

Въведение

A decorative graphic consisting of several overlapping, wavy, light gray lines that flow from the right side of the slide towards the left, positioned below the title.

Информация

☞ *Веселин Евгениев Георгиев*

- ✓ кабинет: 2300A
- ✓ e-mail: veg@tu-sofia.bg
 - ✗ *Subject: FKSU_student*

☞ **График:**

- ✓ Лекции: 2 часа седмично
- ✓ Лаб. упражнения: 4 часа през седмица

Учебен план

- ➡ **План за валидация и верификация на разработвано програмно осигуряване – изисквания, особености, стандарти.**
- ➡ **Особености на валидация и верификация на системен софтуер (операционни системи, виртуализатори и др.)**
- ➡ **Особености на валидация и верификация на хетерогенни системи.**
- ➡ **Особености на валидацията и верификацията на вградени системи и системи за реално време**

Учебен план

- ➡ **Основни понятия.**
- ➡ **Изисквания за осигуряване на качеството и надеждността на разработваното ПО.**
- ➡ **Основни принципи и стратегии процесите на валидацията и верификация на ПО.**
- ➡ **Методи за верификация на програмно осигуряване**
- ➡ **Методи за валидация на програмно осигуряване**
- ➡ **Стандарти, средства и среди за реализация на етапите на валидация и верификация**

Литература

- ➡ **Adam Barr, Find the Bug A Book of Incorrect Programs, Addison-Wesley**
- ➡ **Gary McGraw, Secure Programming with Static Analysis, Addison-Wesley**
- ➡ **John W. Horch, Practical Guide to Software Quality Management, 2nd Edition, Artech House**
- ➡ **Jerry Zeyu Gao, H.-S. Jacob Tsao, Ye Wu , Testing and Quality Assurance for Component Based Software, Artech House**
- ➡ **IEEE P1012/D12 - Draft Standard for Software Verification and Validation**
- ➡ **Abhik Roychoudhury, Embedded Systems and Software Validation, Morgan Kaufmann**

Литература

- **Dorothy Graham, Erik van Veenendaal, Isabel Evans, Rex Black, Foundations of Software Testing**
- **PRABHAT MISHRA, NIKIL D. DUTT, Functional Verification of Programmable Embedded Architectures - A Top-Down Approach, Springer**
- **G. NICOLESCU, A.A JERRAYA, Global Specification And Validation of Embedded Sysytems, Springer**
- **Guide to software verification and validation, European Space Agency**
- **Ian Sommerville, Software_Engineering, 8th Edition, Addison-Wesley**
Stephan Merz, Nicolas Navet , Modeling and Verification of Real-Time Systems - Formalisms and Software Tools, Wiley

Литература

- ➡ **T. M. B. Steffen, Leveraging Applications of Formal Methods, Verification, and Validation, Springer**
- ➡ **R. K. Kandt, Software Engineering Quality Practices, Auerbach Pub.**
- ➡ **D. Bjørner , Software Engineering 3 - Domains, Requirements, and Software Design, Springer**
- ➡ **T.Grotker, U. Holtmann, H. Keding, M. Wloka, The Developers Guide to Debugging, Springer**
- ➡ **Glenford J. Myers, The Art of Software Testing, Second Edition, Wiley**
- ➡ **A. ENGEL , Verification, Validation and Testing of Engineered Systems, Wiley**
- ➡ **Jeff Tian, Interscience Software Quality Engineering - Testing, Quality Assurance and Quantifiable Improvement, Wiley**

Izpit

➡ Продължителност: 3 часа

➡ Тип:

- ✓ Тест: 60 въпроса

- ✓ Тип на въпросите: един от много (single choice), много от много (multiple choice), подреждане на списък, подреждане в групи, свързване в последователност, свързване в съответствие, т.н.

Lecture 1

Current Trends

“While the volume of application development has grown steadily, so has the list of applications that people want to develop. It seems that the more programs we write, the more we understand what we need, and the more programs we add to the application-development backlog. So far, the rate of growth in application demand has continued to accelerate.”

WATTS S. HUMPHREY

Current Trends

- ☞ The growth in demand is increasing
 - ✓ Application programming is a very effective way to meet many kinds of human needs.
 - ✓ As long as people continue devising new and cleverer ways to work and to play, can be expected the growth in computer applications to continue.
 - ✓ The more we learn about computing systems and the more problems we solve, the better we understand how to address more complex and interesting problems.
 - ✓ The principal limitation on the expanding use of computing systems is our ability to find enough skilled people to meet the demands.

Broad Categories of The Software World

Applications

- ✓ By 'application programming' we refer to solving human problems with computing systems. Here the focus is on defining the problem and figuring out how to solve it with an automated system.

Systems

- ✓ Systems programming focuses on how to provide automated systems, aids, tools, and facilities to help people produce and run application programs.

Application Program Quality

- ☞ The quality of application software today is spotty at best:
 - ✓ a few programs are of high quality
 - ✓ many are downright bad
- ☞ The discussion has two principal parts:
 - ✓ Computers are being used for progressively more critical business applications.
 - ✓ The size and complexity of application programs is increasing.

The Defect Content of Programs

- ☞ The number of defects in a program increases:
 - ✓ linearly with its size
 - ✓ exponentially with its complexity
- ☞ The rate at which application program users experience problems is largely determined by the number of defects in a program rather than their density.

The Quality Problem

- ☞ We need to consider the fact that defective programs run:
 - ✓ When engineers produce a program and then run extensive tests, they generally can get it to work.
 - ✓ Unless the tests were comprehensive, the tested program will likely contain a great many defects.
 - ✓ Any testing process can only identify and fix the defects encountered in running those specific tests.

The Quality Problem

- ☞ Problems with size of application programs:
 - ✓ Will we be able to test these programs?
 - ✓ How well does the testing process scale up with program size?
- ☞ The answer is not encouraging
 - ✓ The number of tests required to achieve any given level of test coverage increases exponentially with program size.
 - ✓ The time it takes to find and fix each program defect increases somewhere between linearly and exponentially with program size.

The Quality Problem

Conclusions:

- ✓ The demand for quality is increasing.
- ✓ The testing process will not scale up with program size.
- ✓ Current software quality practices will not be adequate in the future.

The Development of Systems

- ❏ Future application programs will be vastly larger and more complex than they are today.
- ❏ The development of systems will change in a number of important ways.
 - ✓ The current cottage-industry approach to developing application programs must give way to a more professional and well-managed discipline.

The Staff

- ☞ Most programming groups will be seriously understaffed.
- ☞ There are only two ways:
 - ✓ Somehow increase the supply of new programmers:
 - ✗ to move more software offshore
 - ✗ to bring more software-skilled immigrants
 - ✓ Figure out how to write more programs without using more programmers.

The Staff

- ☞ Today we hire somebody who was reasonably familiar with the programming language of choice and expect him or her to rapidly become productive at developing application programs.
 - ✓ The required application knowledge and experience will increase as well.
 - ✓ The cost and time required to build application knowledge will be prohibitive.

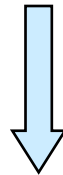
Reuse

- ☞ This is a way to produce more applications with fewer people.
- ☞ History indicates that reuse technology will be largely confined to building progressively larger libraries of language and system functions.
 - ✓ Unfortunately, this added language complexity will cause other problems.
 - ✓ It is just at too low a level to address the application needs of most users.

Reuse

The solution:

“Make software more hard and hardware more soft.”



The component development !

Application Categories

- ➡ Application development work in the future will likely involve three categories of work:
 - ✓ developing prepackaged applications that users can configure to their needs
 - ✓ configuration business systems to use prepackaged application systems
 - ✓ developing and supporting unique applications that cannot be created with prepackaged software

Packaged Applications

- ➡ Rather than custom-designing each application, this industry will increasingly develop families of reconfigurable application systems.
- ➡ Why ?
 - ✓ To handle the volume needs of many users.
- ➡ How ?
 - ✓ By producing essentially prepackaged application systems that can be configured in prescribed ways.
 - ✓ Systems have limited, but generic, capabilities.

The Objectives of Systems Programs

- ☞ The users need a secure, stable, reliable, fast and efficient operating system.
- ☞ The users need a system that provides the functions they need.
- ☞ The system must have a fixed and well-known application programming interface (API) so that many people can write applications to run on the system.

The Objectives of Systems Programs

☞ These programs must provide capabilities for:

- ✓ Multi-processing
- ✓ Multi-programming
- ✓ Distributed processing
- ✓ Interactive computing
- ✓ Continuous operation
- ✓ Dynamic recovery
- ✓ Security
- ✓ Usability
- ✓ Shared data
- ✓ Cooperative computing
- ✓ and much more.

The Trends of System Applications

- ☞ Users might decide to buy a new one for any of four reasons:
 - ✓ They are new computer users.
 - ✓ They need to replace their current computers and either the operating system they have will not run on the new computer or they can't buy a new computer without getting a new operating system.
 - ✓ They need a new version that fixes the defects in the old one.
 - ✓ They need functions that the new operating system provides and that they cannot get with the old system.

The Trends of System Applications

- ☞ How companies make the existing system obsolete:
 - ✓ Somehow tie the operating system to the specific computer on which it is initially installed.
 - ✓ Find defects or problems in the old version and fix them only in the new version.
 - ✓ Offer desirable new functions with the new version and ensure that these functions cannot be obtained by enhancing the old version.
- ☞ This problem suggests a basic business strategy:
 - ✓ Gradually expand the scope of the operating system to encompass new system-related functions.

The Kinds of Software Businesses

- ☞ All people expect the volume of application-development work to continue growing.
 - ✓ This will require people who know and understand various application domains and are also competent programmers.
 - ✓ Skill needs will span the full gamut from traditional business and accounting applications to embedded controllers that manage complex devices and processes.
- ☞ Systems developers will principally be concerned with developing, maintaining, and enhancing the operating and support systems needed by the application-development community.

The Groups of Software Businesses

- ☞ Three rather loosely defined groups:
 - ✓ the software houses like Microsoft and Oracle
 - ✓ the systems businesses like Apple, Sun, and IBM
 - ✓ the growing body of open-source programmers supporting systems like Linux

The Forces on Software Businesses

- ☞ There are many forces on software businesses
 - ✓ The functional content and complexity of systems is increasing rapidly, as is the size of the software parts of these systems.
 - ✓ Increasingly, software plays a central role in controlling and managing systems: it is a crucial part of the products and services in almost all industries.
 - ✓ Most computing systems will be interconnected.
 - ✓ We will see more internal and external threats to our systems. Now, in the interconnected world, these systems must work, be safe, and stay secure, even when exposed to attack by criminals and terrorists.

The Forces: Software Size and Complexity

- ☞ The principal concern with size and complexity is the scalability of the development process.
- ☞ To handle the massive increases in system scale, organizations must employ processes that scale up.
- ☞ We attempt to use the same methods and practices for programs with 1000 lines of code as for systems with 1000000 lines of code.

The Forces: Software Size and Complexity

Organizations that don't anticipate and prepare for scalability problems will someday find that they simply cannot get systems to work safely, reliably, and securely, no matter how much their people test and fix them.

The Forces: The Central Role of Software

- ☞ This is because much or even all of the functions that customers find attractive about modern products and services is embodied in their software.
- ☞ Many executives view software as a problem that they don't understand and have no idea how to manage.

The Forces: The Central Role of Software

When management subcontracts the technologies that make their products unique, they lose the ability to manage their future.

The Forces: Interconnectedness

- ☞ The value of a computing system is increasingly determined by the number of other systems to which it can connect.
- ☞ The keys to broad market acceptance are compatibility, interoperability, and interchangeability.
- ☞ We will need to think in interconnected ways and to remove any limitations that make it hard to interconnect and to interoperate our systems.

The Forces: Interconnectedness

*It is precisely this ability of our systems
to be used in new and innovative ways
that will make them attractive to the
users of the future.*

The Forces: Real-World Threats

- ☞ With stand-alone systems, our exposure to the realities of a dangerous and unpleasant world was limited. But with the Internet, and with the growing interconnectedness of our systems, this is no longer the case.
- ☞ People seek to recover damages from the organizations that built unsafe or insecure systems.

The Forces: Real-World Threats

Secure and safe systems will be an economic necessity, and users will band together to seek damages from suppliers who don't provide such systems.

What is a Software Process?

- ➡ A set of activities whose goal is the development or evolution of software
- ➡ Generic activities in all software processes are:
 - ✓ Specification
 - ✓ Development
 - ✓ Validation
 - ✓ Evolution

What Is a Software Process Model?

- ☞ A simplified representation of a software process, presented from a specific perspective.
- ☞ Examples of process perspectives are
 - ✓ Workflow perspective - sequence of activities
 - ✓ Data-flow perspective - information flow
 - ✓ Role/action perspective - who does what

What Is a Software Process Model?

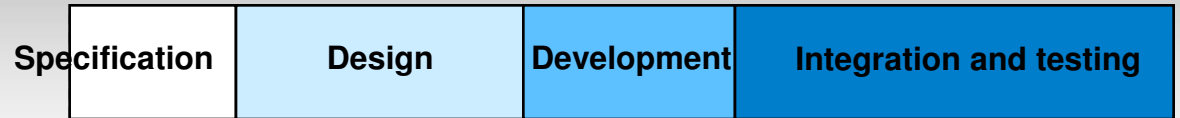
- ➡ A simplified representation of a software process, presented from a specific perspective.
- ➡ Generic process models
 - ✓ Waterfall
 - ✓ Iterative development
 - ✓ Component-based software engineering

What Are The Costs of Software Engineering?

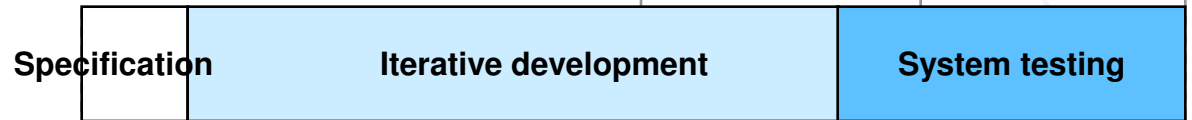
- ☞ Evolution costs often exceed development costs
 - ✓ Custom built vs. mass market software
- ☞ Costs vary depending on
 - ✓ the type of system being developed
 - ✓ the requirements of system attributes such as performance and system reliability
 - ✓ The experience of the development team(s)
- ☞ Distribution of costs depends on the development model that is used.

Activity Cost Distribution

☞ Waterfall



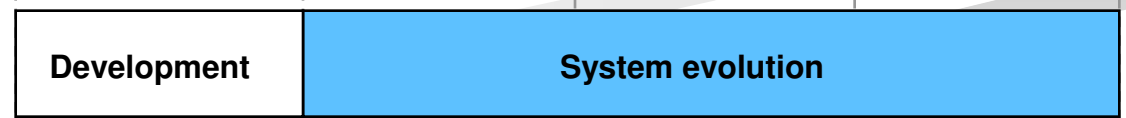
☞ Iteration development



☞ Component-based development



☞ Long-lifetime systems



0

25

50

75

100

Software Engineering

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

FAQs about Software Engineering

- ➡ What is software engineering?
- ➡ What is the difference between software engineering and computer science?
- ➡ What is the difference between software engineering and system engineering?
- ➡ What is a software process?
- ➡ What is a software process model?

FAQs about Software Engineering

- ➡ What are the costs of software engineering?
- ➡ What are software engineering methods?
- ➡ What is CASE (Computer-Aided Software Engineering)
- ➡ What are the attributes of good software?
- ➡ What are the key challenges facing software engineering?

What Is Software?

- ➡ Computer programs and associated documentation such as requirements, design models and user manuals
- ➡ Software products may be developed for a particular customer or may be developed for a general market
- ➡ Software products may be
 - ✓ Generic
 - ✓ Custom
- ➡ New software can be created by developing new programs, configuring generic software systems or reusing existing software

What Is Software Engineering?

- ☞ Software engineering is an engineering discipline that is concerned with all aspects of software production
- ☞ Software engineers should adopt a systematic and organised approach to their work and use appropriate tools and techniques depending on
 - ✓ the problem to be solved
 - ✓ the development constraints
 - ✓ the resources available

What Is Software Engineering?

A key software engineering “axiom”

- Better
- Cheaper
- Faster



pick any two

Software Engineering vs. Computer Science

Computer science

- ✓ It is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software
- ✓ Its theories are still insufficient to act as a complete underpinning for software engineering (unlike e.g. physics and electrical engineering)

Software Engineering vs. System Engineering

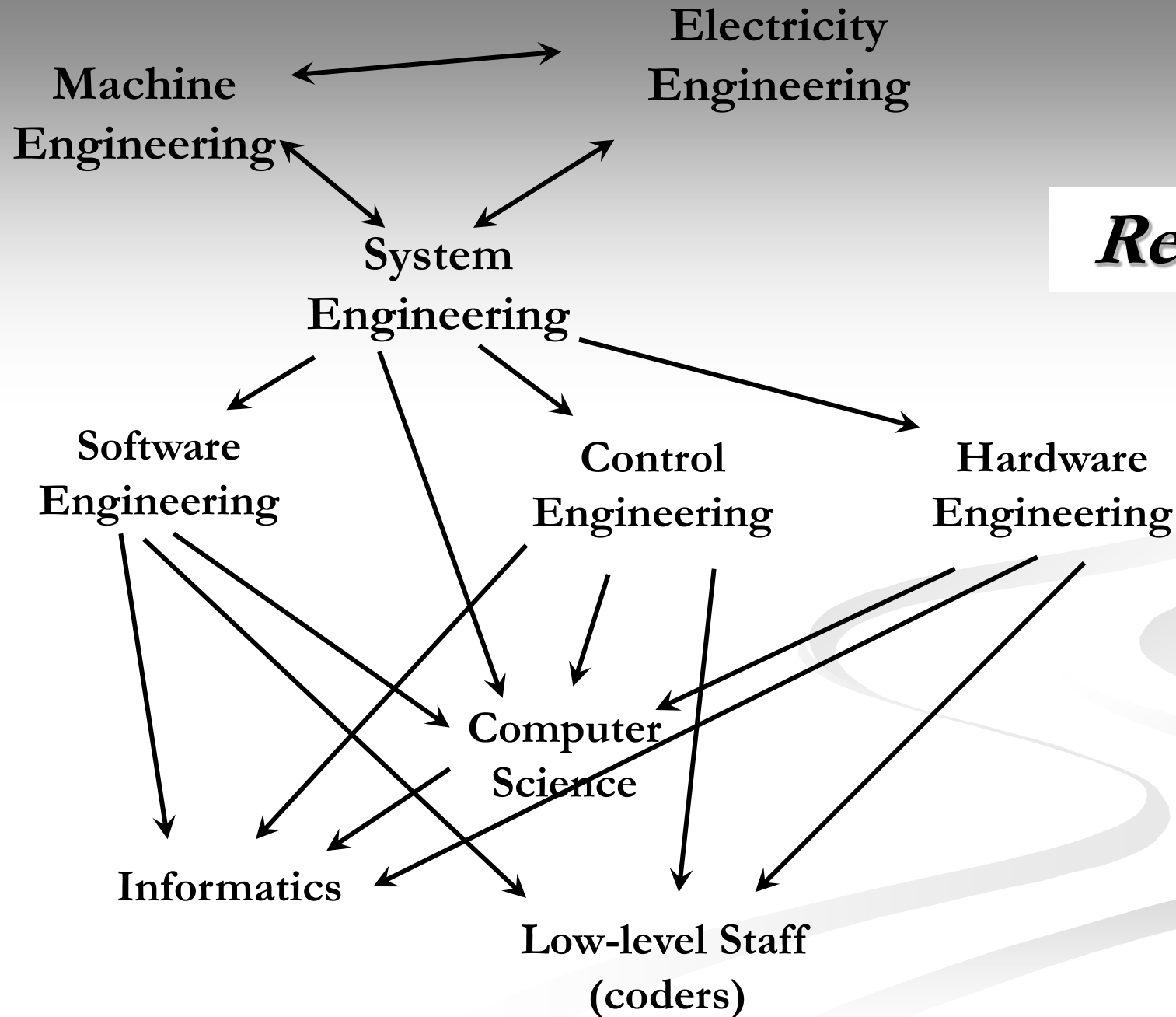
System engineering

- ✓ It is concerned with all aspects of computer-based systems development including hardware, software and process engineering.
- ✓ It is involved in system specification, architectural design, integration and deployment



Software engineering is part of this process concerned with developing the software infrastructure, control, applications and databases in the system

Relations



Cost of not Engineering Software

- ☞ Need to restructure code
- ☞ High maintenance costs
- ☞ Software is unacceptable to end users
- ☞ Unreliable code
- ☞ Need to rewrite code
- ☞ Difficulty integrating system modules
- ☞ Difficult project management
- ☞ Budget and time overruns

What Are Software Engineering Methods?

- ➡ Structured approaches to software development which include system models, notations, rules, design advice and process guidance.
- ➡ Model descriptions
 - ✓ Descriptions of graphical models which should be produced
- ➡ Rules
 - ✓ Constraints applied to system models
- ➡ Recommendations
 - ✓ Advice on good design practice
- ➡ Process guidance
 - ✓ What activities to follow

What is CASE

- ➡ CASE -> Computer-Aided Software Engineering
- ➡ Software systems that are intended to provide automated support for software process activities.
- ➡ CASE systems are often used for method support.
- ➡ Types:
 - ✓ Upper level: Tools to support the early process activities of requirements and design
 - ✓ Lower level: Tools to support later activities such as programming, debugging and testing

What Are The Attributes of Good Software?

- ☞ The software should deliver the required functionality and performance to the user and should be maintainable, dependable and acceptable.
- ☞ Maintainability
 - ✓ Software must evolve to meet changing needs
- ☞ Dependability
 - ✓ Software must be trustworthy
- ☞ Efficiency
 - ✓ Software should not make wasteful use of system resources
- ☞ Acceptability
 - ✓ Software must accepted by the users for which it was designed. This means it must be understandable, usable and compatible with other systems.

What Are The Key Challenges Facing Software Engineering?

Heterogeneity

- ✓ Developing techniques for building software that can cope with heterogeneous platforms and execution environments;

Delivery

- ✓ Developing techniques that lead to faster delivery of software;

Trust

- ✓ Developing techniques that demonstrate that software can be trusted by its users.

Why Software Engineering is not Universal

- ☞ Understanding software development as programming only without recognition of importance of analysis and design
- ☞ Short-sighted technical management
- ☞ Pressure of deadlines
- ☞ Poor project estimation, thus unreasonable deadlines

Working in Teams

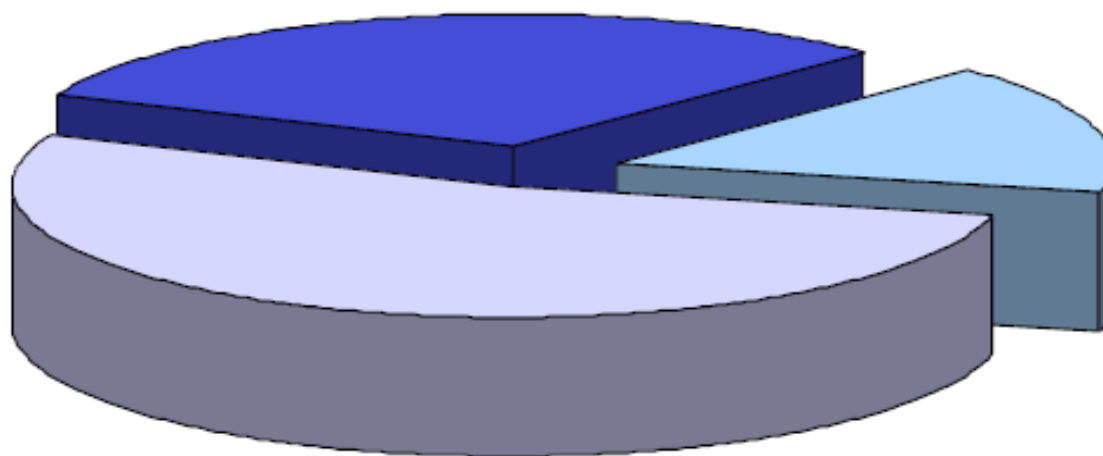
- ➡ Be conscientious about due dates
- ➡ If you are less motivated by the class project, negotiate an equitable grade and take a lesser role
- ➡ Meet regularly with your team
- ➡ Always create an agenda for every team meeting
- ➡ Rotate responsibility for chairing team meetings

Questions ?

Качеството на разработваното ПО

Софтуерни проекти

31 %
прекратени



16 %
успешно
завършили

51 %
спорни

Провалени проекти

👉 Бюджет

под 20 %	15.5%
21-50 %	31.5 %
51-100 %	29.6 %
101-200 %	10.2 %
201-400 %	8.8 %
над 400 %	4.4 %

*Средната себестойност
на провалениите е 189 %
от предвидената*

Провалени проекти

Продължителност

Under 20%	13.9%
21 - 50%	18.3%
51 - 100%	20.0%
101 - 200%	35.5%
201 - 400%	11.2%
Over 400%	1.1%

Средната продължителност на проваленията е 222 % от предвидената

под 20 %	13.9 %
21-50 %	18.3 %
51-100 %	20.0 %
101-200 %	35.5 %
201-400 %	11.2 %
над 400 %	1.1 %

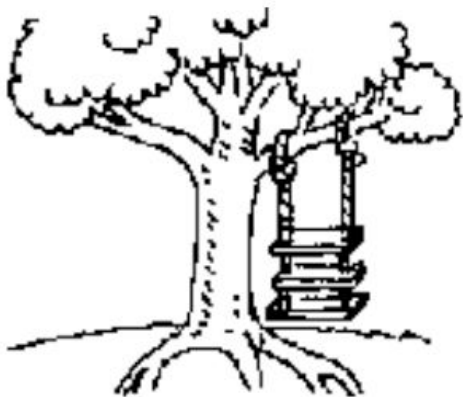
Провалени проекти

☞ Реализирана функционалност

Средното изпълнение при спорните проекти е 61 % от заданието.

под 25 %	4.6 %
25-49 %	27.2 %
50-74 %	21.8 %
75-99 %	39.1 %
100 %	7.3 %

Основен проблем



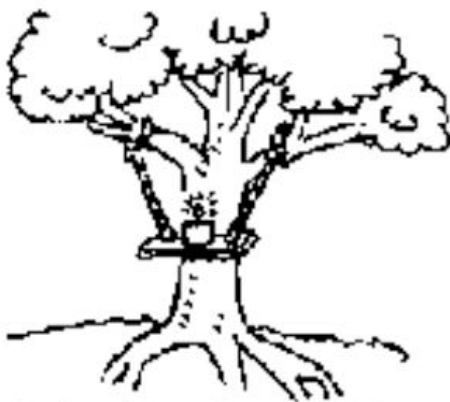
Това е разбрано, че
ще се разработва



Задание в проекта



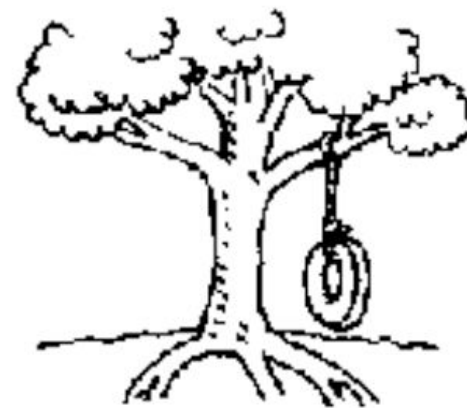
Програмен дизайн



Разработка



Инсталирана версия



Желание на клиента

Основен проблем

- ☞ По малко от половината ръководители на проекти смятат, че качеството на програмните продукти се подобрява.

Значително повече повреди (failures)	27%
Повече повреди (failures)	21 %
Без промяна	11 %
По-малко повреди (failures)	19 %
Значително по-малко повреди (failures)	22 %

Качество на ПО

☞ Под качество на ПО хората разбират две неща:

- ✓ Програмният продукт трябва да прави това, за което е предназначен и нищо друго (“трябва да прави правилните неща”)
- ✓ При да извършва дейностите си коректно и качествено (“трябва да работи правилно”).

Качество на ПО

☞ Дейности по за създаване на качествено ПО:

- ✓ Планиране на качеството
- ✓ Изпълнение на дейностите по валидация и/или верификация на разработваното ПО
- ✓ Измерване и анализ на работата на ПО за установяване на данни, показващи съответното ниво на качество

Стремeж към все по-големи ПС

☞ Това са системи с поне 1 млн. реда и имащи висока сложност:

- ✓ Разработват се голям брой хора.
- ✓ Разработката е в дълъг период от време.

☞ Основни проблеми

- ✓ Безразборно добавяне на функционалност, която често е несъществена.
- ✓ Лош и даже 'глупав' дизайн.
- ✓ Неправилен избор на език за програмиране.
- ✓ Несъстоятелен подбор на технологии за разработка.

Подобряване на качеството

- ☞ Дейностите за подобряване на качеството на разработката са насочени към
 - ✓ Предпазва от появата на някои класове проблеми/повреди.
 - ✓ Премахва възможността от възникване на някои класове проблеми/повреди.
 - ✓ Намалява вероятността и прави по-трудна появата на проблеми/проблеми с използваното ПО.

Управление на дейностите за подобряване на качеството

Управление на дейностите за подобряване на качеството на ПС (software quality engineering)

Техники и дейности за подобряване на качеството на ПС (quality assurance)

- Инспекции
- Отказоустойчивост
- Превенция на дефектите
- Формални методи
- Други

Тестване на ПС

Качество на ПО

- ☞ Алтернативният въпрос е:
 - ✓ “Какви са характеристиките на висококачественото ПО ?”
- ☞ За дефинирането на тези характеристики е необходимо да се отчитат различни гледни точки и очаквания:
 - ✓ Потребители
 - ✓ Разработчици
 - ✓ Ръководители на проекти
 - ✓ Специалисти по маркетинг на ПО
 - ✓ Мениджъри на софтуерни фирми
 - ✓ Специалисти по поддръжка и съпровождане на ПО

Качество на ПО

☞ Прието е да се използват пет основни гледни точки, дефиниращи характеристиките за качество:

- ✓ Абстрактни критерии
- ✓ Потребителско разбиране
- ✓ Производствено разбиране
- ✓ Продуктова същност
- ✓ Стойностно-базирани критерии

Качество на ПО

☞ Според ролите си по отношение на разработваното ПО съществуват две основни групи:

✓ Клиенти

- ✗ Купувачи (customer) –избират какво и защо да се купи
- ✗ Потребители (user) – само използват вече закупено/осигурено ПО
 - В тази група влизат и ‘нехора’-потребители, т.е. друго ПО или апаратна част, взаимодействащи с разработваното ПО

✓ Производители (producers)

- ✗ Директно участващите в разработката (дизайнери, програмисти, ръководители и др.)
- ✗ Хората от трети фирми, подпомагащи разработката, поддръжката и съпровождането на ПО

Купувачи и Качество

- ➡ Основното очакване е ПО да реализира в действителност полезните функции, които казва че реализира:
 - ✓ Функциите отговарят на нуждите на покупателите
 - ✓ При многократна употреба и/или в дълъг период от време се запазва полезността и правилното функциониране на ПО – това е известно като издръжливост на ПО (reliability)

Киленти и Качество

☞ Специфични изисквания на потребителите

- ✓ Сложността на използването може да е много по-важна характеристика за много потребители – това е известно като ‘използваемост’ (usability)
- ✓ Сложността на инсталирането и поддръжката е друга важна характеристика за много хора
- ✓ При групата на ‘нехората’-потребители много важни са адаптивността (adaptability) и възможността за взаимодействие в една по-голяма система (interoperability)

☞ Специфични изисквания на купувачи

- ✓ цена

Производители и Качество

Основната оценка е:

- ✓ Дали разработеното ПО е това, което е поръчано, т.е. дали изпълнява всички изисквания, заложиени в спецификацията на проекта.
- ✓ Колкото по-лесно това се доказва, толкова по-качествено е разработено ПО.

Допълнително изискване:

- ✓ Купувачите пак да се върнат при нас.
- ✓ Да мога да поема друг проект след завършването на текущия.

Производители и Качество

- ☞ Специфични изисквания на потребителите
 - ✓ Сложността на използването може да е много по-важна характеристика за много потребители при съставяне на мнение за ПО
 - ✓ Сложността на инсталирането и поддръжката е друга важна характеристика за много хора
 - ✓ При групата на 'нехората-потребители' много важни са адаптивността (adaptability), възможността за взаимодействие в една по-голяма система (interoperability)
- ☞ Специфични изисквания на купувачи
 - ✓ цена

ISO-9126

- ☞ Стандарт за качество на ПО, дефиниращ 6 характеристики за качество:
- ✓ Функционални: годност (suitability), точност на разработката (accuracy), съвместимост (interoperability), сигурност (security)
 - ✓ Надеждност: работоспособност (maturity), устойчивост на откази (fault tolerance), възстановимост (recoverability)
 - ✓ Използваемост: разбираемост на ПО (understandability), научаемост (learnability), леснота на опериране със ПО (operability)

ISO-9126

☞ Стандарт за качество на ПО, дефиниращ 6 характеристики за качество:

- ✓ Ефективност: измерва връзка ‘функционалност-бързодействие-ресурси’
- ✓ Възможност за поддръжка (maintainability): анализируемост, променяемост (changeability), устойчивост (stability), възможност за тестване на промените (testability)
- ✓ Преносимост: адаптивност, инсталируемост, съответствие между инсталираните версии (conformance), заменяемост на версиите (replaceability)

Други стандарти за качество

☞ CUPRIMDS

- ✓ Capability, Usability, Performance, Reliability, Installation, Maintenance, Documentation, Service
- ✓ Използва се в IBM

☞ Web-базирани приложения

- ✓ Първични характеристики: издръжливост, използваемост и сигурност
- ✓ Вторични характеристики: достъпност (availability), мащабируемост (scalability), възможност за поддръжка и времето за излизане на пазара

☞ Ежедневно използвано ПО

- ✓ Основните критерии в този случай са коректността и съответствие със заданието

Дефекти на ПО (defects)

Повреда (failure)

- ✓ Невъзможност на система или компонент да реализират заложена функционалност според зададени технически характеристики
- ✓ Отнася се за отклонение в поведението на ПО спрямо заданието за разработка

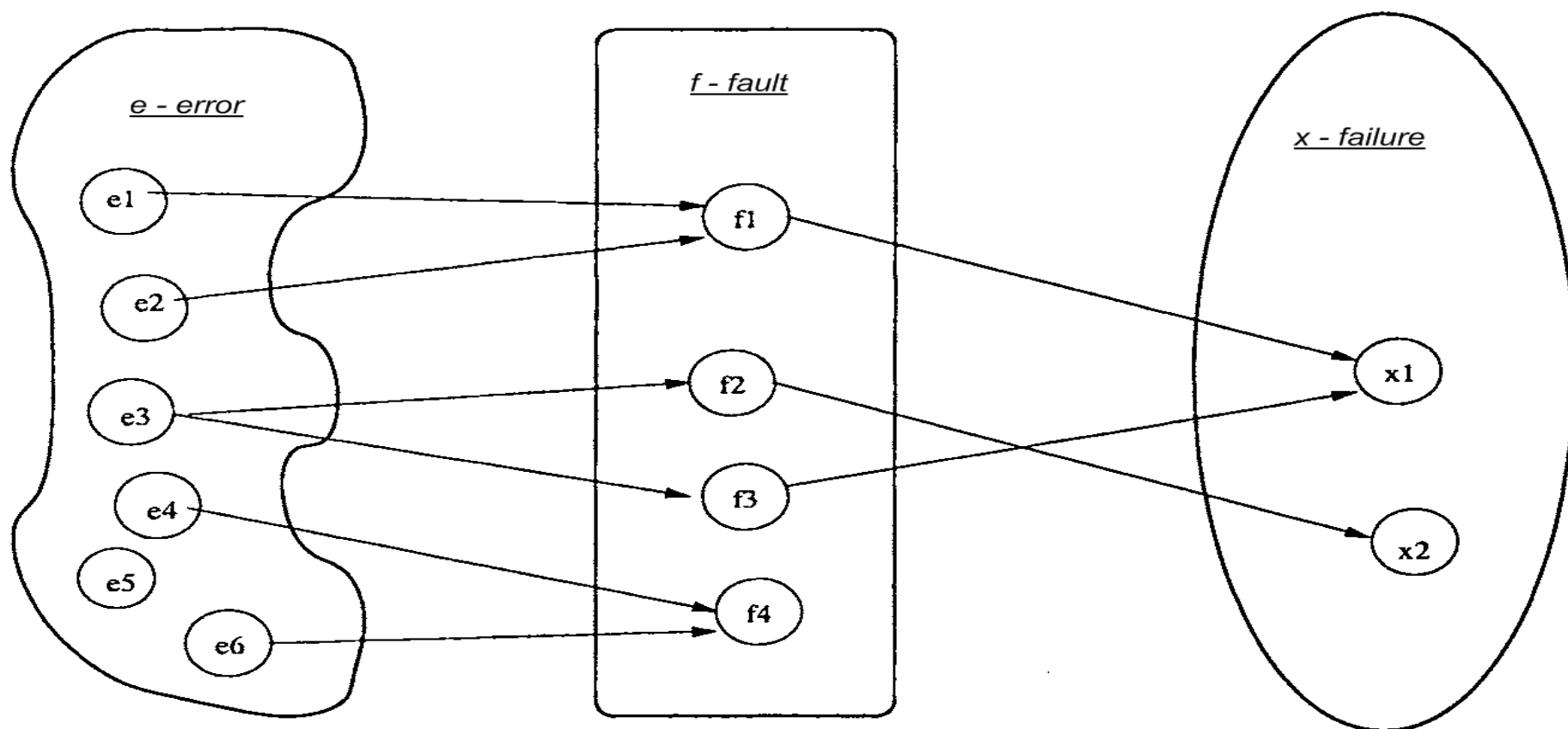
Дефект/опущение (fault)

- ✓ Неправилен процес, стъпка или дефиниция на данни в компютърна програма.
- ✓ Отнася се към наличието на специфични условия в ПО, позволяващи проявлението на дадена повреда.

