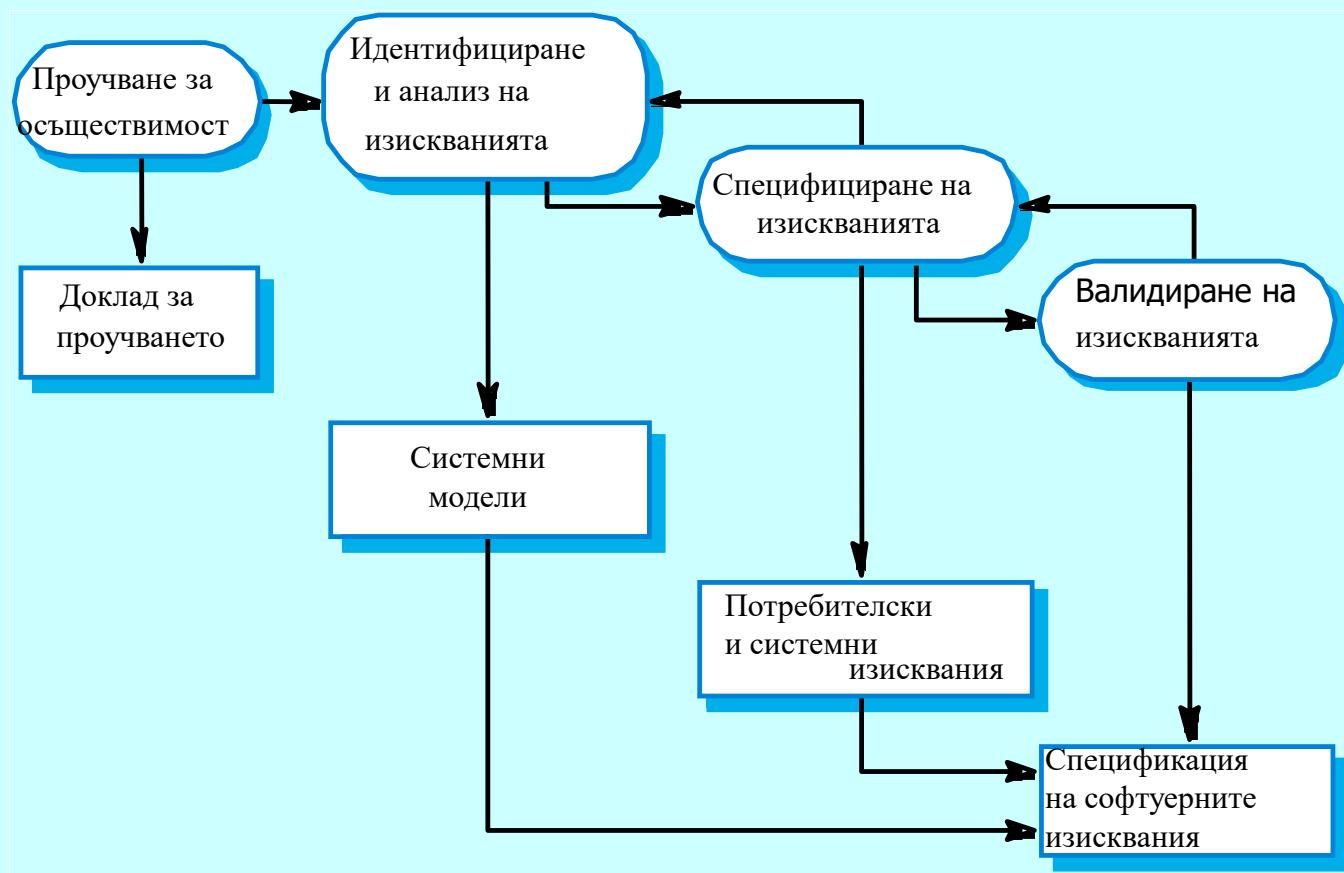
Инженеринг на изискванията (ИИ)

- **Процес на установяване на услугите, които клиент изисква от системата, както и ограниченията на работа и разработване.**
- **Изискванията** са описание на системните **функции и ограниченията**, които се установяват по време на инженеринга на изискванията.
 - Неформално... Дисциплина от софтуерното/системното инженерство, която обхваща дейностите по специфициране на продукта/системата.
- ИИ е изключително важна част от процеса на системно разработване, която направлява всички останали дейности към постигане на резултатите, за които система е предназначена.
- ИИ оказва влияние на целия жизнен цикъл на дадена система

Къде е мястото на ИИ в цялостния процес на разработка на софтуер?



Процес на специфициране на изискванията (Инженеринг на изискванията)



Проучвания за осъществимост

- Проучване за осъществимост се прави с цел да се докаже, че разработването на системата си струва
- Кратко фокусирано проучване, което проверява дали:
 - системата допринася за целите на организацията;
 - системата може да бъде реализирана с настоящите технологии и в определен бюджет;
 - системата може да бъде интегрирана с други използвани системи.

Действия на процеса на ИИ

- Откриване (идентифициране) на изискванията
 - Взаимдействия със заинтересованите лица, за да се установят изискванията. На този етап се определят и изискванията на приложната област.
- Организиране и класификация на изискванията.
 - Свързаните изисквания могат да бъдат класифицирани и организирани в съгласувани групи (клъстери).
- Приоритизация и договаряне (част от анализа на изискванията)
 - Приоритизиране на изискванията и разрешаване на конфлктите.
- Документиране (специфициране) на изискванията
 - Изискванията за документирани и се тръгва по следващия цикъл на спиралата.
- Валидиране на изискванията
- Управление на изискванията

Идентифициране и анализ

- Понякога наричано *извлечане на изискванията* или *откриване*.
- Включва технологии, работещи заедно с клиентите за да разучат
 - приложната област,
 - услугите, които системата трябва да предоставя, както и
 - оперативните ограничения.
- Етапът включва крайни потребители, менажери, софтуерни инженери, инженери по поддръжката, експерти в областта и др. Всички те се наричат **зainteresовани лица** (stackholders).

Идентифициране и анализ: Откриване на изискванията

- Процесът на събиране на информация за предлаганата система и извлечането на потребителските и системни изисквания от тази информация.
- Ресурсите на информация включват
 - документи
 - системните заинтересовани лица, както и
 - спецификациите на подобни системи
 - законодателство, касаещо приложната област

Пример: Заинтересовани лица на софтуер за банкомат

- Клиентите на банката
- Представители на други банки
- Менажерите на банката
- Персоналът на банката
- Администраторите на базата данни
- Менажери по сигурността
- Отдел по маркетинг
- Софтуерните инженери, отговарящи за софтуерната и хардуерна поддръжка.
- Специалисти по банково законодателство

Проблеми при анализа на изискванията

- Заинтересованите лица не са наясно какво точно искат.
- Заинтересованите лица описват изискванията според собствените си разбирания.
- Различните заинтересованите лица могат да имат противоречащи си изисквания.
- Организационни и политически фактори могат да повлият на системните изисквания.
- Изискванията се променят по време на анализа. Могат да се появят нови заинтересованите лица или да се промени бизнес средата.

Гледни точки

- Гледните точки са начин да се структурират изискванията за да се отразят разбиранията на различната заинтересовани лица.
- Заинтересованите лица могат да бъдат класифицирани според различните гледни точки.
- Този многостррен анализ е важен, тъй като няма единствен верен начин да се анализират системните изисквания.

Пример: Йерархия на гледните точки на библиотечната система



Социални и организационни фактори

- Софтуерните системи се използват в социален и организационен контекст. Това може да повлияе и дори да е доминиращо при определяне на системните изисквания.
- Социалните и организационни фактори са повлияни от множество такива.
- Добрият анализ трябва да е чувствителен към тези фактори, но за момента няма систематичен начин да бъде направено това.

Основни техники за извличане на изискванията

1) Интервюта

- Във формални или неформални интервюта екипът по определяне на изискванията задава въпроси на заинтересованите лица за системата, която използват и за тази, която ще бъде разработена.
- Има два вида интервюта:
 - **Затворени интервюта**, при които се задават предварително дефинирани въпроси.
 - **Отворени интервюта**, при които няма предефиниран план и се провежда отворена дискусия.

Интервюта в практиката

- Най-често смесени затворени и отворени интервюта
- Интервютата са добра практика за придобиване на цялостна представа за това какво правят заинтересованите лица и как ще си взаимодействат със системата.
- **Интервютата не са удачни** за разбиране на изискванията на приложната област.
 - Софтуерните инженери не могат да разберат специфичната терминология на приложната област;
 - Някои знания за приложната област изглеждат толкова естествени, че за хората е трудно да ги осмислят и дефинират като конкретни изисквания.

Ефективни интервюиращи

- Интервюиращите трябва да са много широкомислещи, имащи желание да слушат и да нямат предварително дефинирани идеи за изискванията.
- Те трябва да предразполагат интервюираните чрез въпроси или предложения, без да очакват отговори на въпроси като: “Какво искате?”.

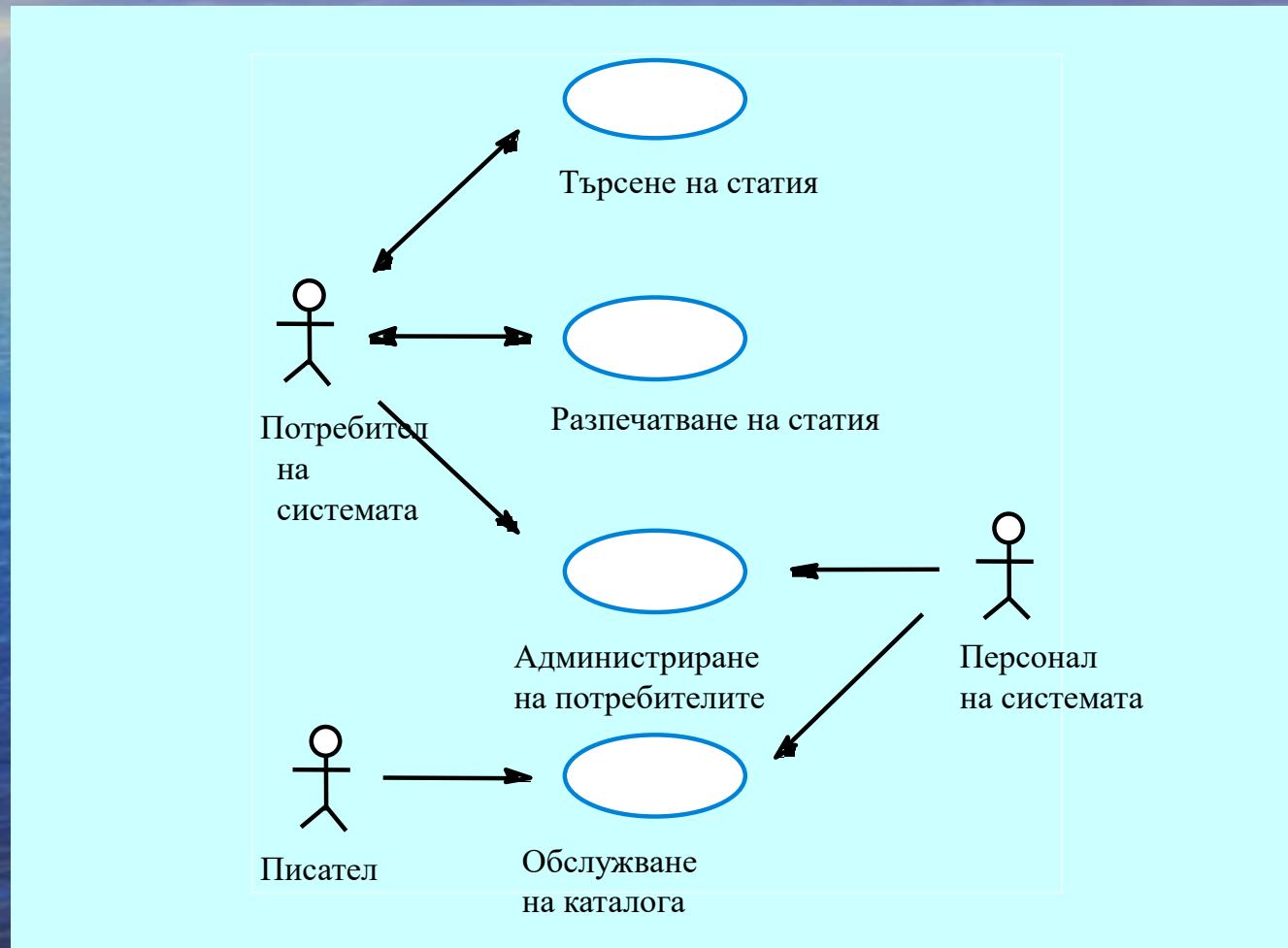
2. Сценарии

- Сценариите са примери от реалния живот за използването на системата.
- Те трябва да включват:
 - Описание на началната ситуация;
 - Описание на нормалния ход на събитията;
 - Описание на това, което може да се обърка;
 - Информация за други конкурентни дейности;
 - Описание на състоянието, с което приключва сценария.

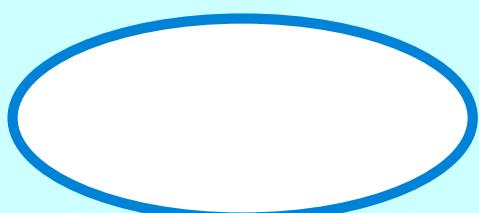
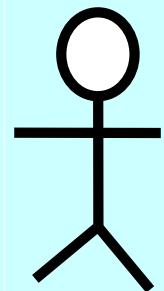
3. Потребителски случаи

- Потребителските случаи (Use-cases) са UML техника, базирана на сценарии, която идентифицира акторите във взаимодействие със системата и описват самите взаимодействия.
- Множеството от потребителски случаи трябва да опише всички възможни взаимодействия със системата.
- Диаграми на последователност (sequence diagrams) могат да бъдат използвани за да детайлизират потребителските случаи, като покажат последователността на обработване на събитията в модули на системата.

Пример: Потребителски случаи на библиотечната система

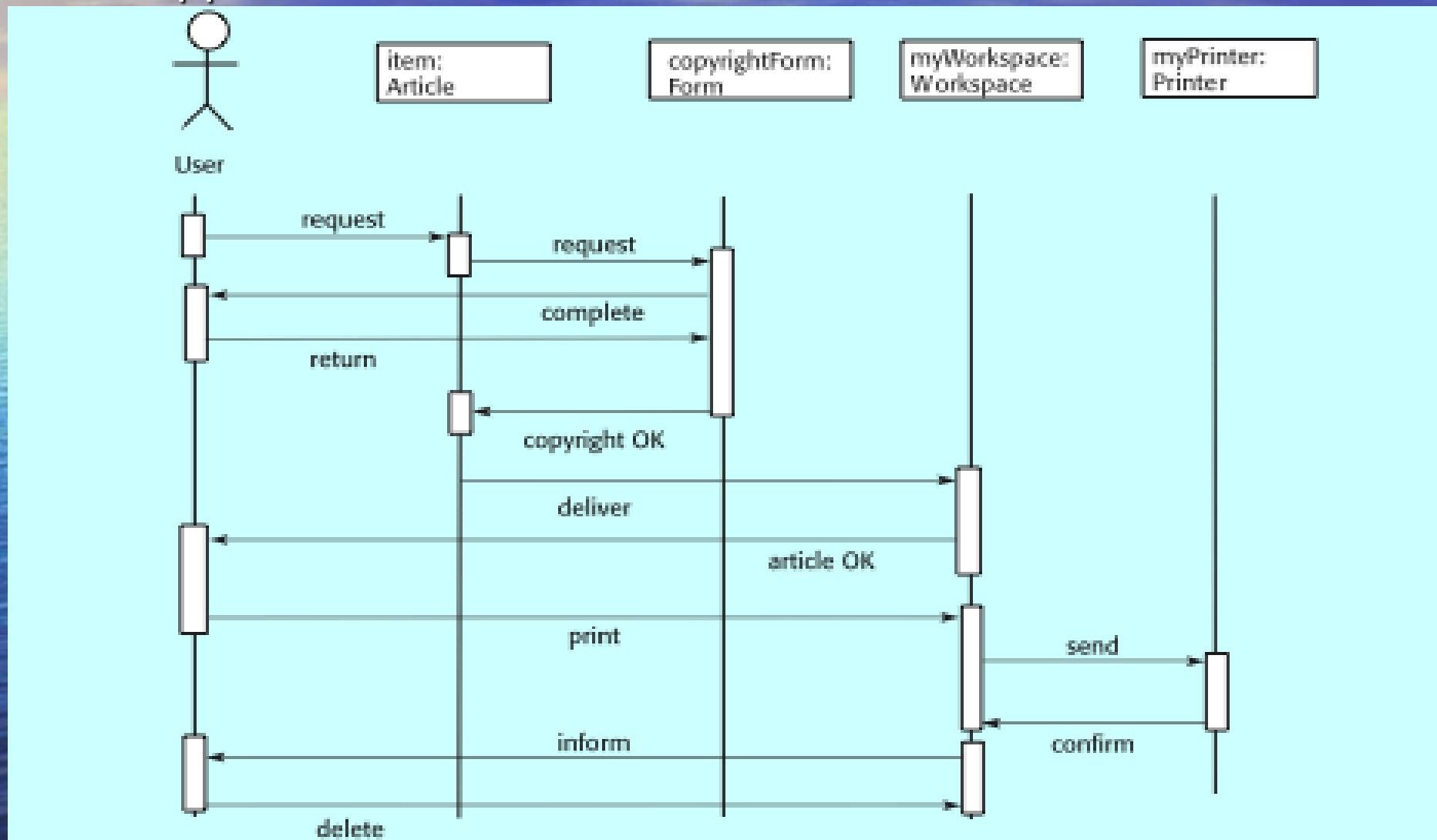


Пример: Потребителски случай на разпечатване на статия



Разпечатване
на статия

Пример: Разпечатване на статия- диаграма на последователностите



4. Етнография

- Социалният изследовател отделя много време да наблюдава и анализира поведението на хората по време на работа.
- Не се налага хората да обясняват и анализират начина си на работа.
- Могат да бъдат взети предвид и други важни социални и организационни фактори.
- Етнографските изследвания са показвали, че най-често работата е много по-сложна отколкото предполагат опростените системните модели.