Грешка (error)

- ✓ Човешка дейност, водеща до получаване на некоректни резултати.
- ✓ Отнася се до отсъстващи или неправилни човешки действия, водещи до появата/залагането на дефекти в ПО.

Недостатъци на ПО (defects)



Вход за разработка
на ПО

Разработено ПО

Сценарии за използване
и резултати от работата
на ПО

Коректността като основа на качеството

Дефиниране

- ✓ От гледна точка на клиентите е свързана с външното проявление на ПО, т.е. наличието на повреди.
 - ✗ Оказва влияние върху използваемостта, преносимостта, възможността за инсталиране, поддръжката и т.н.
- ✓ От гледна точка на производителите е свързана с влиянието на вътрешните особености на ПО, т.е. наличието на дефекти (faults).
 - ✗ Свързана е характеристики като дизайн, размер, сложност и т.н.

Клиентите и Коректността

- ☞ Коректността може да се измерва чрез различни техники за следене на повредите:
 - ✓ Описват се типа и същността на проявлението
 - ✓ Измерват се чрез анализ на броя, разпространението, честотата на проявление.
- ☞ Издръжливостта (reliability) пряко зависи от коректността, по-точно от честотата на проявление на повреди
 - ✓ Дефинира се като време за работа без поява на повреди или като работа при определени входни данни без проява на повреди
- ☞ Безопасността на ПО (safety) също зависи от коректността
 - ✓ Дефинира се като характеристика на ПО, което не води до инциденти при определено ниво на опасност от настъпилата повреда.
 - ✗ Инцидентът е повреда в ПО с тежки последствия за клиента.

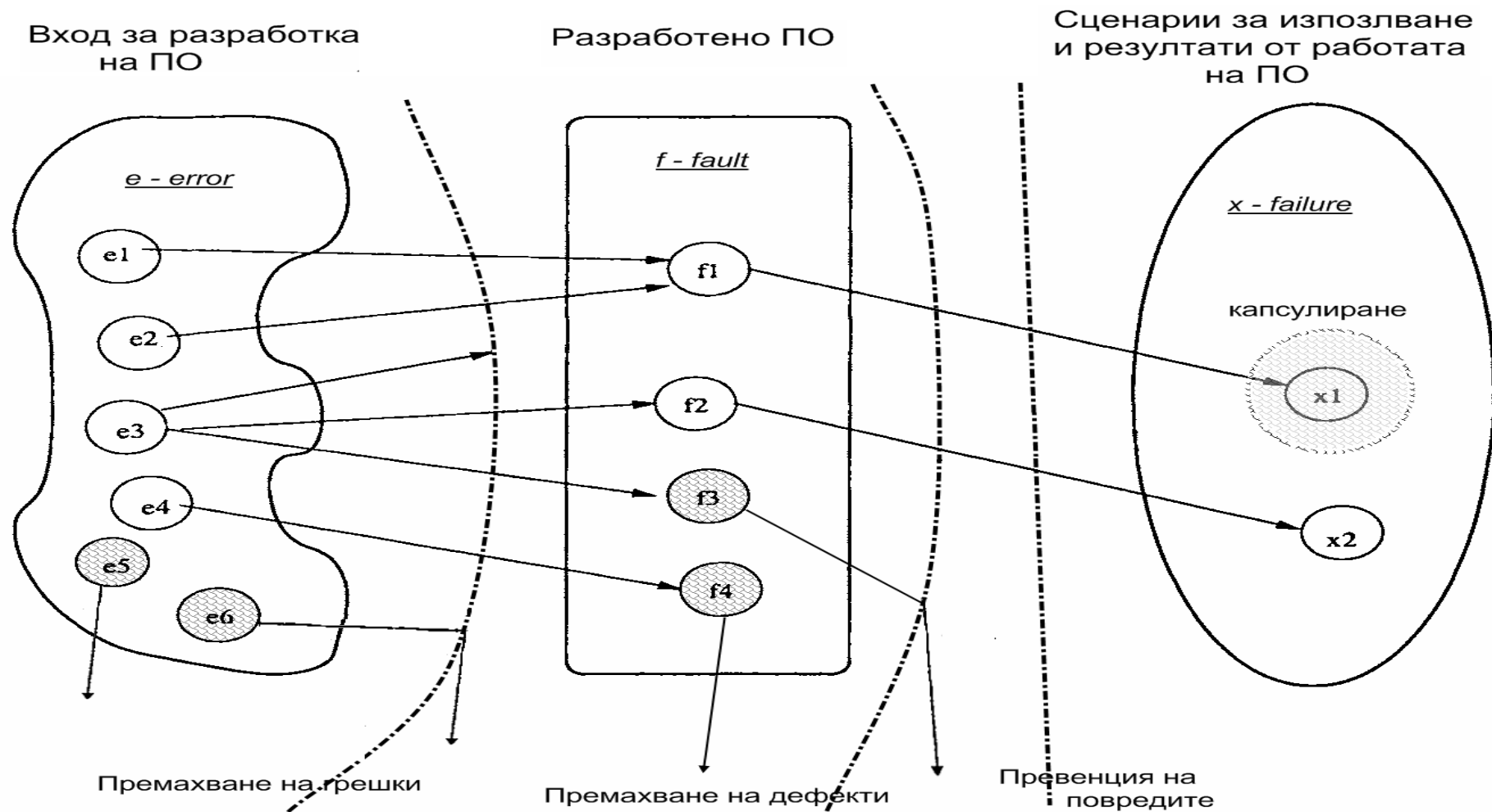
Производители и Коректността

- ☞ Използват се техники и методи за измерването и анализ на дефектите в разработката на ПО:
 - ✓ Ниво индивидуални анализ:
 - ✗ Тип, връзка с определени повреди и инциденти, причини, време и причини за тяхната поява в ПО и др.
 - ✓ Ниво на групов анализ
 - ✗ Изследват се разпространението и гъстотата на дефектите в различните фази на разработка.
 - ✗ Изследват се разпространението и гъстотата на дефектите в различните подсистеми и/или компоненти на ПО.
- ☞ Разработват се специални планове, съдържащи мерки по фази и компоненти за подобряване на коректността, т.е. на качеството на ПО.

Сигурност в качеството на разработката на ПО

- ☞ Сигурността в качеството на разработката на ПО (quality assurance - QA) се дефинира като:
 - ✓ отношение към наличието на дефекти в ПО и техниките за тяхното ограничаване/премахване.
- ☞ Съществуват три генерични категории на проявление на QA:
 - ✓ Превенция на дефектите чрез блокиране на грешките или тяхното премахване
 - ✓ Редуциране на дефектите чрез разпознаване на дефектите и последващото им премахване
 - ✓ Ограничаване на влиянието на повредите чрез превенция и капсулиране (ограничаване на проявленията)

Сигурност в качеството на разработката на ПО



Превенция на дефектите

☞ Обучение на хората

- ✓ Много източници на грешки са резултат от незнание или неразбиране

☞ Формални методи

- ✓ Осигуряват определянето на местата с лош дизайн и/или реализация

☞ Използването на стандарти и добри практики при разработката

- ✓ Силно се ограничавам възможностите и местата за вмъкване (инжектиране) на грешки в разработваното ПО.

☞ Използване на подходящи и сигурни средства за разработка

- ✓ Намалява се възможността за добавяне на грешки от работата на използваното ПО при разработката.

Редукция на дефектите

☞ Инспекция

- ✓ За директно откриване и премахване на дефекти

☞ Тестване

- ✓ За наблюдение на повредите като начин за намиране и премахване на дефектите

☞ Други методи

- ✓ Това са методи за анализ на ПО след създаването му

Инспекция

- ☞ Това е статичен метод за анализ, базиращ се на:
 - ✓ Критично четене на различните документи по създаване на ПО (изисквания, дизайн, код, т.н.).
 - ✓ Реализира се на последователни и координирани сесии от екип от специално обучени специалисти, работещи координирано.
 - ✓ Дефектите се определят директно при провежданите инспекции
 - ✓ Откритите грешки се премахват и се извършва повторна инспекция за поява на нови.
 - ✓ Структурата и същността на процеса силно варират от обикновено преминаване през документите до формализиране и анализ на инспектирания документ.
 - ✓ Процесът задължително се планира като дейности и насоки.

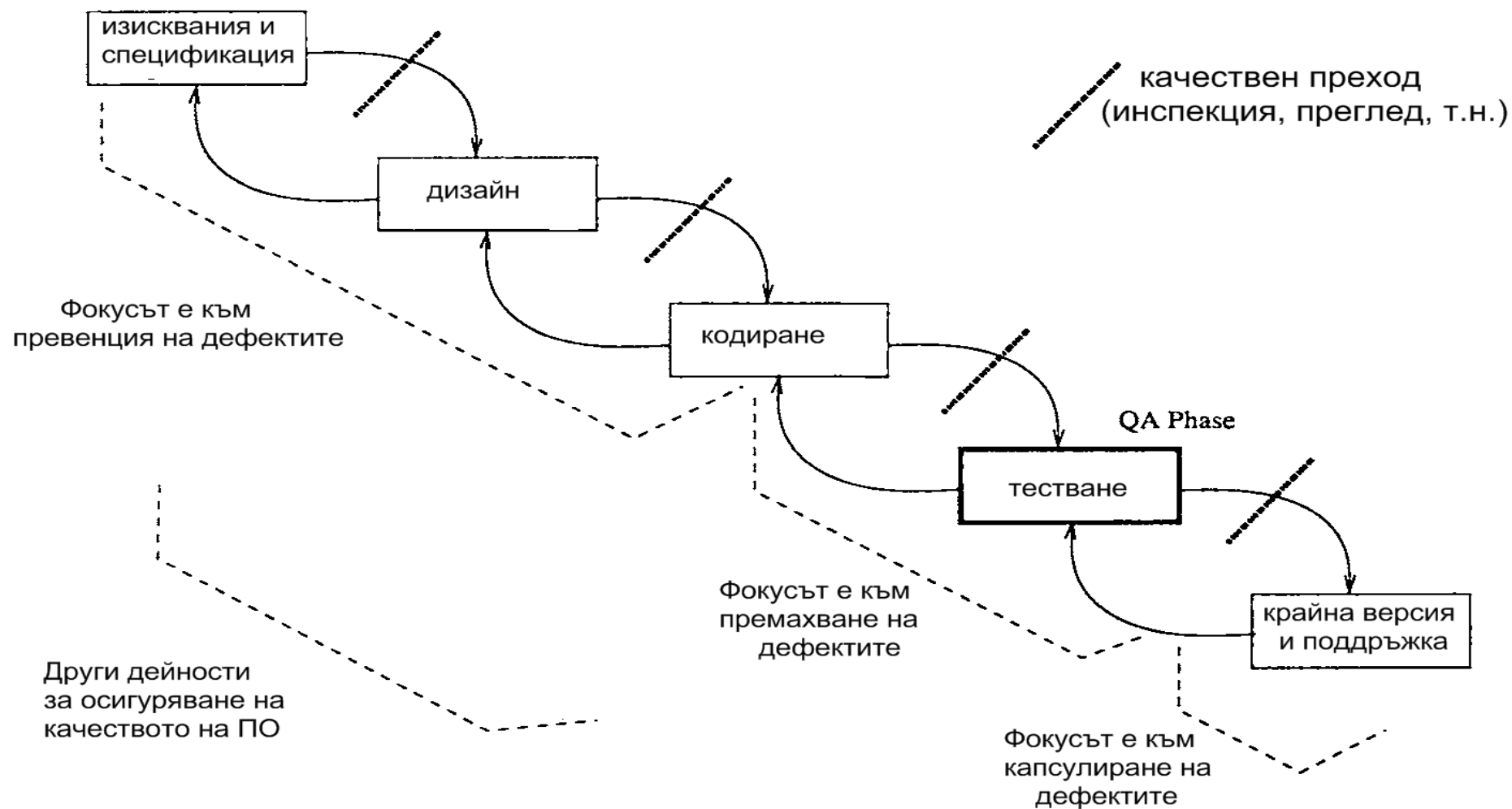
Тестване

- ☞ Кога дефекти трябва да се определят чрез тестване ?
- ☞ Какво да се тества и какъв вид дефекти се намират ?
- ☞ При какво ниво на открити дефекти да се спира тестването ?

Осигуряване на качеството в процесите на разработка

- ☞ Управление на дефектите (defect handling)
 - ✓ Документиране на проявата и начално описание на дефекта (defect logging)
 - ✓ Следене и описание на проявлението на дефекта след началното установяване до крайното премахване (defect tracking)
- ☞ Дейности по управление на дефектите
 - ✓ Анализ на дефектите – дефекти се обработват по групи (вътрешни грешки, грешки на потребителя, други) за да има консистентност по типовете грешки
 - ✓ Периодично наблюдение и описание на състоянието на всички установени дефекти.

Пример



Валидация и качеството на ПО

☞ Валидацията е насочена към установяване на съответствието между функционалността на ПО и очакването на клиентите

- ✓ Проверява дали е реализирана функционалността
- ✓ Проверява дали има отклонение в поведението на ПО спрямо описаното в заданието
- ✓ Проверява как повредите влияят върху възможността за използване на заложената функционалност.

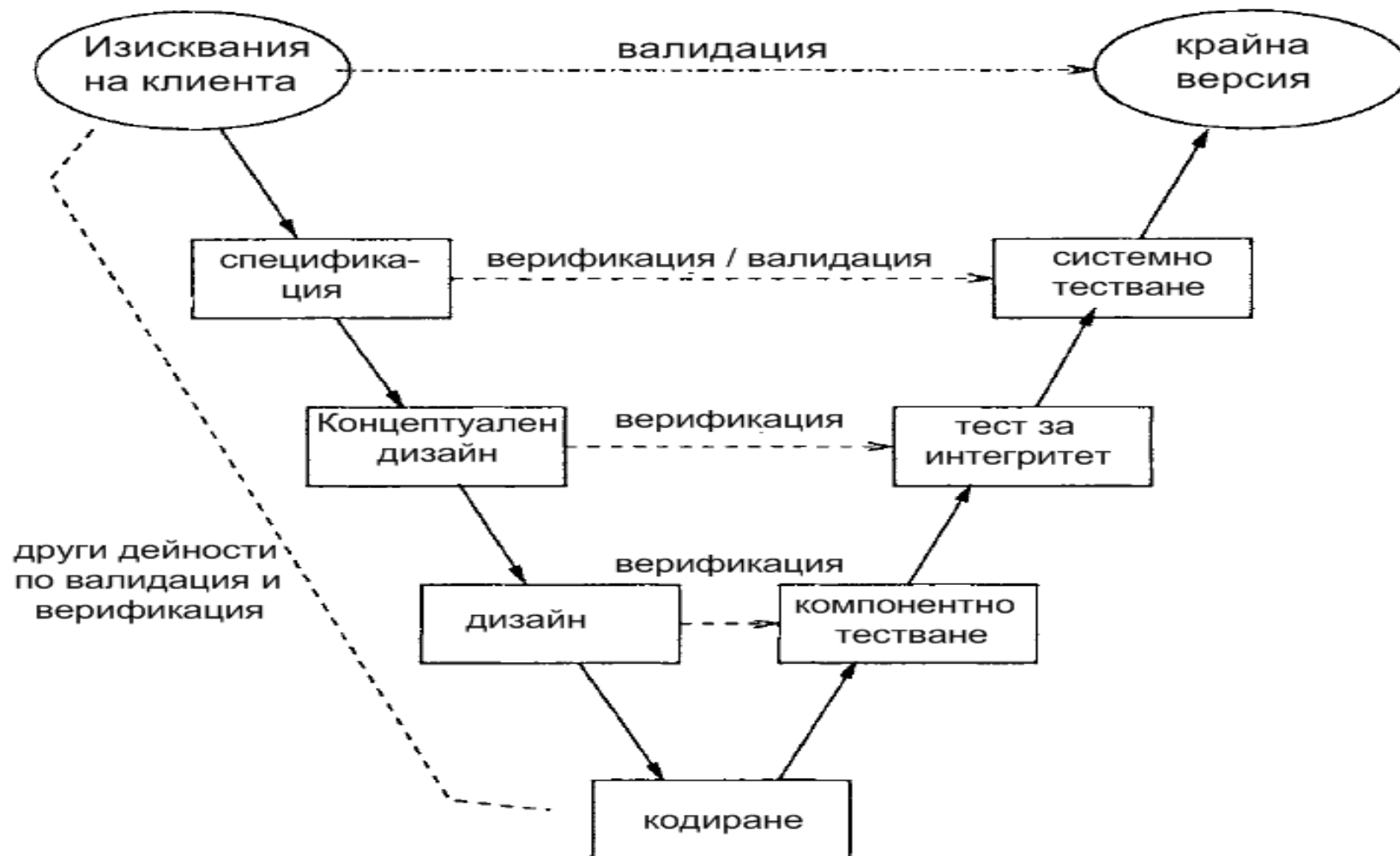
Валидация и качеството на ПО

- ☞ Валидационни дейности, отговарящи за качеството на ПО
 - ✓ Системно тестване (system testing)
 - ✓ Тестване за допустимост (acceptance testing)
 - ✓ Статистически анализ на използваемостта на заложената функционалност
 - ✓ Установяване на дефектоустойчивост (fault tolerance).
 - ✓ Установяване на сигурността на ПО (safety assurance)

Верификация и качеството на ПО

- ☞ Верификацията проверяват дали реализацията е извършена по зададените спецификации
 - ✓ При наличието на повреди се проверяват причините за изменения в поведението на ПО
 - ✓ При други видове проблеми се проверява за грешки и вкарани дефекти.

Валидация и Верификация на ПО като част от качеството на разработката



Questions ?

Основни принципи на V&V

Методи и стратегии на реализация
на процесите на V&V

Възникване на V&V

☞ Валидацията и верификацията (V&V) са резултат от развитие на дейностите по осигуряване и управление на качеството на разработвания продукт.

- ✓ Развитието на системите и методите за управление на процесите на управление на разработката периодично водят до промяна на основната насока и задачи на дейностите по V&V.
- ✓ Появата на нови приложения на ПО води и до необходимостта от разширяване (по-рядко предефиниране) на тези понятия.
- ✓ Това води до съществуващата нееднозначност в разбирането за тези процеси.

Понятието ‘Система’

- ☞ Този термин възниква в 20 век като резултат от развитието на теорията за системите на Bertalanffy.
- ☞ Интуитивно разбиране
 - ✓ Тя се състои от отделни елементи, организирани по някакъв начин в едно цяло, имащи зададена интерфейс помежду си.
 - ✓ Работата на системата създава резултат, който не може да се получи като елементарно сумиране на резултатите от работата на отделните елементи.
 - ✓ Всяка система влияе на заобикалящата я среда, както и се влияе от тази среда – задължително се дефинира връзката между системата и заобикалящата я среда
 - ✗ Формите на вход-изход могат да бъдат най-различни (материя, енергия, информация).

Инженерна система

- ☞ Целта на инженерния процес е разработката на ефективни и надеждни системи (продукти, услуги, решения), отговарящи на определени нужди, които се дефинират чрез задаване на множество от ограничения.
- ☞ Класическият цикъл на живот за всяка инженерна система включва следните фаза: дефиниране, дизайн, реализиране, интеграция, окачествяване и производство.

Инженерна система

Основни характеристики

- ✓ Търсене на единично решение, базиране на уникален дизайн и ориентирано към специфичен проблем.
- ✓ Поведението на системата трябва да е предвидимо и да бъде прецизно описано.
- ✓ Подходът за реализация е ‘отгоре-надолу’, базиран на фундаменталното разбиране, че всяка система може да бъде описана цялостно чрез описание на поведението на нейните части, връзките им и взаимното им влияние.

Инженерна система

Основни атрибути:

- ✓ Предсказуемост: във всяка ситуация системата работи по предсказуем начин.
- ✓ Надеждност: в зададен период от време и при зададени условия на работа системата изпълнява изискваните дейности.
- ✓ Прозрачност: структурата на системата и на процесите и може да се опише експлицитно.
- ✓ Управляемост: системата може да се управлява директно според зададените инструкции при зададените условия на работа.

Инженерна дефиниция за система

“Система е множество от различни елементи, които заедно създават резултат, който не може да се получи от самостоятелните елементи. Елементи (части на системата) могат да включват хора, апаратура, програмно осигуряване, средства, политики, документи. И всички тези елементи са необходими за получаване на необходимите резултати. Тези резултати включват: свойства, характеристики, функции, поведение, действия. Ползата от системата, като цяло, която се получава извън полезността на всеки от елементите поотделно, основно се получава от връзката между отделните части, т.е. от начина на свързването им.”

INCOSE

V&V – основна цел

Елиминиране на дефектите

или

Доказване, че няма дефекти

V&V – основна цел

☞ Елиминирание на дефектите

- ✓ Необходимо е да се премахнат колкото е възможно повече дефекти при ограничението за пари, време, ресурси (апаратура, хора, т.н.).
- ✓ Основен подход не само за ПО, но и за всички инженерни проекти - позволява да се оцени и подобри качеството на продукта.

☞ Доказване, че няма дефекти

- ✓ Много необходимо от маркетингова гледна точка, но неправилно от инженерна.
- ✓ В болшинството от случаите е невъзможна за реализация – изключителни обеми от работа с висока сложност и необходимост от безкрайни ресурси.

V&V – основна цел

Елиминиране на дефектите

или

Доказване, че няма дефекти

Атрибути на качеството

☞ За оценка на процеса на V&V се използват следните атрибути:

- ✓ Коректност (correctness)
 - ✗ Ако входните данни са валидни, ПО ще работи ли както се очаква ?
- ✓ Пълното (completeness)
 - ✗ Реализира ли ПО всички изисквания, поставени за него ?
- ✓ Консистентност (consistency)
 - ✗ По еднакъв начин ли работи ПО в идентични ситуации ?
Аналогичното ПО от дадената фамилия също ли има този начин на работа в тези ситуации ?
- ✓ Надеждност (reliability)
 - ✗ Във всички случаи ли системата има очакваното поведение, даже и при силно променени условия ?

Атрибути на качеството

☞ За оценка на процеса на V&V се използват следните атрибути:

- ✓ Полезност (usefulness)
 - ✗ ПО дава ли на потребителите полезни услуги ?
- ✓ Използваемост (usability)
 - ✗ Удобна ли е системата за употреба ?
- ✓ Ефективност (efficiency)
 - ✗ Ефективно ли използва програмното осигуряване ресурси като време, памет, периферия, комуникационни канали и други ?
- ✓ Сполучливост (effectiveness)
 - ✗ От гледна точка на вложенияте пари това едно ефективно решение ли е ?

Същност на V&V

☞ Верификация

- ✓ Тя има отношение към писаните спецификации на системата
- ✓ Тя включва коректността на вътрешната структура на ПО
- ✓ Свързана е със процесите на разработка от целия цикъл на живот за ПО

☞ Валидация

- ✓ Използва се за ^{unbiased} сравняване на системата от изисквания на клиентите (stakeholders), които могат да се променят във времето.
- ✓ За реализацията и е необходимо наличие на формално или неформално специфициране на множеството от желания на клиентите за функционалност на разработваното ПО преди началото на процеса.

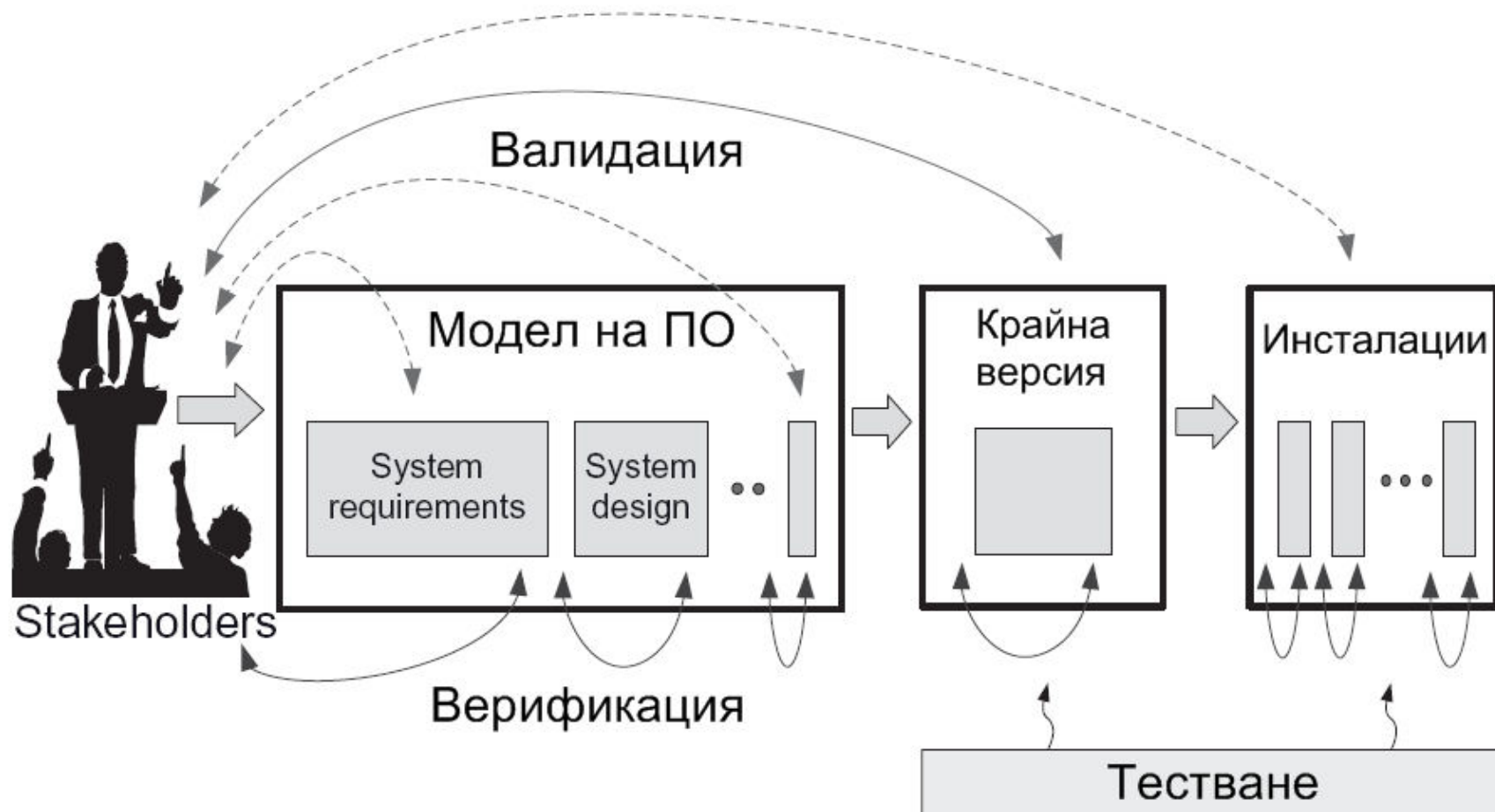
☞ Тестване

- ✓ То включва някои видове проверки на ПО – това са статични или динамични методи за оценка на коректността на функциониране.
- ✓ То се извършва като подмножество на процесите на валидация или верификация.

Същност на V&V

- ☞ Това са дейности, които трябва да се извършват се през целия жизнен цикъл на ПО за да позволят дефектите да се установяват колкото може по-рано.
- ☞ Дейностите трябва да се планират, документират и ръководят от безпристрастна група специалисти.
- ☞ Проверката на отделните части на ПО не гарантира задължително качеството на ПО в цялост. Цялостната проверка на системата е много сложен процес и не винаги може да се реализира.

Същност на V&V



Верификация

☞ Основни дефиниции

✓ IEEE-610:

- ✗ Процес на оценка на система или компонент за определяне дали продуктът в дадената фаза на разработка отговаря на условията, определени в началото на фазата.

Дали създаването на продукта

✓ ANSI/EIA 632

(написан, разработен, кодиран, сглобен и интегриран)

- ✗ Потвърждение чрез проучване и на основата на обективни данни, че специфицираните изисквания, по които *е извършено* кодиран или сглобен, са изпълнени.

Валидация

Основни дефиниции

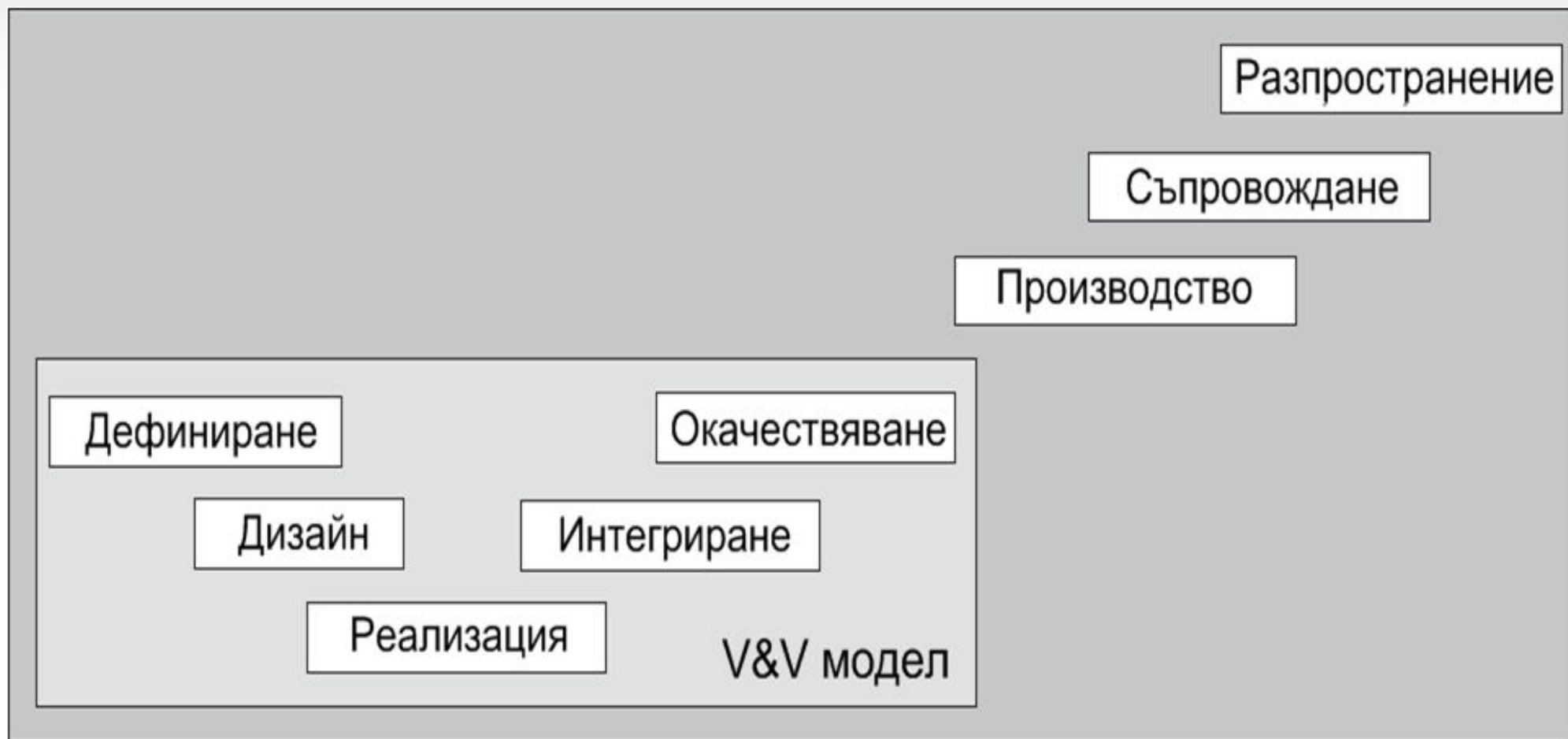
✓ IEEE-610:

- ✗ Процес на оценка на система или компонент по време на разработка или в края на разработката дали отговаря на специфицираните изисквания.

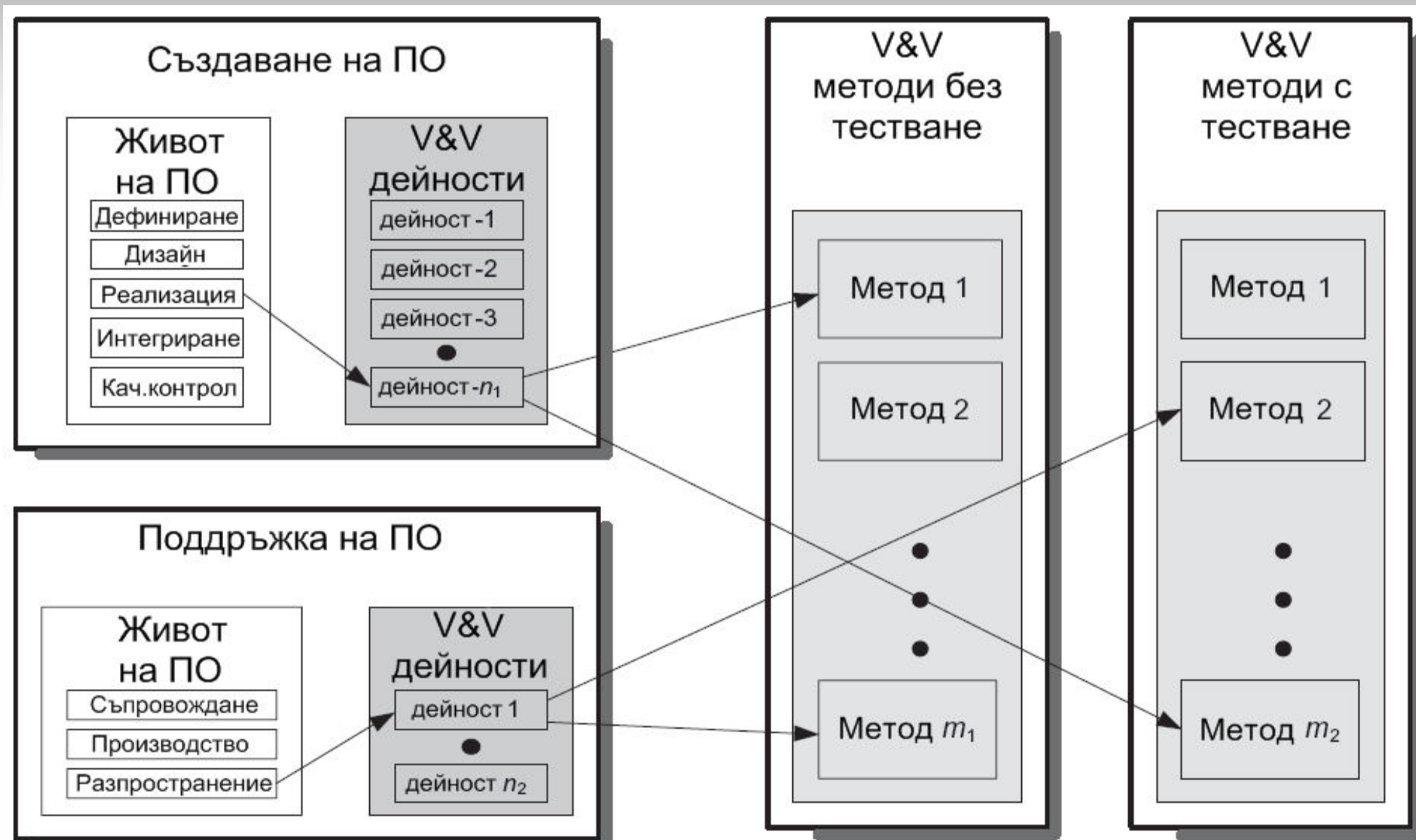
✓ ANSI/EIA 632

- ✗ Потвърждение чрез проучване и на основата на обективни данни, че зададената употреба на продукта (разработен или закупен) или съвкупността от продукти е извършено в желаната среда.

Място на V&V модела



Реализация на V&V модела



Реализация на V&V модела

☞ Фраза “Дефиниране”

- ✓ Определят се изискванията към разработваното ПО.

☞ Дейности за реализация на V&V модела

- ✓ Моделът е насочен е към дефиниране на ясни, пълни, непротиворечащи си изисквания.
- ✓ Дефинират се правилата за измерване на качеството.
- ✓ Създава се план за окачествяване на ПО на основата на дефинираните изисквания.

Реализация на V&V модела

☞ Фраза “Дизайн”

- ✓ Определя се техническата концепция и програмната архитектура за реализация.

☞ Дейности за реализация на V&V модела

- ✓ Обвързване на изискванията от заданието с конкретни подсистеми и компоненти.
- ✓ Конкретизиране на начините за измерване на качеството.
- ✓ Избиране на стратегиите за верификация.
- ✓ Дефиниране на изискванията за качество за следващите фази, които зависят от избрания дизайн.

Реализация на V&V модела

☞ Фраза “Реализация”

- ✓ Реализация на дизайна в реално система.

☞ Дейности за реализация на V&V модела

- ✓ Практическа верификация на ниво компоненти и подсистеми според ограниченията и изискванията, наложени от дизайна на ПО.
- ✓ Генериране на предложения за подобряване на реализацията на отделни елементи на ПО.
- ✓ Валидиране на основните изисквания според обвързването им с конкретни подсистеми и/или компоненти.

Реализация на V&V модела

☞ Фраза “Интегриране”

- ✓ Свързване на отделните подсистеми и компоненти в едно цяло.

☞ Дейности за реализация на V&V модела

- ✓ Верификация на интерфейса между отделните елементи.
- ✓ Верификация на интерфейса между системата и обкръжаващата я среда.
- ✓ Валидиране на изисквания, които са резултат от интеграцията на няколко подсистеми и/или компоненти.
- ✓ Завършване на плана за окачествяване.

Реализация на V&V модела

☞ Фраза “Окачествяване”

- ✓ Извършва се оценка на качеството чрез прилагане на тестове, специфицирани в стандарти, добри практики или определени от възложителя.

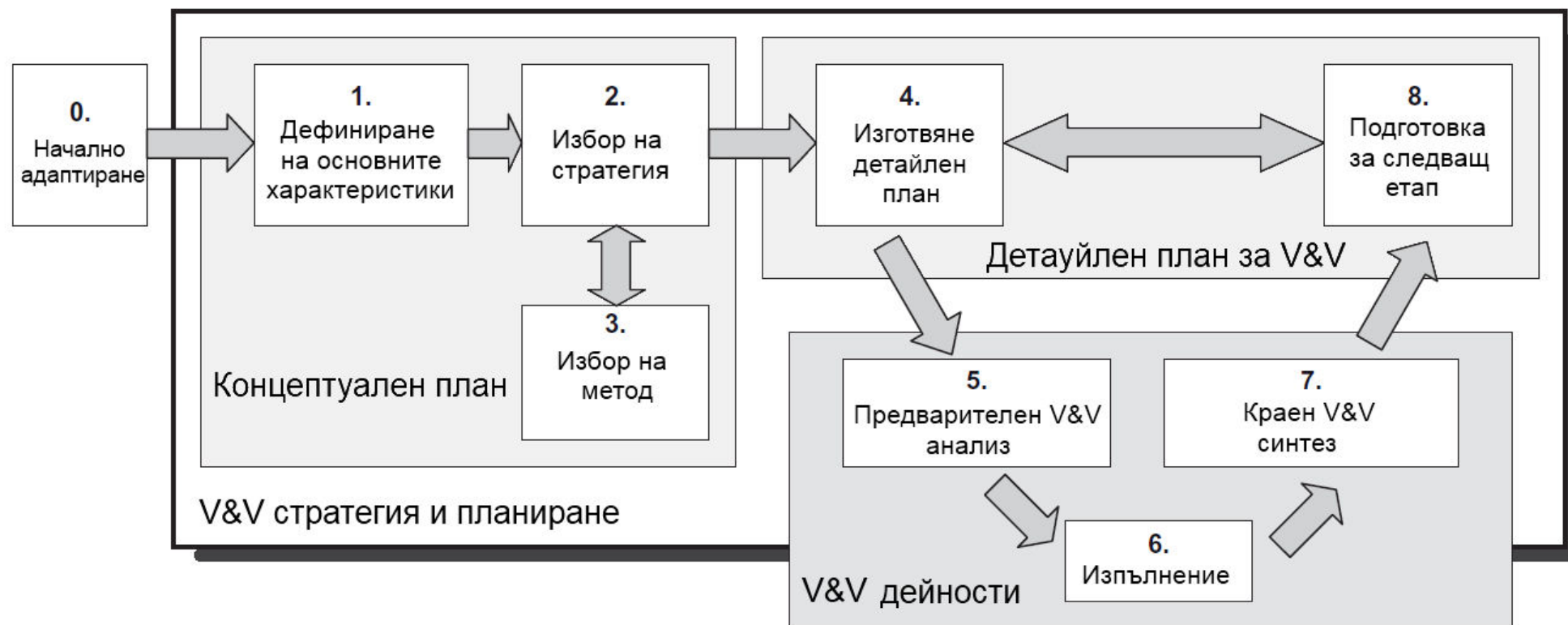
☞ Дейности за реализация на V&V модела

- ✓ Валидация на верифицираните изисквания с цел проверка дали правилно са отразени и реализирани реалните изисквания на възложителя.
- ✓ Проверяват се за правилност избраните методи за верификация (анализ, тестване, инспекция, прототипна демонстрация или други).

Реализация на V&V модела

- ☞ Дейности за реализация на V&V модела в другите фази на жизнения цикъл
- ✓ Валидация на новите изисквания за промяна за консистентност със съществуващите изисквания.
 - ✓ Валидация на изисквания за пълното при премахване на съществуваща функционалност.
 - ✓ Верификация на реализацията на новите подсистеми и/или компоненти.
 - ✓ Валидацията на интеграцията.
 - ✓ Крайна валидация на новата версия от реализация на изискванията на възложителя.

Методология за V&V



Инженерна дефиниция за система

- ☞ Verification, Validation and Testing of Engineered Systems - 047052751X - Wiley.pdf
 - ✓ 1.2.2
- ☞ Guide to software verification and validation.pdf
 - ✓ sydyrvanie
- ☞ Verification and Validation in Systems Engineering - Assessing UML-SysML Design Models.pdf
 - ✓ Glava 5

Как да разбираме препоръките по темата в литературата

Has To / Must

- ✓ Това е най-високото ниво задължителност за дадената препоръка, защото описания процес, процедура или подход *РАБОТИ САМО ПО ТОЗИ НАЧИН.*

Shall

- ✓ В този случай има няколко възможни потенциални варианта, но препоръчаният е най-желателен. Много често препоръката се дължи и на данни, които не са част от описанието.

Should

- ✓ Това е случая, когато авторът е експерт в дадената област и препоръката отразява опита му, че предложеното решение е най-доброто и най-правилното.

Questions ?

Методи за верификация на програмно осигуряване

Артефакти при разработка на ПО

Технически

- ✓ Основни – описание на изискванията, на различните проекти (идеен и детайлен), кода на програмата, ръководства за потребителя, за инсталиране
- ✓ Помощни – модели на обкръжението, формални модели на поведение, тестови набори, т.е. елементи, необходими за провеждане на верификацията.

Организационни

- ✓ Различните видове планове – управление на риска, на качеството, график на дейностите и т.н.
- ✓ Описание на процесите и процедурите за извършване на отделните дейности.

Задачи на верификацията

- ☞ Може и трябва да се прилага за всички видове артефакти, създавани при разработката и съпровождането на ПО.
- ☞ Верификация на организационни документи и процеси
 - ✓ Те се оценяват спрямо задачата на проекта, ограниченията по пари и време.
 - ✓ Оценява се квалификацията на персонала и стратегиите за преквалификация

Задачи на верификацията

☞ Верификация на изискванията – извършва се най-често според стандартите IEEE 830 и IEEE 1233, т.е. проверява се за:

- ✓ Еднозначност
- ✓ Съгласуваност и непротиворечивост
- ✓ Пълнота
- ✓ Минималност
- ✓ Проверяемост и проследимост
- ✓ Систематичност

Задачи на верификацията

☞ Верификацията на проектни решения оценява:

- ✓ Връзката “функционалност-изискване”
- ✓ Пълнота на документите за разработка
- ✓ Коректност на документите за разработка
- ✓ Коректност на решенията, особено свързани със сигурност, надеждност и други задължително изисквания към разработката.

Задачи на верификацията

- ☞ Верификацията на програмния код оценява:
 - ✓ Връзката “код-функционалност-изискване”
 - ✓ Спазването на семантичните и синтактични изисквания, т.нар. правила за кодиране (coding rules)
 - ✓ Проверка за различни видове неточности при реализация – цикли, пътища, инициализации и т.н.

Верификация на програмен код

- ☞ Изследвани типове дефекти:
 - ✓ Използване на данни (data references)
 - ✓ Декларативни грешки (data declaration)
 - ✓ Изчислителни грешки
 - ✓ Грешки при сравнения
 - ✓ Входно/Изходни грешки
 - ✓ Грешки при управление на извършване на действията (control flow)
 - ✓ Интерфейсни грешки
 - ✓ Други

Задачи на верификацията

☞ Верификацията на цялостното разработвано ПО оценява:

- ✓ Системата и нейните компоненти могат ли да работят в съответното обкръжение. Описано в заданието
- ✓ На основата на сценарии се проверява коректността и пълнотата на поведението на системата и компонентите и.

Задачи на верификацията

- ☞ Верификацията на тестовите планове оценява:
 - ✓ Пълнотата на плановете според оценката на риска за проекта
 - ✓ Избраните методи за определяне на дефектите според очакванията за типовете дефекти.
 - ✓ Използваните вторични технически артефакти

Класификация

- ☞ Съществуват 4 основни категории:
 - ✓ Неформални (informal)
 - ✗ Разчитат изцяло на човешката интерпретация на артефактите и не използват математически формализми. Затова ги наричат и *‘експертизи’*.
 - ✓ Статични
 - ✗ Оценяват статичния дизайн и реализацията му, без оценка на изпълнението върху конкретна машина
 - ✓ Динамични
 - ✗ Проверяват поведението на ПО чрез изпълнение върху конкретен компютър.
 - ✓ Формални (formal)
 - ✗ Използват се различни математически формализми за определяне на дефектите в ПО.

Класификация



Експертизи

- ☞ Възникнали са като необходимост от подготовка на специалистите по тестване и настройка (debugging) на ПО
 - ✓ Наричат ги 'Човешко тестване' (human testing)
- ☞ Изключително ефективни при намирането и отстраняването на дефекти в ПО
 - ✓ Приема се, че във всеки проект трябва да присъстват поне един метод от тази група
 - ✓ Прилага се към всички групи артефакти и за всички етапи
- ☞ Имат две основни предимства
 - ✓ Ранното отстраняване на грешките намалява времето за разработка и стойността на проекта
 - ✓ Има чисто психологическо значение за програмистите, водещо до по-малко грешки спрямо работа само с тестване.

Експертизи

Видове

- ✓ Техническа експертиза (technical reviews)
- ✓ Инспекции на кода (inspections)
- ✓ Прегледи (walktroughs)
- ✓ Одит (audit)
- ✓ Други
 - ✗ Оценка на архитектурата чрез сценарии (scenario based evaluations)
 - ✗ Самоконтрол чрез четене (desk-checking)
 - ✗ Разглеждане между равни (peer reviews)

Статичен анализ

☞ Същност:

- ✓ На основата на формални методи се определя коректността на създаване на различни артефакти, които са част от проекта.
- ✓ На основата на шаблони, чрез формални техники, се проверява за специфични групи дефекти.

☞ Основна дилема: “Строгост на анализа – Количество на дефектите”

Статичен анализ

☞ Предимства:

- ✓ Осигурява добра възможност за автоматизация на процеса за анализ.
- ✓ Широко разпространение, особено за някои видове ПО и използвани езици за програмиране и методи за разработка.

☞ Продукти

- ✓ PolySpace Verifier, Coverity Prevent и Klocwork K7

Формални методи

- ➡ Използват се за анализ на формалния модел на изискванията, поведението на ПО и на обкръжението му.
- ➡ Може да се прилага само към артефакти или части от тях, които са изразени формално или за които може да се създаде формален модел.
- ➡ Формалните модели не могат да се създават автоматизирано, но техните характеристики могат да се проверяват автоматизирано.
- ➡ Много добри за откриването на сложни дефекти, които не могат да се открият с тестване или експертиза.

Формални методи

☞ Биват:

- ✓ Логико-алгебрични модели (property-based models)
 - ✗ съждения и тяхното изчисление, предикатна логика и изчисление на изрази в нея (включително и от по-висок ред), λ -изчисления, модални логики, времеви логики, релационна алгебра, алгебрични модели на основата на абстрактни типове, т.н.
- ✓ Изпълними модели (executable models)
 - ✗ виртуални машини, крайни автомати, системи на основата на маркирани преходи, разширени крайни автомати, йерархични автомати, взаимодействащи си автомати, времеви автомати, мрежи на Петри, машини на основата на абстрактни състояния
- ✓ Междинни модели
 - ✗ Логика на Хоар (Hoare), динамични и програмни логики, програмни контракти

Динамични методи

- ☞ Изисква наличието на работещо ПО (може и отделни части от него).
 - ✓ Може да се използват и прототипи - в този случай се говори за 'имитационна динамична верификация'.
- ☞ Могат да се намерят дефекти, свързани само с работата на програмното осигуряване, но не и с други етапи от разработката, напр. съпровождане, инсталация и т.н.
- ☞ Изискват допълнителна подготовка
 - ✓ Най-често това са тестови набори, симулационна среда, мониторингова система, система за записване на резултатите.

Динамични методи

☞ Особено подходящ подход за определяне на някои от характеристиките на ПО

- ✓ Функционалност

- ✗ При проверка на функционалността се следят получаваните резултати, вътрешни състояния, обменени съобщения, ред на изпълнение на дейностите, по-рядко и времеви зависимости.

- ✓ Преносимост

- ✗ При проверка на преносимостта се следи за тези елементи на функционалните изследвания, които се влияят от средата на работа на ПО.

Динамични методи

☞ Особено подходящ подход за определяне на някои от характеристиките на ПО

- ✓ Производителност

- ✗ Записват се и се оценяват времето за изпълнение на операциите, използваната памет (всички видове), натоварването на комуникационните канали, вътрекомпонентните обмени на информация

- ✓ Надеждност

- ✗ Записва се броя и времето на поява на дефектите (по класове, по ресурси, по компоненти, по подсистеми).
- ✗ Използват се сценарии за симулиране на промени в обкръжаващата среда (напр. отказ на оборудване).

Динамични методи

☞ Биват:

✓ Мониторинг

- ✗ Това са дейности по следене работата и записване на текущото състояние на системата и получаваните резултати.
- ✗ След завършване на записването се провежда оценка на получените резултати от мониторинга като средство за определяне на неправилно функциониране.

✓ Тестване

- ✗ Мониторинг, но за специално изградени сценарии на използване.
- ✗ Генерирането на сценариите е с цел създаване на ситуации, при които ПО може да има неправилно поведение.

Синтетични методи

- ➡ Основната идея е съчетаване на предимствата от основните видове за сметка на ограничаване на недостатъците им.
- ➡ Най-често това е съчетание на динамични с формални методи:
 - ✓ Тестване на основата на модел на системата (model-based testing, model driven testing)
 - ✓ Мониторинг на формални свойства (runtime verification, passive testing)

Инспекции и Прегледи

- ☞ Извършва се задължително от група от специалисти, които четат или визуално проверяват програмата (според оценяваните артефакти).
- ☞ Участниците имат ясно дефинирани задачи, които трябва да извършат в процеса на работа.
- ☞ Основната фаза е ‘Срещата на участниците’
 - ✓ Обсъждат се резултатите от индивидуалната работа като начин за намиране на дефектите и проблемните зони.
 - ✓ Участниците нямат за задача да набележат начините за премахване на грешките.

Инспекции и Прегледи

Предимства:

- ✓ Много по-ефективни от самоконтрола, защото се извършва от външни хора, а не от автора.
- ✓ Намаляват разходите за настройка, защото локализират мястото на дефекта много точно.
- ✓ Добри са за определяне на логически грешките при разработката или грешки в кода.
- ✓ Особено полезни при промени на съществуващо ПО.

Недостатъци

- ✓ Не са добри за определяне на грешките от етапа на събиране и анализ на изискванията, т.е. в дизайна от високо ниво.

Инспекции

➡ Дефиниция

- ➡ Това са процедури и техники за откриване на дефекти в ПО на основата на групово изследване чрез четене.

➡ Същност:

- ✓ Критическо изследване на програмните артефакти, целящо откриването и отстраняването на дефектите.
- ➡ Една от най-използваните техники за верификация и подобряване на качеството на ПО.
- ➡ Директно определя и коригира дефектите в ПО.
- ➡ Могат да се прилагат за много видове програмни артефакти.

Инспекции

- ☞ Същност:
 - ✓ Критическо изследване на програмните артефакти, целящо откриването и отстраняването на дефектите.
- ☞ Една от най-използваните техники за верификация и подобряване на качеството на ПО.
- ☞ Директно определя и коригира дефектите в ПО.
- ☞ Могат да се прилагат за много видове програмни артефакти.

Инспекции: особености

- ➡ Реализира се от група *инспектори* в рамките на паралелни и координирани проверки.
- ➡ Реализира се на няколко последователни фази, между които има координационни сесии.
- ➡ Дефекти се откриват като резултат от индивидуална инспекция, обобщаване на резултатите в сесиите или паралелна инспекция на една и съща част от ПО.
- ➡ Дефектите се отстраняват от разработчика и се извършва повторна проверка на ПО.
- ➡ Дълбочината на инспекция може да варира според нуждите и типа на ПО.

Фаганови Инспекции (Fagan inspections)

- ➡ Една от най-ранните системи е разработена от Fagan през 1976 г.
 - ✓ Планиране – какво ще се инспектира, кой ще участва и каква ще му е ролята.
 - ✓ Начална среща – разпределят се инспекторите по задачи и се определя как ще се извършва работата
 - ✓ Реализация – всеки инспектор подготвя списък с потенциални дефекти и проблемни области
 - ✓ Инспекционно събиране – формира се списъка с дефектите чрез обобщаване и обсъждане на индивидуалните инспекции.
 - ✓ Фиксиране на дефектите - разработчикът отстранява намерените дефекти.
 - ✓ Контролна проверка – прави се валидация на ПО след фиксирането на дефектите.

Инспекции

☞ Промени в метода на Фаган

- ✓ Различни техники за реализация на индивидуалните инспекции с цел намаляване на фалшивите съобщения и интензифициране на работата.
- ✓ Разлики в големината на групите, задачата на модератора и на сесийния ръководител.
- ✓ Промени в техниките за определяне дефектите – прилагане на все повече систематични техники и намаляване на ad-hoc дейностите.
- ✓ Използване на резултатите на инспекциите като основа на добри практики за разработка на ПО

Инспекции: фази на процеса

- ☞ Планиране и реализация
 - ✓ Каква са целта и обекта на инспекция ?
 - ✓ Какво ще се инспектира ?
 - ✓ Кой ще извършва инспекцията ?
 - ✓ Кой трябва допълнително да се привлече, в каква роля и с какви отговорности ?
 - ✓ Какъв ще е процесът, какви техники ще се използват и какъв ще е контролът на резултатите ?
- ☞ Инспекция и определяне на дефектите
- ☞ Корекция и контрол на резултатите.

Инспекции: състав

- ☞ Модератор
 - ✓ Висококвалифициран програмист, но не разработчика на програмата.
- ☞ Програмист
 - ✓ Разработчикът на инспектираното ПО
- ☞ Системен дизайнер
- ☞ Специалист по тестването

Прегледи на ПО (walkthroughs)

- ➡ Подобно е на инспекциите, но процедурите и техниките за откриване на дефекти са различни.
- ➡ Общи характеристики
 - ✓ Провежда се от група от 3-5 разработчика, като само един е автор на проверяваното ПО.
 - ✓ Задължителните роли са модератор, тестер, програмист и секретар.
 - ✓ Допълнително участват: много висококвалифициран програмист; специалист по програмни езици; програмист извън проекта; този, който ще поддържа програмата; може и други програмисти от проекта.
 - ✓ Сесиите са по 2-3 часа без прекъсване.

Прегледи на ПО (walkthroughs)

Реализация

- ✓ Всеки от участниците предварително е получил и проучил съответния материал
- ✓ По време на срещата се симулира работата на компютъра.
- ✓ Тестерът предварително е приготвил група тестове, които се проиграват умствено. Така тестовите данни преминават (*walk through*) през логиката на програмата.
- ✓ Състоянието на данните и системните ресурси се следи непрекъснато на дъска.

Прегледи на ПО (walkthroughs)

Цел:

- ✓ Подготвените тестове не трябва да обхващат дефектите, а само да дадат основа за виртуални експерименти с програмата.
- ✓ Намерените дефекти не се адресират към програмиста, а се записват за по-нататъшен анализ.

Отстраняване и контрол на дефектите

- ✓ По подобие на инспекциите и този процес определя с голяма точност мястото, типа и причината на грешката.
- ✓ Проверката след поправката е аналогична с инспекциите.

Самоконтрол чрез четене (desk-checking)

☞ Същност:

- ✓ Авторът на програмата прилага техника от инспекцията или прегледа върху собствения си код.

☞ Недостатъци

- ✓ Много по-неефективен
 - ✗ Човек трудно открива собствените си грешки – стилът на мислене е един и същ.
 - ✗ Няма груповия поглед, позволяващ едновременно да се огледа от всички страни един проблем.
 - ✗ Психологически – никой не иска да си изтъква собствените грешки.
- ✓ Няма дисциплиниращ ефект върху разработчиците.

Разглеждане между равни (peer reviews)

Същност:

- ✓ Техника за определяне на критерии за оценка на качеството, полезността, възможността за разрастване и съпровождане, яснотата

Реализация

- ✓ Администраторът на процеса подбира група програмисти, като всеки представя по няколко собствени работи.
- ✓ Работите се разглеждат анонимно и случайно разпределени, като всеки дава оценки по предварително дефинирана таблица.
- ✓ При събирането всеки защитава и обяснява оценките си и така се създава обща система за оценяване и критерии за прилагането и.

Мониторинг

- ☞ Мониторингът е подход от групата на динамичните техники.
 - ✓ monitoring, runtime verification, online verification, passive testing
- ☞ Когато се следи само на производителността се нарича '*профилиране*' (profiling).
- ☞ Използваните техники зависят от:
 - ✓ От типа и предназначението на записаната информация.
 - ✓ От типа на получаване на информацията за следените данни.
 - ✓ От типа на получаване на оценъчни характеристики.

Мониторинг

- ☞ Влияние на типа и предназначението на записаната информация
 - ✓ Общи данни и метрики, свързани с проверяваните характеристики
 - ✗ параметри на извикване и резултати, обем и количество на динамичното заделяне на памет, количество на съобщенията и др.
 - ✓ Данни, които могат да служат за определяне на грешки и проблемни ситуации
 - ✗ динамична памет и указатели, синхронизационни механизми, статистика на мрежовите комуникации/сесии

Мониторинг

- ☞ Влияние от типа на получаване на информация за следените данни.
 - ✓ Получаване от изходния код или получаване от двоичния код
 - ✓ Използваните техники биват:
 - ✗ Ръчни
 - ✗ Компилаторно-асистирани
 - ✗ С използване на бинарни трансляции
 - ✗ Инжекция на времево изпълнение
 - ✓ Симулаторен мониторинг

Мониторинг

- ☞ Влияние от типа на получаване на оценъчни характеристики.
 - ✓ Базирани на събития
 - ✓ Базирани на статистики

Въпроси ?

Тестване на ПО

Тестването като задача

- 👉 Нуждата от тестване се определя от желанието за повишаване на стойността на продукта, който се продава.
 - ✓ Повишаването на стойността е за сметка на подобряване на качеството и надеждността.
 - ✓ Тестването се използва за подобряване на качеството и надеждността като процедура за откриване на налични грешки в създаваното ПО.
- 👉 Тестването е техническа задача, но тя включва важни аспекти, свързани с икономика и човешка психология.

Психология и Тестването

- ❏ Неправилно разбиране на предназначението на процеса на тестване на разработваното ПО.
 - ✓ “Тестването е процес на показване отсъствието на грешки.”
 - ✓ “Целта на тестването е да покаже, че програмата изпълнява необходимите си функции коректно.”
 - ✓ “Тестването е процес на създаването на убеденост, че програмата прави това, за което е предназначена.”

Тестването е процес на изпълняване на програма с цел намиране на грешки в нея.

Психология и Тестването

- ☞ “Тестването е процес на показване отсъствието на грешки.”
 - ✓ Понятията ‘Успешен тест’ или ‘Неуспешен тест’
- ☞ “Целта на тестването е да покаже, че програмата изпълнява необходимите си функции коректно.”
 - ✓ Невъзможността за реализиране на задачата определя задълбочеността на реализирания процес.
- ☞ “Тестването е процес на създаването на убеденост, че програмата прави това, за което е предназначена.”
 - ✓ Понятието ‘не работи правилно’
 - ✗ Не само това, което не прави по задание, а и това, което прави повече отколкото трябва.

Психология и Тестването

- ☞ Познаването на идеологията на създаване на тестваната програма
 - ✓ Тестването трябва да се извършва от човек, който не е писал програмата, не е запознат детайлно с програмата и не е част от групата, която е създавала програмата.
- ☞ Дисциплиниращо действие
 - ✓ Никой не иска да съществува документ, в който са описани неговите грешки.

Икономика и Тестването

- ➡ Не може да се определят всички грешки.
- ➡ Тестването е времеемък и скъп процес.
- ➡ Някои грешки са по-важни, други са по-маловажни.

10^{те} принципа на тестването

- ➡ #1: Дефинирането на очаквания резултат/изходни данни са задължителна част от тестова процедура.
- ➡ #2: Програмистите трябва да избягват желанието да тестват собствените си програми.
- ➡ #3: Групата, създала ПО не трябва никога да отговаря за тестването на собствените си програми.
- ➡ #4: Резултатите от всеки тест трябва да се изследват внимателно и задълбочено.
- ➡ #5: Генерираните тестове трябва да обхващат както случаите с невалиден или неочакван входен набор от данни, така и с валиден и очакван входен набор.

10-те принципа на тестването

- 👉 #6: Проверката дали програмата прави това, за което е създадена е половината от работата. Другата половина е за проверка дали програмата прави неща, за които не е предназначена.
- 👉 #7: Избягвайте безсмислените тестове освен ако програмата не е безсмислена.
- 👉 #8: Не планирайте тестове с презумпцията, че няма грешки.
- 👉 #9: Вероятността за наличие на още грешки в дадена част от програмата е пропорционална на броя на намерените вече грешки.
- 👉 #10: Тестването е изключително интелектуален и творчески процес.

Генерация на тестовия набор

- ☞ Независимо от големината и спецификата на генерирания тестов набор не може да се гарантира, че с него ще се открият всички грешки.
- ☞ Ограничението по време и стойност на процеса е наложило правилото:
 - ✓ От всички възможни входни набори трябва да се определи такова подмножество, което да позволи установяването на най-много грешки.
- ☞ Съществуват няколко различни стратегии:
 - ✓ Случаен подбор на подмножеството от тестови данни.
 - ✓ Метод на черната кутия
 - ✓ Метод на бялата кутия
 - ✓ Комбиниран метод

Видове

☞ Метод на бялата кутия

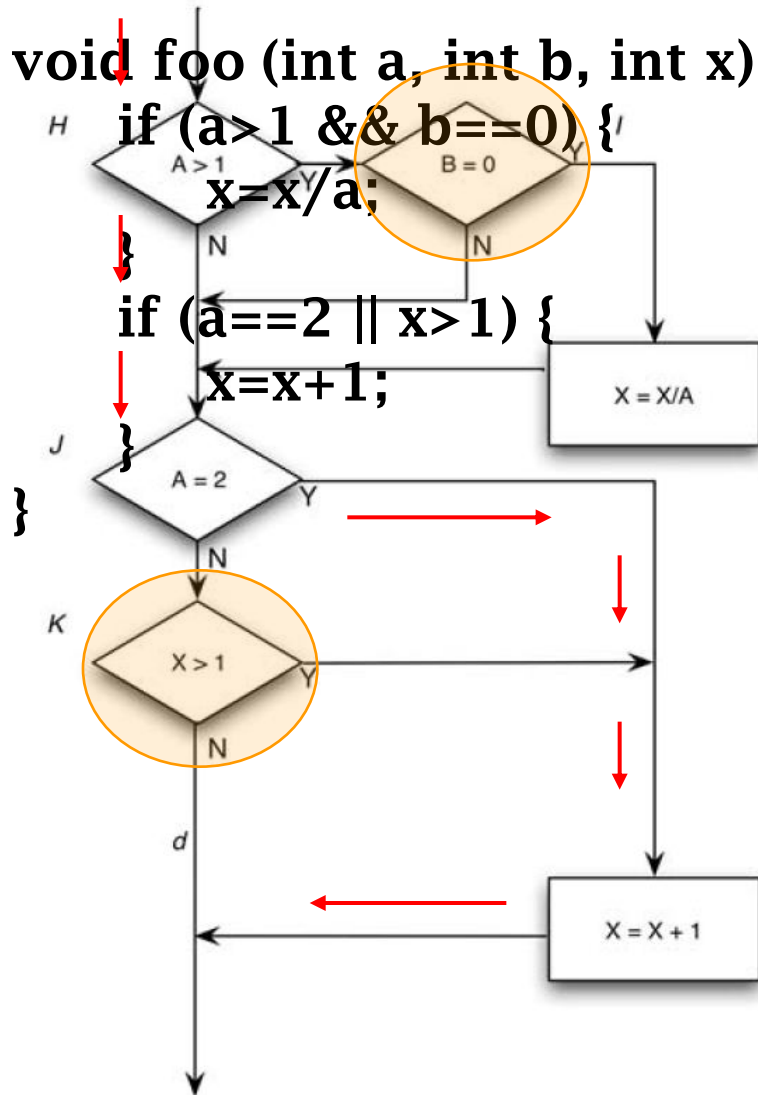
- ✓ Основан на покритие на логиката на програмата
 - ✗ Покриване на оператор (statement coverage)
 - ✗ Покриване на решение (decision coverage)
 - ✗ Покриване на решение условие (condition coverage)
 - ✗ Покриване 'решение -условие' (decision-condition coverage)
 - ✗ Покриване 'множествено условие' (multiple-condition coverage)

☞ Метод на черната кутия

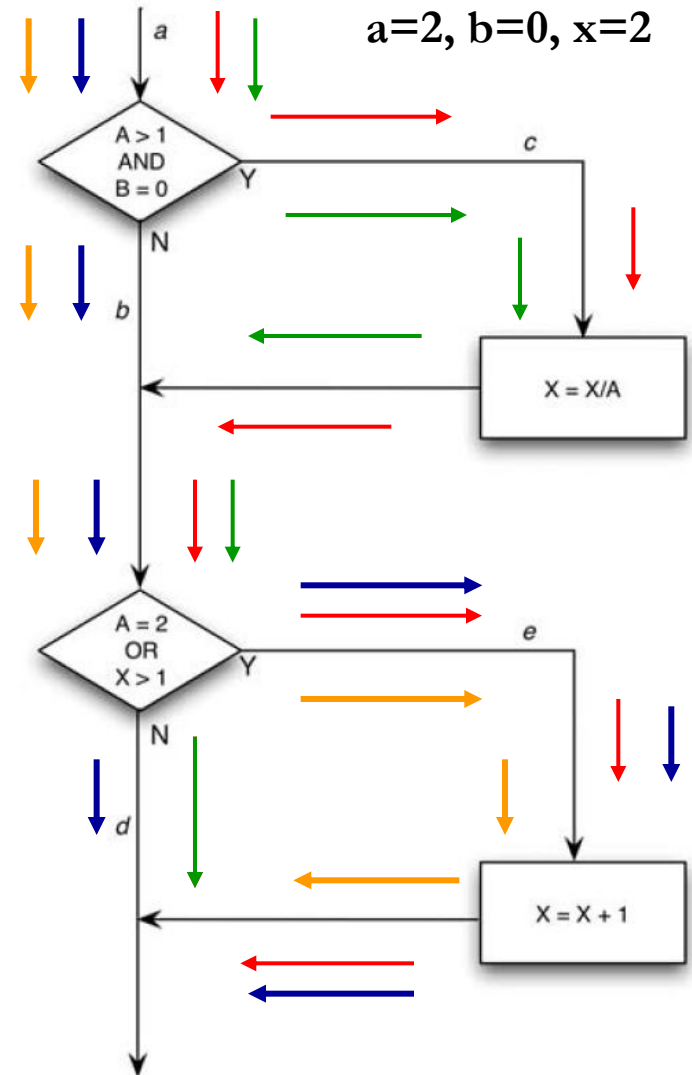
- ✓ Еквивалентно (равнозначното) разделяне
- ✓ Анализ на гранични стойности
- ✓ Графи 'Причина-Ефект'

Логическо покриване

```
void foo (int a, int b, int x) {  
  if (a>1 && b==0) {  
    x=x/a;  
  }  
  if (a==2 || x>1) {  
    x=x+1;  
  }  
}
```



`a=2, b=0, x=2`

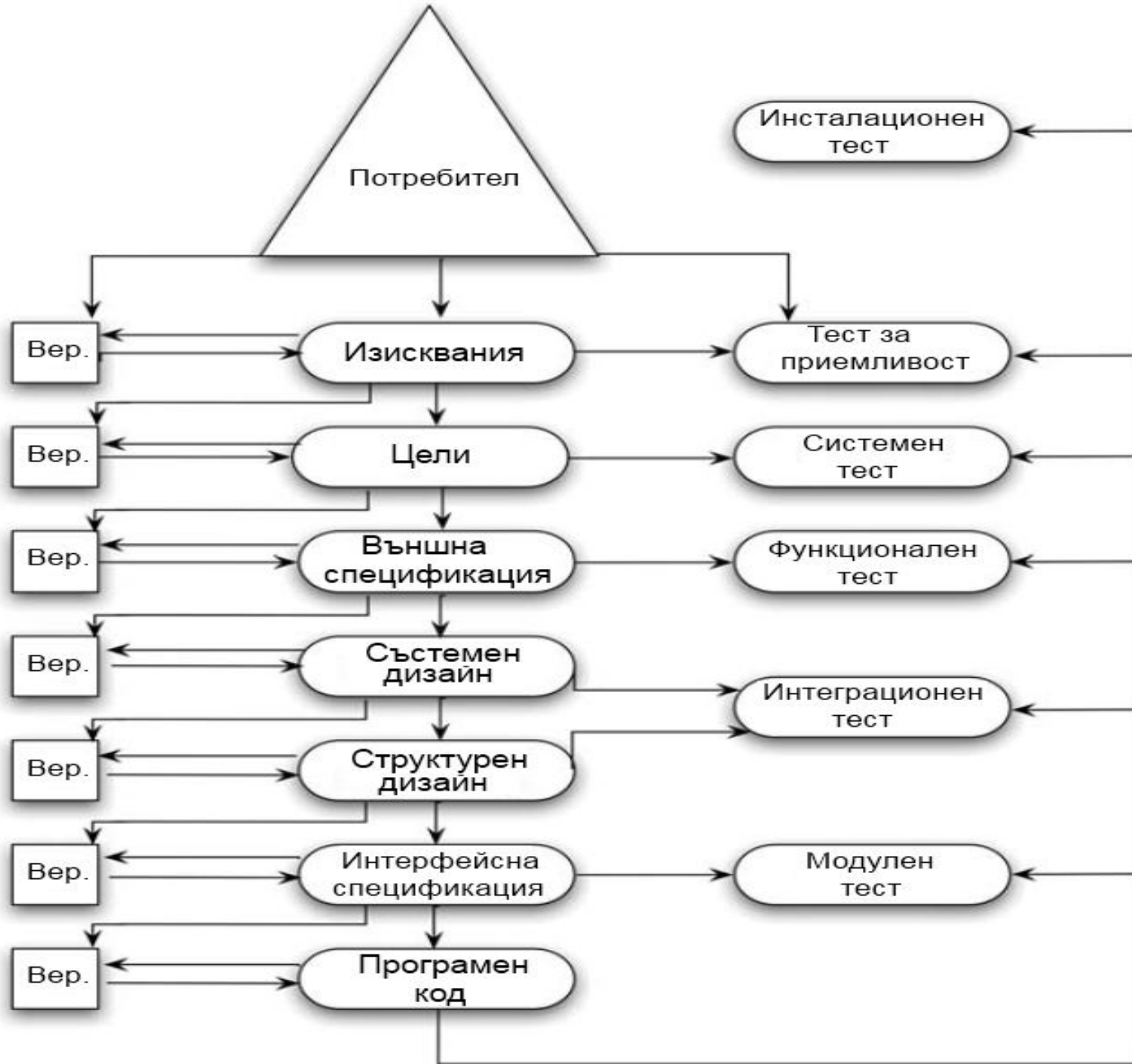


Стратегии при избора на тестов набор

1. Ако спецификацията съдържа комбинация от входни условия най-добре е използването на графи 'Причина-Ефект'
2. Винаги да се използва анализа на граничните стойности – независимо че тези стойности могат да се получат и от графите 'Причина-Ефект'.
3. Идентифицирайте валидните и невалидните равнозначни класове за входните и изходните данни, и ако е необходимо да се допълни тестовия набор, създаден от предходните стъпи.

Стратегии при избора на тестов набор

4. На основата на 'отгатване' да се попълни тестовия набор с още тестове.
5. Анализирайки логиката на програмата се генерират допълнителни тестове за случаите на комбинация от условия, които не изглеждат невъзможни и които не са били създадени на предните етапи (използват се всички обхват).