Анализ на изискванията

- Проверка на извлечените изисквания дали правилно, точно, пълно, атомарно описват изискванията на клиента.
- Приоритизация и догваряне
 - Приоритизиране на изискванията и разрешаване на конфликтите.

Валидиране на изискванията

- Целта е да се демонстрира, че изискванията определят системата, която клиентът наистина иска.
- Грешките при определяне на изискванията струват много скъпо и за това валидирането е особено важно.
 - Поправяне на грешка допусната при определяне на изискванията след доставянето на системата може да струва до 100 пъти повече, отколкото поправянето на грешка от имплементацията.

Проверка на изискванията

- **Валидност.** Дали системата наистина предоставя функциите , които да отговорят най-добре на нуждите на клиента?
- **Консистентност.** Има ли противоречиви изисквания?
- **Пълнота.** Включени ли са всички функции, изискани от клиента?
- **Реалистичност.** Могат ли изискванията да бъдат реализирани предвид наличните бюджет и технологии?
- **Верифицируемост.** Могат ли изискванията да бъдат променени?

Техники за валидиране на изискванията

- Ревюта на изискванията
 - Систематичен ръчен анализ на изискванията.
- Прототипиране
 - Използване на работещ модел на системата, за да се проверят изискванията.
- Генериране на тестови случаи
 - Разработване на тестове за изискванията за да се провери тяхната тестабилност.

Ревюта на изискванията

- Периодични ревюта трябва да бъдат провеждани, при процеса на дефиниране на изискванията.
- В ревютата трябва да бъдат включени хора както от страна на клиента, така и специалисти от страна на разработващата организация.
- Ревютата могат да бъдат формални (с попълване на документация) или неформални.
- Добрата комуникация между разработчиците, клиентите и потребителите може да разреши проблеми в най-ранните фази на разработване.

Ревюто проверява

- **Верифицируемост.** Дали изискванията могат да бъдат реалистично верифицирани/ тествани?
- **Понятност.** Дали изискването е правилно разбрано?
- **Проследимост.** Дали произхода на изискването е ясно дефиниран?
- **Адаптивност.** Може ли изискването да бъде променено, без това да повлияе силно на останалите изисквания?

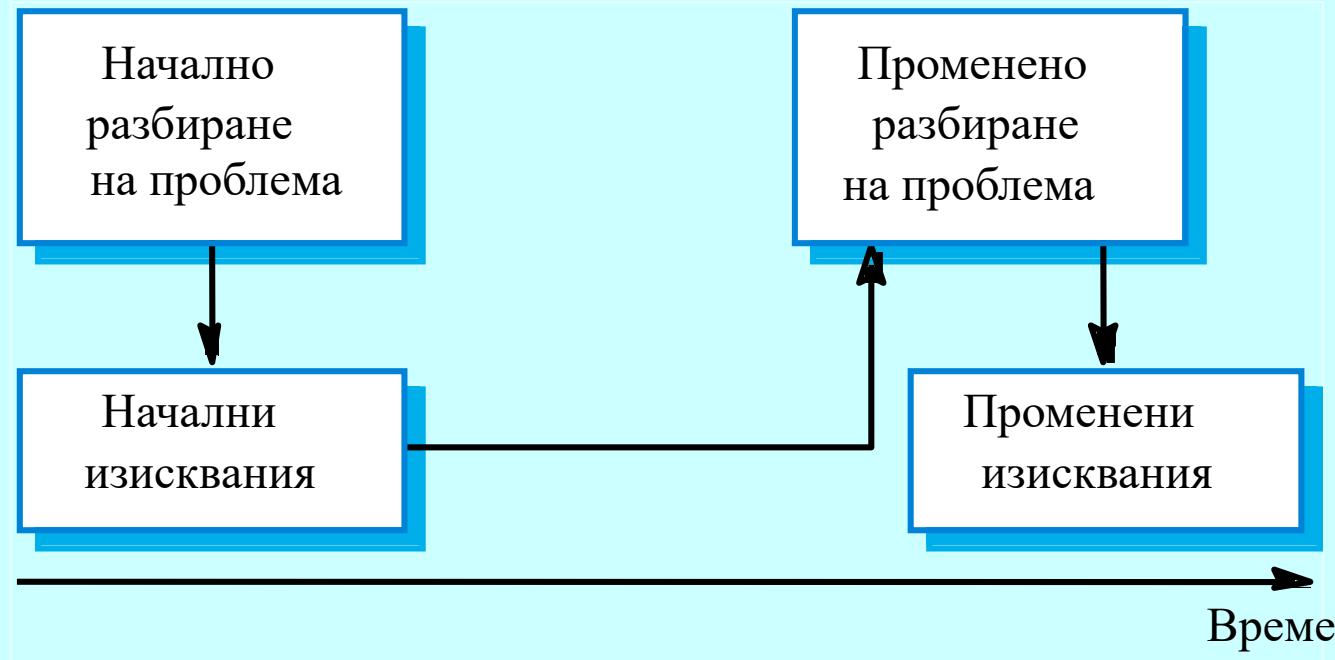
Управление на изискванията

- Управлението на изискванията е процес на управление на променящите се изисквания по време на определянето на изискванията и системното разработване
- Изискванията са неизбежно непълни и противоречиви
 - Нови изисквания възникват по време на процеса, тъй като се **променят бизнес нуждите** и се придобива **по-добро разбиране за системата**;
 - Различните гледни точки имат различни изисквания, много често противоречиви.

Изискванията се променят

- Повечето от изискванията, определени от различните гледни точки се променят по време на процеса на разработване.
- Клиентите на системата могат да определят изисквания от бизнес гледна точка, които да са в конфликт с тези на крайните потребители.
- Бизнес и технологичното обкръжение на системата могат да се променят по време на разработването и.

Еволюиране на изискванията



Постоянни и променящи се изисквания

Постоянни изисквания. Стабилни изисквания произтичащи от основни дейности в организацията на клиента, от приложната област.

Например – в болницата винаги има лекари, сестри и т.н.

Променливи изисквания. Изисквания, които се променят по време на разработването или по време на използване на системата.

В болницата – изисквания произтичащи от здравната система.

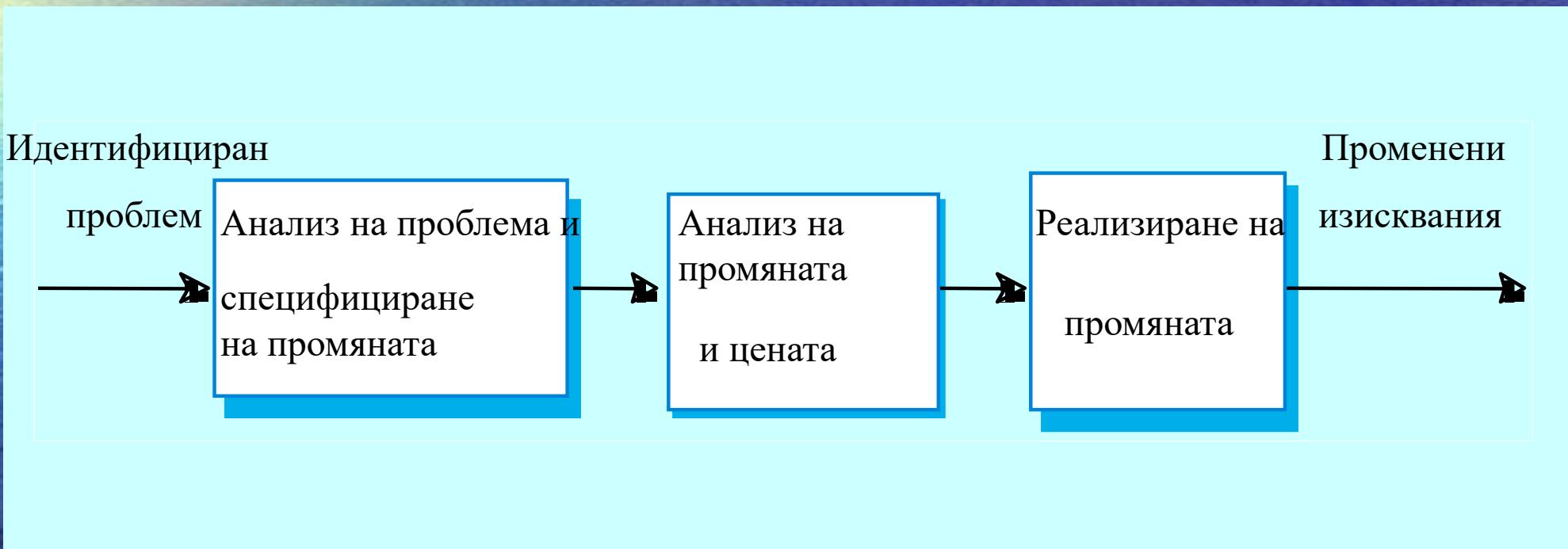
Класификация на изискванията

| Тип на изискването | Описание |
|-------------------------|---|
| Мутации изисквания | Изисквания, които се променят, поради помени в средата, в която работи организацията. |
| Появяващи се изисквания | Изисквания, които се появяват въледствие на по-доброто разбиране за системата, което клиентът придобива по време на системното разработване. Дизайн процесът може да разкрие нови изисквания. |
| Последстващи изисквания | Изисквания, произтичащи от въвеждането на компютърната система – това може да промени процесите в организацията и да открие нови начини на работа, които да генерират нови изисквания. |
| Съвместими изисквания | Изисквания, които зависят от конкретна система или бизнес процес в организацията. Ако те се променят, възникват изисквания за съвместимост. |

Планиране на управлението на изискванията

- По време на процеса на специфициране на изискванията трябва да се планира:
 - Идентифицирането на изискванията
 - Как изискванията са индивидуално идентифицирани;
 - Процеса на управление на промените при изискванията
 - Процесът, които следва след като се анализира промяна в изискванията;
 - Политиката на проследяемост
 - Обемът информация за връзките между изискванията, които трябва да се поддъжат;
 - Използване на CASE системи
 - Необходимо е използването на системи, които да подпомогнат управлението на промени при изискванията;

Процес на управление на промените



Управление на промените при изискванията

- Трябва да бъде приложено за всички предложени промени на изискванията.
- **Основни стъпки**
 - Анализ на проблема. Дискутиране на проблема с изискванията и предложение за промяна;
 - Анализ на промените и цената им. Оценка на влиянието на промяната върху други изисквания;
 - Реализиране на промяната. Промяна на документа с изискванията и другите документи за да се отрази промяната.

Обобщение

- Системите имат много заинтересовани лица с различни изисквания.
- Изискванията се влияят от социални и организационни фактори.
- Процесът на инженеринг на изискванията включва
 - проучване за осъществимост,
 - идентифициране и анализ на изискванията,
 - специфициране на изискванията,
 - валидиране на изискванията и
 - управление на изискванията.
- Идентифицирането и анализът на изискванията са итеративни и включват
 - разбиране за приложната област,
 - събиране на изискванията,
 - класифициране, структуриране, приоритизиране и валидиране.

Обобщение 2

- Валидирането на изискванията цели да провери дали изискванията са валидни, консистентни, пълни, реалистични и верифицируеми.
- Промените в бизнес средата водят до промени в изискванията.
- Управлението на изискванията включва планиране и управление на промените.



Софтуерни метрики и формални модели на софтуерни системи



Agenda/Съдържание

- ▶ Софтуерни метрики
- ▶ Понятие за качество на софтуерните системи
 - ▶ Измерване на качеството
- ▶ Формални модели

Софтуерни метрики

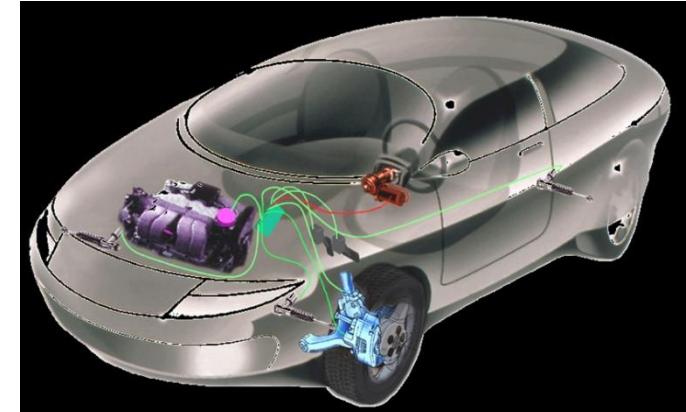
- ▶ Показател за това доколко дадена софтуерна система или процес притежава някакво свойство.
- ▶ Какво обикновено се измерва за софтуера?
 - ▶ Функционалност?
 - ▶ Нещо друго?

Нефункционални изисквания

- ▶ Функционалните изисквания определят **какво** трябва да прави софтуерната система
- ▶ Нефункционалните изисквания определят **как** софтуерната система да работи
 - ▶ На практика качествата поставят ограничения върху начина по който системата ще се изпълнява

Нефункционални изисквания

- ▶ Доста често ги наричаме и качествени изисквания
- ▶ Други имена:
 - ▶ Нефункционани свойства/атрибути (Properties/Attributes)
 - ▶ Ограничения (constraints)
 - ▶ “-ilities”
 - ▶ Performance
 - ▶ Reliability
 - ▶ Availability
 - ▶ Modifiability
 - ▶ Usability
 - ▶ Testability
 - ▶ Survivability
 - ▶ Maintainability
 - ▶ Accessibility
 - ▶ Etc.





Как се измерва качеството на софтуерните
системи?

Мерни единици за наличност (availability)

- ▶ Наличността на системата се дефинира като вероятността тя да бъде в изправност, когато има нужда от нея
- ▶ Може да се представи като:

$$\alpha = \frac{\Delta t_f}{\Delta t_f + \Delta t_c},$$

- ▶ Където Δt_f е средното време между отказите, а Δt_c е средното време за отстраняване на повредата

Интерпретация на наличността

- ▶ Наличността обикновено се изразява като процент от времето, през което системата е достъпна за предоставяне на услуги (напр. 99,95%).
- ▶ Това обаче не отчита два фактора:
 - ▶ Броят на потребителите, засегнати от прекъсването на услугата. (Напр. загуба на достъп нощно време може да не е от значение за редица системи, за разлика от загубата на услуга по време на периоди на пикова употреба).
 - ▶ Продължителността на прекъсването. (Напр. няколко кратки прекъсвания е по-малко вероятно да бъдат разрушителни от 1 дълго прекъсване.)

Availability vs. Reliability

- ▶ Наличност и надеждност са различни понятия.
- ▶ Дадена система може да има висока степен на наличност и в същото време да е ненадеждна и обратно
 - ▶ Ако системата отказва редовно - за по 1 милисекунда на всеки час, то тя има наличност от над 99,9999 процента, но е много ненадеждна.
 - ▶ Система, която "никога" не отказва, но е недостъпна за две седмици всеки август (например за профилактика), има висока надеждност, но само ~96 процента наличност

Мерни единици за производителност (Performance)

- ▶ Количествено, резултатите в сценариите за производителност могат да се характеризират чрез различни параметри, напр.:
 - ▶ Латентност (latency) – времето между пристигането на заявката и обработката ѝ;
 - ▶ Времева граница (Навременност) (deadline) – максимално време за наблюдавана реакция на системата;
 - ▶ Отклонение (jitter) – вариация в латентността;
 - ▶ Пропускливост (throughput) – броя заявки, които системата може да обработи за определен интервал от време;
 - ▶ Брой на необработените заявки (поради претовареност).

Мерни единици за сигурност

- ▶ Време/усилия/ресурси, необходими за заобикаляне на сигурността с вероятност за успех;
- ▶ Вероятност за разкриване на атаката
- ▶ Вероятност за разкриване на самоличността на извършителите
- ▶ Процент на работоспособност (напр. при DoS)
- ▶ Обхват на пораженията

Мерни единици за сигурност

- ▶ A list of popular software vulnerabilities and their measures, together with scoring of vulnerabilities are used, both represented by industry standards
- ▶ System security is given by:

$$Sec = \sum_n (P_n \times W_n),$$

where W_n is the weight of the n-th software weakness, which is defined by the vulnerabilities that activate it and P_n represents the risk of the corresponding weakness.

Testability

- ▶ Някои от най-популярните метрики се основават на различни обектно ориентирани характеристики
 - ▶ Дълбочина на дървото на наследяване (Depth of Inheritance Tree – DIT) – размерът на най-дългия път в дървото на наследяване на клас
 - ▶ Number of children – броят класове, които наследяват даден клас
 - ▶ Number of methods – броят на методите, които предоставя даден клас
 - ▶ Брой полета

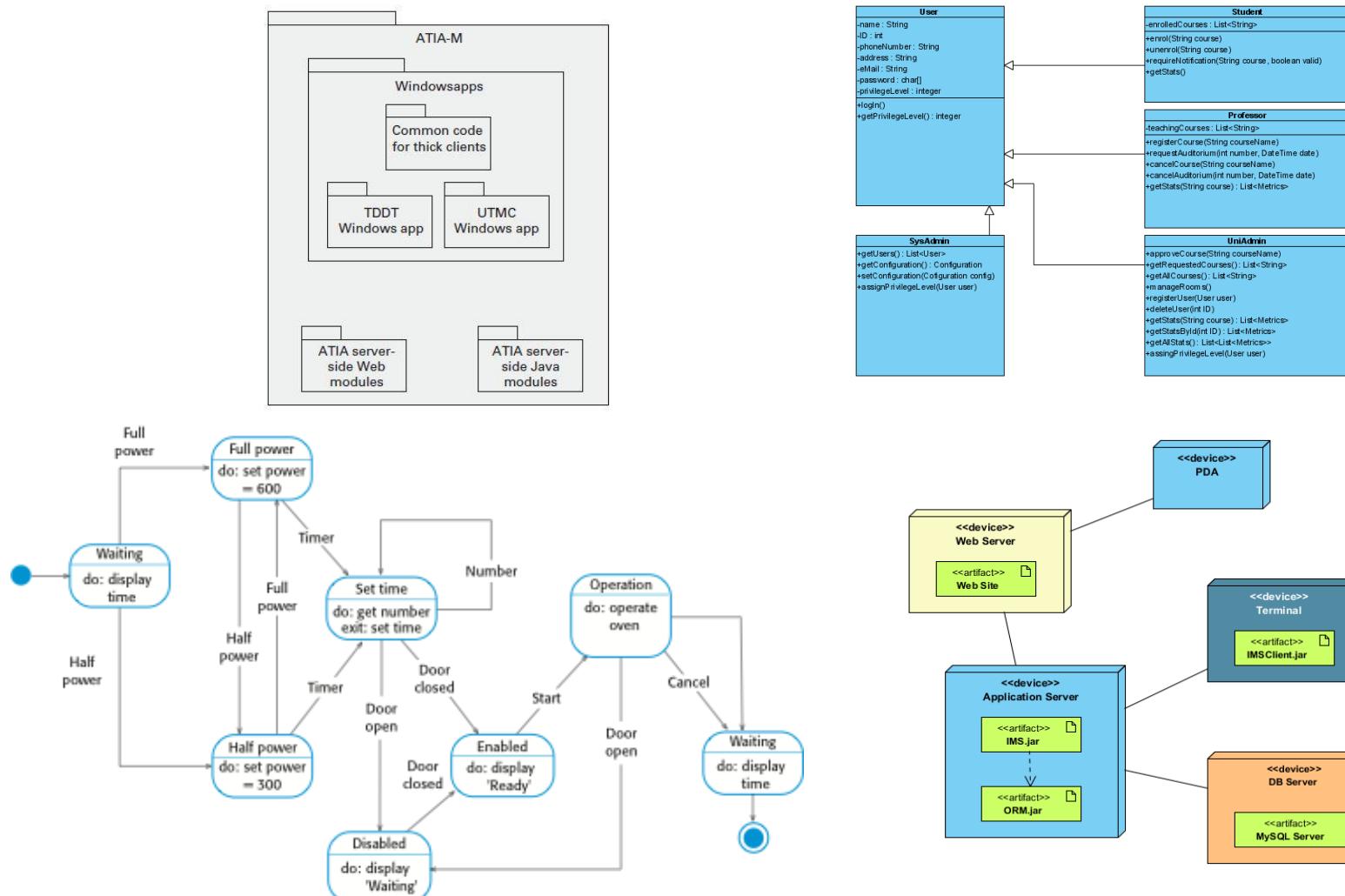
Мерни единици за възможност за промяна (modifyability)

- ▶ Стойност
 - ▶ Например – брой променени елементи, усилие, пари
- ▶ Доколко промяната се отразява на останалата функционалност и свойства
- ▶ Различни метрики за комплексност
 - ▶ Cyclomatic complexity
 - ▶ Halstead complexity



Формални модели в софтуерното инженерство

Какво разбираме под модел на софтуерна система?



Модели на софтуерните системи

- ▶ Достатъчен ли е UML?
- ▶ Може ли да се моделира семантиката на софтуера с UML?
- ▶ Може ли да се моделират нефункционални характеристики с UML?

Формални методи

- ▶ Формалното моделиране представлява част от по-общ набор от технологии, познати като “формални методи”
- ▶ Базират се на математическо представяне и анализ на софтуера
- ▶ Формалните методи в софтуера включват
 - ▶ Формално моделиране (спецификация)
 - ▶ Анализ и доказателство на спецификацията
 - ▶ Разработка, базирана на трансформации (Transformational development)
 - ▶ Програмна верификация

Необходимост от формални методи в софтуерното инженерство

- ▶ Формален анализ на софтуерни системи
 - ▶ Динамични архитектури
 - ▶ Генерация на код
-
- ▶ Всички те обаче са неприложими без съответните инструменти (*tool support*)

Формални модели на софтуерните системи

- ▶ CSP (communicating sequential processes)
- ▶ π – calculus
- ▶ Z schema
- ▶ Мрежи на Петри
- ▶ И т.н.

Различни ADLs

- ▶ MetaH
- ▶ Rapide
- ▶ UniCon
- ▶ Darwin
- ▶ Wright
 - ▶ Базира се на CSP за формално описание
- ▶ xADL
- ▶ ACME
- ▶ AADL

Примерен модел на архитектурата с EOA Wright

System Example

Component CompA

 Port provide [provide protocol]
 Spec [ComponentA specification]

Component CompB

 Port request [request protocol]
 Spec [ComponentB specification]

Connector A B connector

 Role CompA-[ComponentA protocol]
 Role CompB [ComponentB protocol]
 Glue [glue protocol]

Instances

 a: CompA;
 b: CompB;
 ab: A B connector;

Attachments

 a.provide as ab.CompA;
 b.request as ab.CompB;

end Example

Клиент-сървър с Wright

```
System SimpleExample
    component Server =
        port provide [provide protocol]
        spec [Server specification]
    component Client =
        port request [request protocol]
        spec [Client specification]
    connector C-S-connector =
        role client [client protocol]
        role server [server protocol]
        glue [glue protocol]
Instances
    s: Server
    c: Client
    cs: C-S-connector
Attachments
    s.provide as cs.server;
    c.request as cs.client
end SimpleExample.
```

Модел на connector с Wright

```
connector C-S-connector =  
    role Client = (request!x→ result?y → Client) ⋱ §  
    role Server = (invoke?x→ return!y → Server) ⋱ §  
    glue = (Client.request?x→ Service.invoke!x →Service.return?y→Client.result!y→glue)  
    ⋱ §
```

- ▶ С въпросителен знак се маркират входни данни
- ▶ С удивителен знак се маркират изходни данни

Примерно описание на СМ – общо поле за данни

```
connector Shared Data1 =  
    role User1 = set→User1 ∩ get→User1 ∩ §  
    role User2 = set→User2 ∩ get→User2 ∩ §  
    glue = User1.set→glue ∏ User2.set→glue  
        ∏ User1.get→glue ∏ User2.get→glue ∏ §
```

Примерно описание на СМ – общо поле за данни (2)

В този модел съществува компонент, който инициализира полето

```
connector Shared Data2 =
    role Initializer =
        let A = set→A ∏ get→A ∏ §
            in set→A
        role User = set→User ∏ get→User ∏ §
        glue = let Continue = Initializer.set→Continue
                User.set→Continue
                Initializer.get→Continue
                User.get→Continue §
            in Initializer.set→Continue §
```

Примерно описание на СМ – общо поле за данни (3)

Този модел е подобен на предишния, но не е задължително полето да се инициализира от компонента `Initializer`

```
connector Shared Data3 =  
    role Initializer =  
        let A = set→A ∏ get→A ∏ §  
        in set→A  
    role User = set→User ∏ get→User ∏ §  
    glue = let Continue = Initializer.set→Continue  
            █ User.set→Continue  
            █ Initializer.get→Continue  
            █ User.get→Continue █ §  
        in Initializer.set→Continue  
            █ User.set→Continue █ §
```

Fault-tolerant pool от сървъри – описание с Wright

Component Client

Port Primary = $\$ \sqcap ((\overline{\text{request}} \rightarrow \text{reply} \rightarrow \text{Primary}) \sqcup \text{down} \rightarrow (\$ \sqcap \text{up} \rightarrow \text{Primary}))$

Port Secondary = $\$ \sqcap \overline{\text{request}} \rightarrow \text{reply} \rightarrow \text{Secondary}$

Computation = UsePrimary

where UsePrimary = $\overline{\text{internalCompute}} \rightarrow (\text{TryPrimary} \sqcap \$)$

TryPrimary = $\overline{\text{primary.request}} \rightarrow \text{primary.reply} \rightarrow \text{UsePrimary}$

$\sqcup \text{primary.down} \rightarrow \text{TrySecondary}$

UseSecondary = $\overline{\text{internalCompute}} \rightarrow (\text{TrySecondary} \sqcap \$)$

TrySecondary = $\overline{\text{secondary.request}} \rightarrow \text{secondary.reply} \rightarrow \text{UseSecondary}$

$\sqcup \text{primary.up} \rightarrow \text{TryPrimary}$

Z

- ▶ Z-нотация (notation)
- ▶ Разработена в края на 80-те години
- ▶ Първоначално не е била предназначена за описание на софтуерни системи
- ▶ Комбинира формално и неформално описание и има възможност за графичен модел

Пример

- ▶ Система, която документира имена на хора и дати
- ▶ Birthday book
- ▶ [NAME;DATE]

BirthdayBook

known : P NAME

birthday : NAME → DATE

known = dom birthday

Birthday book

AddBirthday _____

$\Delta \text{BirthdayBook}$
 $\text{name?} : \text{NAME}$
 $\text{date?} : \text{DATE}$

$\text{name?} \notin \text{known}$

$\text{birthday}' = \text{birthday} \cup \{\text{name?} \mapsto \text{date?}\}$

FindBirthday _____

$\exists \text{BirthdayBook}$
 $\text{name?} : \text{NAME}$
 $\text{date!} : \text{DATE}$

$\text{name?} \in \text{known}$

$\text{date!} = \text{birthday}(\text{name?})$

Birthday book

Remind _____

$\exists \text{BirthdayBook}$
 $\text{today?} : \text{DATE}$
 $\text{cards!} : \mathbb{P} \text{ NAME}$

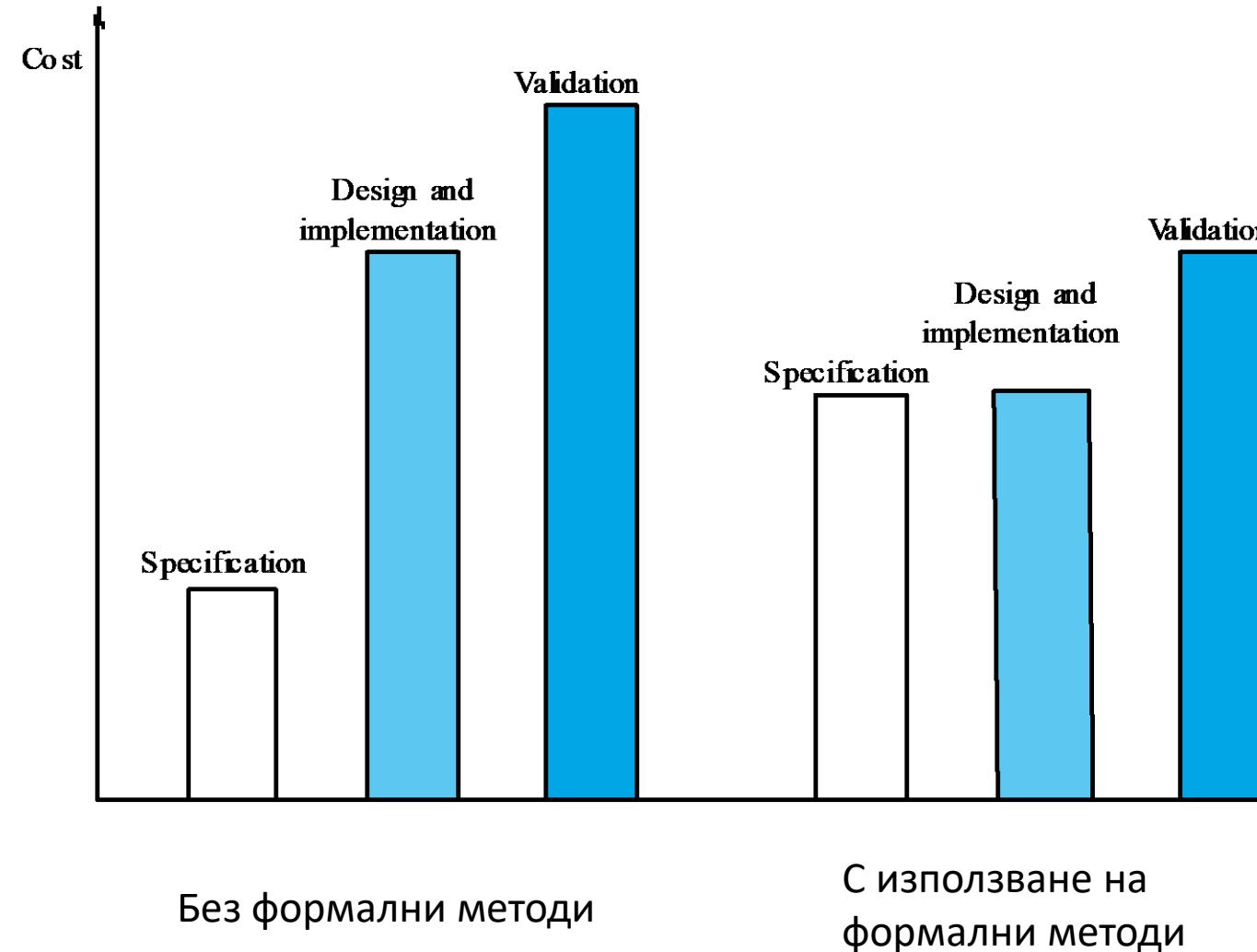
$\text{cards!} = \{ n : \text{known} \mid \text{birthday}(n) = \text{today?} \}$

AlreadyKnown _____

$\exists \text{BirthdayBook}$
 $\text{name?} : \text{NAME}$
 $\text{result!} : \text{REPORT}$

$\text{name?} \in \text{known}$
 $\text{result!} = \text{already_known}$

Разходи за разработката



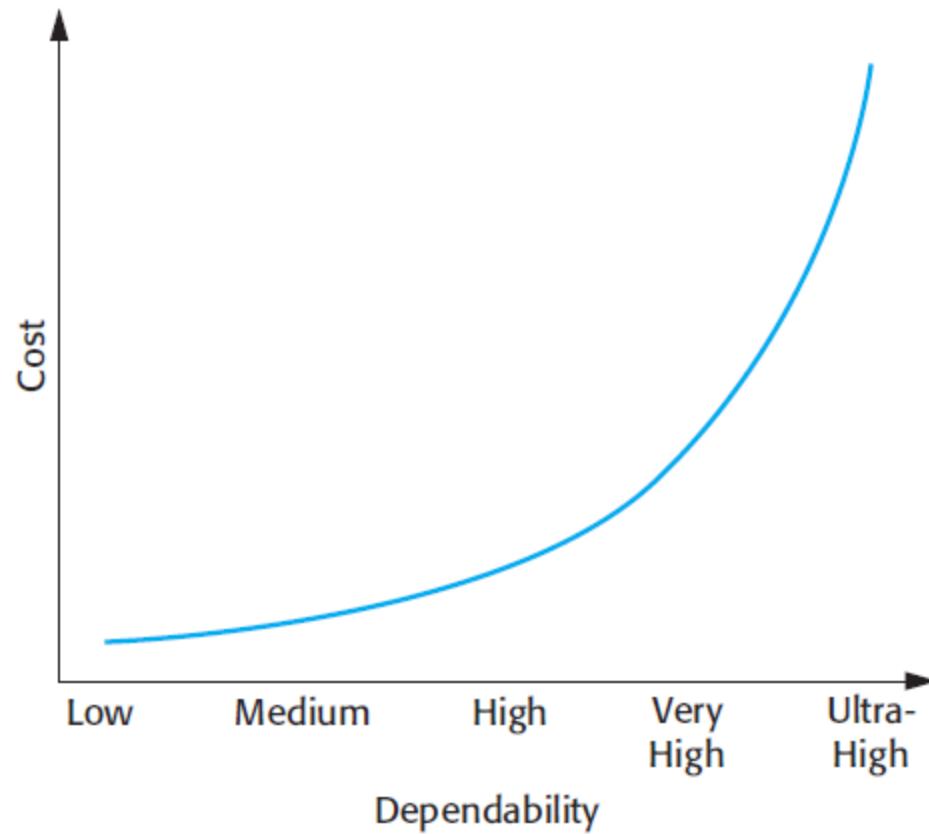
Минуси на формалните методи

- ▶ Ефективността на неформалните технологии в СЕ в редица проблемни области, намалява необходимостта от формални методи
- ▶ Формалните методи не намаляват времето за разработка (time-to-market)
- ▶ Не са подходящи за описание на потребителски интерфейс и взаимодействие
- ▶ Проблеми с приложението в големи и развиващи се системи

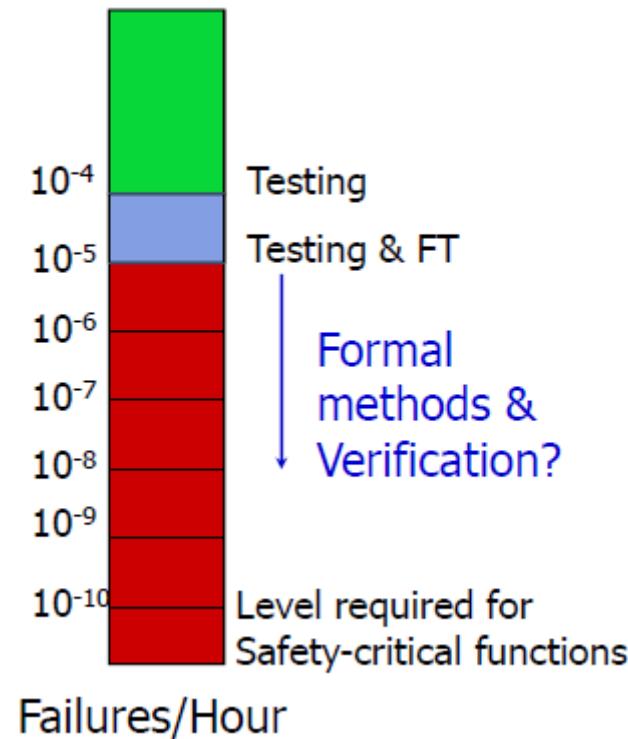


Формални модели на качествените изисквания

Cost of dependability/reliability



Attainable levels of SW reliability



- ▶ FAA (Federal Aviation Administration) & NASA safety=critical requirement is less than 10^{-10} failures per 10 hrs of flight.