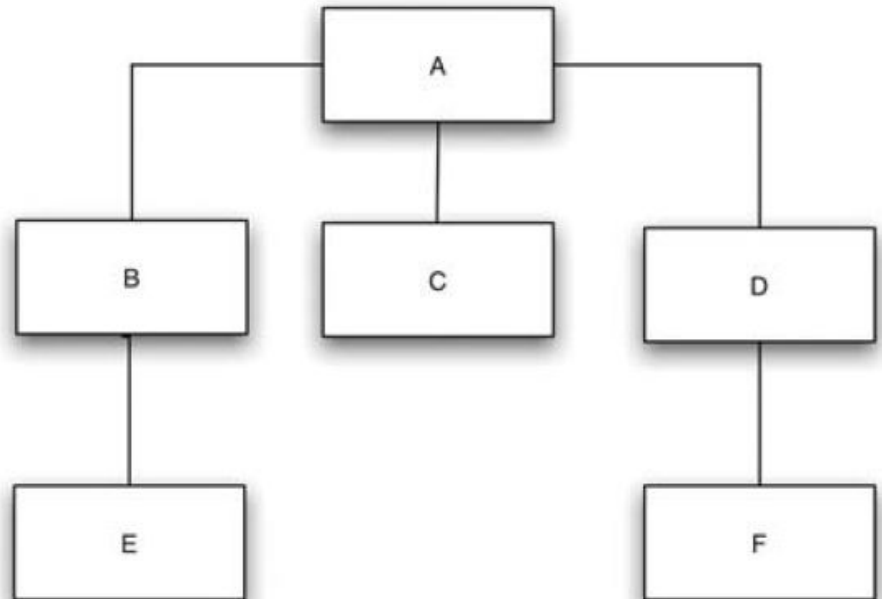


Тестване на модули

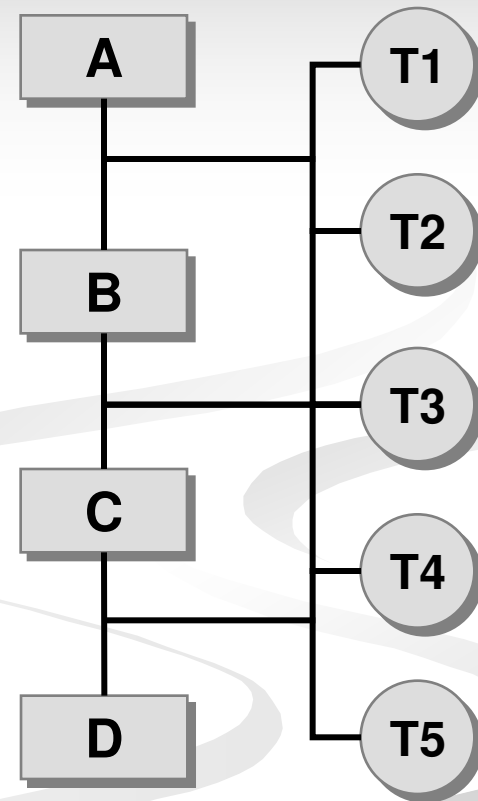
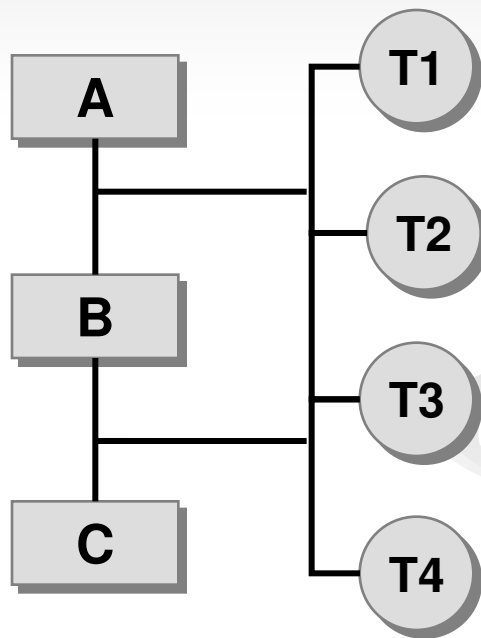
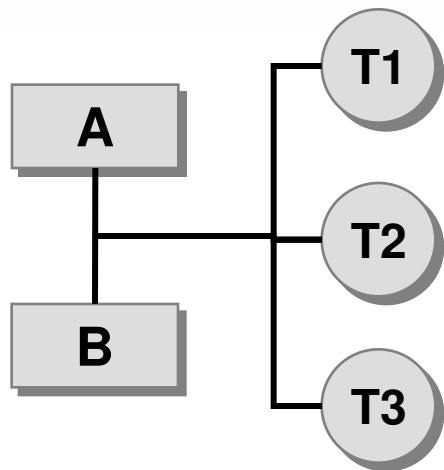
- ➡ Тестването на модули (unit testing) е насочено към проверка на отделните блокове (подсистеми, функции и др.) самостоятелно.
- ➡ Основно е ориентирано към проверка по метода на бялата кутия.
- ➡ Предимства:
 - ✓ Лесна комбинация на методите – елементите са малки по размер и сложност.
 - ✓ Паралелност на обработката на отделните елементи.
 - ✓ Лесна локализация на грешката при използване на резултатите от тестването.

Инкрементално тестване

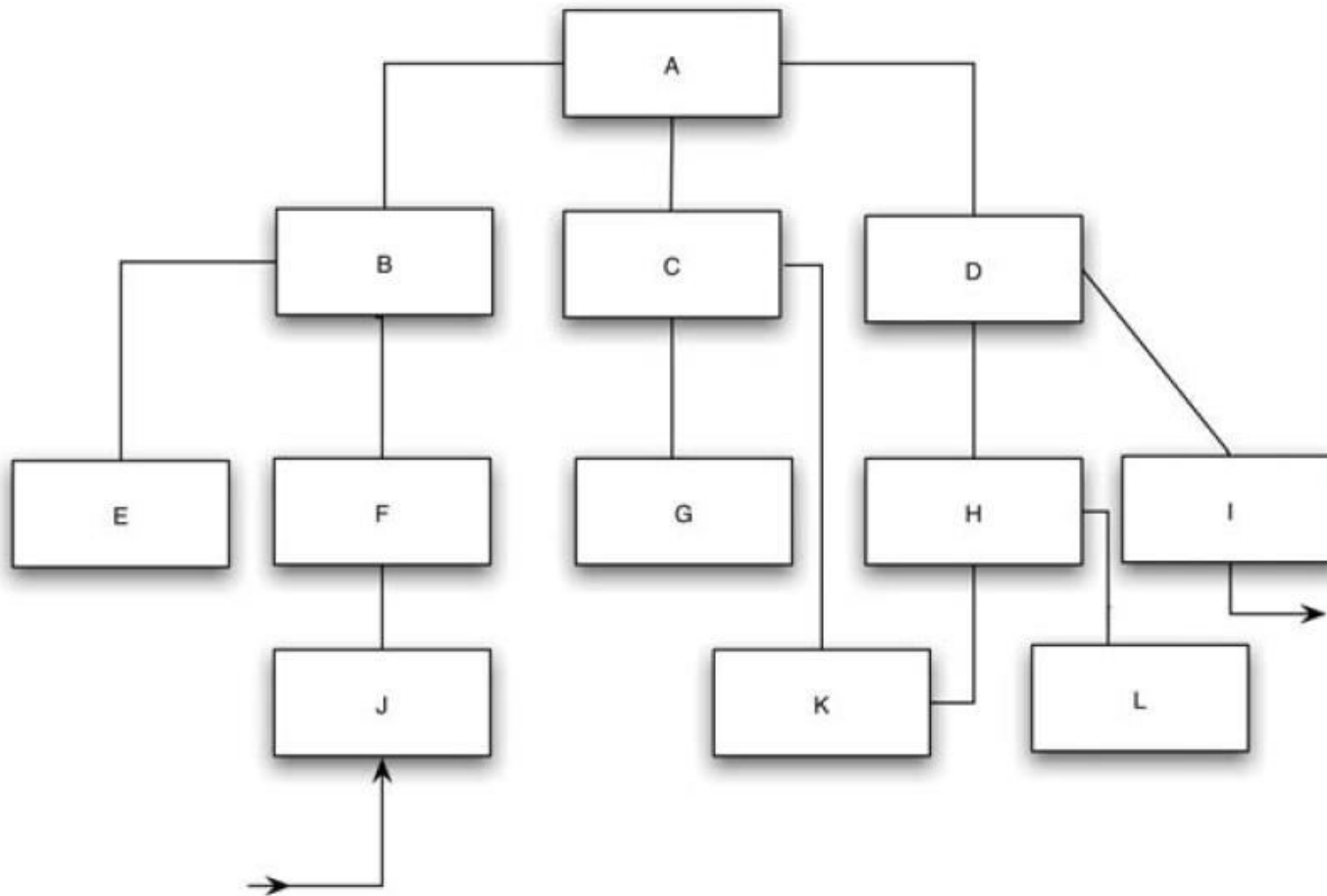
- ➡ Това е техника за тестване на модули, при която дадения модул се обвързва със съседен/-и модул/-и (вече тестван/-и) и се тестват заедно.
- ➡ Два подхода
 - ✓ Отдолу-нагоре
 - ✓ Отгоре-надолу



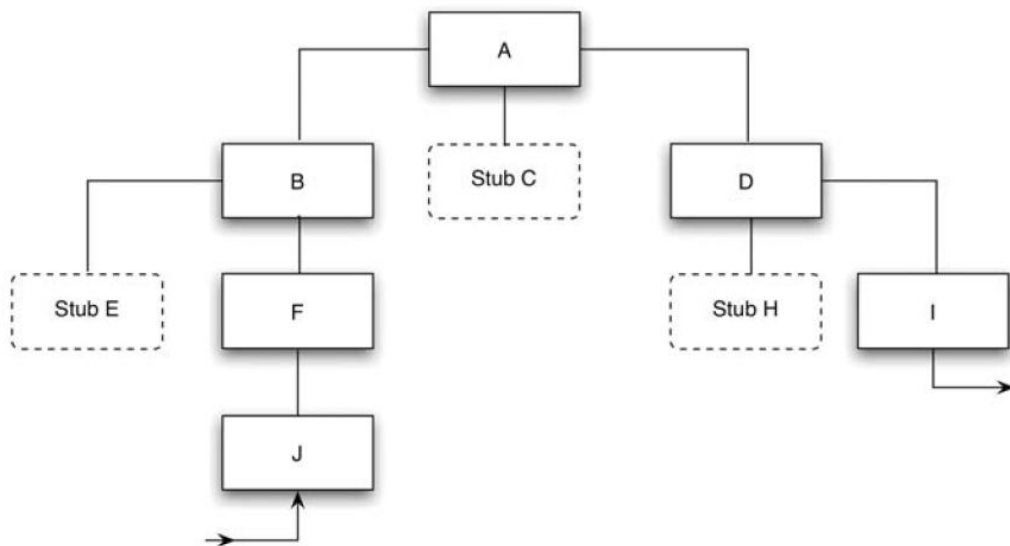
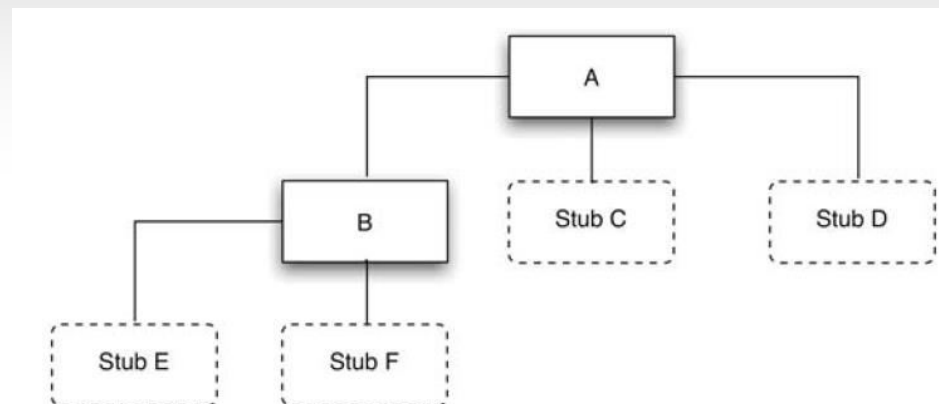
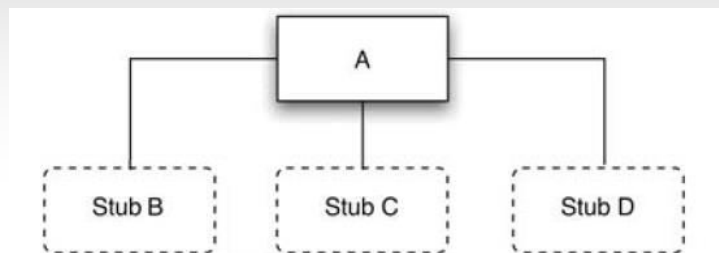
Инкрементално тестване



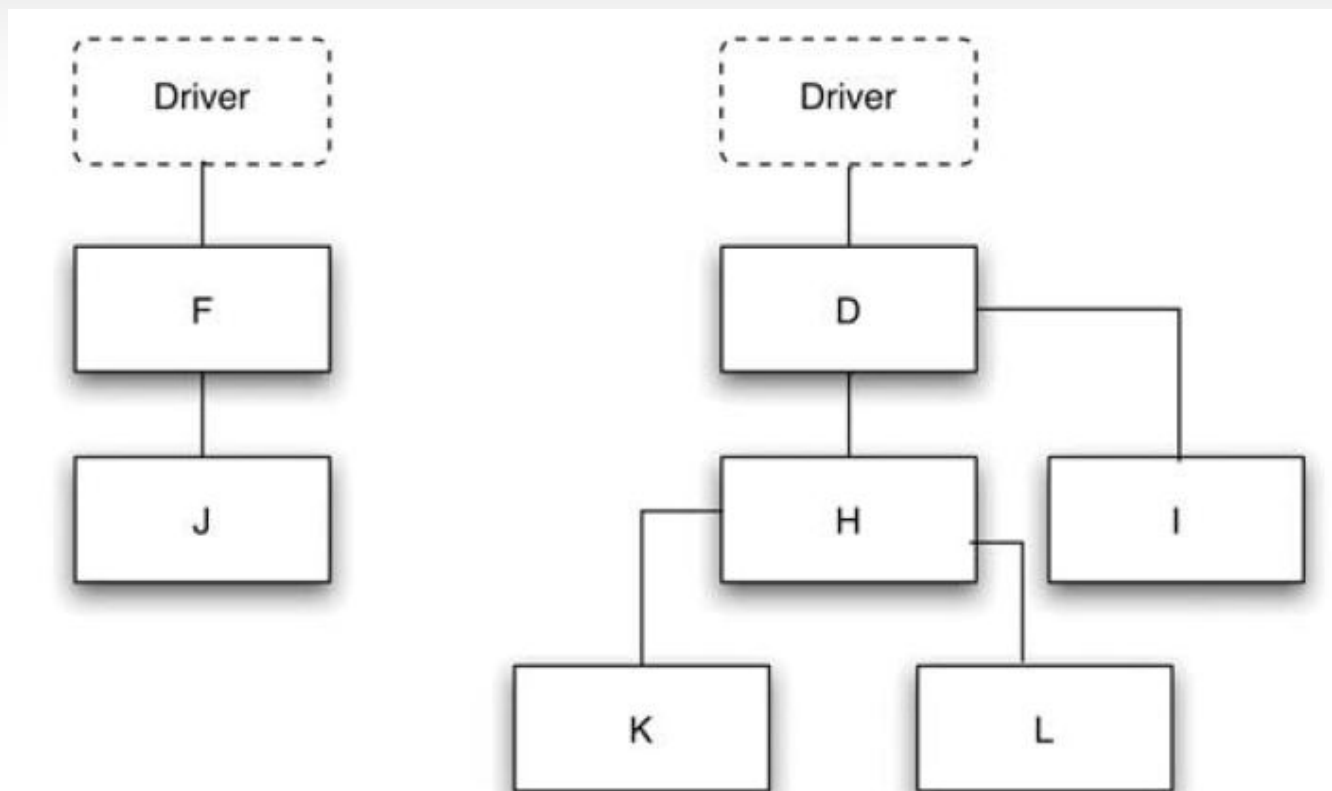
Инкрементално тестване



Отгоре-надолу



Отдолу-нагоре



Тестване на ПО на по-високо абстрактно ниво

- ☞ Модулното тестване е първия етап от процеса.
 - ✓ Това е особено вярно при големи и/или сложни системи.
- ☞ Тестваното на по-високо абстрактно ниво се дължи на факта, че задачата на ПО е да трансферира информация между елементите и/или да преобразува тази информация от един вид в друг.
 - ✓ Грешките най-често са резултат от прекъсване на предаването на информацията, неправилно преобразуване или появата на шум.

Функционален тест

☞ Предназначение:

- ✓ Откриване на различия между програмата и външните спецификации, задаващи поведението на програмата от гледна точка на потребителя.

☞ Реализация

- ✓ При малки програми се използва метода на черната кутия.
- ✓ При по-големи програми се използват методи от групата на бялата кутия (по същество всички без логическо покритие).

☞ Препоръки

- ✓ Да се доразвива тестването по посока на случаите с най-много открити грешки.
- ✓ Специално внимание да се обърне на невалидни и/или неочаквани входни данни.

Системен тест

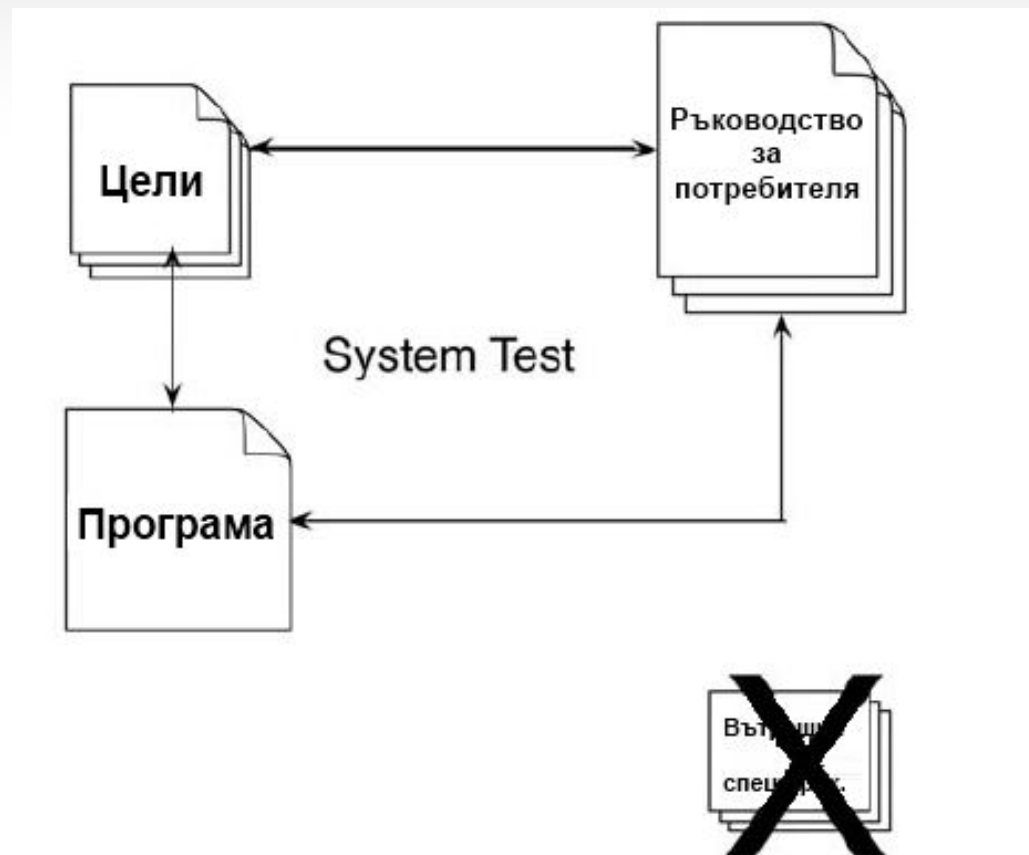
Предназначение

- ✓ Задачата му не е да проверява функционалността на системата като цяло – това е задача на функционалния тест.
- ✓ Системният тест е насочен да провери как системата (като цяло) реализира определените и цели.
 - ✗ Задължително трябва да се измерват метрики за измерване на степента на реализация на целите.
 - ✗ Задължително се извършва като част от дефинираната работна среда.

 Извършва се от външна група с опит както като разработчици, така и като потребители.

Системен тест

👉 Генерация на тестовите набори



Външни спецификации

Системен тест

Видове:

- ✓ Реализирани обработки (facility testing)
- ✓ Тестване по размер (volume testing)
- ✓ Тестване с екстремно натоварване (stress testing)
- ✓ Тест за използваемост (usability testing)
- ✓ Тест за сигурност (security testing)
- ✓ Тест за производителност (performance testing)
- ✓ Тест за съхраняемост на информацията (storage testing)
- ✓ Конфигурационен тест (configuration testing)

Системен тест

Видове:

- ✓ Тест за съвместимост (compatibility/configuration/conversion testing)
- ✓ Тестване на възможност за инсталиране (installability testing)
- ✓ Тест за издръжливост (reliability testing)
- ✓ Тест за възстановяемост (recovery testing)
- ✓ Тест за системна поддръжка (serviceability testing)
- ✓ Тест на документацията (documentation testing)
- ✓ Тест за част от процедури (procedural testing)
- ✓ Конфигурационен тест (configuration testing)

Тест за приемливост

- ➡ Тестът за приемливост (acceptance testing) е проверка как програмата реализира наразните си изисквания.
- ➡ Най-често се извършва от крайния потребител, т.е. разработчиците най-често не отговарят за организацията на този процес.
- ➡ Стратегия за реализация:
 - ✓ От документа с изискванията се проверяват едно по едно кое не е реализирано.

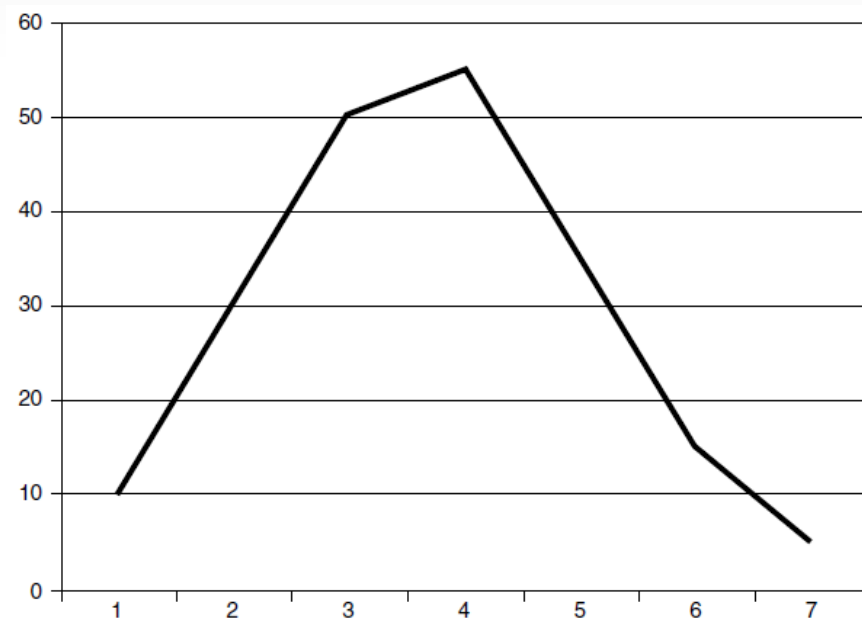
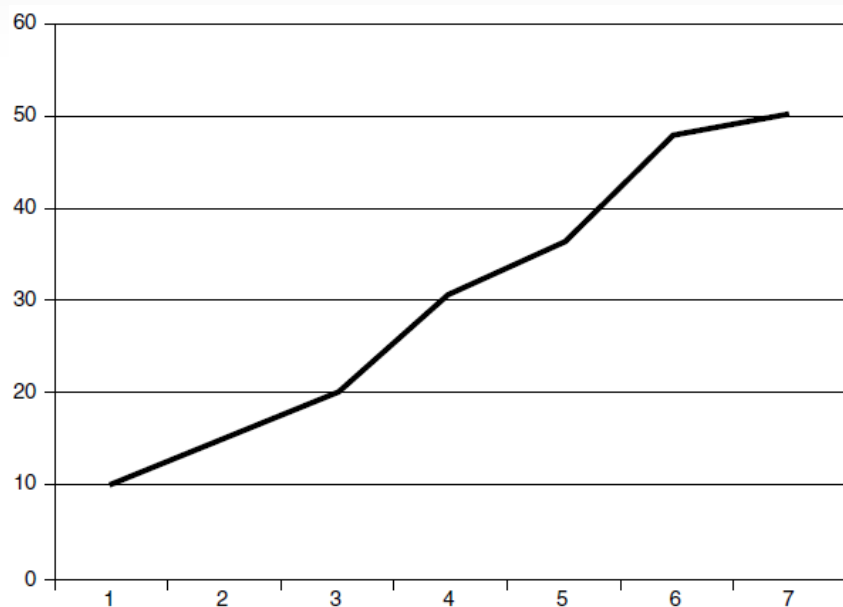
Инсталационен тест

- ➡ Инсталационният тест (installation testing) е специален тип тестване, защото при него не се търсят програмни грешки, а се проверява за грешки, резултат от процеса на инсталиране на програмата на конкретна система.
- ➡ Основни грешки:
 - ✓ Валидни опции при конфигуриране на системата за инсталация.
 - ✓ Задаване на валидна апаратна среда.
 - ✓ Нужда от комуникация с други програми в процеса на инсталация.
 - ✓ Дали се прехвърлят и/или използват необходимите файлове и библиотеки.
- ➡ Създава се и се изпълнява от разработчиците.

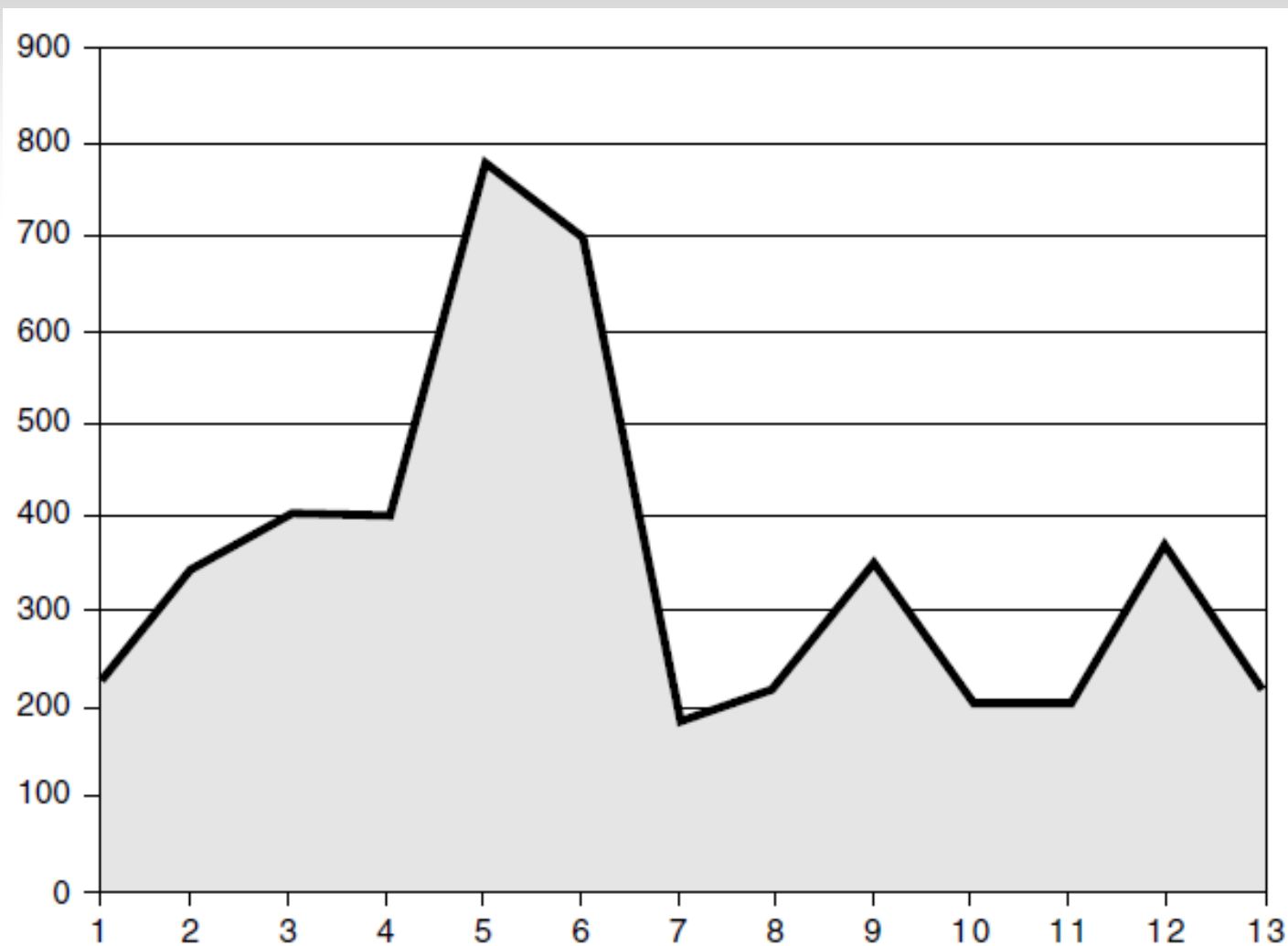
Кога свършва тестването

- ☞ Това е един от най-важните въпроси
 - ✓ Знае се, че не може да се намерят всички грешки.
 - ✓ Не се знае какъв процент от грешките е намерен.
 - ✓ Има сериозна стойност, т.е. струва пари.
 - ✓ Сериозни проблеми с графици, като най-често сериозно забавя етапите.

Кога свършва тестването



Кога свършва тестването



Използване на външна група

- ➡ Много честа практика, защото някои видове тестване е не трябва да се организират и извършват от разработчиците.
- ➡ Изисква се групата да е специализирана за нашия клас ПО, иначе резултатите от тестването ще са приемливи само за определени тестове.
- ➡ Необходимо е подготовка на изчерпателна документация преди да се започне процеса.
- ➡ Изполва се за доказване на качество на разработваното ПО.

Въпроси ?

Валидация на Програмно Осигуряване

Дефиниция

- ☞ Валидацията на програмно осигуряване е задължителен елемент от дейностите и процедурите по подобряване на качеството на създаваното програмно осигуряване.
- ✓ “Процес на оценяване на система или нейни компоненти в процеса на разработка или в нейния край за определяне дали задоволяват специфицираните изисквания”
- ✓ “Процес на събиране на информация, показваща че ПО и асоциираните продукти задоволяват системните изисквания, зададени за ПО в края на всеки цикъл от живота на ПО за решаване на някой сериозен проблем, както и да задоволяват потребителски нужди и зададено използване.”

Предимства от използването му

- ➡ Подобрява използването на отделните технологии.
- ➡ Подобрява бизнес-резултатите.
- ➡ Подобрява връзките между 'Възложител-Производител-Потребител-Засегнати'.
- ➡ Подобрява ефективността на използването.
- ➡ Намалява риска за откази и грешки.
- ➡ Изисква структурен подход към разработката.
- ➡ Подобрява обвързаността с регулациите в областта.

Същност

- ➡ Валидацията е базирана на методологиите за управление на процеса на производство.
- ➡ Свързана е с последователните дейности по пътя на разработката.
- ➡ Задължително отчита средата, в която ще функционира програмния продукт.

Същност

Спецификация на
потребителски
изисквания

Оценяване на
пригодността

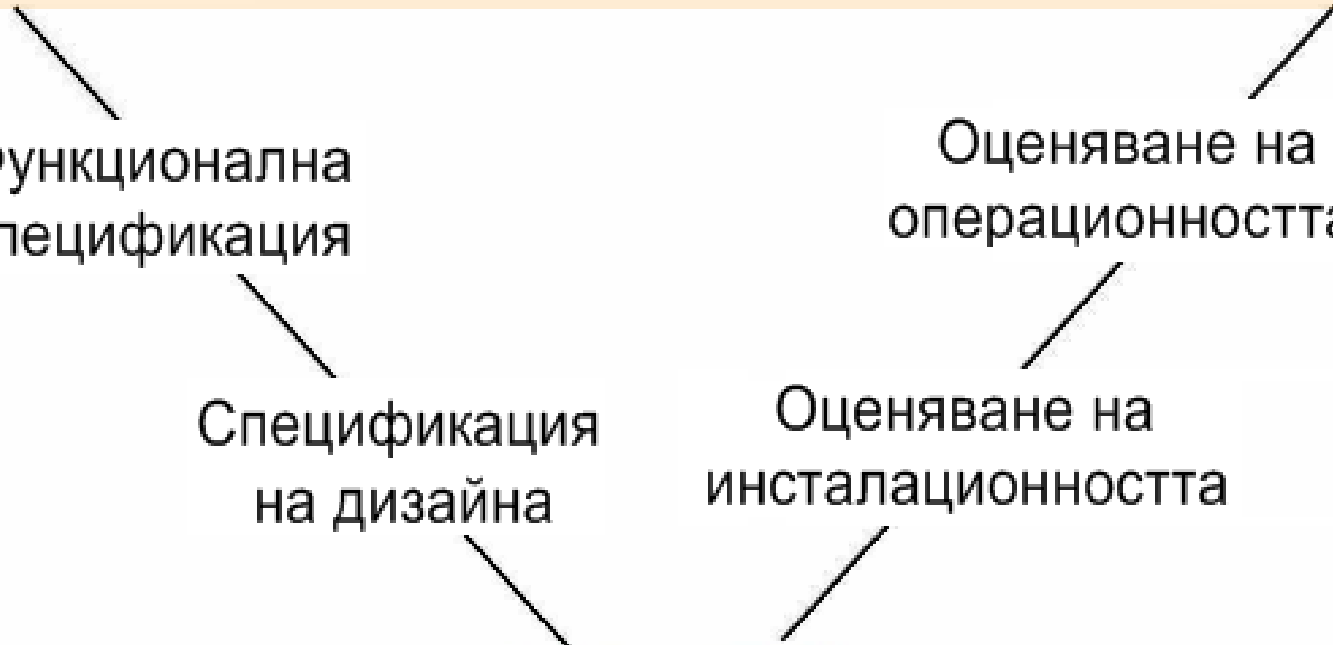
Функционална
спецификация

Оценяване на
операционността

Спецификация
на дизайна

Оценяване на
инсталационността

Програмен продукт



Характеристики

- ➡ Много повече от тестване за приемливост.
- ➡ Показва управление на производството и работата на ПО.
- ➡ Осигурява съгласуваност.
- ➡ Генерира знания и умения.
- ➡ Показва потенциални бъдещи изисквания.
- ➡ Изисква структурен подход към разработката.

Разлика при валидацията на софтуер и хардуер

- ➡ При програмните продукти може да се трасира причината за появата на грешката назад до спецификацията.
- ➡ Качеството на програмните продукти много-много по-малко се влияе от средата за производство.
- ➡ Големината на софтуерния продукт не е свързана със сложността му.
- ➡ Тестването не позволява гарантирането на правилното функциониране.
- ➡ Програмният продукт не е физически обект, т.е. той не се износва и не остарява физически.

Разлика при валидацията на софтуер и хардуер

- ☞ Дефектите в програмните продукти са резултат от специфични входни набори (могат да са свързани и с определени моменти във времето), т.е. появяват им е без предварително предупреждение.
- ☞ Скоростта е леснотата на ПО много често е причина за възникване на повече дефекти, отколкото дизайна и условията на производство.
 - ✓ Малки промени могат да доведат до сериозни проблеми на различни места в програмния продукт.

Разлика при валидацията на софтуер и хардуер

- ☞ Най-често промените в програмните продукти се правят от хора, които не са участвали в оригиналния проект.
- ☞ Софтуерните компоненти не са така добре стандартизирани и взаимозаменяеми, както е при хардуерните компоненти.

Принципи на програмната валидация

- ➡ Зависимост от спецификацията на изискванията
- ➡ Превенция на дефектите
- ➡ Оценка на времето и усилията
- ➡ Връзка с жизнения цикъл на ПО
- ➡ Планиране
- ➡ Избор на процедурите за реализация.
- ➡ Дълбочина след промяна на ПО.
- ➡ Независимост на оценката от производителя.
- ➡ Гъвкавост и отговорност на процедурата.

Основни дейности в различните етапи от жизнения цикъл

☞ Планиране на качеството



Планиране на качеството

Основни дейности

- ✓ Специфициране на задачите за всяка от дейностите в жзнения цикъл.
- ✓ Определяне и изброяване на важните фактори за качеството.
- ✓ Определяне на методите и процедурите за всяка задача.
- ✓ Определяне на критериите за приемане на всяка от задачите.
- ✓ Определяне на критериите за дефиниране и документиране на резултатите, позволяващо оценка на съответствието им с изискванията.
- ✓ Вход и изход за всека задача.
- ✓ Роли, ресурси и отговорности за всяка задача.
- ✓ Оценка на рисковете и предположения за появата.
- ✓ Документиране на потребителските изисквания.

Планиране на качеството

- ☞ Типичният план съдържа:
- ✓ План за управление на риска.
 - ✓ План за управление на конфигурациите.
 - ✓ План за осигуряване на качеството
 - ✗ План за валидация и верификация
 - Дейности и критерии за приемане на дейностите
 - График и планиране на ресурсите
 - Описание на изискванията за провеждане
 - ✗ Изисквания за формалния анализ на дизайна
 - ✗ Описание на изисквания за други технически спецификации.
 - ✓ Начин за описание на проблемите и методите им за решаване.
 - ✓ Други съпътстващи дейности.