The problem with testing

- ▶ Reliability of life-critical and real-time software is infeasible to be quantified by testing [Butler & Finelli 1993]
- ▶ Only small reliabilities (99,999%) are possible to be estimated in a obtainable period of time for testing
- ▶ For example assuring that a program has failure rate of about 10^{-7} per hour may require thousands (and even more) years of testing
- ▶ There may not exist effective oracle to carry out statistical testing

Софтуерна надеждност

- ▶ Леко се различава от традиционната математическа теория на надеждността
 - ▶ Отказите са детерминистични събития, поведението на потребителя не е
 - ▶ Дефинира се като вероятност за отказ в рамките на даден времеви интервал
 - ▶ На практика се използва като индикатор за това колко тестване на системата е достатъчно



Как измерваме надеждността на софтуер

► Има различни явни метрики

- ▶ Probability of failure (success)
- ▶ Mean time to failure μ
- ▶ Failure rate λ
 - ▶ $\lambda = 1/\mu$

► Неявни метрики

- ▶ Брой редове код
- ▶ Test coverage
- ▶ Други...

-
- ▶ Измерването на надеждността на софтуер е трудно, поради сложната природа на софтуерните продукти, която все още остава неразбрана (недефинирана)
 - ▶ Дори метрики, които би трябвало да са очевидни, като размер на софтуера, все още нямат общоприета дефиниция
 - ▶ Отказите на софтуера, за разлика от тези на хардуера са детерминистично явление, поведението на потребителя не е детерминистично
 - ▶ Трудно е да се оцени надеждността на компонентно-базирана система, поради сложността на възможните взаимодействия между съставящите я компоненти

Методи за оценка на надеждността

► Модели от тип черна кутия

- ▶ Основани са на статистическа обработка на данни, най-често от тестване на системите
- ▶ Software Reliability Growth Models (SRGMs)

► Модели от тип бяла кутия

- ▶ Базират се на данни за вътрешната организация на системата, напр. - архитектурата
- ▶ Формализират процеса на отказите на компонентите в системата
- ▶ Интегрира се поведението на отказите със модел на архитектуата



Методи за оценка на надеждността

- ▶ Методите на бялата кутия не изключват тези на черната
- ▶ Съществува значителен проблем с гарантирането на много висока надеждност $\sim (1-10^{-9})$ само чрез тестване



Изчисляване чрез данни от тестването

- ▶ Доказано е, че е неподходящо надеждност на критични системи да се изчислява само чрез тестване [Butler & Finelli 1993]
- ▶ Само малки надеждности (99,999%) може да се изчислят на базата на данни, придобити от разумно дълъг период на тестване
- ▶ Например, за да сме сигурни че дадена система има надеждност от порядъка на ($1-10^{-7}$), трябва да тестваме хиляди години без прекъсване

Модели на черната кутия

- ▶ Разработени са десетки такива модели
- ▶ Те се основават на данни от тестването на софтуер, като време между отказите, брой откази и т.н.
- ▶ Правят се редица опростяващи допускания, които влошават качеството на оценката за надеждност

Модели на черната кутия

- ▶ Допуска се че отказите се подчиняват на някакво статистическо разпределение
 - ▶ Експоненциално – модел на Jelinski-Moranda
 - ▶ Поасново (Nonhomogeneous Poisson process – NHPP) – модели на Musa

Базов модел за надеждност на Муса

- ▶ Честотата на отказите на системата е:

$$\lambda(\mu) = \lambda_0 \left[1 - \frac{\mu}{v_0} \right]$$

- ▶ Където:

- ▶ λ_0 е началната честота на отказите в началото на тестването
- ▶ μ е средния (очакван) брой откази на системата в даден момент от време τ :

$$\mu(\tau) = v_0 \left[1 - \exp \left[-\frac{\lambda_0}{v_0} \tau \right] \right]$$

- ▶ v_0 е общия брой бъгове в системата

Пример

Нека допуснем, че в даден софтуер има общо 100 бъга и началната честота на откази е 10 отказа/час. В настоящия момент на тестване са регистрирани 50 отказа.

Тогава честотата на отказите е:

$$\lambda(\mu) = \lambda_0 \left[1 - \frac{\mu}{\nu_0} \right] = 10 \left[1 - \frac{50}{100} \right] = 5 \text{ отказа/час}$$

Броят откази след 10 часа е:

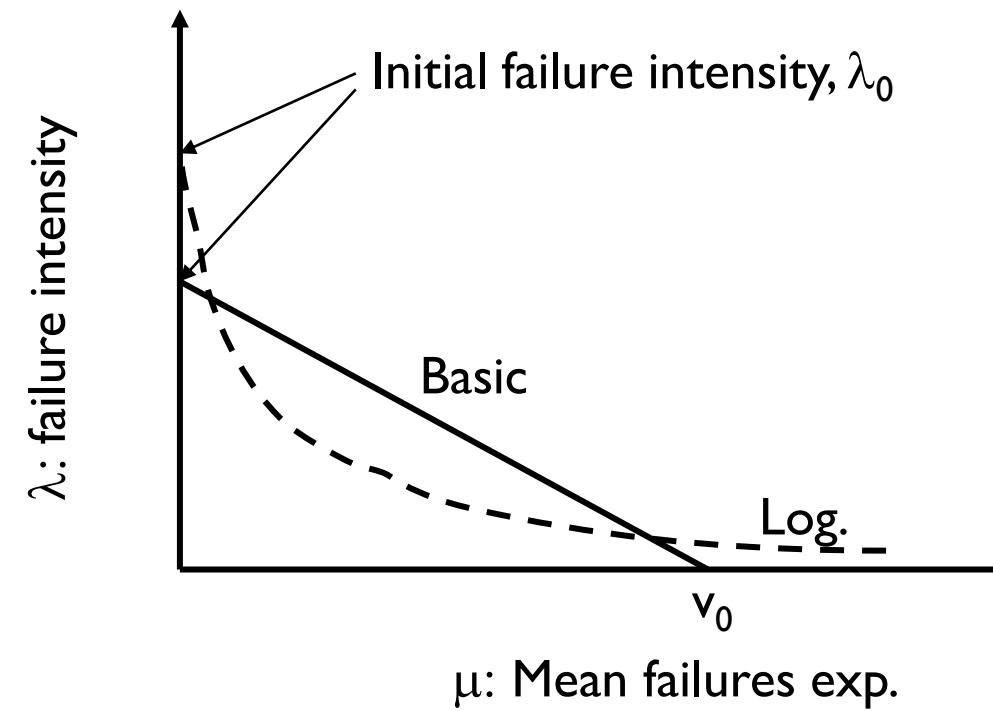
$$\mu(\tau) = 100 \left(1 - \exp \left[-\frac{10}{100} (10) \right] \right) = 100 [1 - \exp(-1)] = 63$$

Съответно след 100 часа:

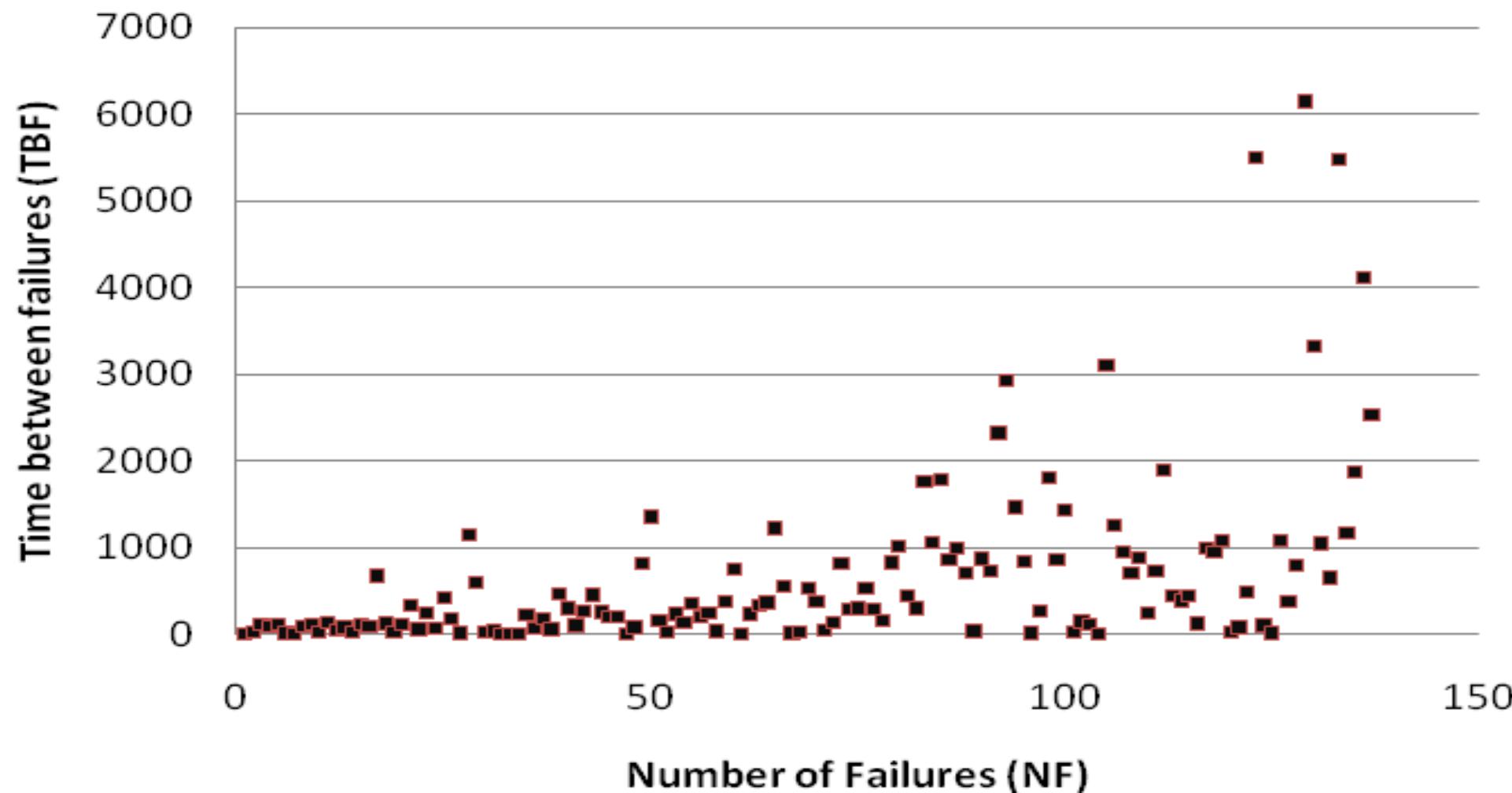
$$\mu(\tau) = 100 \left(1 - \exp \left[-\frac{10}{100} (100) \right] \right) = 100 [1 - \exp(-10)] = 100$$

Failure Intensity Reduction Concept

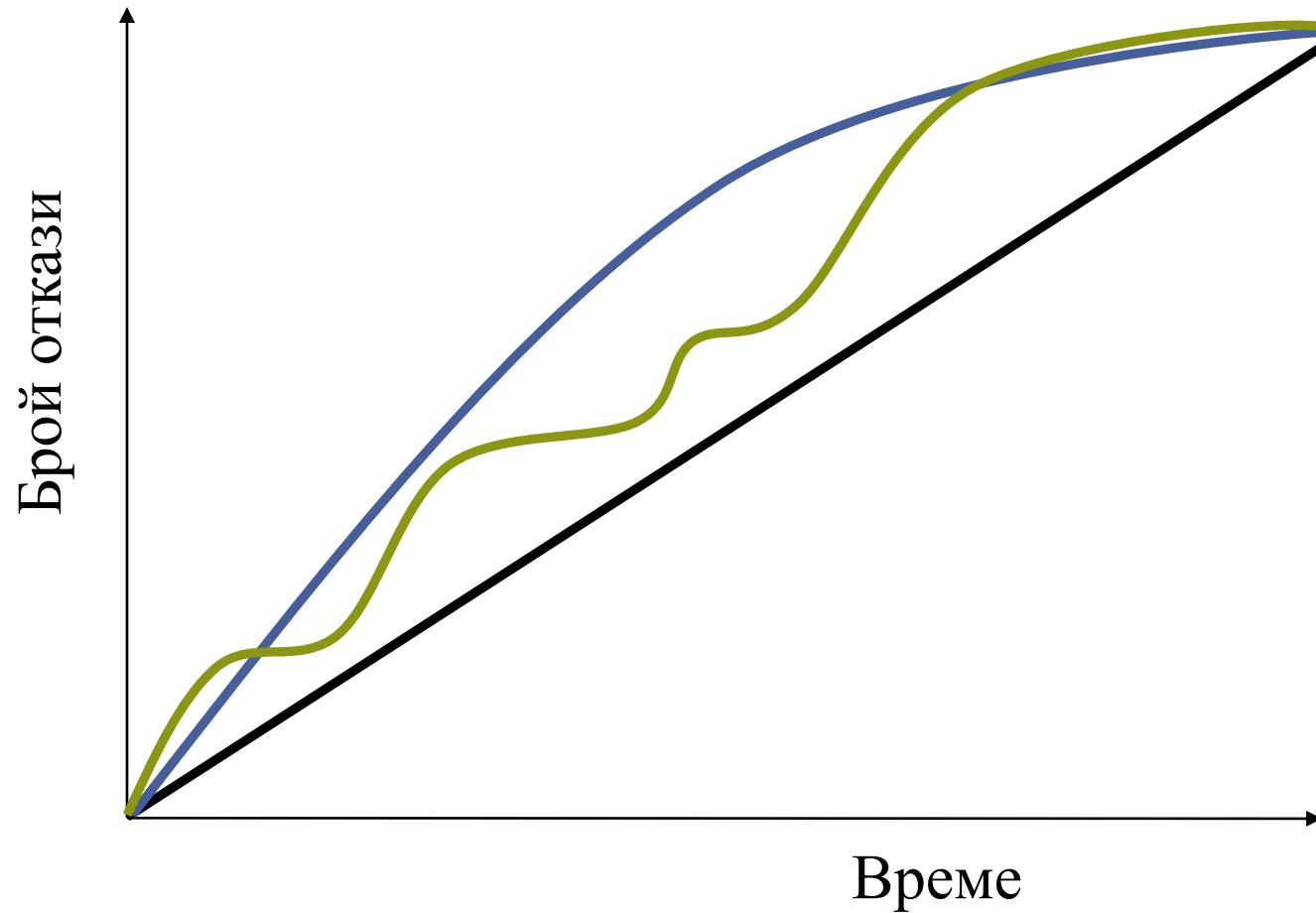
λ : Failure intensity
 λ_0 : Initial failure intensity
at start of execution
 μ : Average total number of
failures at a given point
in time
 v_0 : Total number of failures
over infinite time



Примерни данни за оценка на надеждност



Характерни модели на поведението на грешките в системата



Модели от тип „бяла кутия“

- ▶ Модели на състоянието (state-based models)
- ▶ Модели на пътя на изпълнение (path-based models)

[Goseva-Popstojanova & Trivedi, 2001]

Модели на състоянието

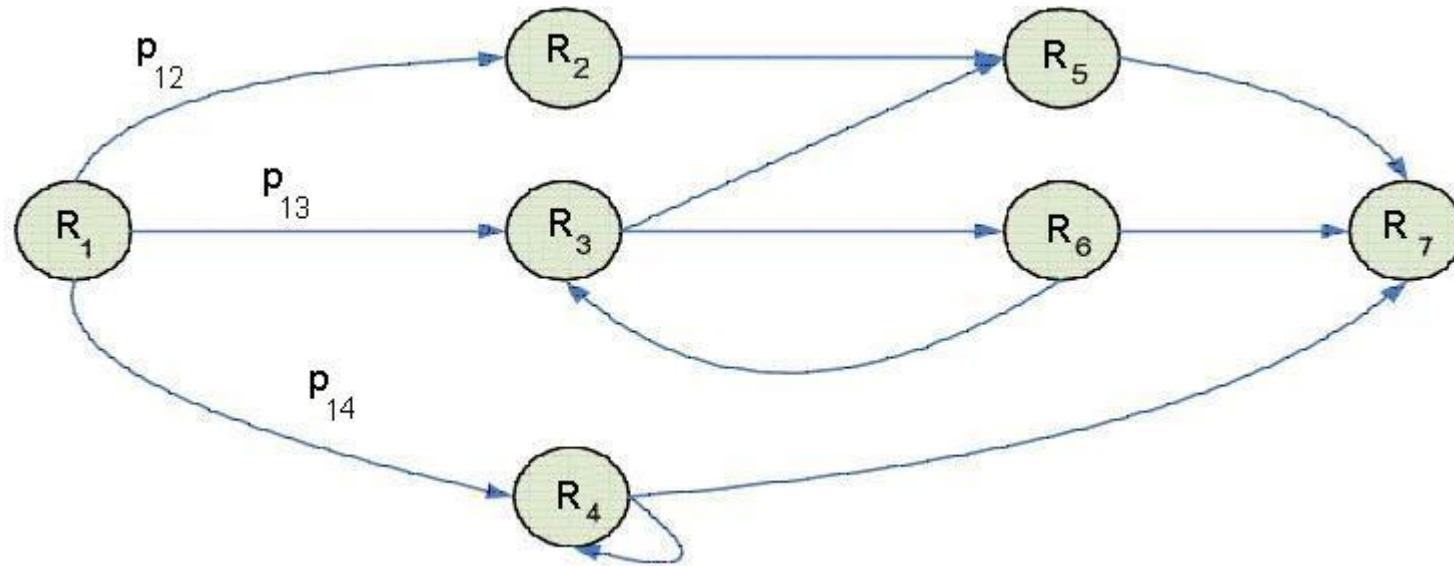
► Основни стъпки при моделирането

- ▶ Идентификация на модулите в системата
- ▶ Построяване на модел на архитектурата на системата
 - ▶ Характерно тук е че трява да се определи т.нар. профил на употреба – т.е. кои компоненти в системата с каква вероятност/честота се използват
- ▶ Определяне на поведението на отказите на всеки модул – намиране на конкретна стойност за надеждността
- ▶ Комбиниране на модела на архитектурата с поведението на отказите

Някои общи за всички модели допускания

- ▶ Данните за надеждността на компонентите са предварително известни
- ▶ Профилът на употреба е предварително известен
- ▶ Отказите на компонентите са независими събития
- ▶ Преходите (предаването на изпълнението) от един компонент към друг са независими събития
- ▶ Вероятността за отказ на даден компонент е константа и не се променя във времето

Модели на състоянието



- ▶ Използва се верига на Марков
 - ▶ Състоянията моделират компонентите
 - ▶ R_i – Надеждност на компонента i
 - ▶ P_{ij} – вероятност за преход от компонент i към компонент j по време на изпълненето на системата (това е профил на употреба)

Модели на състоянието

- ▶ Модел на Ръоснер/Ченг [Cheung 1978, Reussner et al 2003]
 - ▶ Съставя се матрица S със следните елементи

$$S_{ij} = p_{ij} R_i$$

- ▶ Добавят се ред и колона в S , които представлят начално и крайно състояние – съответно с индекси I и J
- ▶ Може да се докаже, че надеждността на системата е:



$$r_{system} = (I - S)^{-1}_{(I,J)}$$

Модели на пътя на изпълнение

- ▶ Отново се използва верига на Марков за модел на архитектурата
- ▶ Изпълняват се N теста на системата, върху различни точно определени пътища на изпълнение в модела на системата
- ▶ Надеждността на всеки път на изпълнение е:

$$R_t = \prod_{\forall m \in M(t)} R_m$$

- ▶ Окончателно надеждността на системата е:

$$R_c = \frac{\sum_{\forall t \in T} R_t}{|T|}$$

- ▶ където T е множеството на тестовете на системата

Модели на пътя на изпълнение

- ▶ Надеждността на системата намалява значително при наличие на циклично изпълнение на даден компонент
- ▶ Надеждността на системата е по-чувствителна от профила на употреба, отколкото от архитектурата на системата

Допълнителни материали

- ▶ Goseva-Popstojanova, K., A. P. Mathur and K. S. Trivedi, Comparison of Architecture-Based Software Reliability Models, 12th IEEE International Symposium on Software Reliability Engineering (ISSRE 2001), Hong Kong, Nov. 2001, pp.22-31.
- ▶ Dimov A. (2013). Measurement of Software Quality Characteristics. Computer Science and Technologies. Vol. I, 2013. 199-204.



Развитие и поддръжка на софтуерни системи

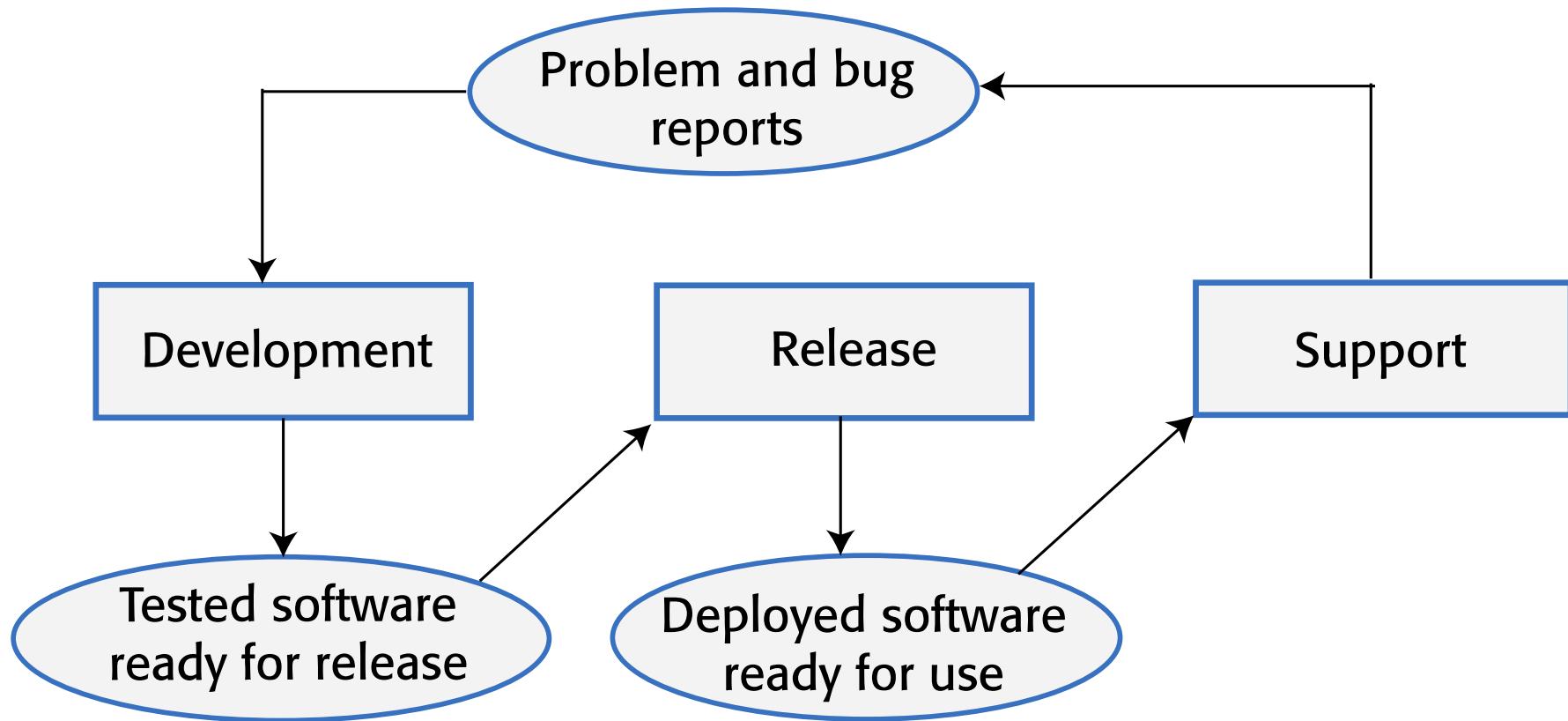


Evolution and code management of software systems

Software support

- ▶ Traditionally, separate teams were responsible software development, software release and software support.
- ▶ The development team passed over a ‘final’ version of the software to a release team. This team then built a release version, tested this and prepared release documentation before releasing the software to customers.
- ▶ A third team was responsible for providing customer support.
 - ▶ The original development team were sometimes also responsible for implementing software changes.
 - ▶ Alternatively, the software may have been maintained by a separate ‘maintenance team’.

Figure 10.1 Development, release and support



DevOps

- ▶ There are inevitable delays and overheads in the traditional support model.
- ▶ To speed up the release and support processes, an alternative approach called DevOps (Development+Operations) has been developed.
- ▶ Three factors led to the development and widespread adoption of DevOps:
 - ▶ Agile software engineering reduced the development time for software, but the traditional release process introduced a bottleneck between development and deployment.
 - ▶ Amazon re-engineered their software around services and introduced an approach in which a service was developed and supported by the same team. Amazon's claim that this led to significant improvements in reliability was widely publicized.
 - ▶ It became possible to release software as a service, running on a public or private cloud. Software products did not have to be released to users on physical media or downloads.

DevOps principles

- ▶ ***Everyone is responsible for everything***

All team members have joint responsibility for developing, delivering and supporting the software.

- ▶ ***Everything that can be automated should be automated***

All activities involved in testing, deployment and support should be automated if it is possible to do so. There should be minimal manual involvement in deploying software.

- ▶ ***Measure first, change later***

DevOps should be driven by a measurement program where you collect data about the system and its operation. You then use the collected data to inform decisions about changing DevOps processes and tools.

Benefits of DevOps

► **Faster deployment**

Software can be deployed to production more quickly because communication delays between the people involved in the process are dramatically reduced.

► **Reduced risk**

The increment of functionality in each release is small so there is less chance of feature interactions and other changes causing system failures and outages.

► **Faster repair**

DevOps teams work together to get the software up and running again as soon as possible. There is no need to discover which team were responsible for the problem and to wait for them to fix it.

► **More productive teams**

DevOps teams are happier and more productive than the teams involved in the separate activities. Because team members are happier, they are less likely to leave to find jobs elsewhere.

Code management

- ▶ During the development of a software product, the development team will probably create tens of thousands of lines of code and automated tests.
- ▶ These will be organized into hundreds of files. Dozens of libraries may be used, and several, different programs may be involved in creating and running the code.
- ▶ Code management is a set of software-supported practices that is used to manage an evolving codebase.
- ▶ You need code management to ensure that changes made by different developers do not interfere with each other, and to create different product versions.
- ▶ Code management tools make it easy to create an executable product from its source code files and to run automated tests on that product.

A code management problem

- ▶ Alice and Bob worked for a company called FinanceMadeSimple and were team members involved in developing a personal finance product. Alice discovered a bug in a module called TaxReturnPreparation. The bug was that a tax return was reported as filed but, sometimes, it was not actually sent to the tax office. She edited the module to fix the bug. Bob was working on the user interface for the system and was also working on TaxReturnPreparation. Unfortunately, he took a copy before Alice had fixed the bug and, after making his changes, he saved the module. This overwrote Alice's changes but she was not aware of this.
- ▶ The product tests did not reveal the bug as it was an intermittent failure that depended on the sections of the tax return form that had been completed. The product was launched with the bug. For most users, everything worked OK. However, for a small number of users, their tax returns were not filed and they were fined by the revenue service. The subsequent investigation showed the software company was negligent. This was widely publicised and, as well as a fine from the tax authorities, users lost confidence in the software. Many switched to a rival product. FinanceMade Simple failed and both Bob and Alice lost their jobs.

Code management and DevOps

- ▶ Source code management, combined with automated system building, is essential for professional software engineering.
- ▶ In companies that use DevOps, a modern code management system is a fundamental requirement for ‘automating everything’.
- ▶ Not only does it store the project code that is ultimately deployed, it also stores all other information that is used in DevOps processes.
- ▶ DevOps automation and measurement tools all interact with the code management system

Code management fundamentals

- ▶ Code management systems provide a set of features that support four general areas:
 - ▶ **Code transfer** Developers take code into their personal file store to work on it then return it to the shared code management system.
 - ▶ **Version storage and retrieval** Files may be stored in several different versions and specific versions of these files can be retrieved.
 - ▶ **Merging and branching** Parallel development branches may be created for concurrent working. Changes made by developers in different branches may be merged.
 - ▶ **Version information** Information about the different versions maintained in the system may be stored and retrieved

Code repository

- ▶ All source code files and file versions are stored in the repository, as are other artefacts such as configuration files, build scripts, shared libraries and versions of tools used.
- ▶ The repository includes a database of information about the stored files such as version information, information about who has changed the files, what changes were made at what times, and so on.
- ▶ Files can be transferred to and from the repository and information about the different versions of files and their relationships may be updated.
 - ▶ Specific versions of files and information about these versions can always be retrieved from the repository.

Features of code management systems

- ▶ ***Version and release identification***

Managed versions of a code file are uniquely identified when they are submitted to the system and can be retrieved using their identifier and other file attributes.

- ▶ ***Change history recording***

The reasons why changes to a code file have been made are recorded and maintained.

- ▶ ***Independent development***

Several developers can work on the same code file at the same time. When this is submitted to the code management system, a new version is created so that files are never overwritten by later changes.

- ▶ ***Project support***

All of the files associated with a project may be checked out at the same time. There is no need to check out files one at a time.

- ▶ ***Storage management***

The code management system includes efficient storage mechanisms so that it doesn't keep multiple copies of files that have only small differences.

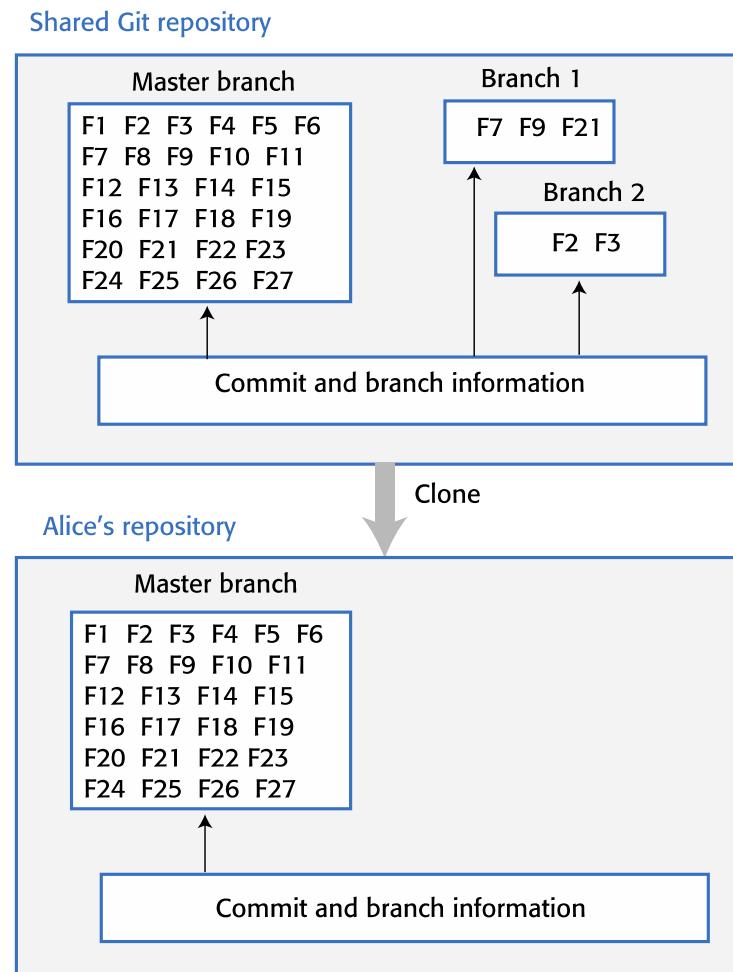
Git

- ▶ In 2005, Linus Torvalds, the developer of Linux, revolutionized source code management by developing a distributed version control system (DVCS) called Git to manage the code of the Linux kernel.
- ▶ This was geared to supporting large-scale open source development. It took advantage of the fact that storage costs had fallen to such an extent that most users did not have to be concerned with local storage management.
- ▶ Instead of only keeping the copies of the files that users are working on, Git maintains a clone of the repository on every user's computer

Branching and merging

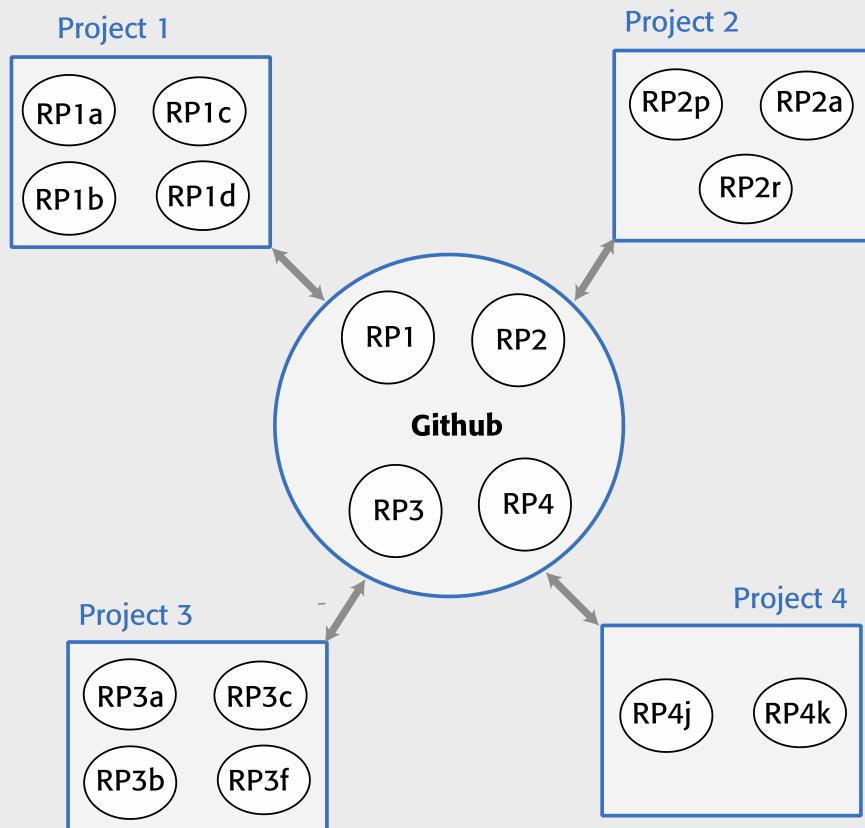
- ▶ Branching and merging are fundamental ideas that are supported by all code management systems.
- ▶ A branch is an independent, stand-alone version that is created when a developer wishes to change a file.
- ▶ The changes made by developers in their own branches may be merged to create a new shared branch.
- ▶ The repository ensures that branch files that have been changed cannot overwrite repository files without a merge operation.
 - ▶ If Alice or Bob make mistakes on the branch they are working on, they can easily revert to the master file.
 - ▶ If they commit changes, while working, they can revert to earlier versions of the work they have done. When they have finished and tested their code, they can then replace the master file by merging the work they have done with the master branch

Repository cloning in Git



Git repositories

Figure 10.6 Git repositories



DevOps automation

- ▶ By using DevOps with automated support, you can dramatically reduce the time and costs for integration, deployment and delivery.
- ▶ *Everything that can be, should be automated* is a fundamental principle of DevOps.
- ▶ As well as reducing the costs and time required for integration, deployment and delivery, process automation also makes these processes more reliable and reproducible.
- ▶ Automation information is encoded in scripts and system models that can be checked, reviewed, versioned and stored in the project repository.