Спецификация на изискванията

☞ Спецификациите на софтуерни продукти включват:

- ✓ Всички входни данни и сигнали.
- ✓ Всички резултати от работата на ПО.
- ✓ Всички функции, които ПО трябва да реализира.
- ✓ Всички изисквания за изпълнение на функциите, които трябва да се спазват (вид, характер и качество на данните, надеждност, времеви ограничения и други).
- ✓ Дефиниране на всички вътрешни, външни и потребителски интерфейси за ПО
- ✓ Описание как потребителя ще взаимодейства с ПО

Спецификация на изискванията

- ☞ Спецификацииите на софтуерни продукти включват:
- ✓ Как ще се установяват грешките и как ще се управляват.
 - ✓ Максималното време за отговор.
 - ✓ Описание на средата за функциониране на ПО, ако от това зависи дизайна му (платформи, ОС, БД и др.)
 - ✓ Всички диапазони, гранични стойности, стойности по подразбиране, специфични стойности, които ПО ще получава.
 - ✓ Описание на всички изисквания, спецификации и характеристики, функциониране на системата за сигурност, отказоустойчивост и самовъзстановяемост.

Спецификация на изискванията

☞ Основни дейности, свързани с валидацията:

- ✓ Предварителен анализ на риска.
- ✓ Анализ на възможностите за трасиране
 - ✗ Програмни изисквания – Системни изисквания
 - ✗ Програмни изисквания – Анализ на риска
- ✓ Описание на характеристиките на потребителя
- ✓ Оценяване на изискванията
- ✓ Анализ на изискванията за потребителски интерфейс
- ✓ Създаване на план за системно тестване
- ✓ Създаване на план за валидационно тестване
- ✓ Описание на двусмислиците в документите за анализ и описание.

Спецификация на изискванията

- ➡ Минимална форма на валидация:
 - ✓ Няма вътрешна неконсистентност.
 - ✓ Всички изисквания за изпълнение на функциите са описани ясно, пълно и подробно
 - ✓ Изискванията за надеждност, сигурност и отказоустойчивост са коректни и пълни.
 - ✓ Разпределението на функциите е пълно и коректно.
 - ✓ Изискванията са подходящи за избраното ниво на риск.
 - ✓ Изискванията са описани по начин, позволяващ измерване на резултатите и верифициране на целите.

Дизайн

- ☞ Основни дейности, свързани с валидацията:
 - ✓ Дефиниране на критерии за приемливост при специфициране на изискванията
 - ✓ Анализ на риска
 - ✓ Специфициране на стандарти и добри практики за разработка на кодиране.
 - ✓ Изисквания за системна документация и документацията за разработка
 - ✓ Изисквания към апаратната част, която ще се използва.
 - ✓ Определяне на параметрите, които ще се измерват.
 - ✓ Изисквания за модела и типа на данни.

Дизайн

- ☞ Основни дейности, свързани с валидацията:
 - ✓ Изисквания за логическата структура и използваните алгоритми.
 - ✓ Дефиниране на класовите грешки, алармите и предупредителните съобщения.
 - ✓ Изисквания за качеството на съпътстващото програмно осигуряване.
 - ✓ Изисквания за модела и технологиите за реализация на различните видове интерфейси (вътрешни, външни, с потребителя)
 - ✓ Метрики, методи и методологии за измерване на сигурността
 - ✓ Други изисквания, метрики и ограничения.

Дизайн

☞ Минимална количество дейности, свързани с валидация на ПО:

- ✓ Промени в анализа на риска
- ✓ Анализа на трасировката:
 - ✗ Двупосочна връзка 'Спецификация на дизайн – Изисквания'
- ✓ Оценяване на софтуерния дизайн
- ✓ Анализ на дизайна на интерфейсите
- ✓ Генерация на дизайна на тестването (от ниво модул нагоре).

Реализация и Кодиране

☞ Минимална количество дейности, свързани с валидация на ПО:

- ✓ Анализа на трасировката:
 - ✗ 'Код – Спецификация на дизайн'
 - ✗ 'Тест – Код'
 - ✗ 'Тест – Спецификация на дизайн'
- ✓ Анализ на дизайна на интерфейсите в програмния код.
- ✓ Генерация на тестовото множество (от ниво модул нагоре).

Тестване

☞ Метрики на основата на структурно покритие:

- ✓ Ниво оператор
- ✓ Ниво разклонение
- ✓ Ниво условие
- ✓ Ниво многоусловни разклонения
- ✓ Цикли
- ✓ Пътища
- ✓ Поток на данните

Тестване

☞ Метрики на основата на функционално покритие:

- ✓ Нормално използване
- ✓ Насочено към резултатите
 - ✗ Обхващане на всички възможни резултати
- ✓ Устойчивост на неочаквани входни въздействия
- ✓ Насочено към входните данни и сигнали
 - ✗ Обхваща различни комбинации на входни данни и сигнали.

Тестване

☞ Минимална количество дейности, свързани с валидация на ПО:

- ✓ Планиране на тестовете.
- ✓ Определяне на тестови набори за структурното покритие
- ✓ Определяне на тестови набори за функционално покритие
- ✓ Анализа на трасировката:
 - ✗ ‘Модулен тест – Дизайн от ниско ниво’
 - ✗ “Тест за интеграция – Дизайн от високо ниво”
 - ✗ ‘Системен тест – Изисквания’
- ✓ Изпълнение на тестовете
- ✓ Оценка на резултатите и провеждането на тестовете.
- ✓ Определяне на грешките
- ✓ Създаване на отчет за тестването.

Съпровождане и Промяна на ПО

- ☞ Допълнителни дейности, оказващи влияние върху валидацията на ПО:
 - ✓ Преразглеждане на валидационните планове.
 - ✓ Оценка на аномалиите.
 - ✓ Идентификация на проблемите и трасировка на дейностите по отстраняването им.
 - ✓ Оценка на очакваните/направените промени.
 - ✓ Повторение на дейности от предходни етапи на разработка.
 - ✓ Обновяване на документацията.

Специфика и създаване на ПО

- ☞ Валидационната процедура зависи от двата основни аспекта в процеса на създаване на ново програмно осигуряване:
 - ✓ Връзката 'програма-пазарен продукт'
 - ✓ Стратегия за създаване на нов програмен продукт
- ☞ Три вида на програмите като пазарен продукт:
 - ✓ когато ПО е част от система и не може да функционира самостоятелно
 - ✓ когато сами по себе си са продукт
 - ✓ когато се използват за създаване на други продукти

Специфика и създаване на ПО

☞ Съществуват 4 основни начина за създаване на нов програмен продукт:

- ✓ *Нов продукт* - изцяло новоразработено програмно осигуряване
- ✓ *Конфигуриране/Преконфигуриране* - ПО на основата на конфигурация и/или генерация по конфигурация от съществуващи модули
- ✓ *Модифициране на ПО* - когато се използва подобно и съществуващо ПО, което се променя за постигане на новите изисквания и за което цялата документация е налична
- ✓ *Използване на готово ПО* - когато се използва съществуващо ПО, отговарящо на спецификациите на заданието (или на части от него), но документацията е няма или е много малко

Деятности

☞ Реализирани чрез ревюта, инспекции и прегледи на различни документи, част от процеса на производство на ПО:

- ✓ обхват на процедурата по тестване на продукта
- ✓ резултати от валидационното тестване
- ✓ документа със записи на аномалиите, открити в ПО
- ✓ верификационната документация
- ✓ документа със записите, описващи проблемите при работа и съпровождане на ПО
- ✓ записите за трасируемост на промените

Нов продукт

Валидационен процес

- ✓ Реализира се на основата на пълната документация за разработваното ПО.
- ✓ Не зависи от валидацията на системите, които съдържат/използват новото ПО.
- ✓ Започва се най-често от спецификациите на системните изисквания
 - ✗ Особено важно е при 'критични' системи (socio-critical, safety-critical).
- ✓ Документа за началото на процеса за валидация трябва да съдържа минимум следните елементи: системните спецификационни изисквания; валидационен тестов план, тестови процедури, тестов дизайн и тестово множество, спецификация на компютърната система.

Нов продукт

Валидационен процес

- ✓ Статичната и динамична симулация на входните сигнали трябва да се реализира така, че да се възпроизведе нормалната среда на работа на ПО.
 - ✗ нормални операции, проявления, ограничения
- ✓ При 'критични' системи е необходимо да се създаде представителен тест, който да демонстрира реакциите на всеки входен параметър поотделно, както и на комбинациите от тези параметри.
- ✓ Специално се изследва верификационната процедура(-и) дали е обхванало всички функции на ПО.
- ✓ Задължително трябва да се създадат условия за проверка на времеви ограничения, ограничения по ресурси и други.

Нов продукт

Резултати от валидационния процес

✓ Валидационен отчет

- ✗ Описва състоянието (успешно/неуспешно), като при неуспешно дава обяснение на отделните причини за дефекти.
- ✗ Не дава препоръки как да се коригират откритите дефекти.

✓ Отчет за откритите аномалии

- ✗ Задължително описва всички аномалии, възникнали в процеса на валидация на ПО.
- ✗ За всяка аномалия се описват следните данни: описание и място на проявление; оценка на влиянието; причина за появата и същност на грешката; колко е критичен дефекта за ПО; препоръки за действие.
- ✗ При наличие на открити аномалии не може да завърши процедурата по валидация.

Модифициране

- ➡ За новите части важат правилата като за ново ПО.
- ➡ В случаите, при които се променя функционалността на съществуващото ПО се извършват всички валидационни действия за дадената подсистема/модул, проведени и описани в документацията.
- ➡ В случаите на промяна на базовите спецификации на съществуващото ПО се прилага пълна схема на валидация (като за изцяло нов продукт).
- ➡ Промените на конфигурационно ниво на продукта най-често се проверяват с верификация на интеграцията.

Модифициране

- ☞ За да може да се използва за нуждите на разработката съществуващото ПО трябва да отговаря на следните изисквания:
- ✓ Разработката трябва да е била на основата на добри практики и заложените метрики в плана за качество трябва да са съизмерими с необходимите за новото ПО.
 - ✓ Съществува документация за историята на работа и използване на ПО, която да показва надеждност на ПО.
 - ✓ Съществува пълна и прецизна документация за потребителя.

Модифициране

☞ При отсъствие на програмните кодове:

- ✓ Трябва да се извърши анализ на:
 - ✗ Функционални и интерфейсни спецификации
 - ✗ Документация за дизайна на системата
 - ✗ Тестовите данни
- ✓ Анализът трябва да се съсредоточи специално върху използванията на прекъсвания, рекурсии и заделянето на памет и други системни ресурси.
 - ✗ Трябва генерирането на нови тестове, проверяващи коректното функциониране в тези случаи.
 - ✗ Необходими са тестове за определяне на коректността на управление на ресурсите и за самопроверка (напр. управлението на стека).
 - ✗ Това е особено съществено за критични системи и в този случай е препоръчително провеждане на валидационна процедура като за ново ПО.

Използване на части от ПО

- ☞ В този случай не може да се разчита на документите за разработката на старото ПО
 - ✓ верификационните документи от етапите на разработка
 - ✓ документация за програмния код
 - ✓ изискванията за програмния продукт
- ☞ Анализира се информацията за историята на използване на ПО
 - ✓ Оценява се приложимостта му за новото ПО
 - ✓ Специално внимание се обръща на:
 - ✗ версиите на ПО, тяхната поява и същност на промените
 - ✗ документите за описание на грешките при използване и как са били отстранени

Използване на части от ПО

- ➡ Провеждат се нови верификационни и нови тестови проверки в случаите на съмнение в достоверността или на отсъствие на резултати за предни проверки.
- ➡ Цялостната проверка (старо + ново ПО) след това е като при разработката на ново ПО.

Конфигуриране / Преконфигуриране

- ☞ Извършва се в две основни направления:
 - ✓ Валидация на базовото ПО
 - ✓ Валидация на конфигурационните данни.
- ☞ Валидацията на базовото ПО се извършва по методите за използване на съществуващо ПО (по един от двата варианта).
 - ✓ За всяка подсистема и за всеки модул, които отговарят за някоя от функциите на новото ПО.
 - ✓ За всяка отделна операция на базовото ПО, които водят до промяна на данните за конфигуриране/преконфигуриране на ПО.
- ☞ Валидацията на конфигурационните данни е като за нов програмен продукт.

Валидация на данните

- ☞ Валидацията на данните се разделя на две дейности:
 - ✓ Валидация на стойностите на данните
 - ✓ Валидацията на техниките и методите за съхранението, модификацията и използването им.
- ☞ И в двата случая насоката на валидацията е да се установи дали данните се използват коректно и дали те позволяват да се въздаде ПО, което да функционира коректно.

Валидация на стойностите на данните

- ☞ Насоката е да се определи коректността на всяка данна във всеки момент на работа на ПО.
- ☞ Съществуват различни форми, като някои са изцяло ръчни, а други изцяло автоматизирани.
 - ✓ Проверка дали входните сигнали се подават в правилните места.
 - ✓ Проверка дали входните сигнали отговарят на критериите за качество и коректност.
 - ✓ Проверка за коректност на реакциите, породени от входните сигнали.
- ☞ Препоръчително е първо да се извърши валидация в специално разработена тестова среда, а след това в реалната среда на използване.

Валидация на системата за управление на данните

- ➡ Това е валидацията на техниките и методите за съхранението, модификацията и използването им.
- ➡ Това се прилага във всички случаи, независимо дали има автоматизирани средства за управление на БД или няма.
- ➡ Подборът на техниките за валидация задължително зависи от типа на съхраняваната информация и нейното използване.
 - ✓ Проверяват се само тези елементи, които оказват влияние върху функционирането на конкретното ПО.
 - ✓ За автоматизирани системи за управление на БД най-често се използват изискванията на стандарта IEC 880.

Като част от система

Особенности

- ✓ В този случай част освен дейностите по валидация на програмното осигуряване самостоятелно се провеждат дейности по валидация на системата като цяло.
- ✓ Задължително се оценява връзката 'Изисквания към изискванията – Изисквания към ПО'
- ✓ При някои случаи на разработка (използване на части от ПО, конфигурация/реконфигурация) може директно да се извърши валидация на ниво система.

Като система за производство на продукти

- ☞ Валидацията е резултат от успешната валидация на следните елементи:
- ✓ Оценява се както като част от система.
 - ✓ Валидация като част от технологичен процес.
 - ✓ Валидация на произведения продукт.

Въпроси ?

Валидация и верификация на вградени системи

Вградена система

Дефиниция

- ✓ “Това е компютърна система, която е създадена като компонент на по-голяма система и е ориентиран към работа върху един процесор. Задачите на тази система са много по-различни от системите с общо предназначение.”
- ✓ “... устройство, което има компютърна логика в ИС и не може да бъде директно програмирано от потребителя. Захранва се от батерия или директно от мрежата. ИС управлява една или повече функции на устройството.”

Вградена система

Дефиниция

- ✓ “Комбинация от апаратна част и програма, заедно с допълнителни механична и/или други части (ако е необходимо), използвани за постигане на определена функционалност.”
- ✓ “Вградените системи са компоненти, интегриращи апаратна и програмна част, специално разработване за свързване и реализация на специфична задача.”

Вградена система

Особенности

- ✓ Всяка от отделните части (апаратна и програмна) не може самостоятелно да осигури нужната функционалност.
- ✓ Основни ограничения
 - ✗ Реактивни системи
 - ✗ Най-често 'критични' системи
 - ✗ Автономни системи
 - ✗ С високо ниво на сигурност и надеждност
 - ✗ Много често имат ясно дефинирани ограничения по време.

Вградена система

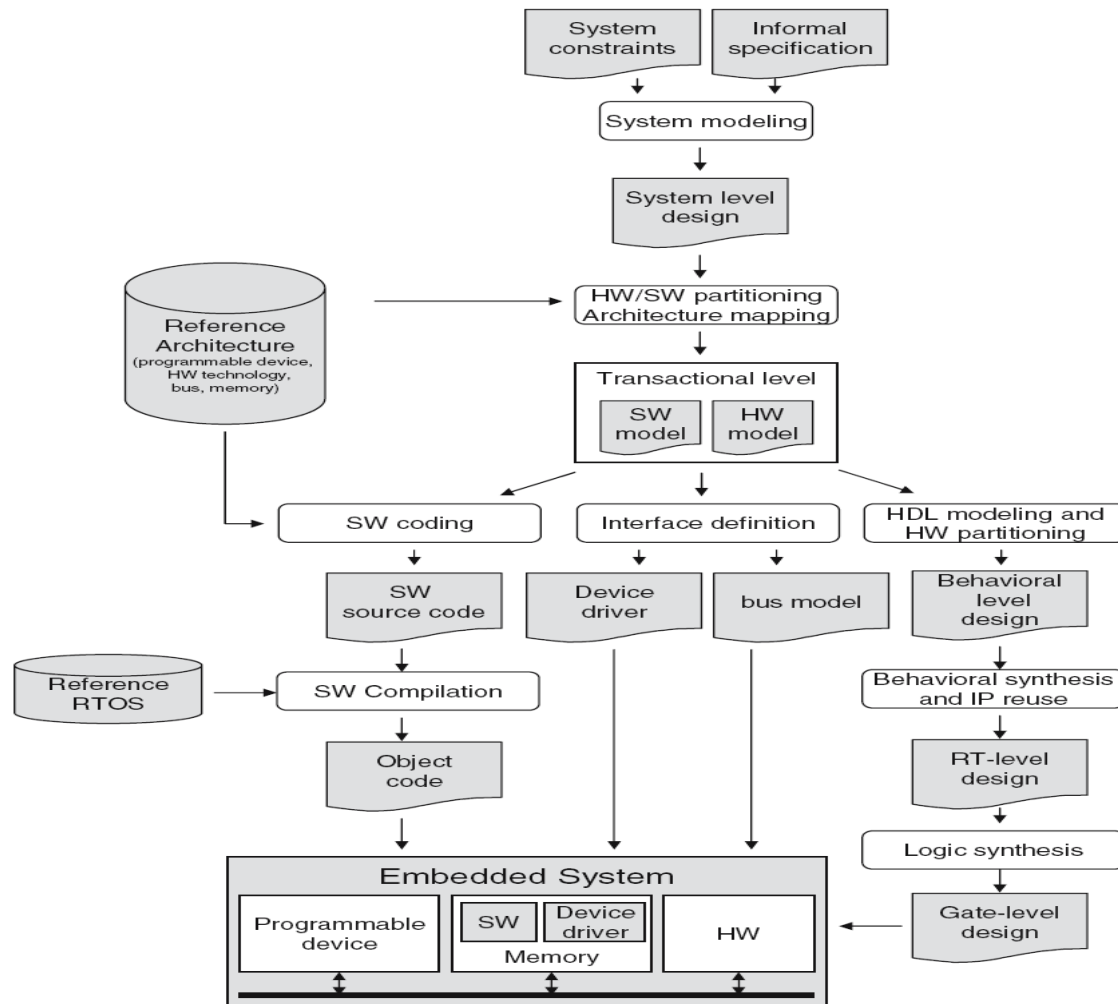
👉 Особенности

- ✓ Ясно дефинирани промени в разбиранията за 'ограничения' между различните приложения и фамилии.
 - ✗ Силиконови системи (system-on-chip)
 - ✗ Класически системи
 - ✗ Други

👉 Валидация и верификация

- ✓ Верификация на различните етапи на разработка
- ✓ Валидация
 - ✗ на модела, на комуникациите, на работата и функционална валидация

Верификация на модела на изграждане



Източник:

Dynamic and Formal Verification of Embedded Systems: A Comparative Survey

Верификация на модела на изграждане

☞ Нива в дизайна на вградената система

- ✓ 'Система'
- ✓ 'Архитектура'
- ✓ 'Програмен код'
- ✓ 'Изпълним код'
- ✓ 'Дефиниция на интерфейси'
- ✓ 'Синтез на поведение'
- ✓ 'Логически синтез'

Дизайн на системата: 'Система'

- ➡ Няма разлика между апаратна и програмна част, т.е. методите за валидация и верификация оценяват системата като цяло.
- ➡ Не се разглеждат комуникационни протоколи между блоковете, анализ на закъснението на разпространение на сигналите, изчислителни алгоритми.
- ➡ Оценяват се гъвкавостта на дизайна, ефективността и функционалността му.

Дизайн на системата: ‘Архитектура’

- ☞ Оценява се обвързването на програмна и апаратна функционалност (на ниво блокове).
 - ✓ Модел на паметта
 - ✓ Комуникационната архитектура между програмна и апаратна част и топология на шината
 - ✓ Програмируемите устройства, върху които ще се изпълнява програмната част
 - ✓ Апаратната технология, която се използва за реализация на апаратните задачи.
- ☞ Оценка на транзакционността - поради отделеността на комуникация от изчисление се използват методи за симулация.

Дизайн на системата: 'Програмен код'

- ☞ Оценява се проблемите от избора на език за програмиране.
- ☞ Основно се оценяват проблемите, свързани с ограниченията на устройствата, върху които ще се изпълняват програмния код и ограниченията, резултат от комуникационния интерфейс.
 - ✓ Апаратната част участва в процеса като 'черна кутия'.
 - ✓ Комуникацията между апаратна и програмна част е не основата на драйвери.

Дизайн на системата: 'Изпълним код'

☞ Оценява се изборът на решение за изпълнение на програмния код:

- ✓ С използването на ОСРВ (операционна системи за реално време) или директно управление на апаратната част
- ✓ Компилативен или интерпретативен вариант.

Дизайн на системата: 'Интерфейсите'

- ☞ Разделянето на функциите на системата, решавани с апаратни средства между отделни модули налага реализация и оценка на интерфейсите между тези модули.
 - ✓ Времевият модел за апаратната и за програмната част е различен (обикновено събитийно базиран – циклично изпълняван) , т.е. трябва преобразуване в двете посоки.
- ☞ Оценка на драйверите за управление на устройствата и на протокола на шината
 - ✓ функционална прозрачност на апаратната част спрямо програмната
 - ✓ надеждност, устойчивост, пропускателна способност

Динамична верификация

- ☞ Оценява коректността на дизайна на основата на симулативни техники
 - ✓ Оценява се реакцията на системата на множество входни стимули, генерирани от симулаторната среда.
 - ✓ Тестовите набори, създадени за по-високо абстрактно ниво се използват и за по-ниските нива, като непрекъсната се добавят и нови.
- ☞ Използва се за определяне на наличието на грешки, но не може да гарантира отсъствието им.

Динамична верификация

- ☞ Може да се използва както за системата като цяло, така и за апаратната и програмната част поотделно.
- ☞ Необходими елементи за оценката на системата
 - ✓ Симулационен модел на дизайна на системата
 - ✓ Симулатор
 - ✓ Тестова среда
 - ✓ Метод за оценка на коректността на резултатите.
 - ✗ Това е основният проблем поради невъзможност за реализиране на изчерпателен тест при симулация.

Динамична верификация

☞ Според метода за оценка на коректността на резултатите съществуват два вида динамична верификация.

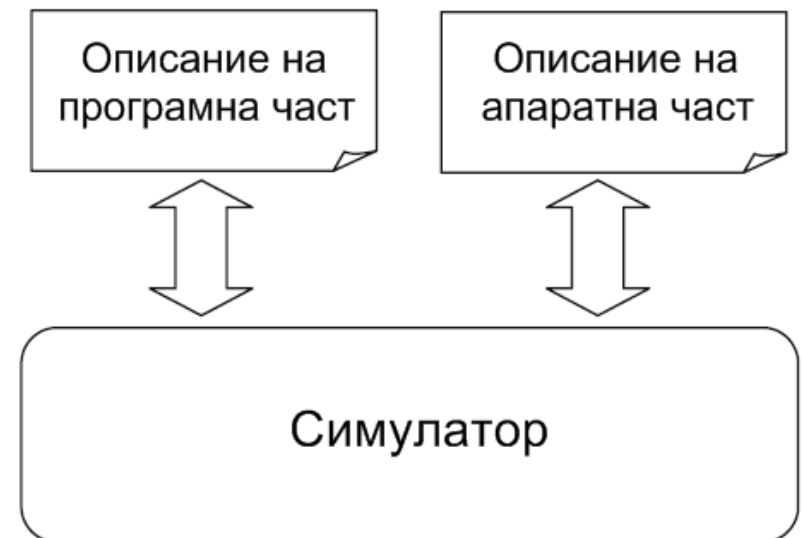
- ✓ Симулация на логиката на системата
 - ✗ Качеството се определя въз основа на покритие на кода (оператор, разклонение, път, цикъл, условие и т.н.)
- ✓ Симулация на дефектите
 - ✗ Качеството се оценява чрез анализ на отношението между открити дефекти и реално дефекти – дефектите се симулират (различни видове и модели) и се оценяват резултатите в с/бездефектна система.
 - ✗ 100 % не гарантират пълно управление на дефектите, а само 100 % откриваемост за симулираните условия.

Ко-симулативни подходи

- ☞ Ко-симулативния подход се използва за оценяване на системата след разделянето на функционалността между програмната и апаратната част.
- ☞ Използват се специално разработени платформи, различаващи се по архитектурна насоченост, ефективност на изпълнението, езици за описание на системата
 - ✓ Хомогенни среди
 - ✓ Хетерогенни среди
 - ✓ Полу-хомогенни среди

Хомогенни среди

- ➡ За симулацията на апаратна и програмна част се използва едно общо ядро.
- ➡ Използва се абстракция от висок ред за представяне на разликата между апаратни и програмни блокове
- ➡ Използва се за
 - ✓ оценка на дизайна
 - ✓ оценка на използване
- ➡ Недостатък
 - ✓ Реализация на много различни средства за апаратна и програмна реализация



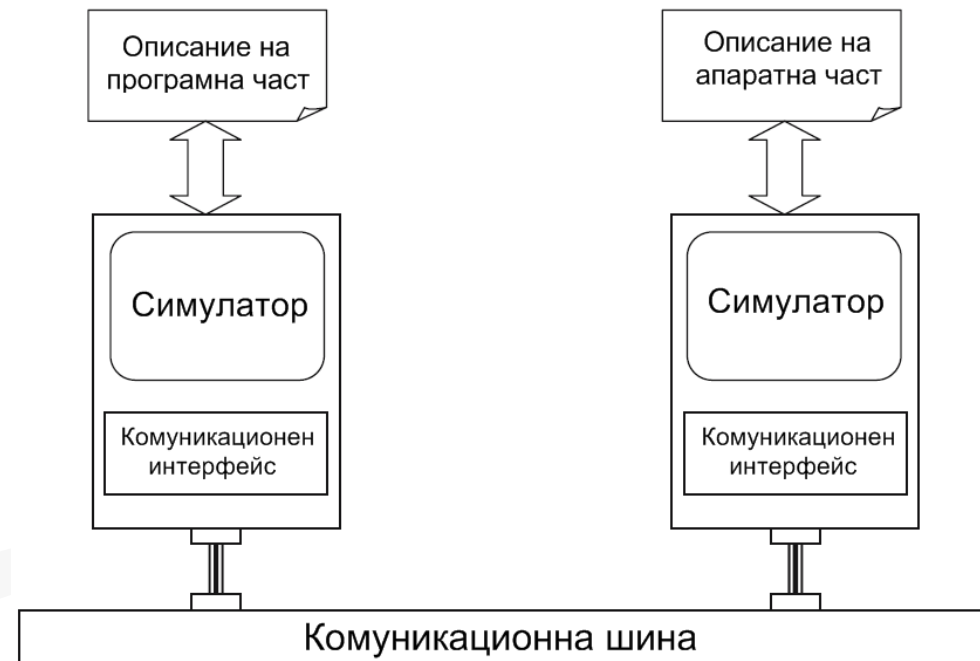
Хетерогенни среди

➡ Много по-точна реализация на поведението на апаратната и програмната част.

- ✓ Използва се по-ниско ниво на абстракция за апаратната част.
- ✓ Програмата част се оценява в изпълним вид.

➡ Същност

- ✓ Изпълняват синхронизация между два и повече симулатора.



Хетерогенни среди

☞ Основен проблем:

- ✓ Ефективността на симулацията на връзката между събитийно-ориентираната апаратна част и симулатора на инструкции, необходим за програмната част
 - ✗ Необходим е симулатор на комуникационния канал (шината) за връзка 'апаратни елементи-модули за изпълнение на програмата' – моделиране на сигналите.

☞ Много бавна симулация.

- ✓ Симулацията на всички сигнали по шината съществено намалява скоростта.

Полу-хомогенни среди

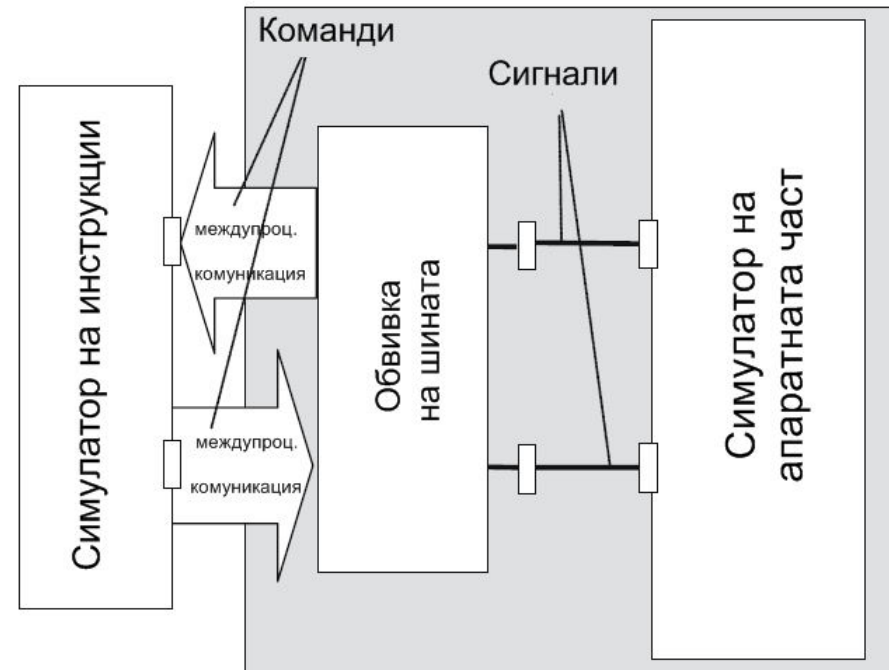
Същност

- ✓ При тях няма симулация на шината на ниско ниво.
 - ✗ Реализира се чрез малък брой функции, предаващи информацията за нужното време без да симулират всички сигнали.
- ✓ Хомогенни от гледна точка на използвания програмния език и хетерогенни от гледна точка на симулацията.

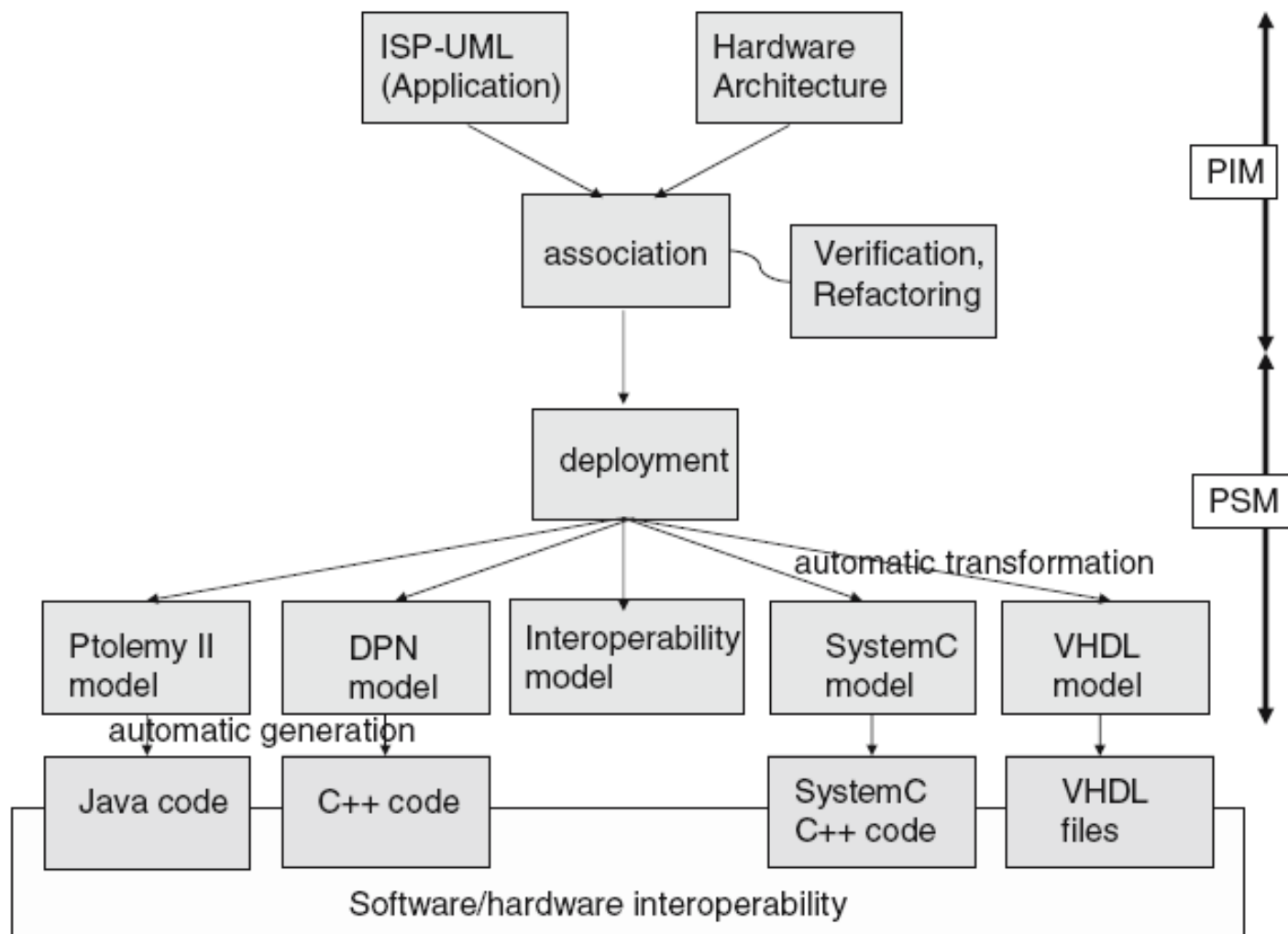
 По-ниска точност от хетерогенните среди, но скоростта на симулацията е по-висока.

Полу-хомогенни среди

- ☞ Две основни системи
 - ✓ Междупроцесна комуникация
 - ✓ Обвивка на шината
 - ✗ Програмно абстрактно нивс
 - ✓ Най-често интерфейсът между обвивката на шината и симулатора на инструкции е променяем



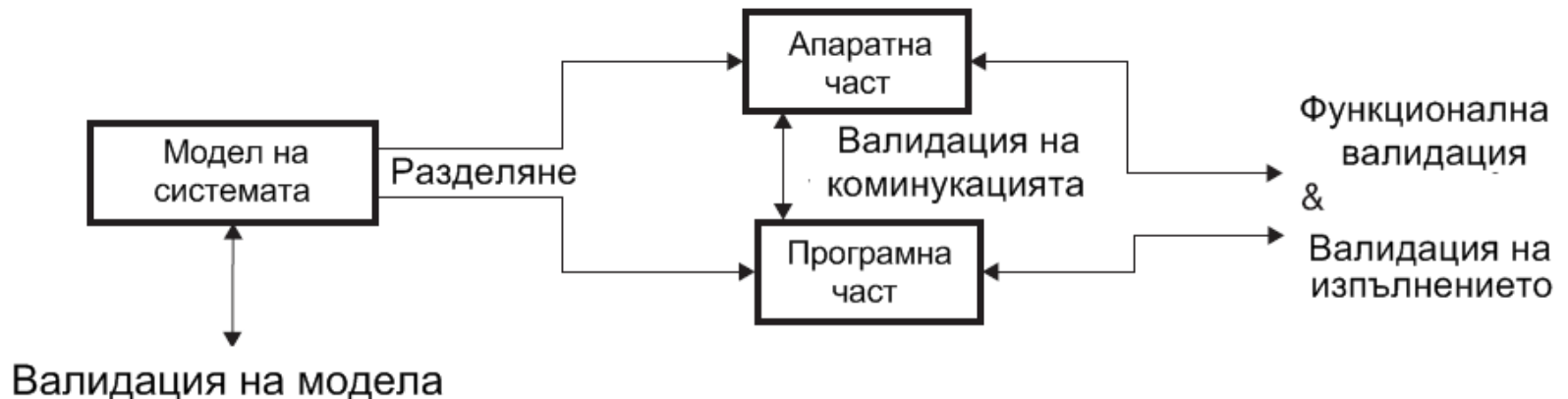
UML-базирана верификация



Источник:
Dynamic and
Formal Verification of
Embedded Systems: A
Comparative
Survey

Валидация на вградени системи

- ➡ Валидация на модела
- ➡ Валидация на реализацията
 - ✓ Валидация на Високо ниво на реализация: комуникация между блоковете
 - ✓ Валидация на ниско ниво
 - ✗ Валидация на изпълнението
 - ✗ Функционална валидация



Валидация на вградени системи

👉 Проблеми:



Валидация на модела

- ☞ **Общо описание на модела**
 - ✓ Крайни автомати (Finite-State Machines - FSMs)
 - ✓ Комуникиращи си крайни автомати (Communicating FSMs)
 - ✓ Базирани на графици от съобщения (Message Sequence Chart–Based Models)
- ☞ **Симулационни модели**
 - ✓ Симулация на крайни автомати
 - ✓ Симулация на модели, базирани на графици от съобщения
- ☞ **Моделно-базирано тестване**
- ☞ **Проверка на модела**
 - ✓ Проверка на спецификацията
 - ✓ Проверка на процедурно ниво

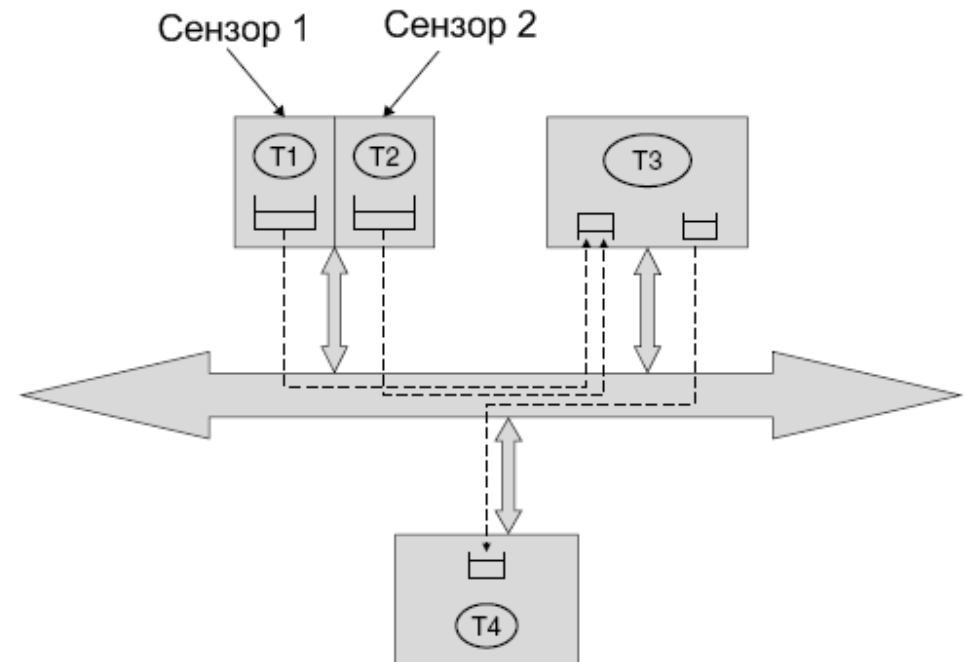
Валидация на модела

Основен проблем

- ✓ Разликата между разработваната система и платформата, върху която ще се имплементира
 - ✗ “Моделът на поведение представлява описание на механизма на промяна на състоянията при комуникация между обекти.”
- ✓ При вградените системи има много силна обвързаност между платформа и разработвана система.
 - ✗ Валидация на платформения модел
 - ✗ Валидация на системния модел на основата на резултатите за платформения модел

Валидация на модела

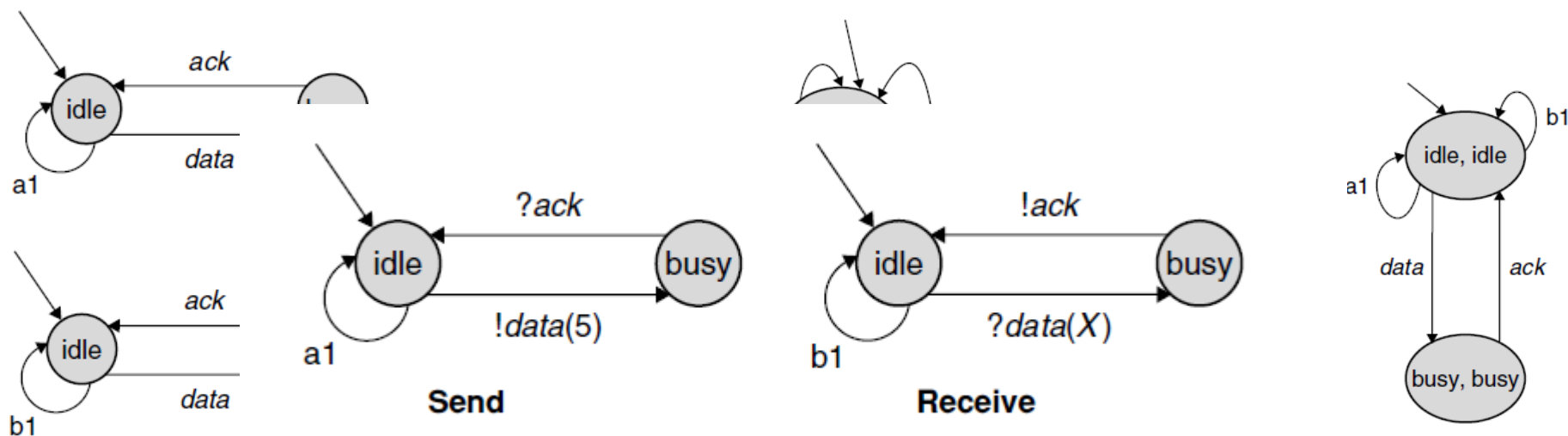
- 👉 Основен проблем
 - ✓ Функционални изисквания
 - ✓ Изисквания за времеви ограничения



Валидация на модела

👉 Общо описание на модела

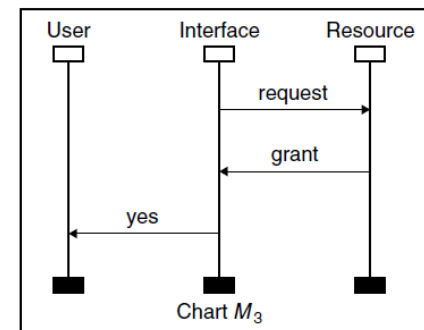
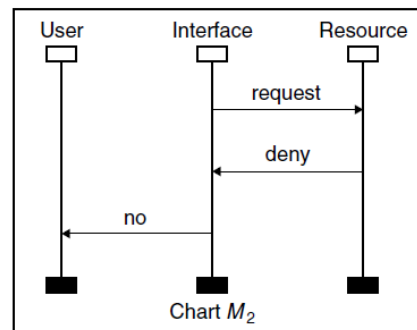
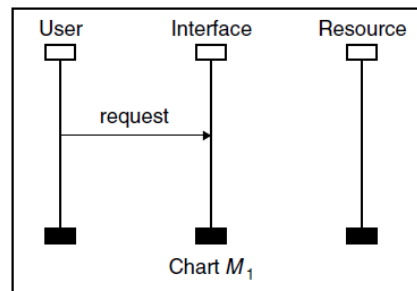
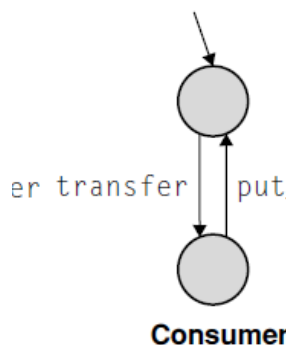
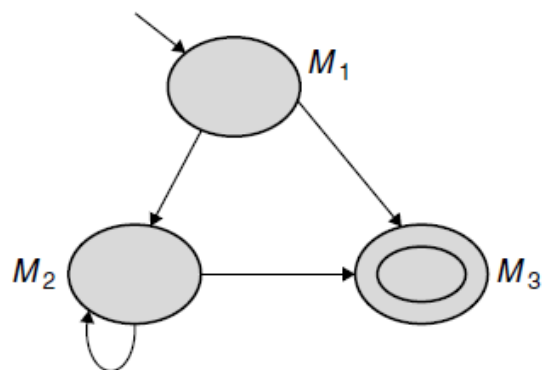
- ✓ Крайни автомати (Finite-State Machines - FSMs)
- ✓ Комуникиращи си крайни автомати (Communicating FSMs)
- ✓ Базирани на графици от съобщения (Message Sequence Chart-Based Models)



Валидация на модела

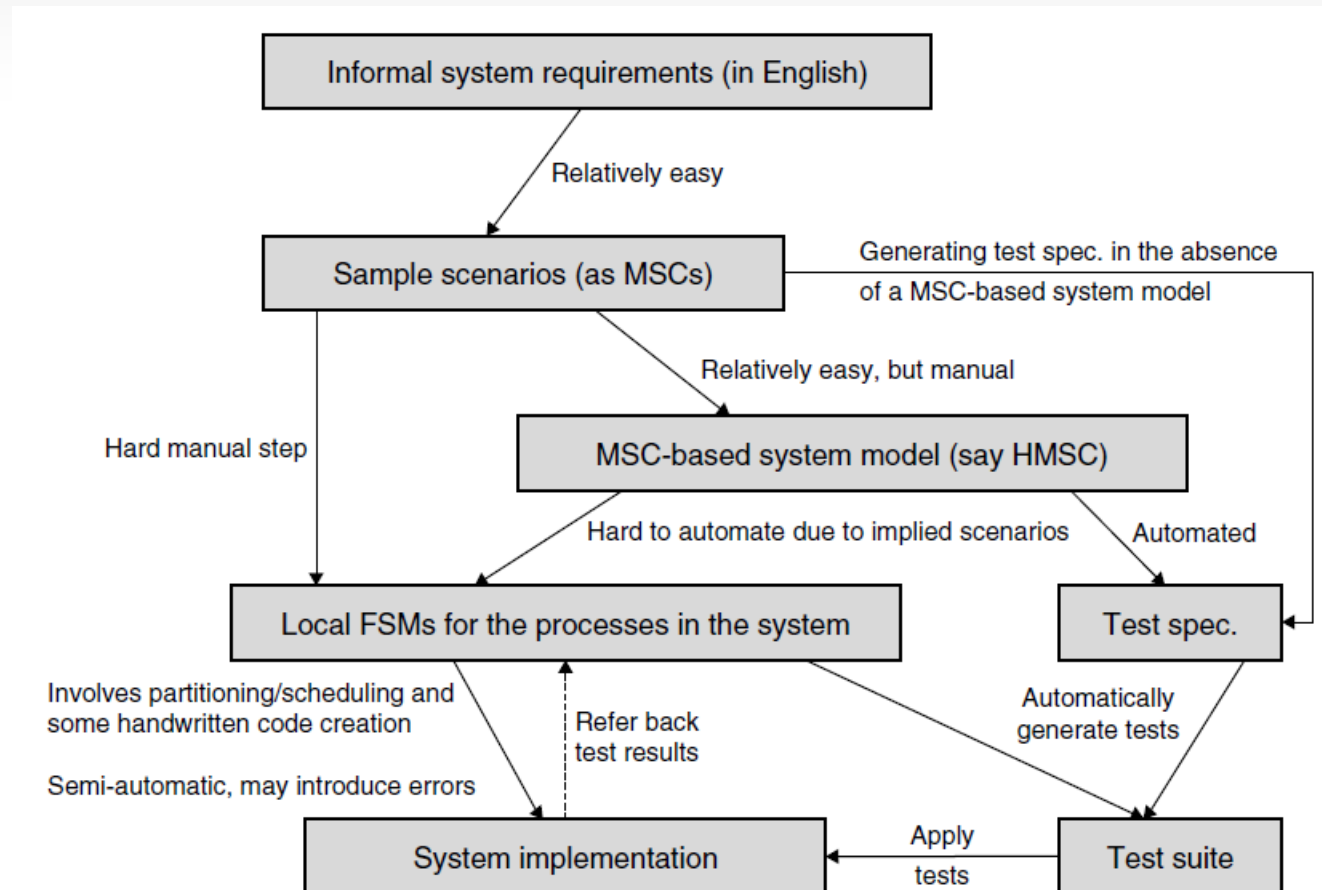
Общо описание на модела

- ✓ Крайни автомати (Finite-State Machines - FSMs)
- ✓ Комуникиращи си крайни автомати (Communicating FSMs)
- ✓ Базирани на графици от съобщения (Message Sequence Chart-Based Models)

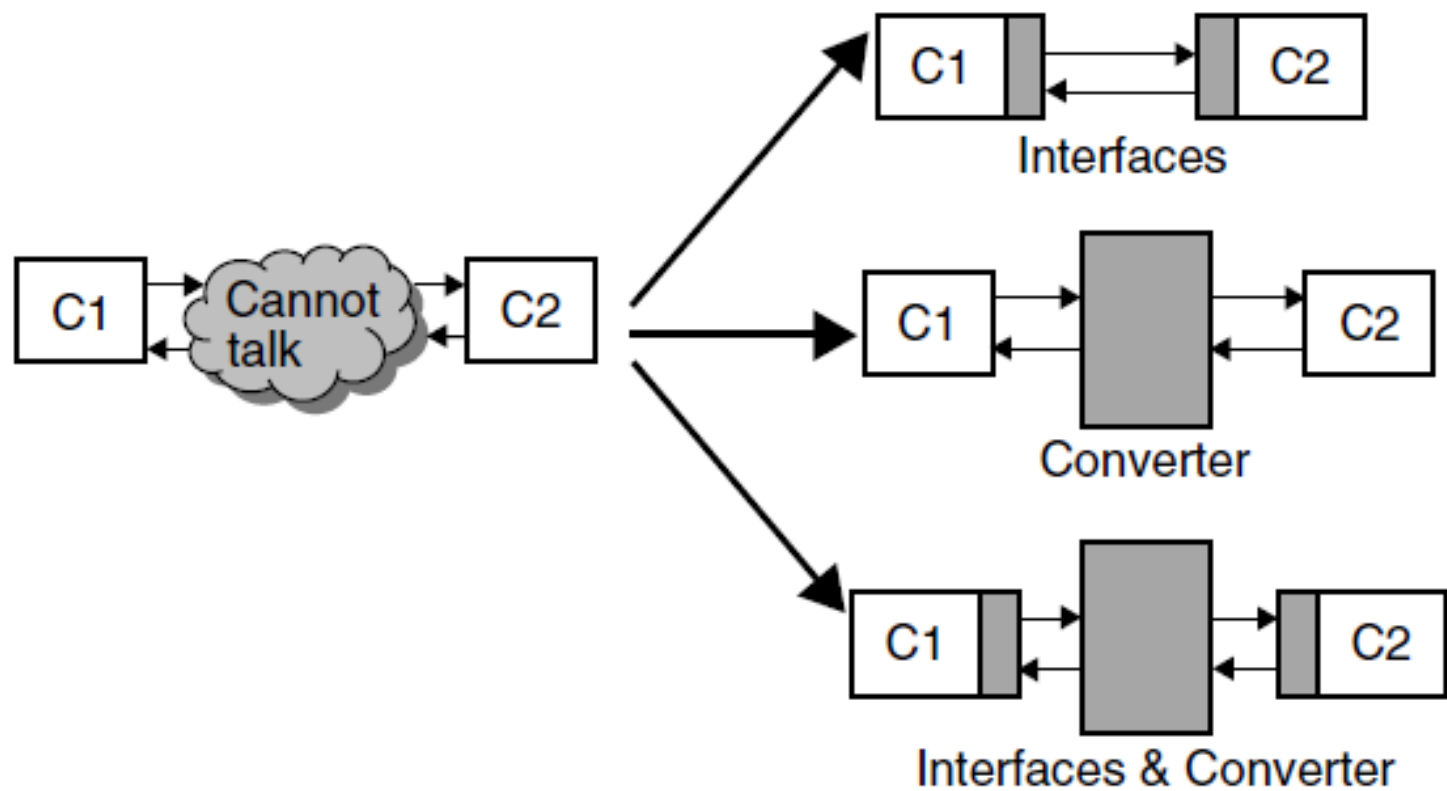


Валидация на модела

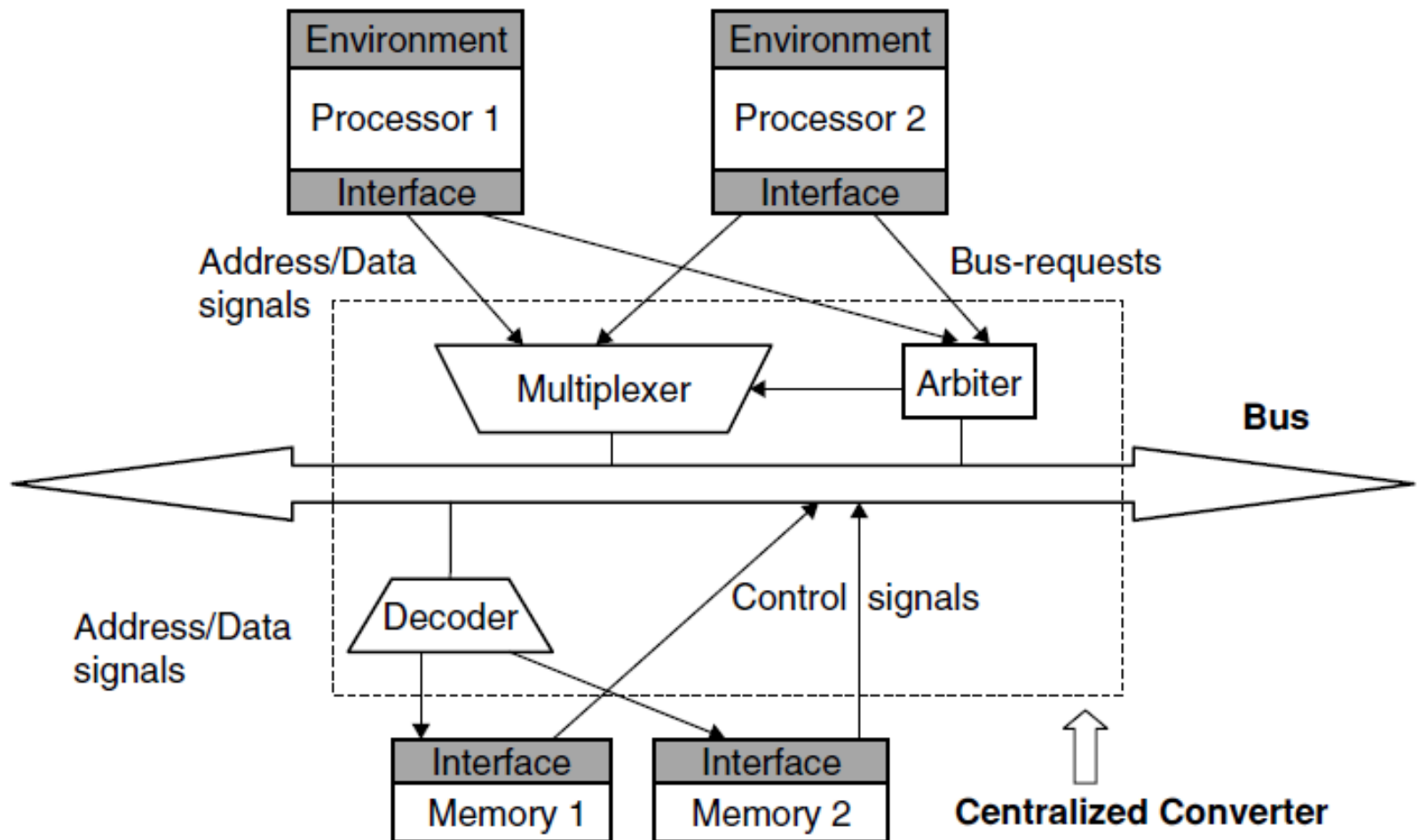
☞ UML-базирано моделиране



Валидация на комуникацията



Валидация на комуникацията

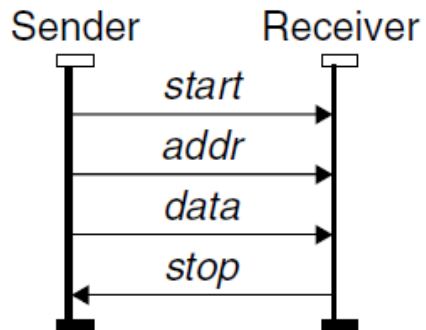


Валидация на комуникацията

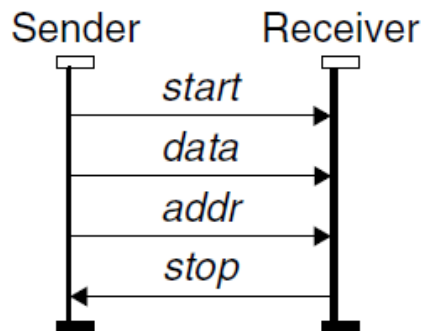
☞ Обща несъвместимост

- ✓ Приемане/Предаване на сигналите в различен ред
 - ✗ Еднакви сигнали, но предавателя има различна задължителна последователност от приемника
- ✓ Използване на различно множество от сигнали
- ✓ Разлика във формат на данните
 - ✗ Например предаване по битове, а приемане на байте
- ✓ Объркване на данните
 - ✗ Най-често данни с адреси
 - ✗ Данни от различни пакети

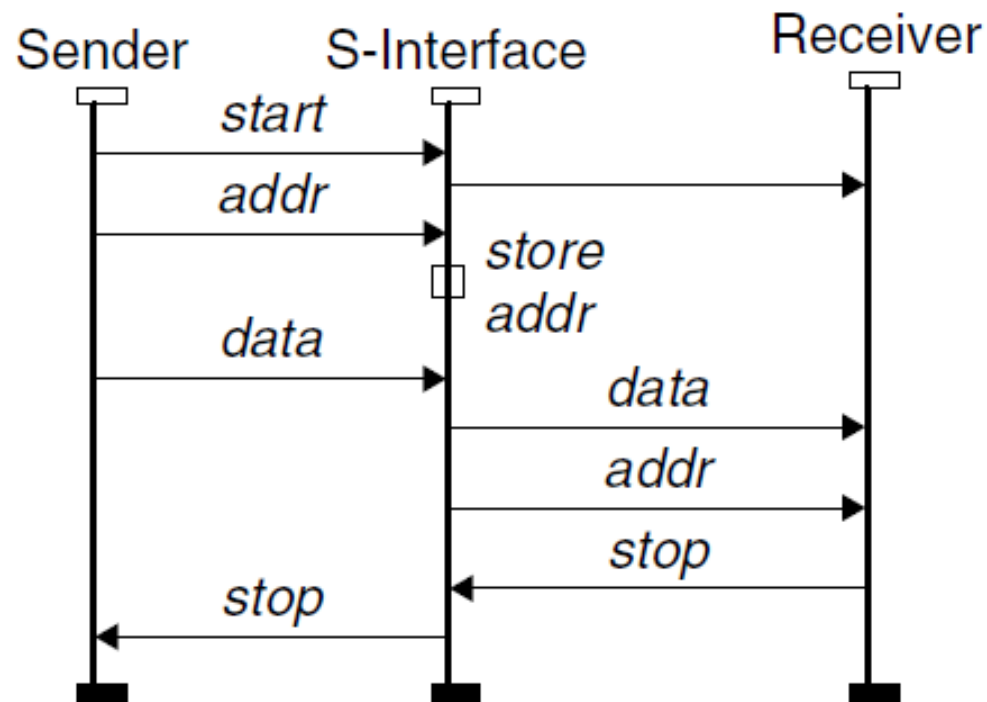
Приемане/Предаване на сигналите в различен ред