Aspects of DevOps automation

► ***Continuous integration***

Each time a developer commits a change to the project's master branch, an executable version of the system is built and tested.

► ***Continuous delivery***

A simulation of the product's operating environment is created and the executable software version is tested.

► ***Continuous deployment***

A new release of the system is made available to users every time a change is made to the master branch of the software.

► ***Infrastructure as code***

Machine-readable models of the infrastructure (network, servers, routers, etc.) on which the product executes are used by configuration management tools to build the software's execution platform. The software to be installed, such as compilers and libraries and a DBMS, are included in the infrastructure model.

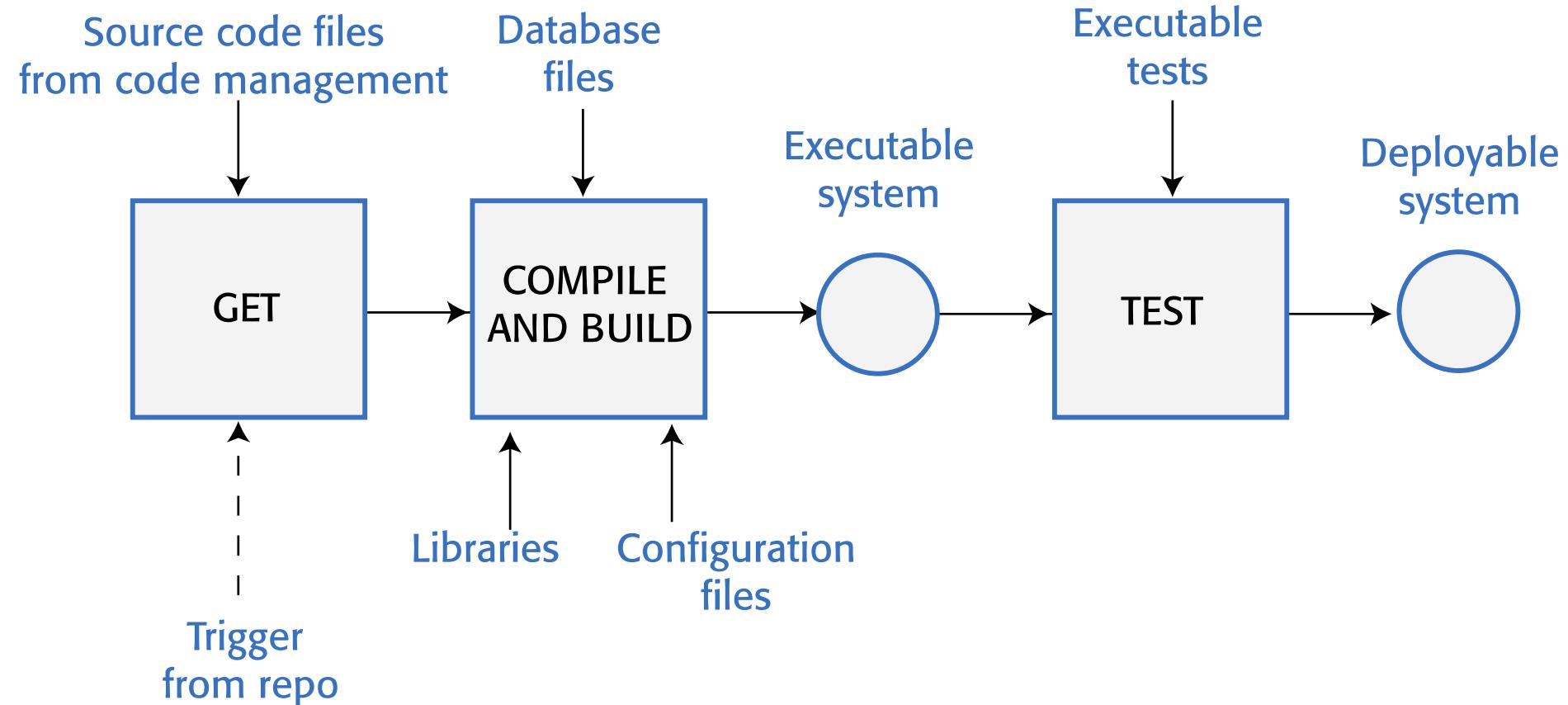
System integration

- ▶ System integration (system building) is the process of gathering all of the elements required in a working system, moving them into the right directories, and putting them together to create an operational system.
- ▶ Typical activities that are part of the system integration process include:
 - ▶ Installing database software and setting up the database with the appropriate schema.
 - ▶ Loading test data into the database.
 - ▶ Compiling the files that make up the product.
 - ▶ Linking the compiled code with the libraries and other components used.
 - ▶ Checking that external services used are operational.
 - ▶ Deleting old configuration files and moving configuration files to the correct locations.
 - ▶ Running a set of system tests to check that the integration has been successful.

Continuous integration

- ▶ Continuous integration simply means that an integrated version of the system is created and tested every time a change is pushed to the system's shared repository.
- ▶ On completion of the push operation, the repository sends a message to an integration server to build a new version of the product
- ▶ The advantage of continuous integration compared to less frequent integration is that it is faster to find and fix bugs in the system.
- ▶ If you make a small change and some system tests then fail, the problem almost certainly lies in the new code that you have pushed to the project repo.
- ▶ You can focus on this code to find the bug that's causing the problem.

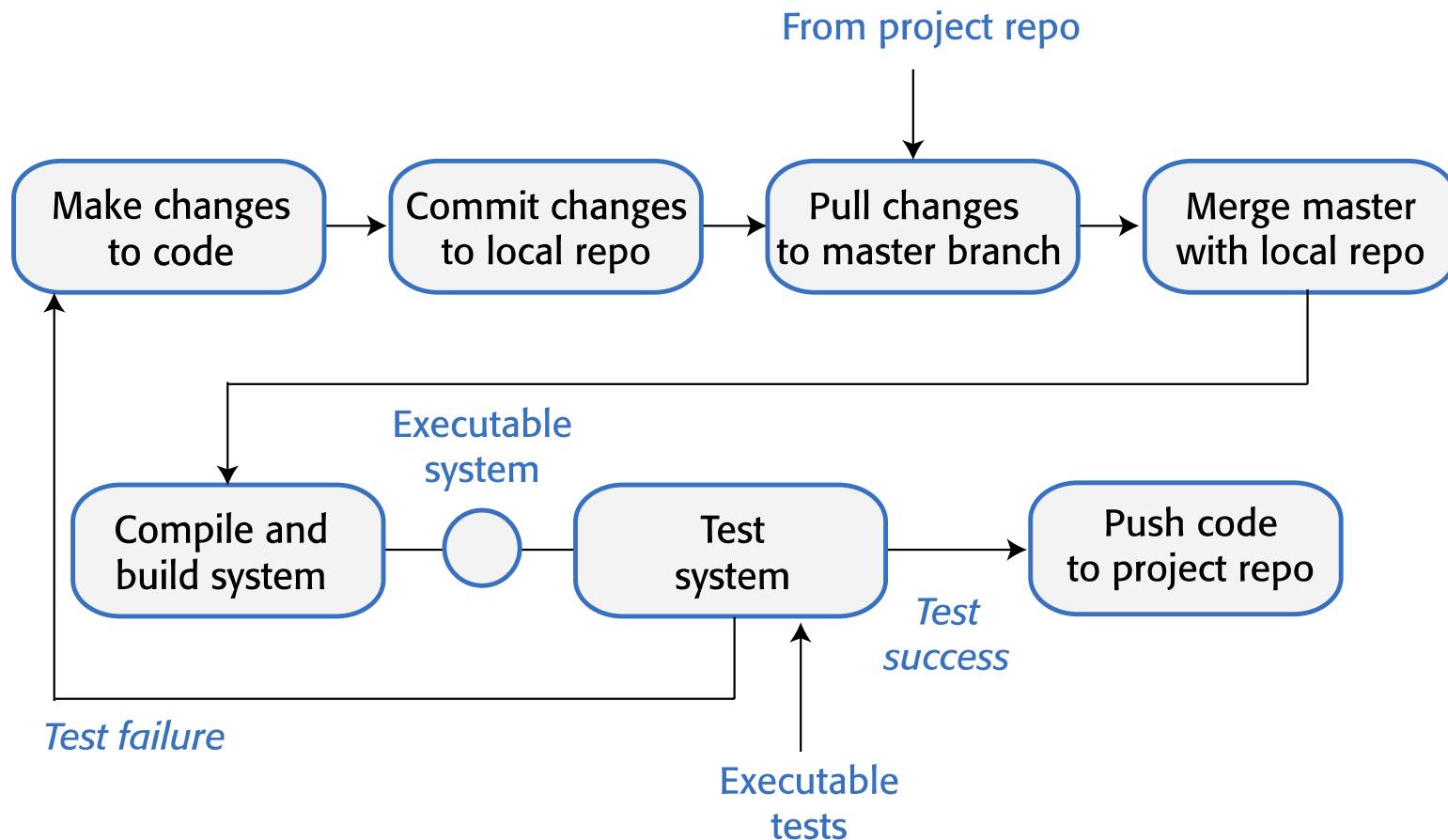
Continuous integration



Breaking the build

- ▶ In a continuous integration environment, developers have to make sure that they don't 'break the build'.
- ▶ Breaking the build means pushing code to the project repository which, when integrated, causes some of the system tests to fail.
- ▶ If this happens to you, your priority should be to discover and fix the problem so that normal development can continue.
- ▶ To avoid breaking the build, you should always adopt an 'integrate twice' approach to system integration.
 - ▶ You should integrate and test on your own computer before pushing code to the project repository to trigger the integration server

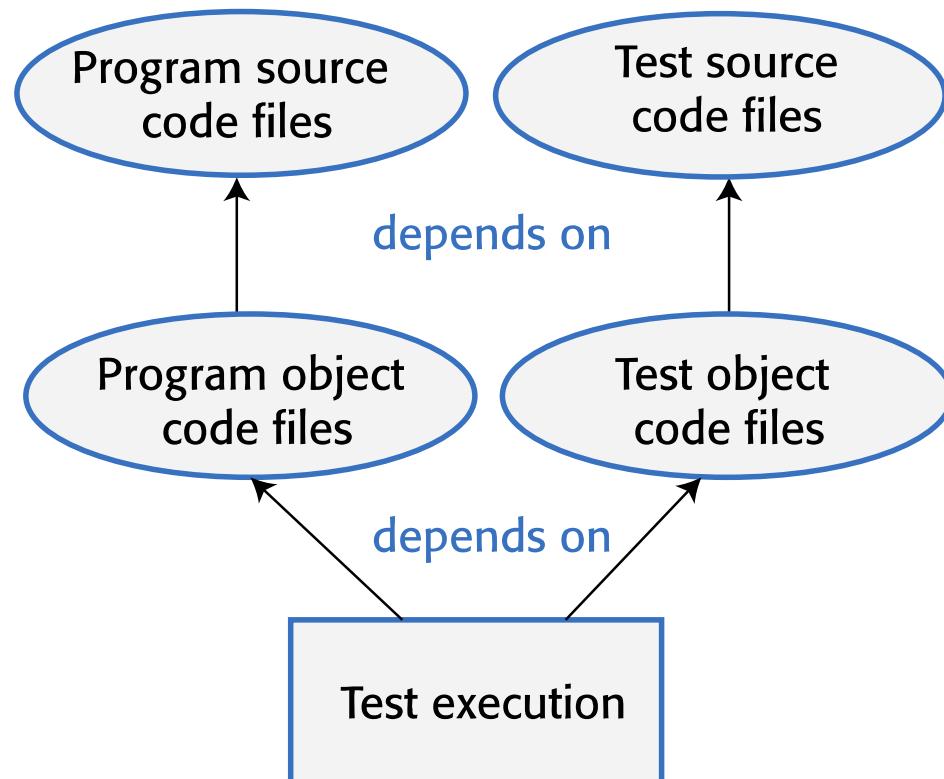
Local integration



System building

- ▶ Continuous integration is only effective if the integration process is fast and developers do not have to wait for the results of their tests of the integrated system.
- ▶ However, some activities in the build process, such as populating a database or compiling hundreds of system files, are inherently slow.
- ▶ It is therefore essential to have an automated build process that minimizes the time spent on these activities.
- ▶ Fast system building is achieved using a process of incremental building, where only those parts of the system that have been changed are rebuilt

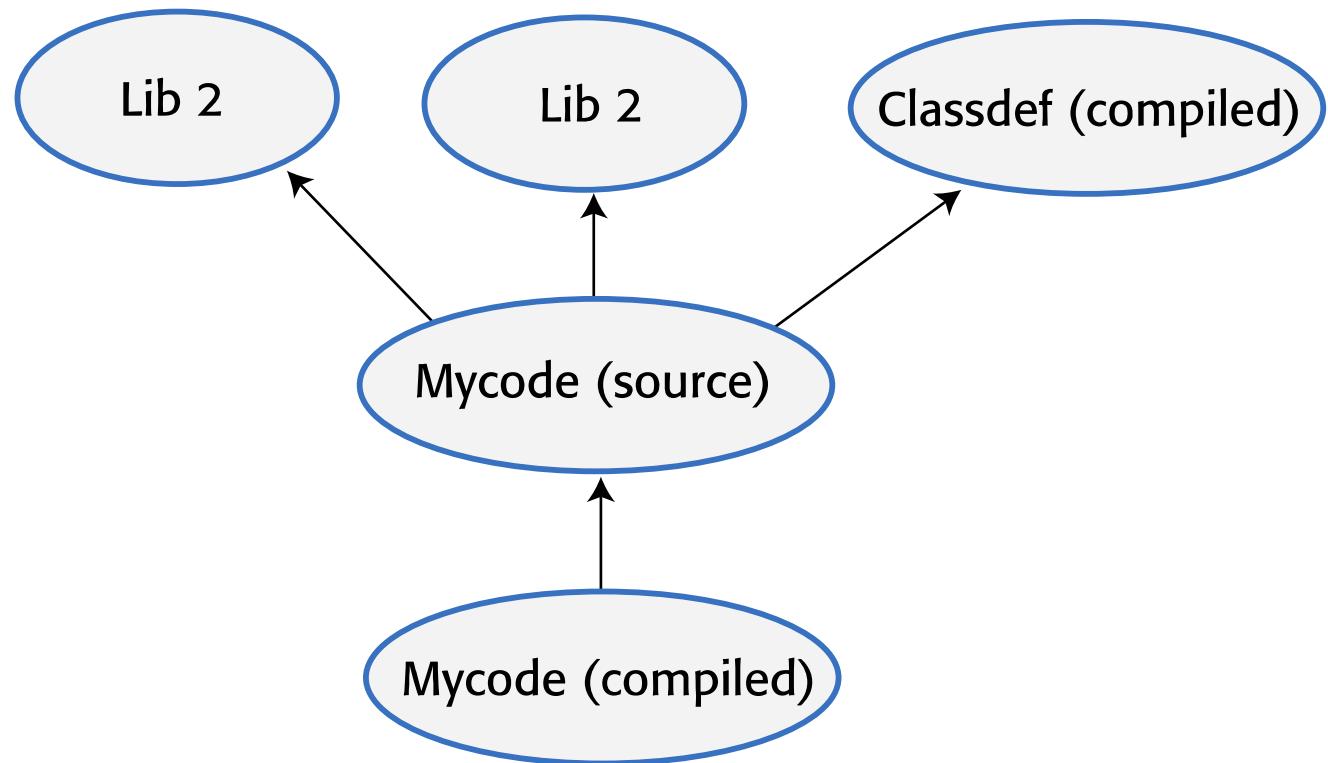
A dependency model



Dependencies

- ▶ This dependency model that shows the dependencies for test execution.
- ▶ The upward-pointing arrow means ‘depends on’ and shows the information required to complete the task shown in the rectangle at the base of the model.
- ▶ Running a set of system tests depends on the existence of executable object code for both the program being tested and the system tests.
- ▶ In turn, these depend on the source code for the system and the tests that are compiled to create the object code.
- ▶ Next slide shows a lower-level dependency model that shows the dependencies involved in creating the object code for a source code files called Mycode.

File dependencies, an example

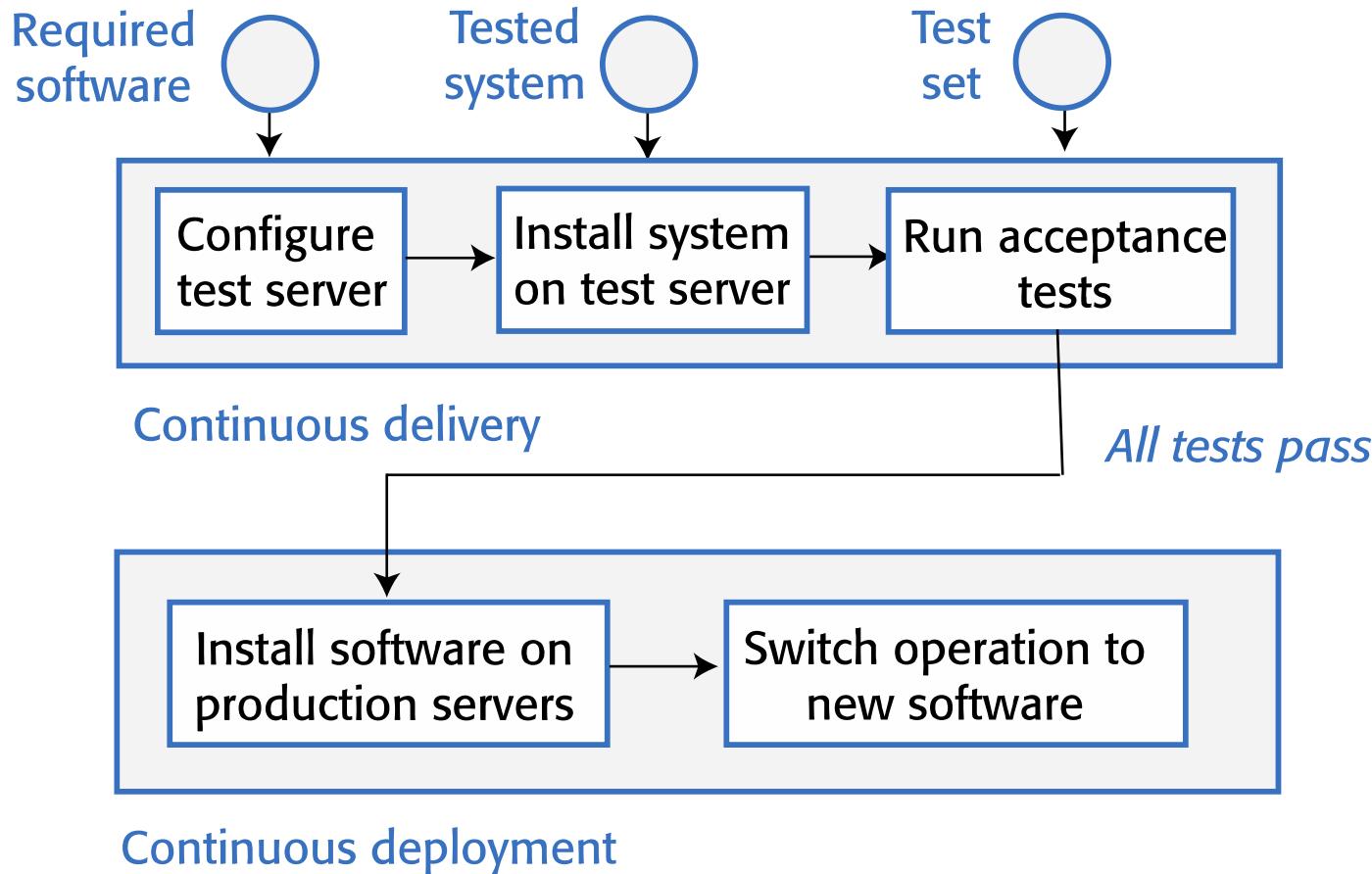


-
- ▶ An automated build system uses the specification of dependencies to work out what needs to be done. It uses the file modification timestamp to decide if a source code file has been changed.
 - ▶ The modification date of the compiled code is after the modification date of the source code. The build system infers that no changes have been made to the source code and does nothing.
 - ▶ The modification date of the compiled code is before the modification date of the source code. The build system recompiles the source and replaces the existing file of compiled code with an updated version.
 - ▶ The modification date of the compiled code is after the modification date of the source code. However, the modification date of Classdef is after the modification date of the source code of Mycode. Therefore, Mycode has to be recompiled to incorporate these changes.

Continuous delivery and deployment

- ▶ Continuous integration means creating an executable version of a software system whenever a change is made to the repository. The CI tool builds the system and runs tests on your development computer or project integration server.
- ▶ However, the real environment in which software runs will inevitably be different from your development system.
- ▶ When your software runs in its real, operational environment bugs may be revealed that did not show up in the test environment.
- ▶ Continuous delivery means that, after making changes to a system, you ensure that the changed system is ready for delivery to customers.
- ▶ This means that you have to test it in a production environment to make sure that environmental factors do not cause system failures or slow down its performance.

Continuous delivery and deployment



The deployment pipeline

- ▶ After initial integration testing, a staged test environment is created.
- ▶ This is a replica of the actual production environment in which the system will run.
- ▶ The system acceptance tests, which include functionality, load and performance tests, are then run to check that the software works as expected. If all of these tests pass, the changed software is installed on the production servers.
- ▶ To deploy the system, you then momentarily stop all new requests for service and leave the older version to process the outstanding transactions.
- ▶ Once these have been completed, you switch to the new version of the system and restart processing.

Benefits of continuous deployment

► **Reduced costs**

If you use continuous deployment, you have no option but to invest in a completely automated deployment pipeline. Manual deployment is a time-consuming and error-prone process. Setting up an automated system is expensive and time-consuming but you can recover these costs quickly if you make regular updates to your product.

► **Faster problem solving**

If a problem occurs, it will probably only affect a small part of the system and it will be obvious what the source of that problem is. If you bundle many changes into a single release, finding and fixing problems is more difficult.

► **Faster customer feedback**

You can deploy new features when they are ready for customer use. You can ask them for feedback on these features and use this feedback to identify improvements that you need to make.

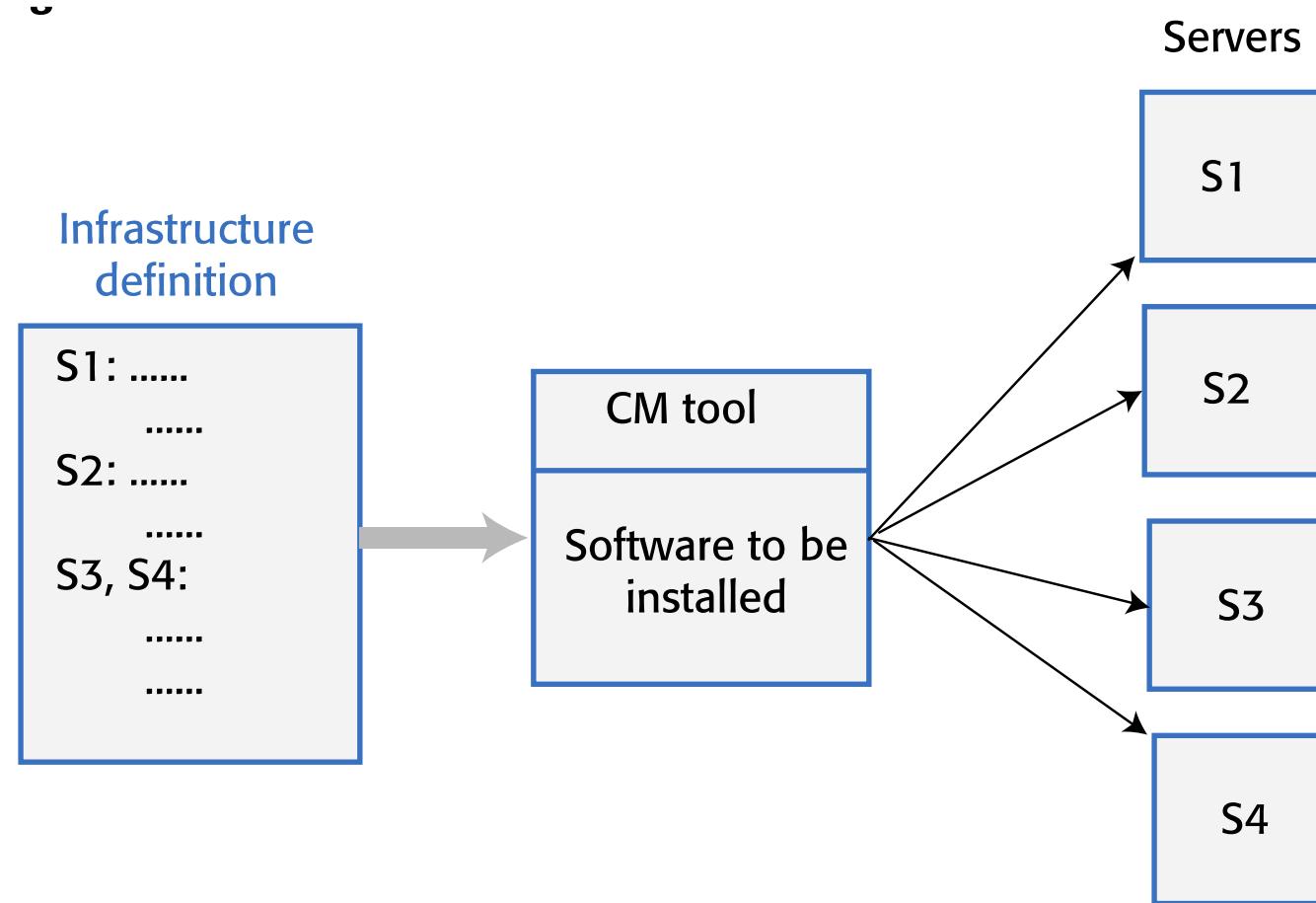
► **A/B testing**

This is an option if you have a large customer base and use several servers for deployment. You can deploy a new version of the software on some servers and leave the older version running on others. You then use the load balancer to divert some customers to the new version while others use the older version. You can then measure and assess how new features are used to see if they do what you expect.

Infrastructure as code

- ▶ In an enterprise environment, there are usually many different physical or virtual servers (web servers, database servers, file servers, etc.) that do different things. These have different configurations and run different software packages.
- ▶ It is therefore difficult to keep track of the software installed on each machine.
- ▶ The idea of infrastructure as code was proposed as a way to address this problem. Rather than manually updating the software on a company's servers, the process can be automated using a model of the infrastructure written in a machine-processable language.
- ▶ Configuration management (CM) tools such as Puppet and Chef can automatically install software and services on servers according to the infrastructure definition

Infrastructure as code



Benefits of infrastructure as code

- ▶ Defining your infrastructure as code and using a configuration management system solves two key problems of continuous deployment.
 - ▶ Your testing environment must be exactly the same as your deployment environment. If you change the deployment environment, you have to mirror those changes in your testing environment.
 - ▶ When you change a service, you have to be able to roll that change out to all of your servers quickly and reliably. If there is a bug in your changed code that affects the system's reliability, you have to be able to seamlessly roll back to the older system.
- ▶ The business benefits of defining your infrastructure as code are lower costs of system management and lower risks of unexpected problems arising when infrastructure changes are implemented.

Characteristics of infrastructure as code

▶ **Visibility**

Your infrastructure is defined as a stand-alone model that can be read, discussed, understood and reviewed by the whole DevOps team.

▶ **Reproducability**

Using a configuration management tool means that the installation tasks will always be run in the same sequence so that the same environment is always created. You are not reliant on people remembering the order that they need to do things.

▶ **Reliability**

The complexity of managing a complex infrastructure means that system administrators often make simple mistakes, especially when the same changes have to be made to several servers. Automating the process avoids these mistakes.

▶ **Recovery**

Like any other code, your infrastructure model can be versioned and stored in a code management system. If infrastructure changes cause problems you can easily revert to an older version and reinstall the environment that you know works.

Containers

- ▶ A container provides a stand-alone execution environment running on top of an operating system such as Linux.
- ▶ The software installed in a Docker container is specified using a Dockerfile, which is, essentially, a definition of your software infrastructure as code.
- ▶ You build an executable container image by processing the Dockerfile.
- ▶ Using containers makes it very simple to provide identical execution environments.
 - ▶ For each type of server that you use, you define the environment that you need and build an image for execution. You can run an application container as a test system or as an operational system; there is no distinction between them.
 - ▶ When you update your software, you rerun the image creation process to create a new image that includes the modified software. You can then start these images alongside the existing system and divert service requests to them.

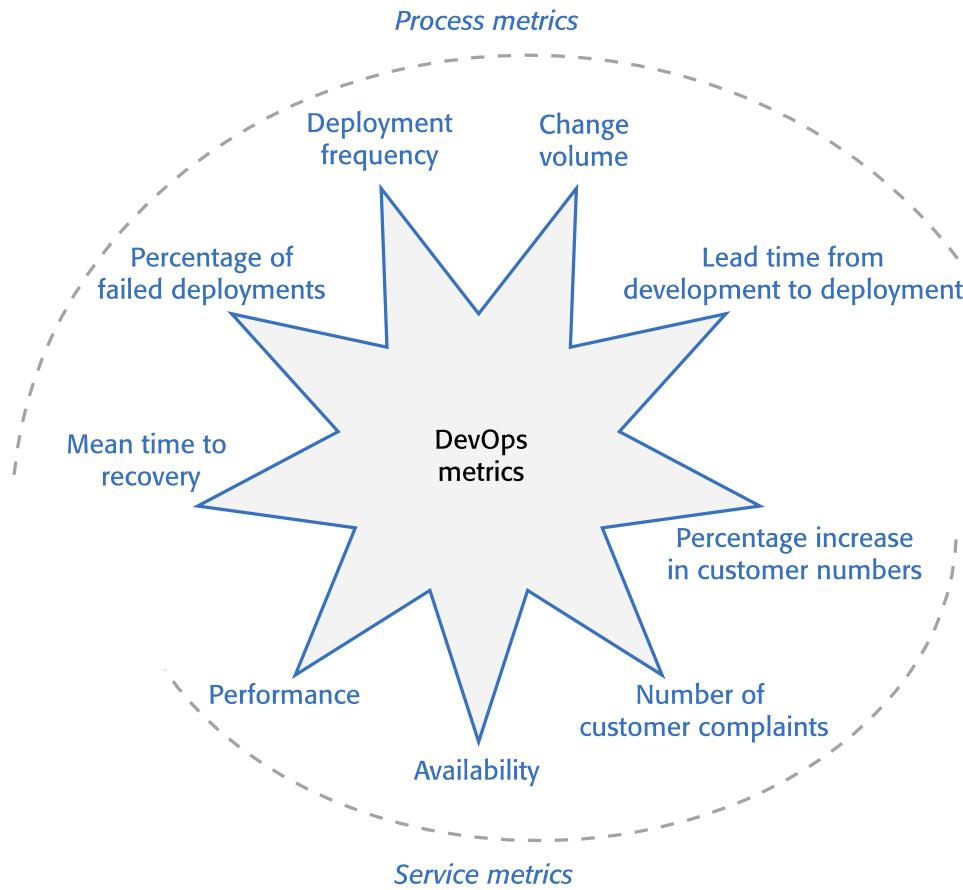
DevOps measurement

- ▶ After you have adopted DevOps, you should try to continuously improve your DevOps process to achieve faster deployment of better-quality software.
- ▶ There are four types of software development measurement:
 - ▶ **Process measurement** You collect and analyse data about your development, testing and deployment processes.
 - ▶ **Service measurement** You collect and analyse data about the software's performance, reliability and acceptability to customers.
 - ▶ **Usage measurement** You collect and analyse data about how customers use your product.
 - ▶ **Business success measurement** You collect and analyse data about how your product contributes to the overall success of the business.

Automating measurement

- ▶ As far as possible, the DevOps principle of automating everything should be applied to software measurement.
- ▶ You should instrument your software to collect data about itself and you should use a monitoring system, as I explained in Chapter 6, to collect data about your software's performance and availability.
- ▶ Some process measurements can also be automated.
 - ▶ However, there are problems in process measurement because people are involved. They work in different ways, may record information differently and are affected by outside influences that affect the way they work.

Metrics used in the DevOps scorecard



Metrics scorecard

- ▶ Payal Chakravarty from IBM suggests a practical approach to DevOps measurement based around a metrics scorecard with 9 metrics:
 - ▶ These are relevant to software that is delivered as a cloud service. They include process metrics and service metrics
 - ▶ For the process metrics, you would like to see decreases in the number of failed deployments, the mean time to recovery after a service failure and the lead time from development to deployment.
 - ▶ You would hope to see increases in the deployment frequency and the number of lines of changed code that are shipped.
 - ▶ For the service metrics, availability and performance should be stable or improving, the number of customer complaints should be decreasing, and the number of new customers should be increasing.