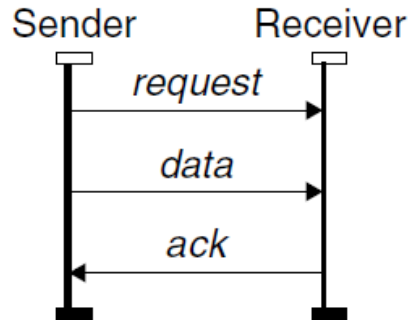
Sender ->



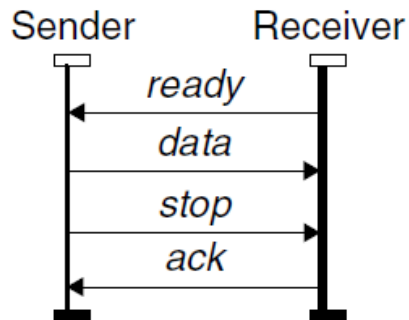
Receiver ->



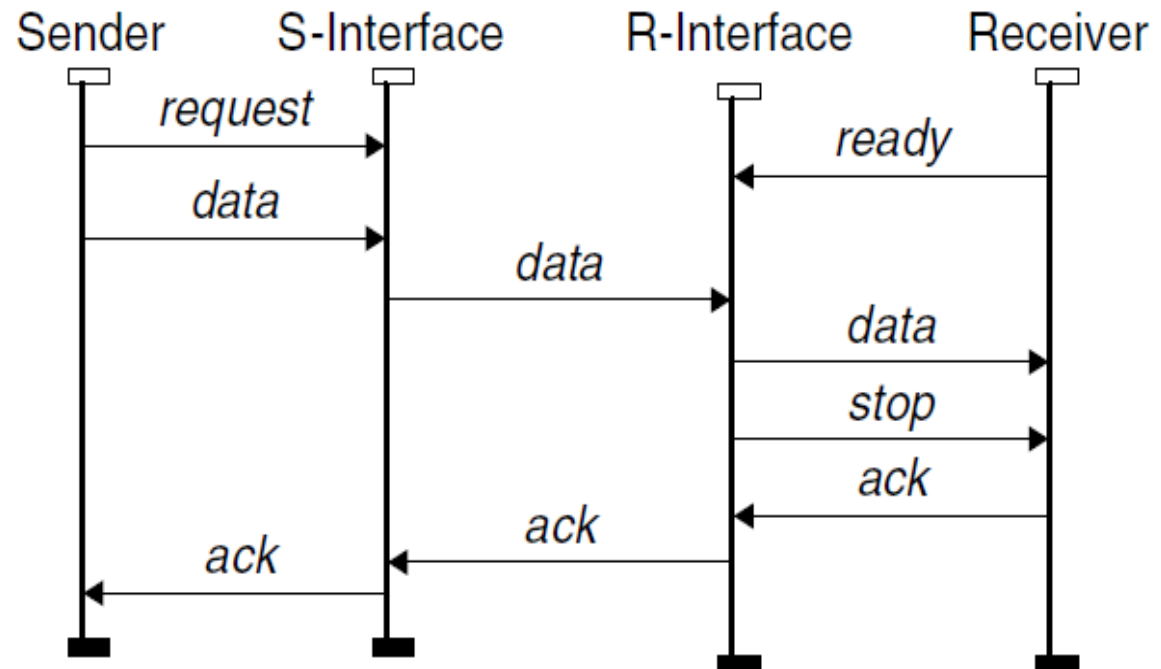
Различно множество от сигнали



Sender ->

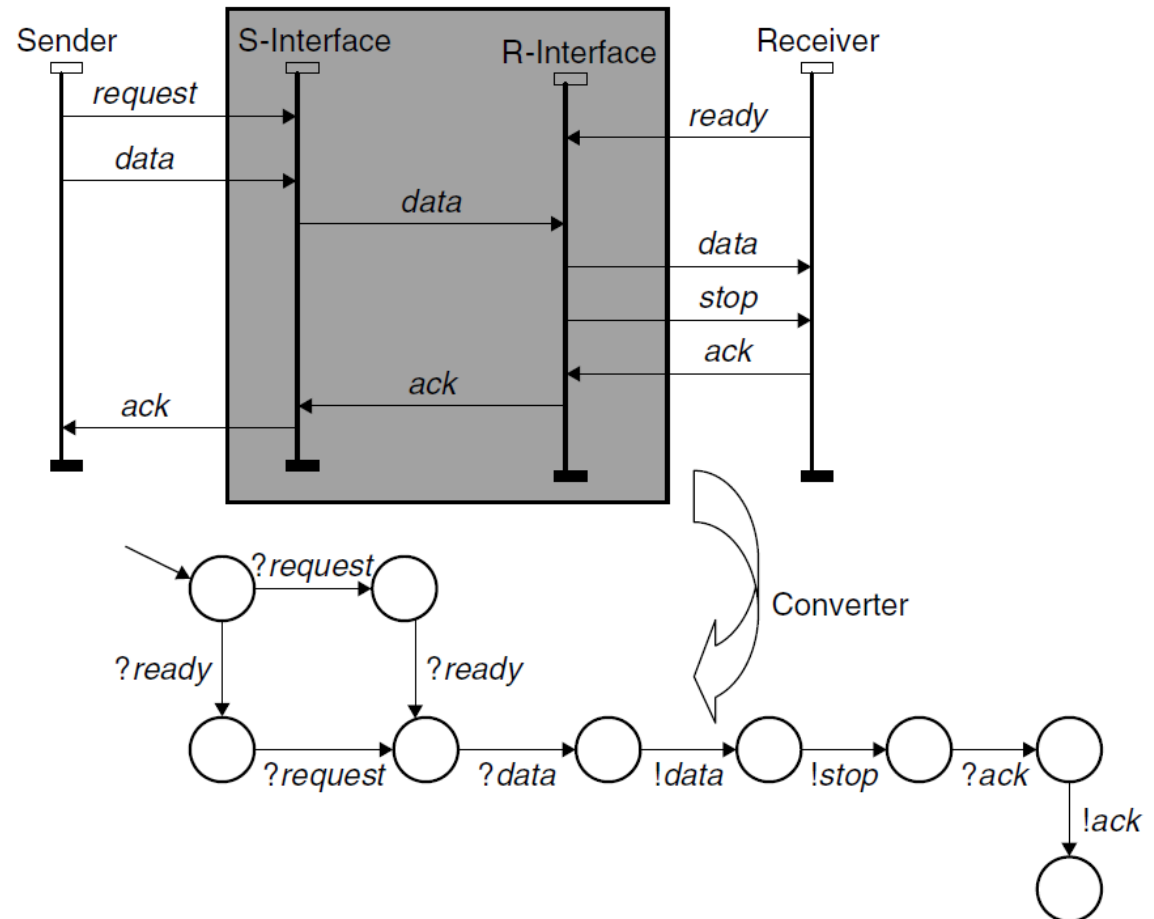


Receiver ->

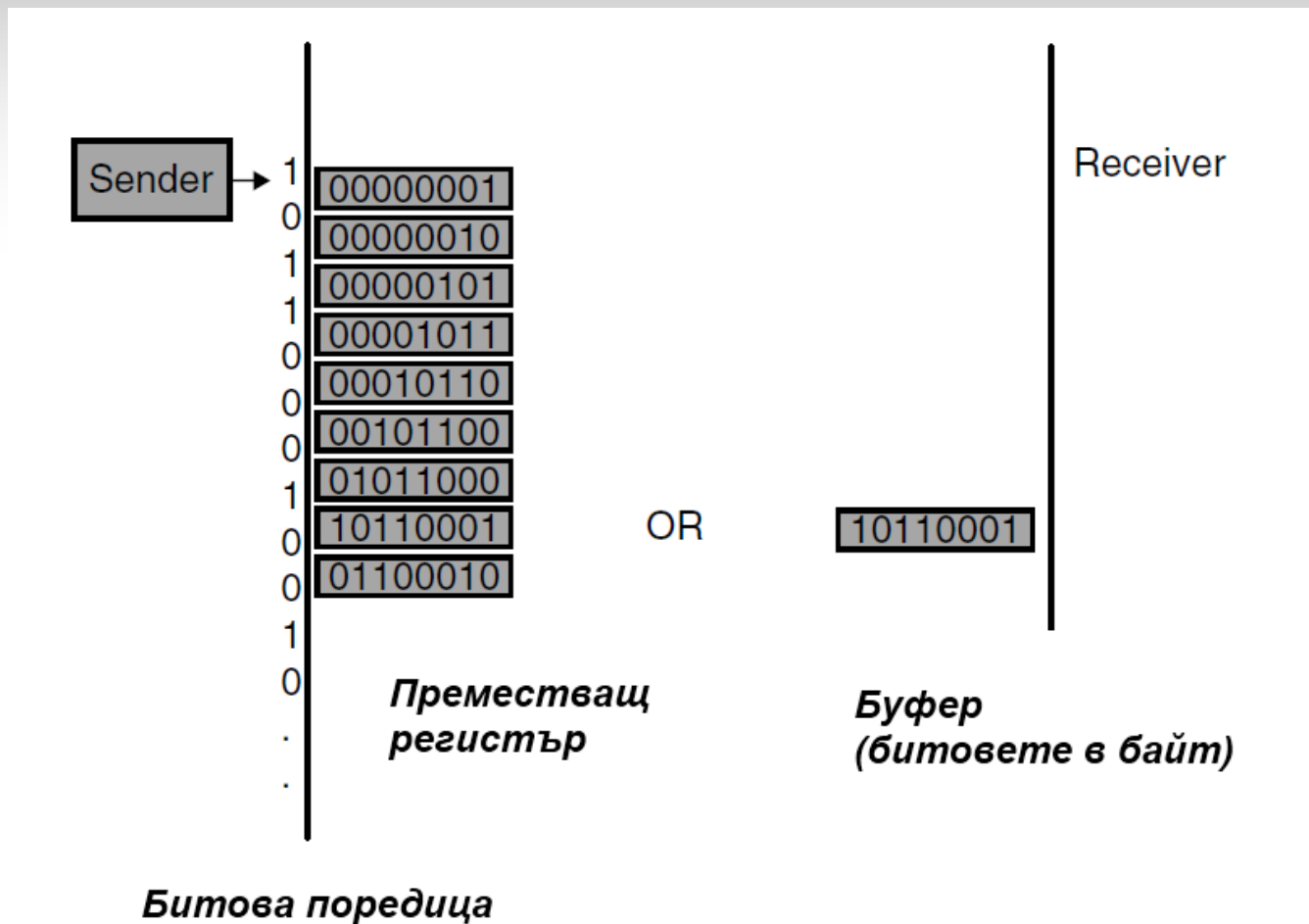


Различно множество от сигнали

☞ Централизиран конвертор на сигнали

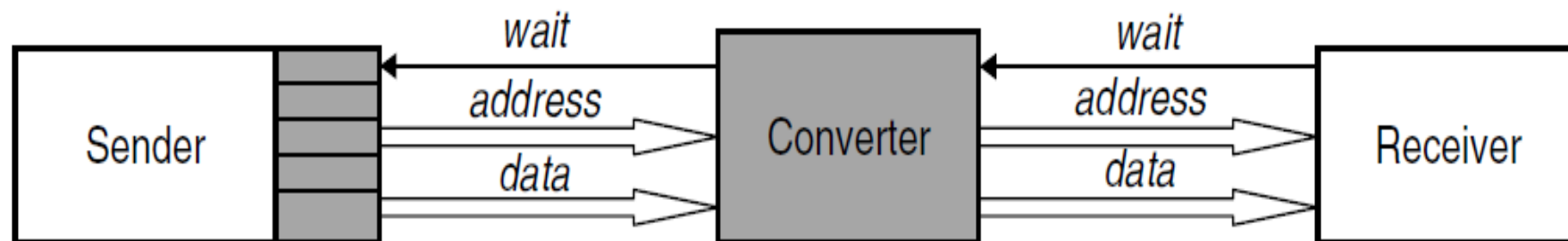


Разлика във формат на данните



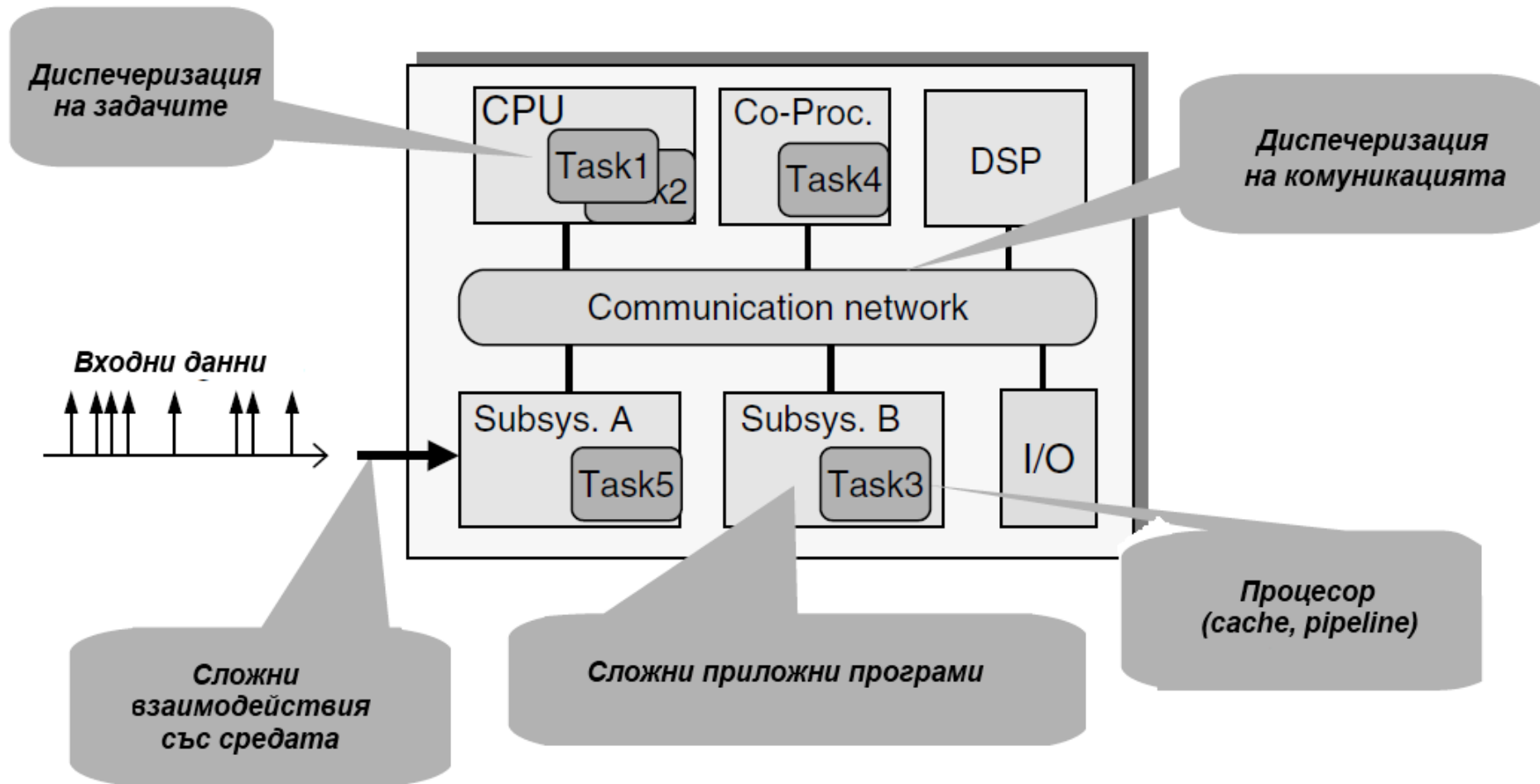
Объркване на данните

👉 Синхронизационни проблеми

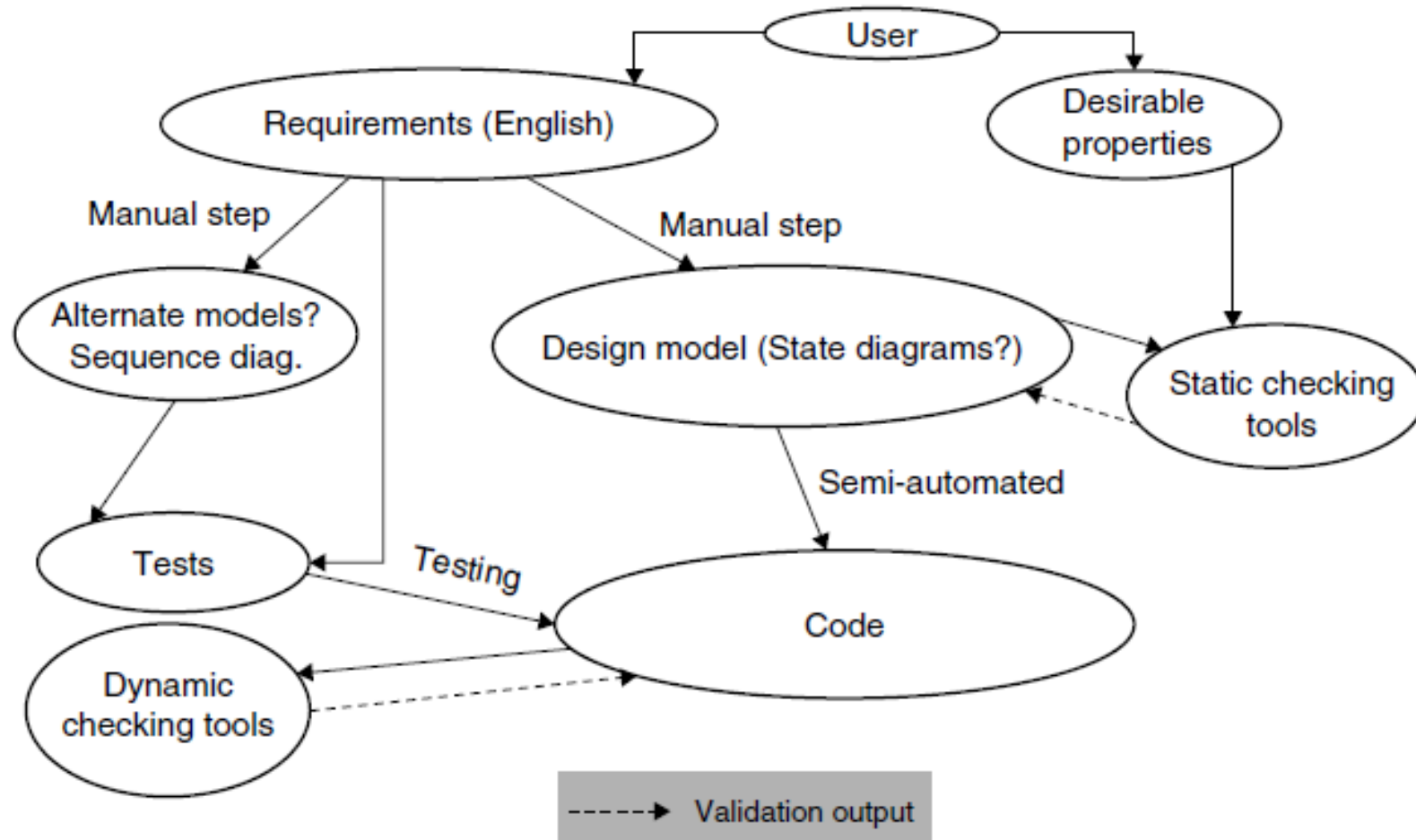


**Буфер за съхранение
на *addr/data* когато
wait е вдигнат**

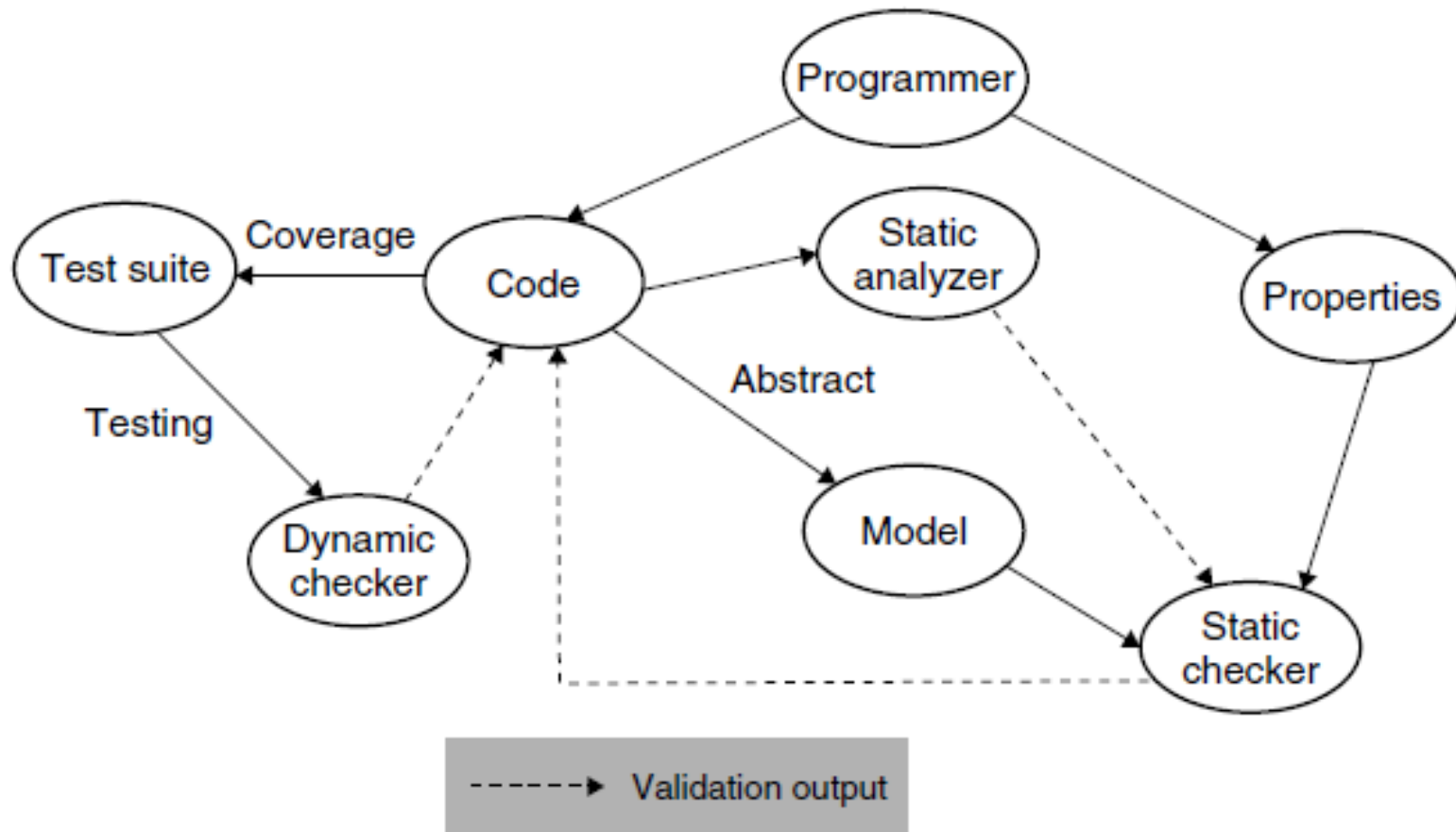
Валидация на изпълнението



Функционална валидация



Функционална валидация



Въпроси ?

Валидация и Верификация на Програмни проекти

Особености при основни архитектурни
стили

Data Flow

☞ Същност:

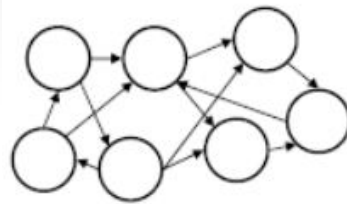
- ✓ Данните контролират изчисленията
 - ✗ Структурата на дизайна се определя от нормалното движение на данни от компонент на компонент
- ✓ Моделът на потока от данни е зададен имплицитно
 - ✗ Това е единствената форма на комуникация между компонентите

☞ Възможни варианти:

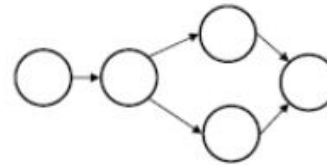
- ✓ Как се управлява контрола – зареждане или извличане (push versus pull)
- ✓ Степен на паралелна обработка (конкурентни процеси)
- ✓ Топология

Data Flow

👉 Модели на потоци от данни



произволен



линеен



цикличен
(с ограничение)

👉 Видове архитектури

- ✓ Пакетна обработка (Batch sequential)
- ✓ Потокови мрежи (pipes&filters)
- ✓ Архитектури със затворен цикъл на управление

Data Flow

👉 Елементи

✓ Компоненти

- ✗ Интерфейсът им е входни и изходни портове
- ✗ Изчислителен модел - *Четене от входен порт; Изчисление; Запис в изходен порт*

✓ Свързващи елементи (конектори)

- ✗ Еднопосочни, най-често асинхронни и буферирани
- ✗ Изчислителен модел - *Транспорт от изходен порт до входен порт*

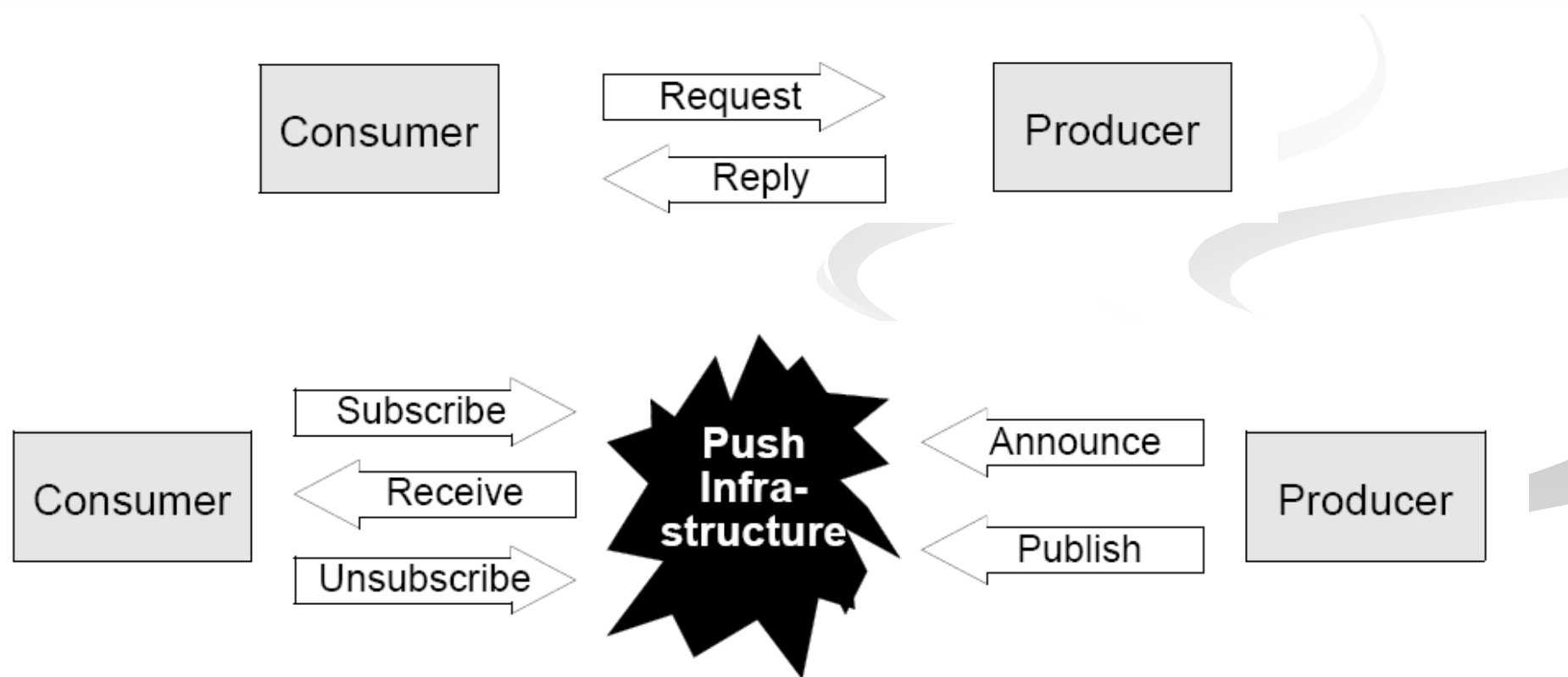
✓ Системи

- ✗ Произволни графи
- ✗ Изчислителен модел - *Функционална композиция*

Push-Based Style

☞ Същност:

- ✓ Класическа задача 'Производител-Потребител'



Push-Based Style

- ➡ Различава се съществено от базираните на извличане (pull-based), например Web-приложенията.
- ➡ Елементи
 - ✓ Компоненти – производител/-и и потребител/-и
 - ✓ Конектори – канали, разпространител (broadcaster), транспортна система, повторители, кеш, прокси
- ➡ Асиметричен комуникационен модел
 - ✓ Производителите не са приемници.
 - ✓ Може да има няколко производителя и всеки да има няколко потребителя (много повече, спрямо архитектурите от вида '*Event-based*')
- ➡ Основени проблеми
 - ✓ Състезание на сигнали (race condition)
 - ✓ Взаимна блокировка (deadlock)

Push-Based Style

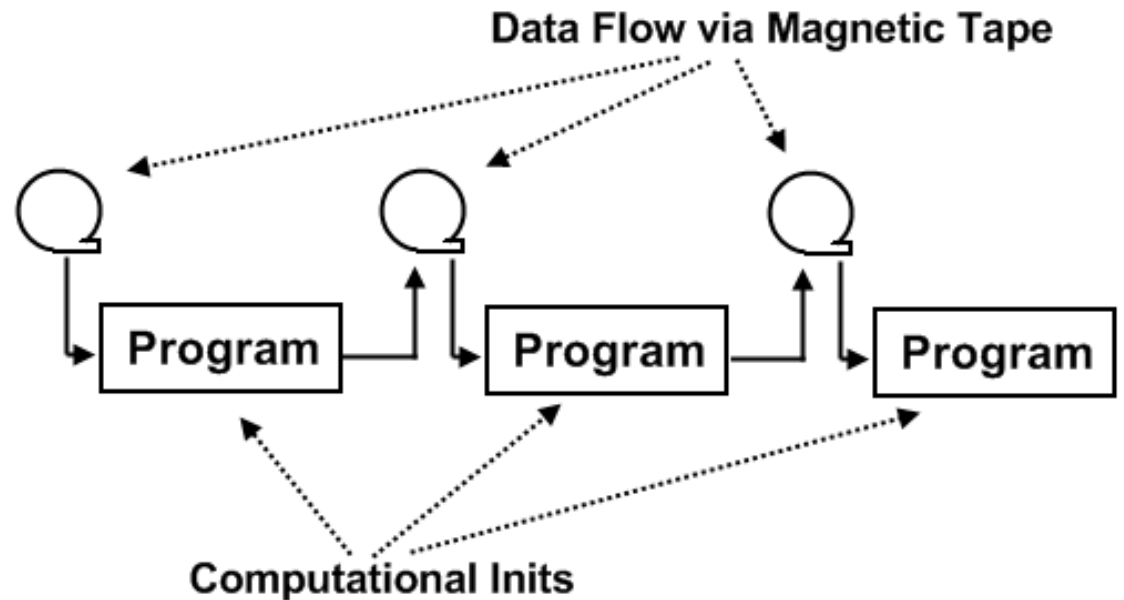
```
int itemCount = 0;
```

```
procedure producer() {  
    while (true) {  
        item = produceItem();  
  
        if (itemCount == BUFFER_SIZE) {  
            sleep();  
        }  
  
        putItemIntoBuffer(item);  
        itemCount = itemCount + 1;  
  
        if (itemCount == 1) {  
            wakeup(consumer);  
        }  
    }  
}
```

```
procedure consumer() {  
    while (true) {  
  
        if (itemCount == 0) {  
            sleep();  
        }  
  
        item = removeItemFromBuffer();  
        itemCount = itemCount - 1;  
  
        if (itemCount == BUFFER_SIZE - 1) {  
            wakeup(producer);  
        }  
  
        consumeItem(item);  
    }  
}
```

Пакетна обработка (Batch Sequential)

- ➡ Компонентите са независими програми
- ➡ Конекторите са някаква форма на медия
- ➡ Всяка програма завършва напълно работата си преди да започне следващата



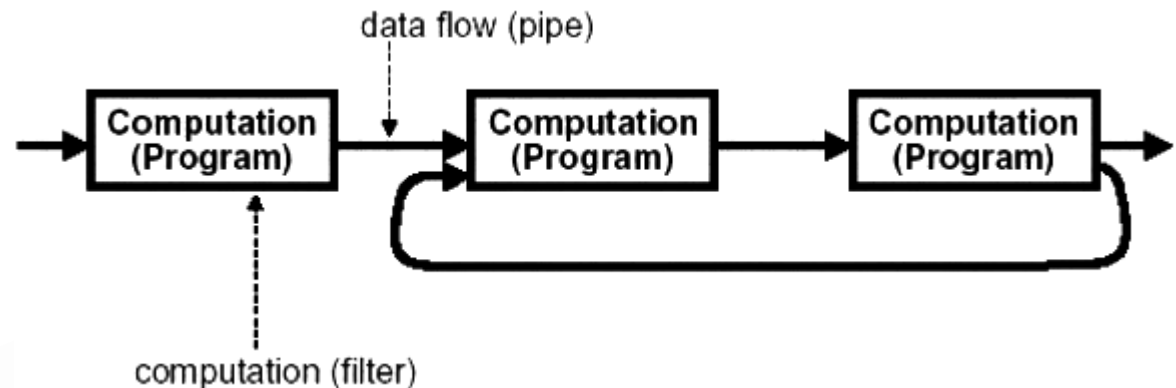
Пакетна обработка

☞ Особенности

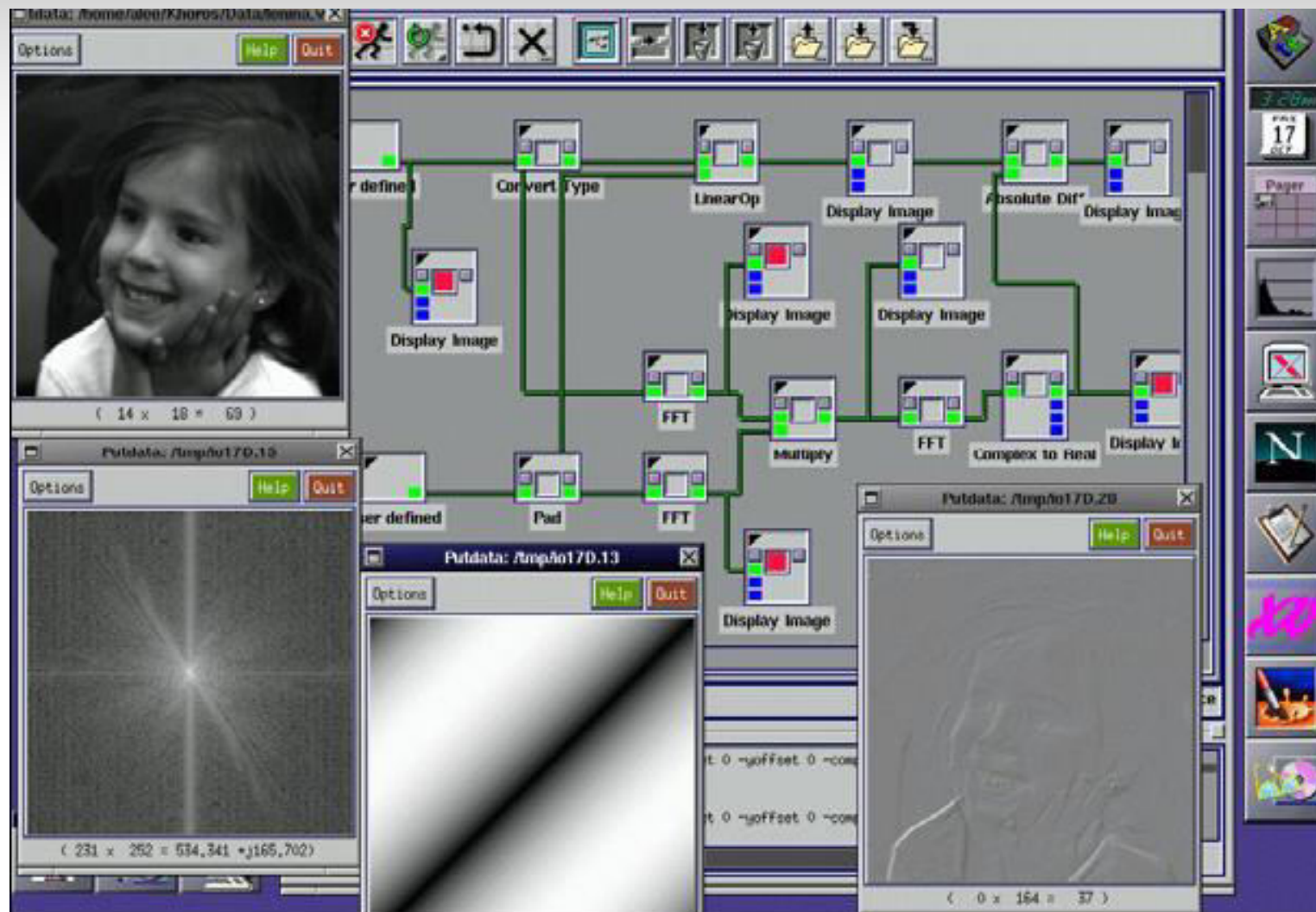
- ✓ Лимитиран обем на преносната медия, напр. дисково пространство.
- ✓ Блокова система на диспечирание на процесорното време
- ✓ Не реално-времеви приложения, ориентирани към пакетна обработка, т.е. използване за бизнес приложения:
 - ✗ Дискретни транзакции на предварително определен тип данни на периодични интервали
 - ✗ Създаване на периодични отчети, въз основа на периодично актуализиране на данни данни

Pipes and Filters

- ☞ Това са системи за потокова обработка на информацията.
 - ✓ Данните се обработват инкрементално.
 - ✓ Всяка стъпка на обработка е реализирана с отделен компонент (filter), като тяхната рекомбинация позволява изграждането на фамилия от системи.
 - ✓ Изискват език за програмиране и ОС.



Pipes and Filters



Pipes and Filters

👉 Варианти

- ✓ Филтрите може да са независими, без общо състояние и памет.
- ✓ Филтърът не знае кой е преди и кой е след него в потока.
- ✓ Коректността на резултата може да не зависи от реда, в който филтрите се изпълняват

👉 Къде ?

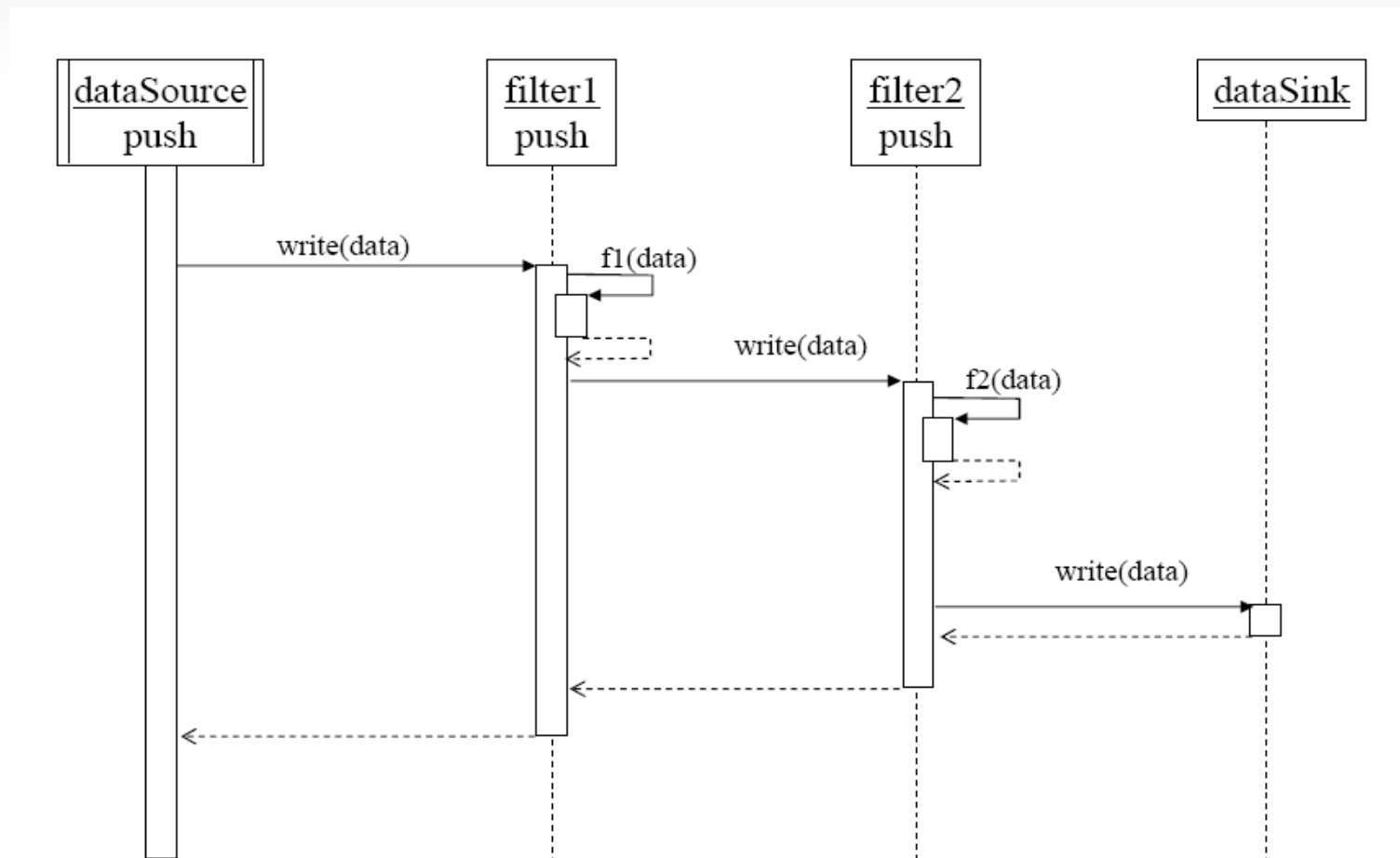
- ✓ Unix pipes
- ✓ lex/yacc-based compiler
 - ✗ Scanning->parsing->semantic analysis->code generation
- ✓ Обработка на изображения
- ✓ Сигнални обработки
- ✓ Аудио и видео стриминг (audio and video streaming)

Pipes and Filters

- ➡ Основен проблем – ‘Какво кара данните да се движат’?
 - ✓ Зареждане (Push): източникът ги зарежда в следващия в потока
 - ✓ Извличане (Pull): приемникът (data sink) ги извлича от предходния
 - ✓ Комбинирана форма (Push/Pull): филтърът активно извлича данните от потока, извършва изчисленията и ги зарежда в потока на следващия.
 - ✗ Ако има повече от един такъв филтър е необходим механизъм за синхронизация.
 - ✓ Пасивно форма: нищи не извърва, а е само генератор на данни или приемник (sink).