Metrics trends

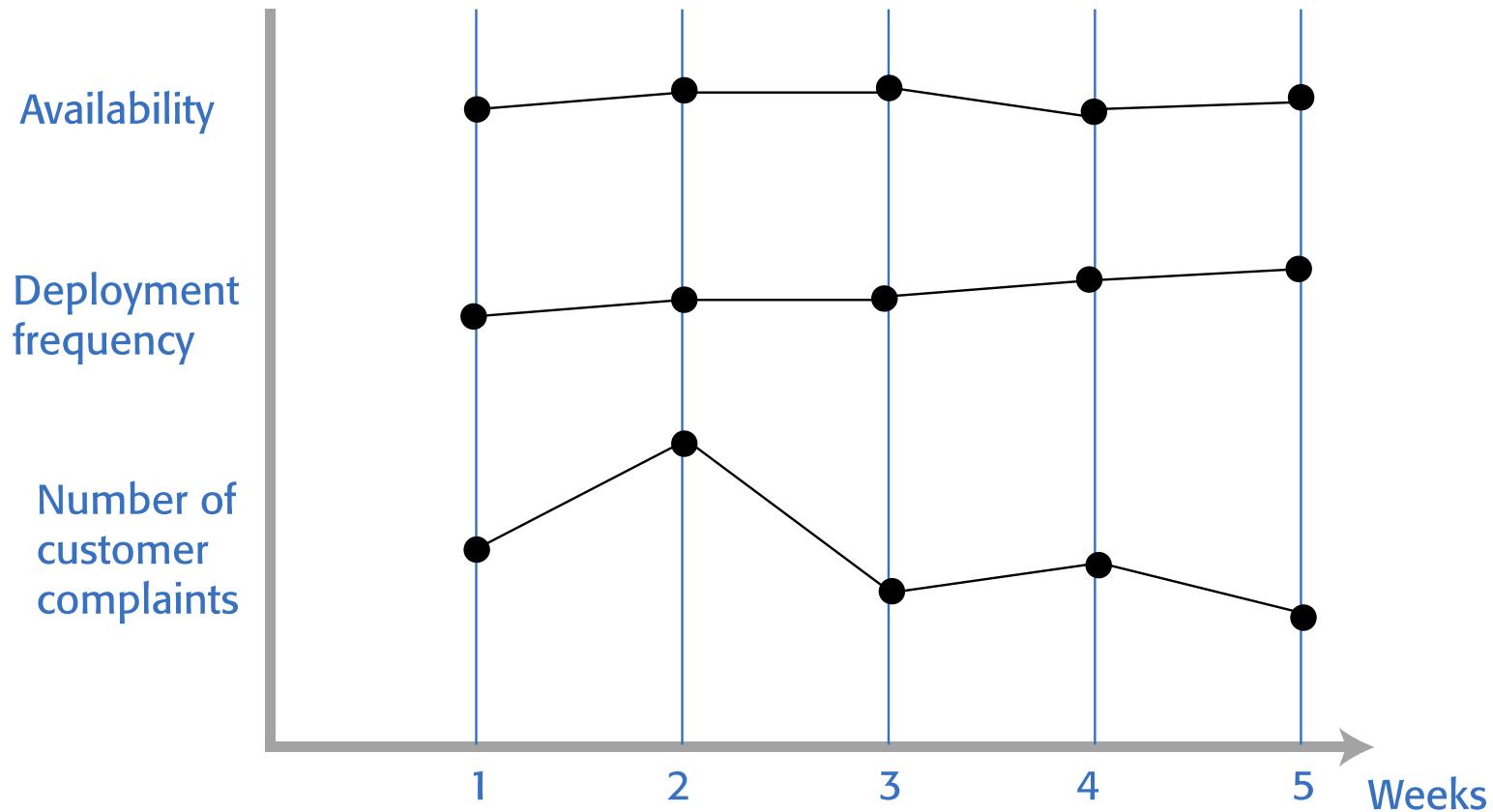
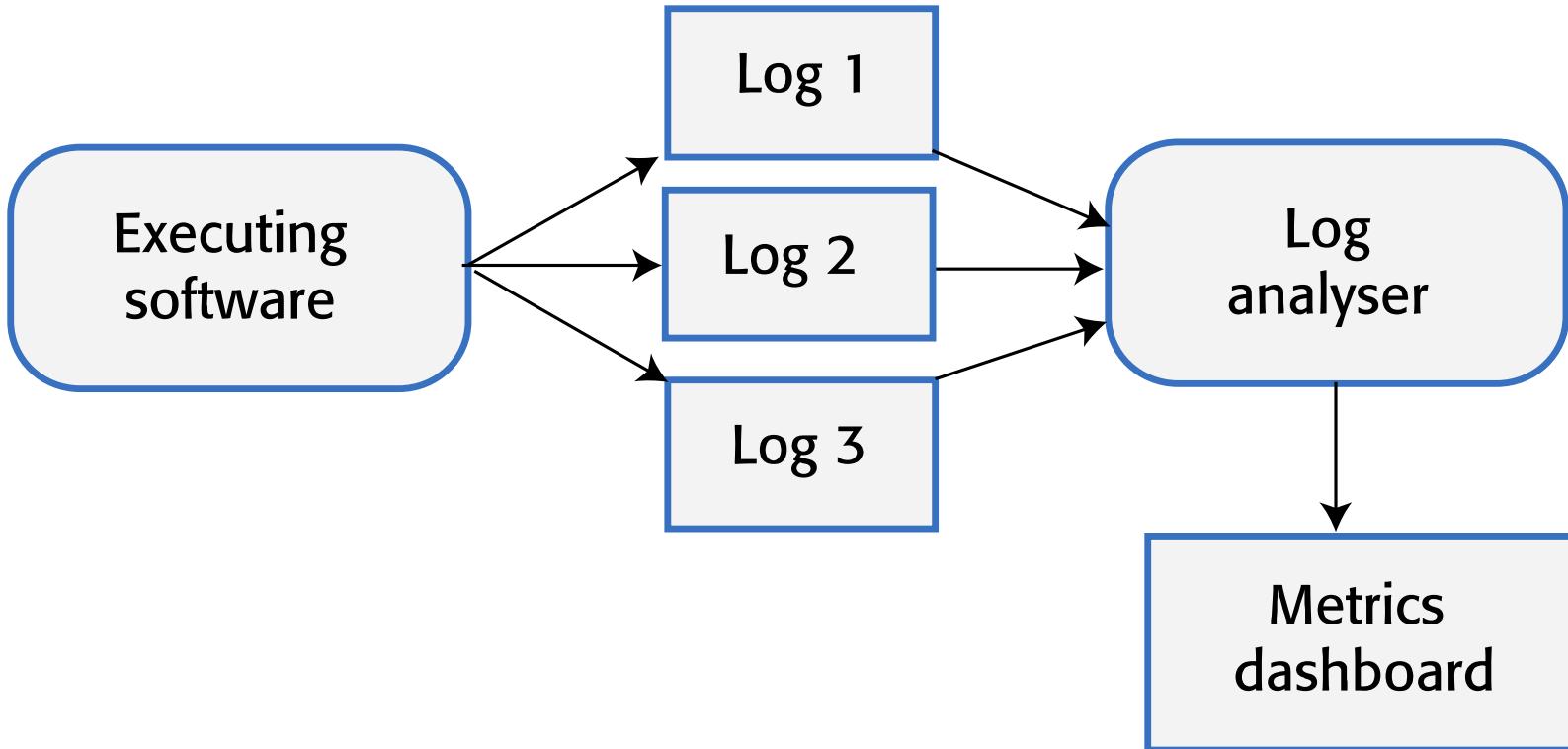


Figure 10.17 Logging and analysis



Key points 1

- ▶ DevOps is the integration of software development and the management of that software once it has been deployed for use. The same team is responsible for development, deployment and software support.
- ▶ The benefits of DevOps are faster deployment, reduced risk, faster repair of buggy code and more productive teams.
- ▶ Source code management is essential to avoid changes made by different developers interfering with each other.
- ▶ All code management systems are based around a shared code repository with a set of features that support code transfer, version storage and retrieval, branching and merging and maintaining version information.
- ▶ Git is a distributed code management system that is the most widely used system for software product development. Each developer works with their own copy of the repository which may be merged with the shared project repository.

Key points 2

- ▶ Continuous integration means that as soon as a change is committed to a project repository, it is integrated with existing code and a new version of the system is created for testing.
- ▶ Automated system building tools reduce the time needed to compile and integrate the system by only recompiling those components and their dependents that have changed.
- ▶ Continuous deployment means that as soon as a change is made, the deployed version of the system is automatically updated. This is only possible when the software product is delivered as a cloud-based service.
- ▶ Infrastructure as code means that the infrastructure (network, installed software, etc.) on which software executes is defined as a machine-readable model. Automated tools, such as Chef and Puppet, can provision servers based on the infrastructure model.
- ▶ Measurement is a fundamental principle of DevOps. You may make both process and product measurements. Important process metrics are deployment frequency, percentage of failed deployments, and mean time to recovery from failure.

Additional reading/Допълнителни материали

- ▶ Sommerville, I. (2020), Engineering Software Products: An Introduction to Modern Software Engineering, 2nd Ed. Chapter 10 (DevOps and Code Management)



Изкуствен интелект в софтуерните системи



Вместо заключение

Machine learning components in software systems

T-Shaped People

Broad-range generalist + Deep expertise

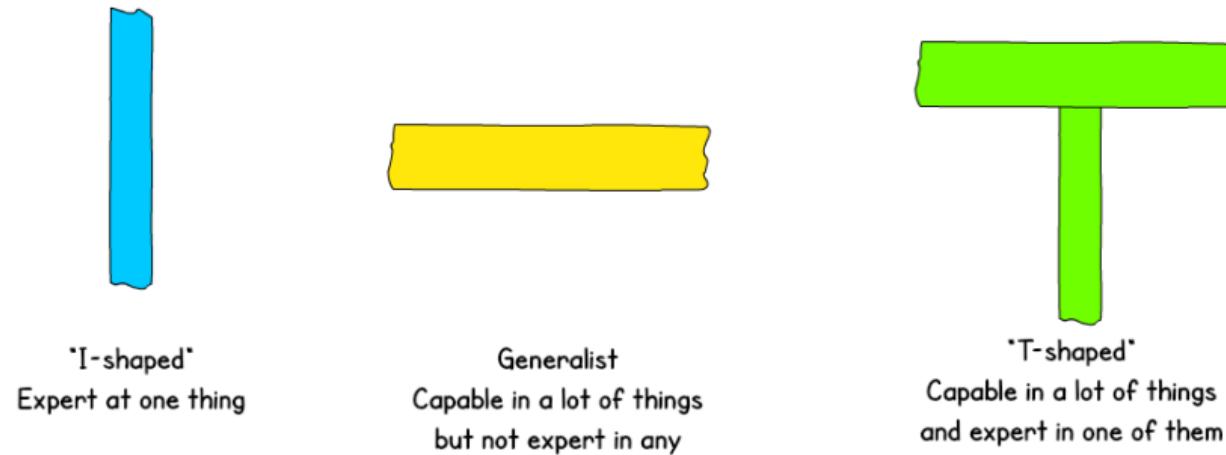


Figure: Jason Yip. [Why T-shaped people?](#). 2018

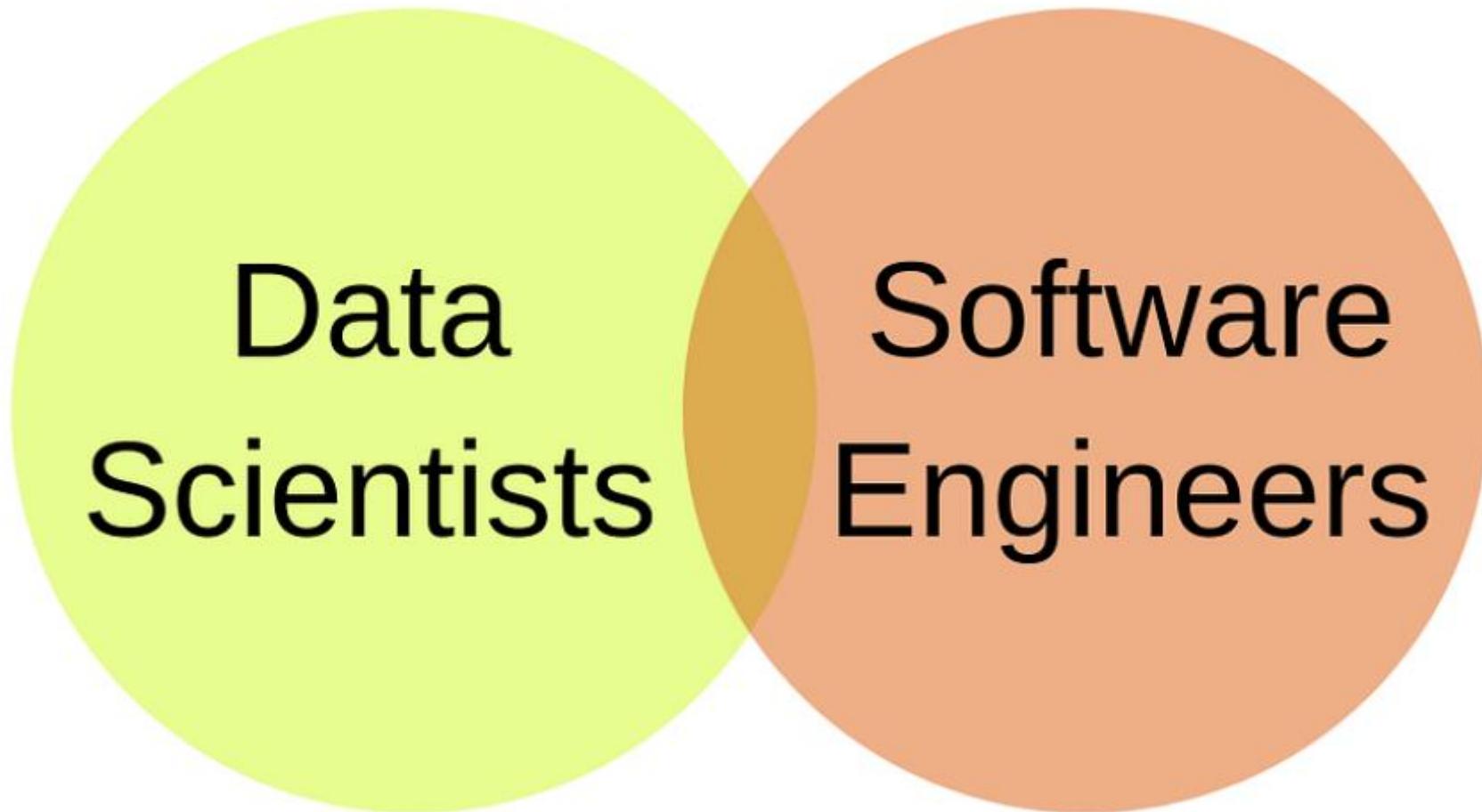
Need for different skills

- *Business skills* to identify the problem and build a product/start-up
- *Domain expertise* to understand the data and frame the goals for the machine-learning task
- *Statistics and data science skills* to select a suitable machine-learning technique and train a model
- *Software engineering skills* to build a system that integrates the model as one of its many components
- *User-interface design skills* to understand and plan how humans will interact with the system (and the model and its mistakes)
- *System operations skills* to handle deployment, scaling and monitoring of the system

More needs

- Help from *data engineers* to extract, move and prepare data at scale
- *Legal expertise* from lawyers who check for compliance with regulations and develop contracts with customers
- Specialized *safety and security expertise* to ensure the system does not cause harms to the users and environment, and does not disclose sensitive information
- *Social science skills* to study how our system could affect society at large, and
- *Project management skills* to hold the whole team together and keep it focused on delivering a product.

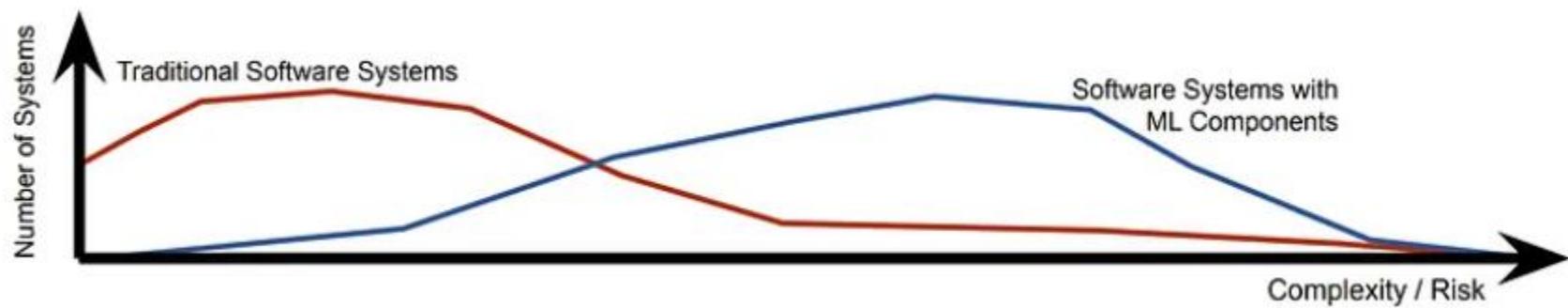
Data scientists and software engineers



Machine-Learning Challenges in Software Projects

- ▶ Lack of Specifications
- ▶ Interactions with environment (the real world)
- ▶ Focus on data and scalability

Traditional vs ML software systems



Допълнителни материали

- ▶ Kästner, C., (2021). Introduction to Machine Learning in Production, available at: <https://ckaestne.medium.com/introduction-to-machine-learning-in-production-eef7427426f1>