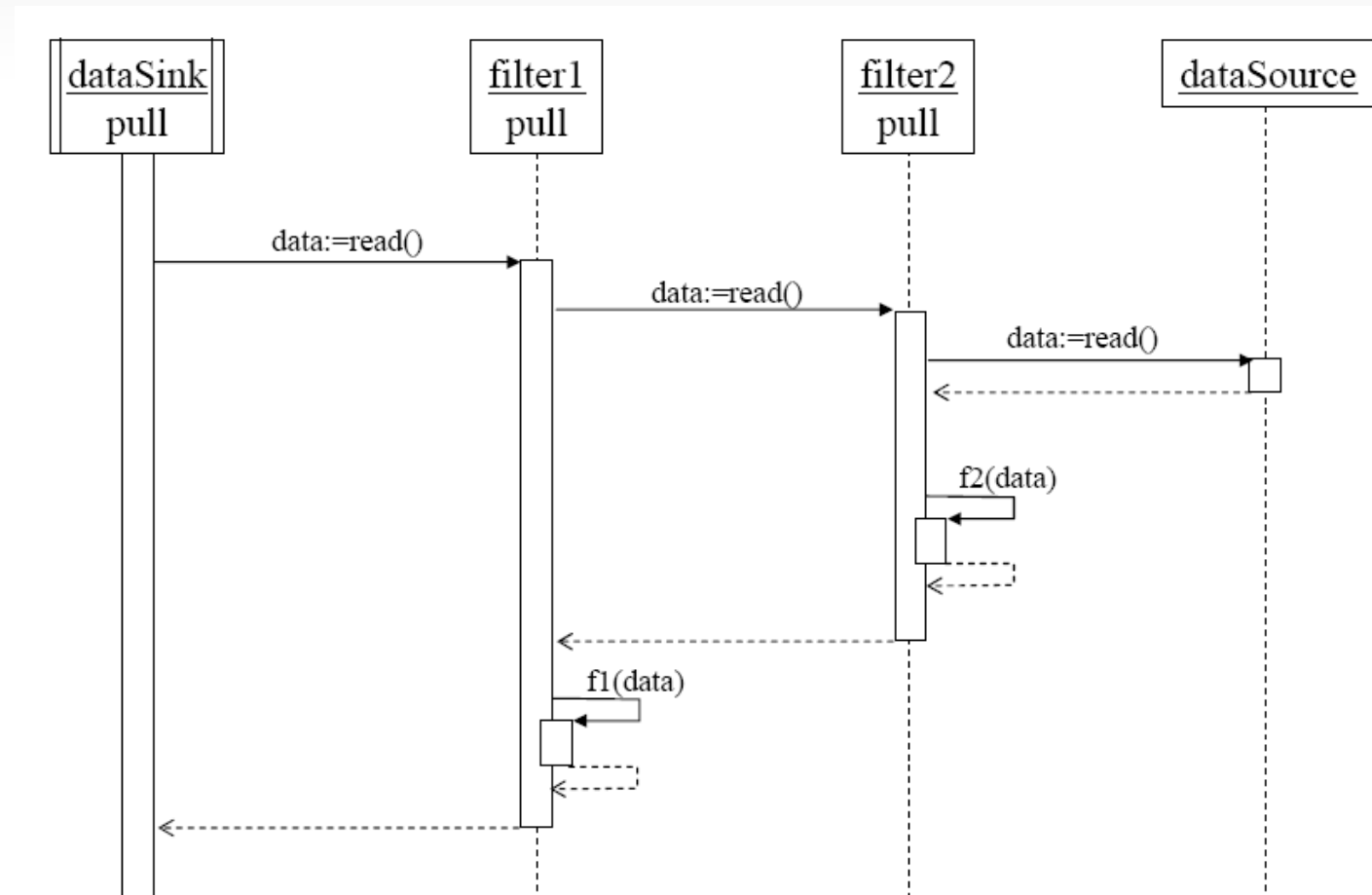
Зареждане и извличане на данни

👉 Зареждане (Push) с един активен източник



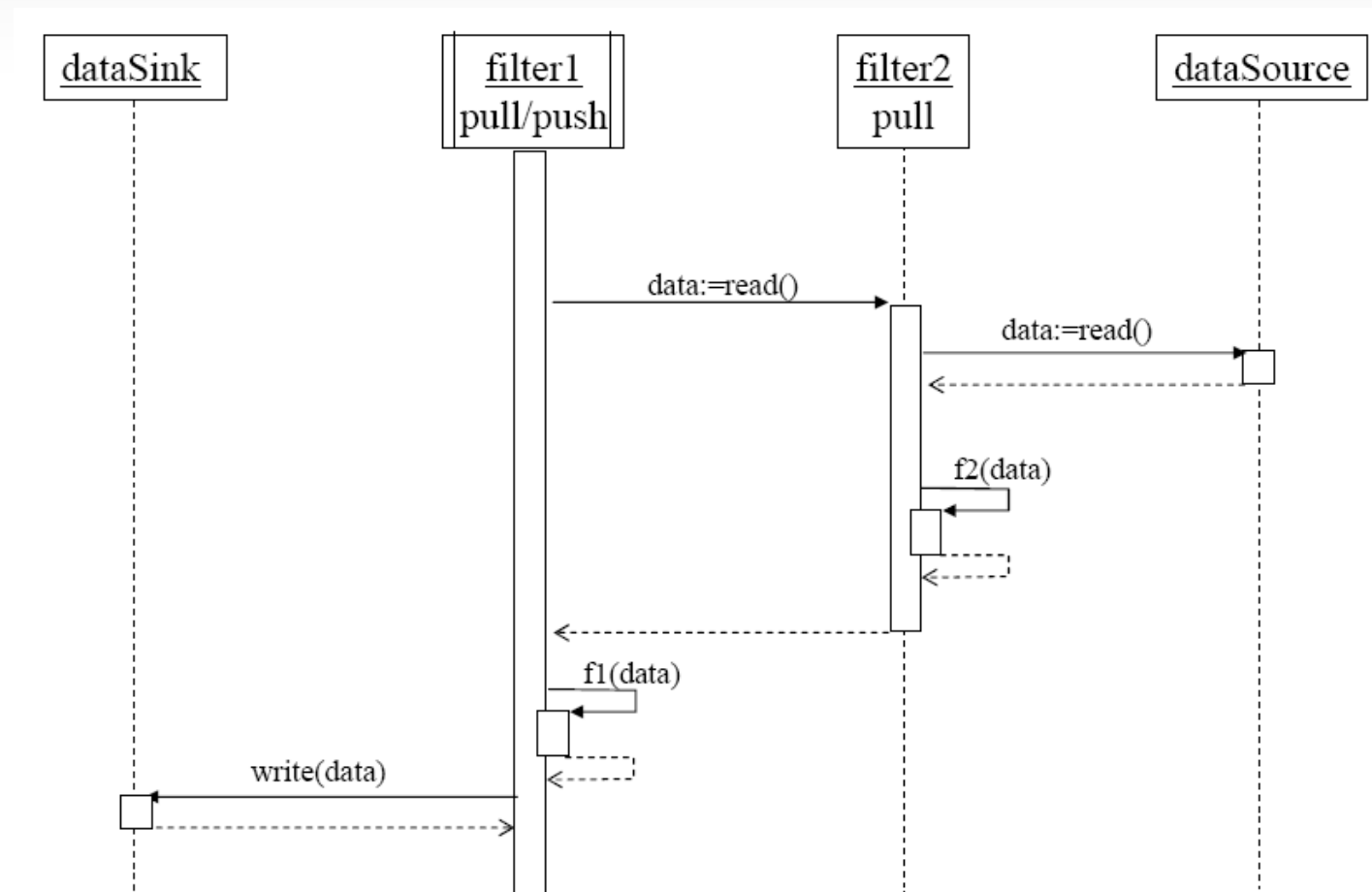
Зареждане и извличане на данни

👉 Извличане (Pull) с един активен приемник



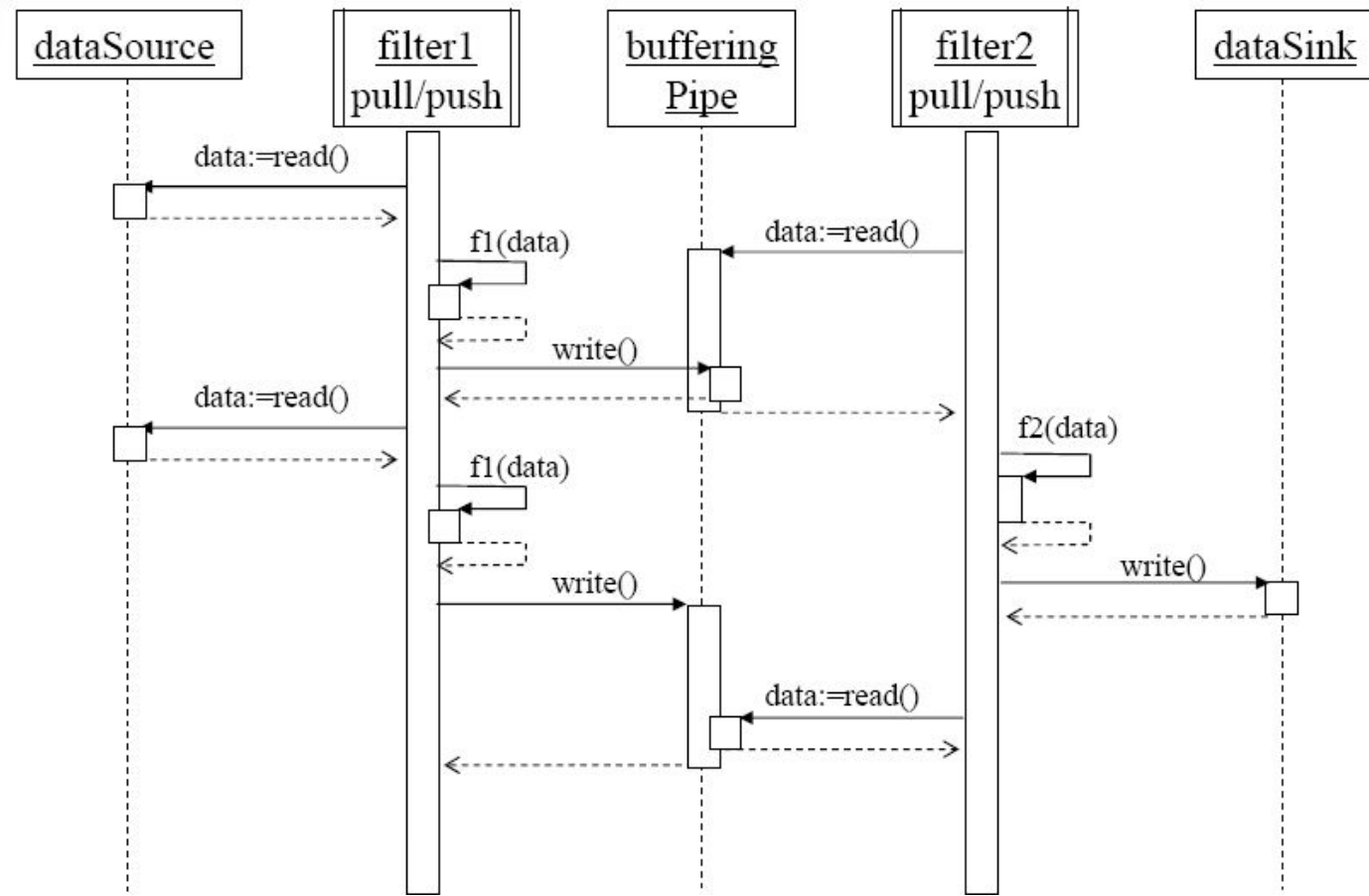
Зареждане и извличане на данни

- Комбинирана форма (Push-Pull) с пасивни източник и приемник



Зареждане и извличане на данни

- Комбинирана форма (Push-Pull) с активни филтри и синхронизационен буфер



Pipes and Filters

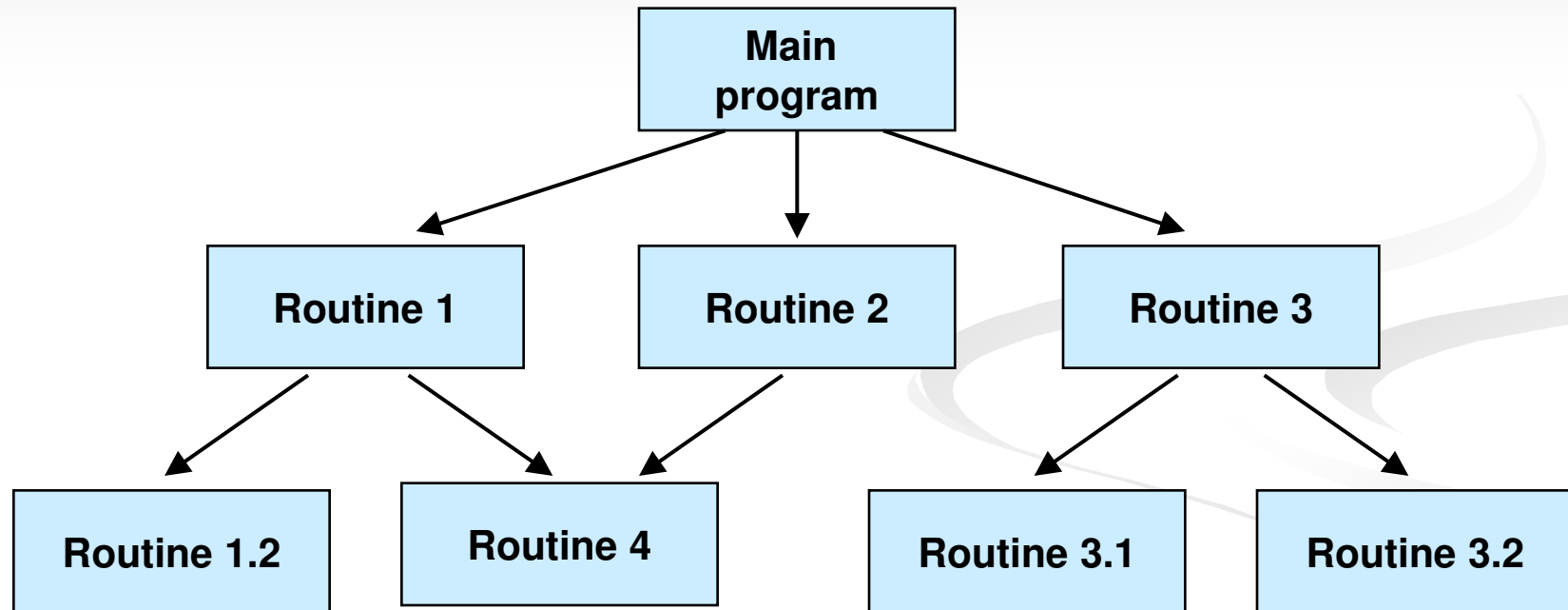
☞ Проблеми за изследване

- ✓ Презиползваемост – формата на данните за пренос
- ✓ Заменяемост
- ✓ Формален анализ, включително определяне на взаимна блокировка (deadlock detection).
- ✓ Паралелност на изпълнението на филтрите.
- ✓ Синхронизацията на потоците може да промени архитектурата.
- ✓ Error handling
- ✓ Изборът на типа на данните за потока.

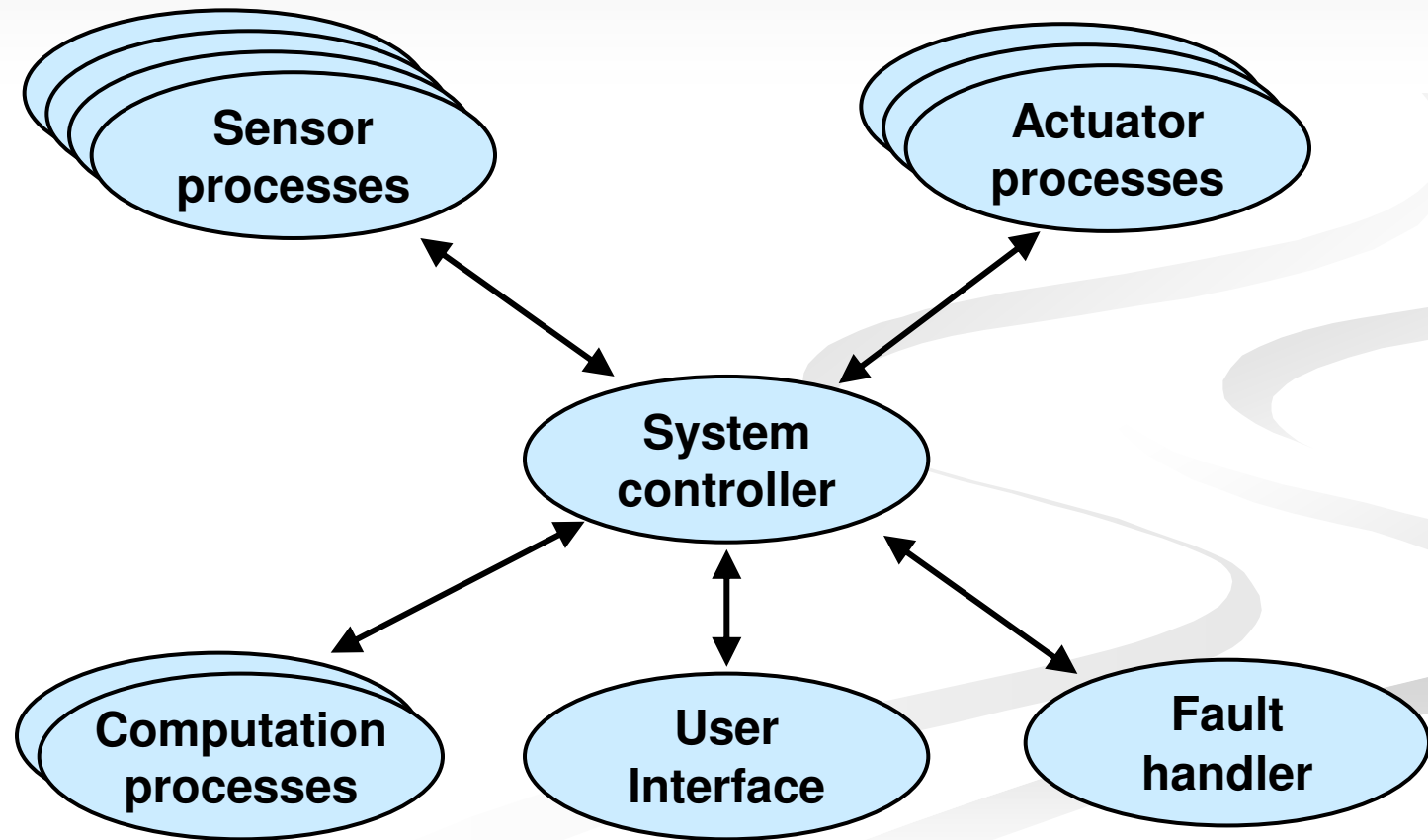
Архитектура с централизирано управление

- ☞ Главна програма + архитектура, базирана на подпрограми
 - ✓ Йерархична декомпозиция
 - ✓ Данните се предават като параметри.
 - ✓ Главната програма управлява последователността от обработки през подпрограмите

Call-Return Model



Management Control



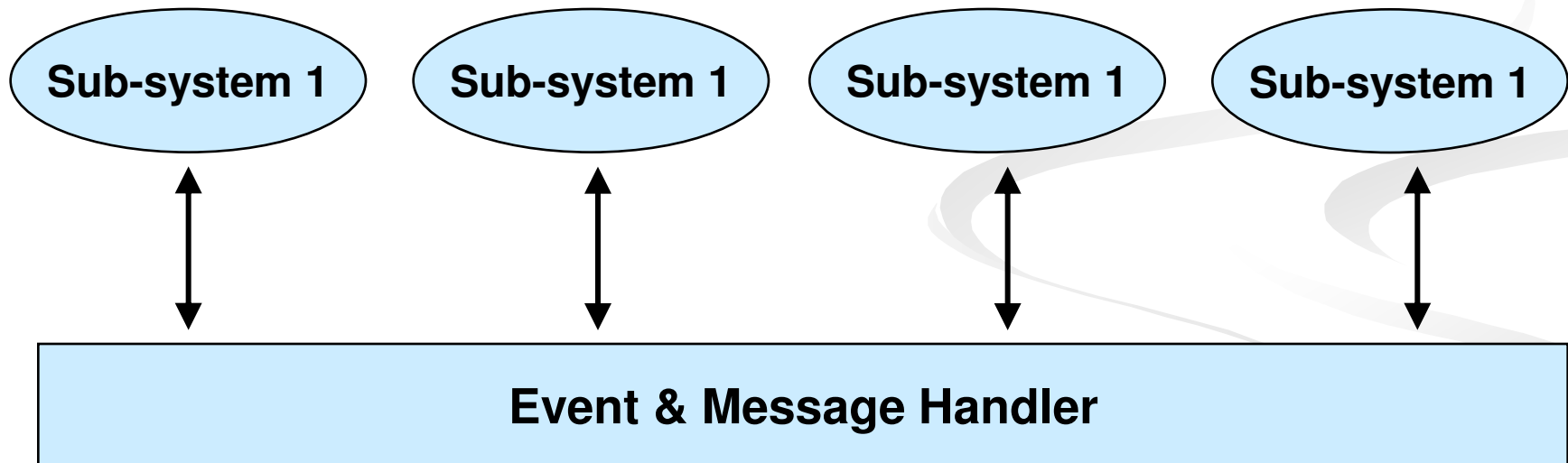
ОО архитектура

- ➡ Без наследственост има архитектурата е от друг вид.
- ➡ Основни проблеми за проверка:
 - ✓ Правилно разделяне на проблемите между свързаните агенти.
 - ✓ Промяната на имплементацията отразява ли се на клиентите ?
 - ✗ Ако В се промени то това ще доведе ли до промяна на използващите го обекти.
 - ✗ А използва В и ако С използва В, то ако промяната на С се отрази върху В то дали има неочаквана промяна на А.
 - ✓ Разпределена функционалност като резултат от сложни динамични връзки.

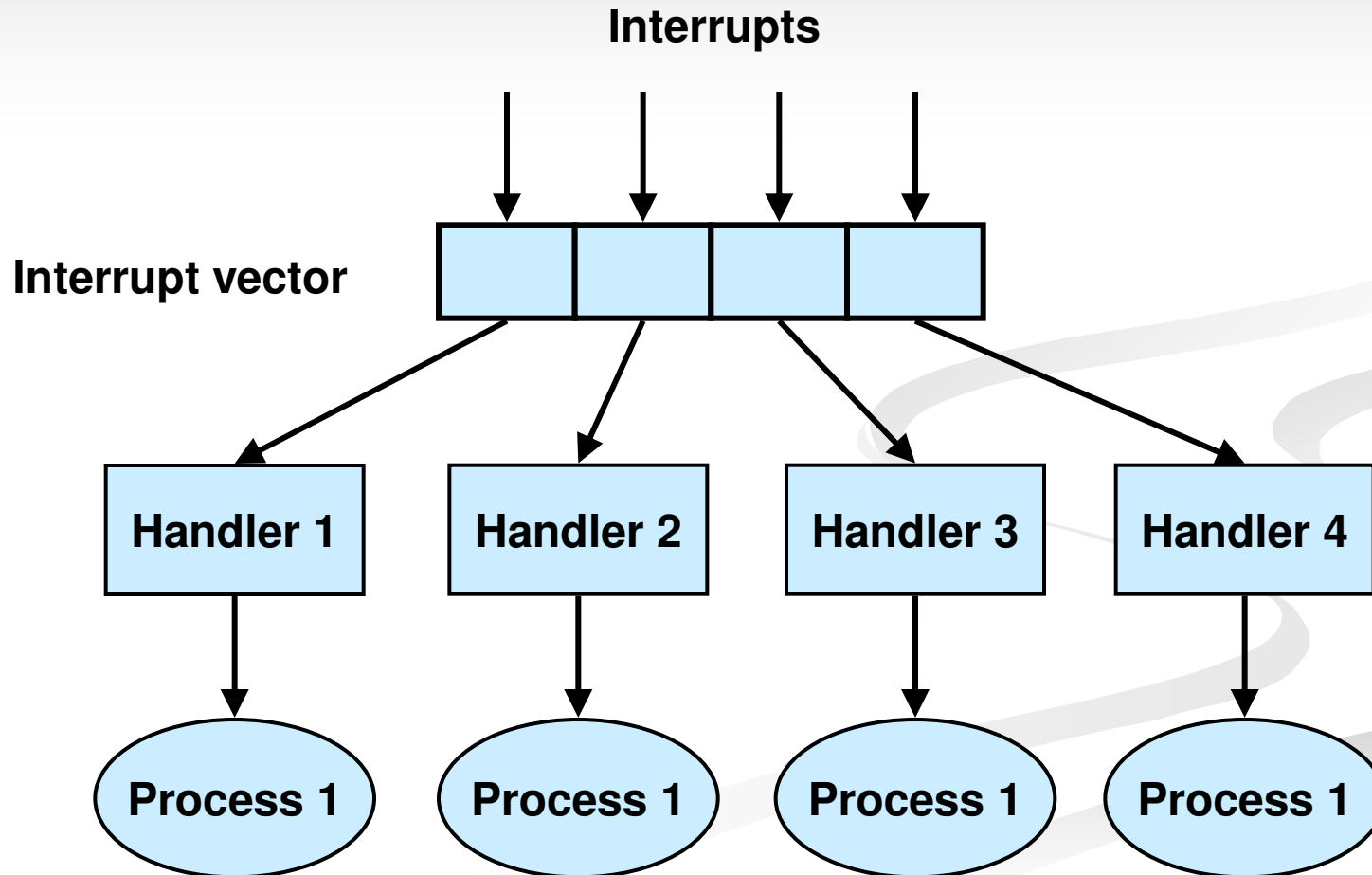
Event-Based

- ☞ Тази архитектура се характеризира със стила на комуникация между обектите:
 - ✓ Директна активизация (procedure invocation), изпращане на съобщение (message) до broadcast или multicast (до регистрирани компоненти), събития (events).
 - ✓ Компонентите не са задължително обекти.
- ☞ Видове
 - ✓ С разпространение (broadcast)
 - ✓ Системи с прекъсване (interrupt-driven)

Broadcasting



Interrupt-Driven Control



Явно активизиране

Основни проблеми

✓ Загуба на контрол

- ✗ При генерирано събитие компонентът не знае дали някой го възприема.
- ✗ Не знае дали някой ще бъде активизиран.
- ✗ Не знае кога свършва работата на активизирания.

✓ Коректността зависи от контекста, при който е възникнала активизацията, т.е. Непредсказуеми взаимодействия.

✓ Общи данни – имплицитното и експлицитна активизация се използват паралелно.

Model-View-Controller

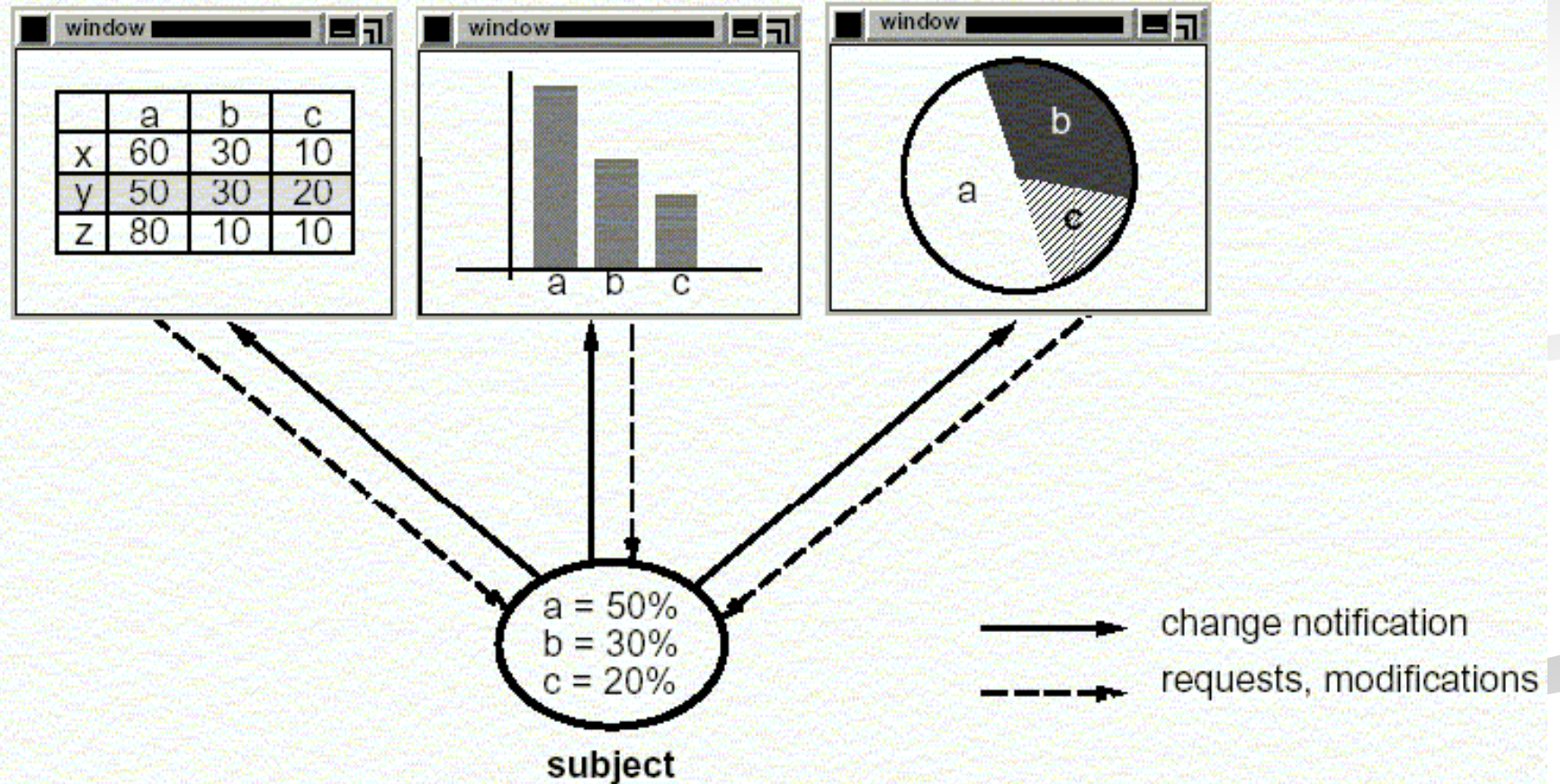
☞ Декомпозиция на интерактивна система на три компонента:

- ✓ Model – основната функционалност и данни
- ✓ Едно или повече 'Views' – визуализират информацията за потребителя
- ✓ Един или повече 'Controllers' – управляват входа от потребителя.

☞ Основен проблем

- ✓ Механизъм за защитена промяна (change-propagation mechanism) – да се осигурява консистентност между потребителския интерфейс и модела
- ✓ Промяната на модела пред един контролер, свързан с даден изглед ('view') трябва автоматично да обнови другите изгледи.
- ✓ Съществува ли обвързаност между контекста на контролера и изгледа – може да доведе до реализирането им като един компонент.

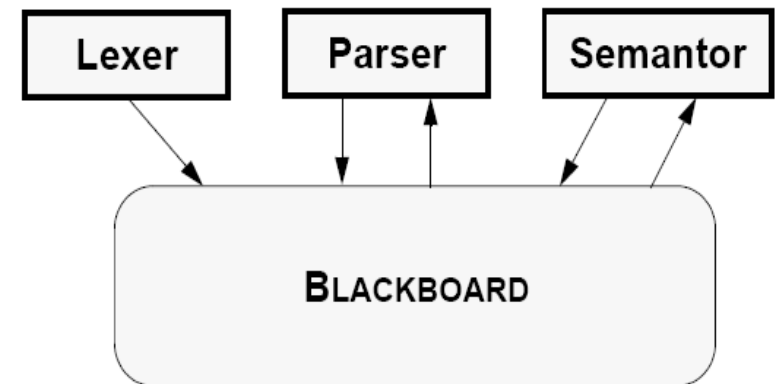
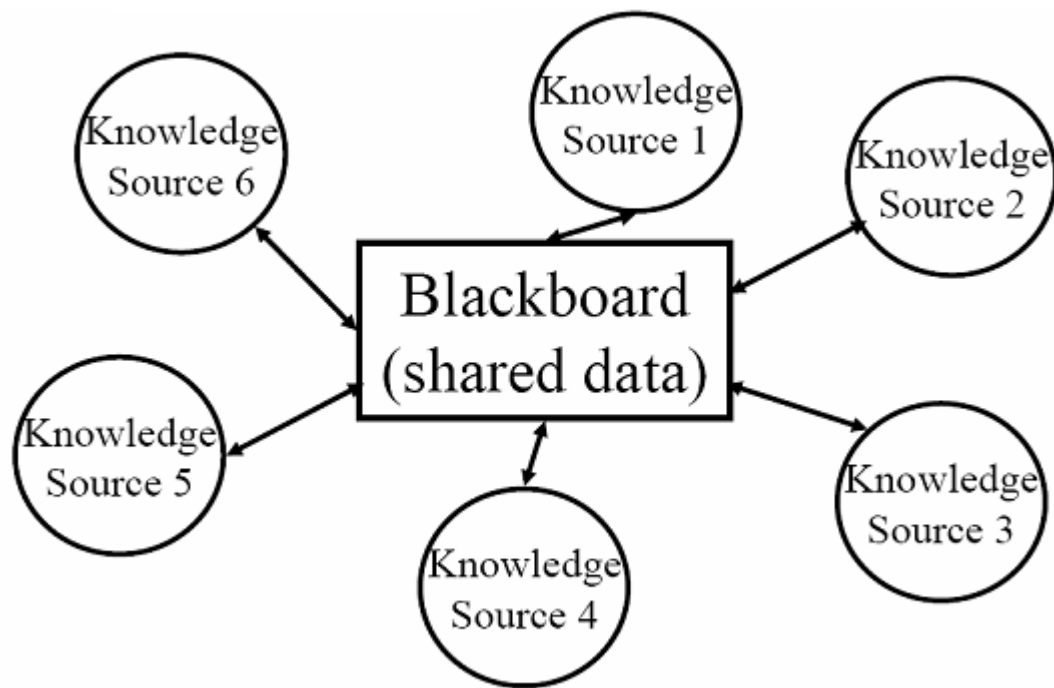
Model-View-Controller



Централизирано съхранение на данните (Repositories / Data Centred)

- ➡ Един централен компонент, съхраняващ централизирано данните и представящ състоянието на системата, и множество независими компоненти, работещи върху съхраненияваните данни.
- ➡ Проблем – връзката между външните компоненти и хранилището:
 - ✓ *Транзакционни БД (Transactional databases):*
 - ✗ Входният поток от транзакции управлява процесите за управление на данните в хранилището.
 - ✗ Пасивна идеология.
 - ✓ *‘Черна дъска’ (Blackboard architecture):*
 - ✗ Текущото състояние на данните в хранилището управлява процесите.
 - ✗ Активна идеология.

Централизирано съхранение на данните



Централизирано съхранение на данните

👉 Примери:

- ✓ Информационните системи
- ✓ Интегрирани среди за разработка
- ✓ Графични редактори
- ✓ Системи за ИИ
- ✓ Други.

👉 Проблеми:

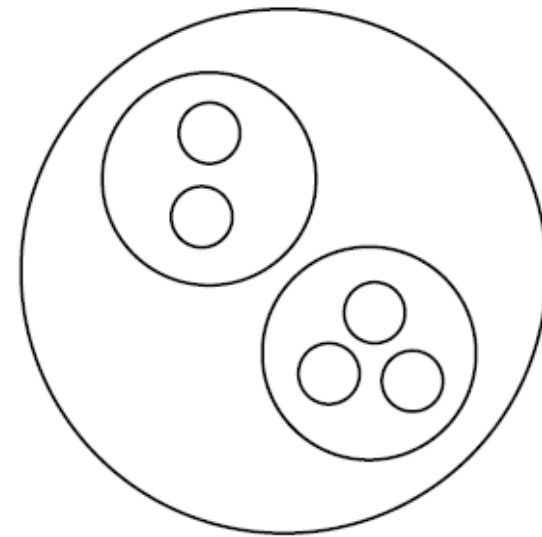
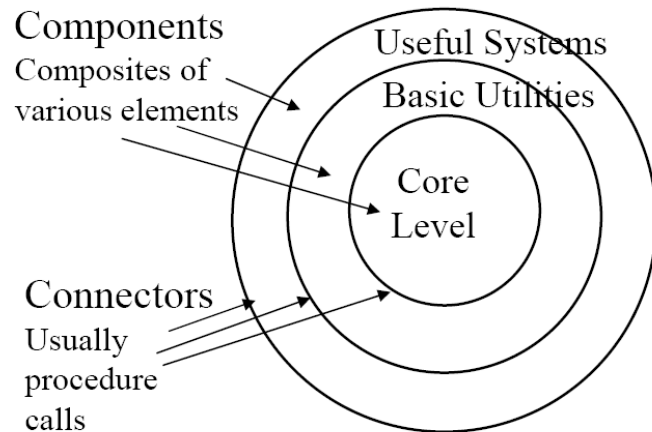
- ✓ Голямо количество на данните
- ✓ Централизирано управление на данните.
- ✓ Разпределени данни
- ✓ Еволюция

Йерархична архитектура

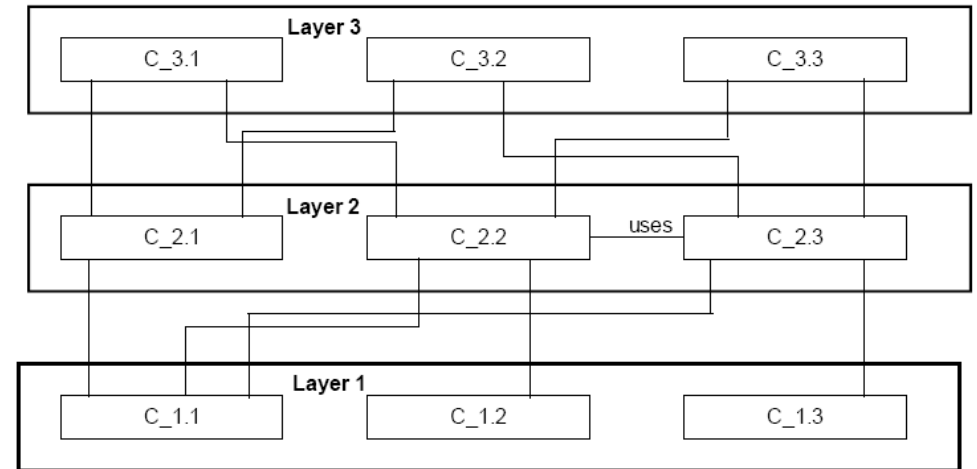
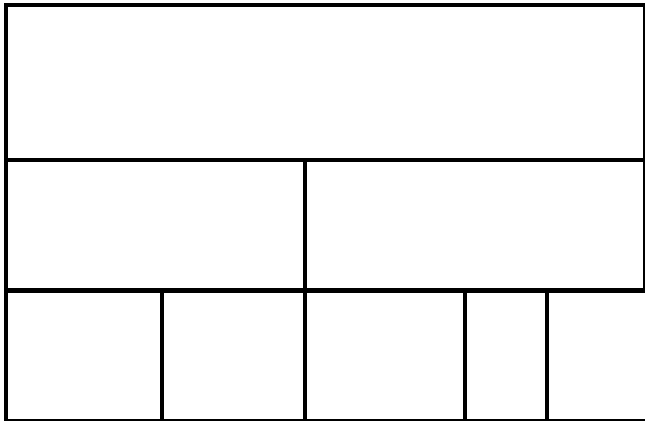
☞ Проблеми

- ✓ Системна организация
 - ✗ Многонивова ‘Клиент-Сървър’
 - ✗ Всяко ниво експортира “API” за горното ниво
- ✓ Всяко ниво се явява едновременно
 - ✗ *Сървър* – за горното ниво
 - ✗ *Клиент* – за долното ниво
- ✓ ‘Чисто’ йерархична система ли е ?
- ✓ Вид:
 - ✗ Интерпретативна или
 - ✗ Многонивова

Йерархична архитектура



Йерархична архитектура



Йерархична архитектура

- ☞ Стратегии за 'error handling'
 - ✓ Има или няма пропация на грешката
 - ✓ Количество на грешките като тип и брой при пропация през нивата
 - ✗ Проблем с обединяването на няколко грешки в една на горното ниво
 - ✗ Проблеми с контекста на грешките при пропация
- ☞ Как промяната на реализацията на дадено ниво се отразява на свързаните с него ?
- ☞ Как се управлява абстракцията ?
- ☞ Как се управлява бързодействието и сигурността ?

Архитектура 'Process-Control'

- ➡ Подходящ за приложения, чиято цел е да поддържа определени свойства на изходите на процеса на достатъчно близо до референтни стойности.
- ➡ Компоненти
 - ✓ Процеси ('Process') – механизми за манипулация на една или повече процесни променливи
 - ✓ Алгоритъм за управление – управлява процеса на манипулация на процесните променливи.

Архитектура 'Process-Control'

☞ Конектори – това са взаимовръзките в потока на данни

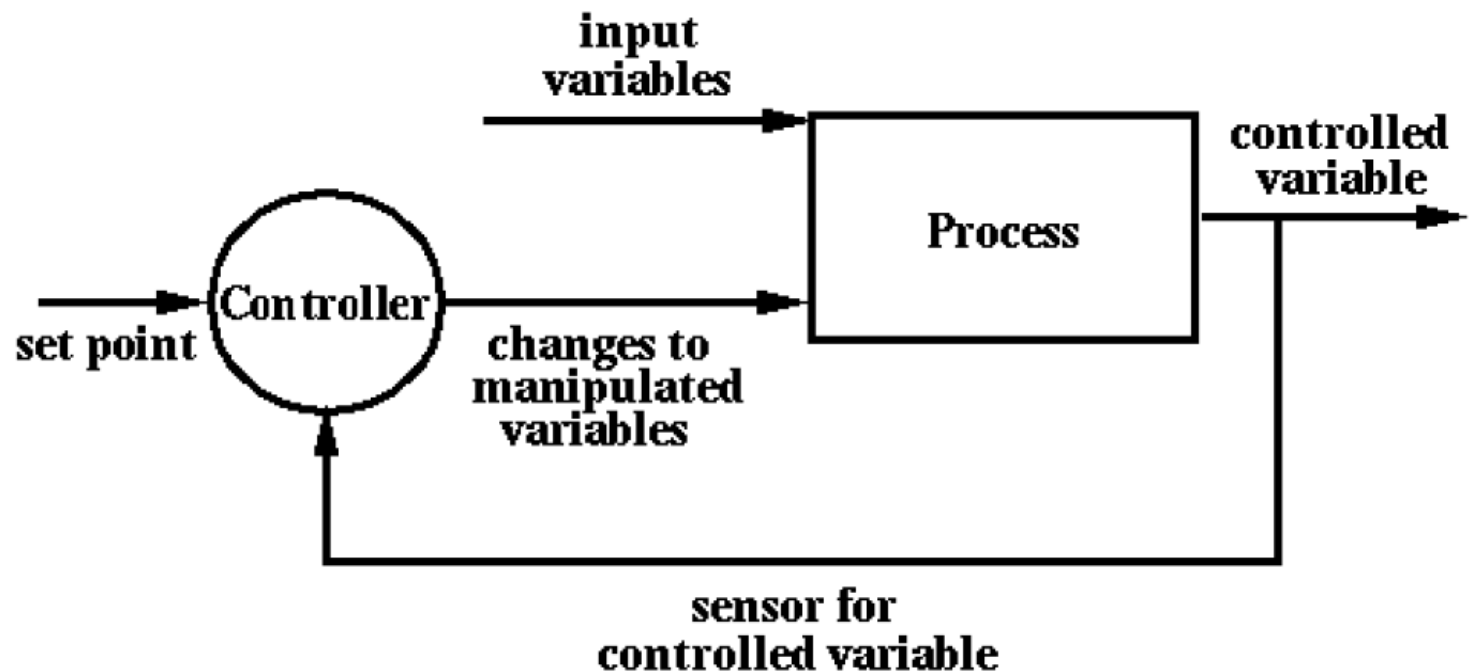
- ✓ Процесни променливи

- ✗ Контролиран променлива –предназначена е да се контролира от системата.
- ✗ Манипулирани променлива - стойността им може да бъде променена от управлението. от администратора.
- ✗ Входната променлива (set point) – това е желаната стойност за манипулирана променлива.

- ✓ Сензори

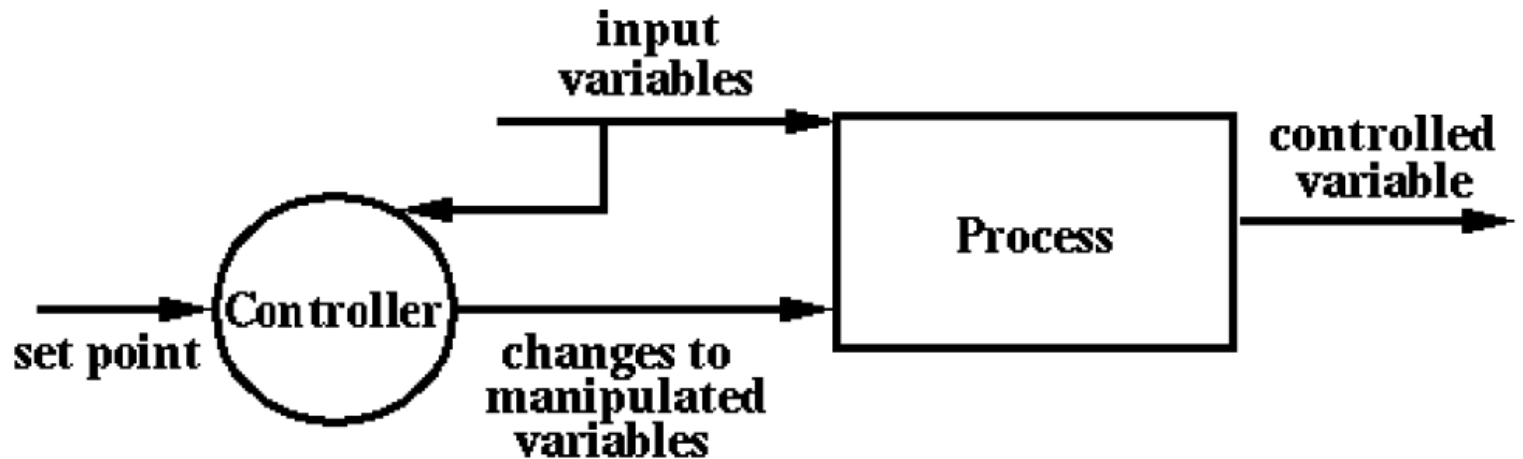
Архитектура 'Process-Control'

☞ Системи с обратна връзка



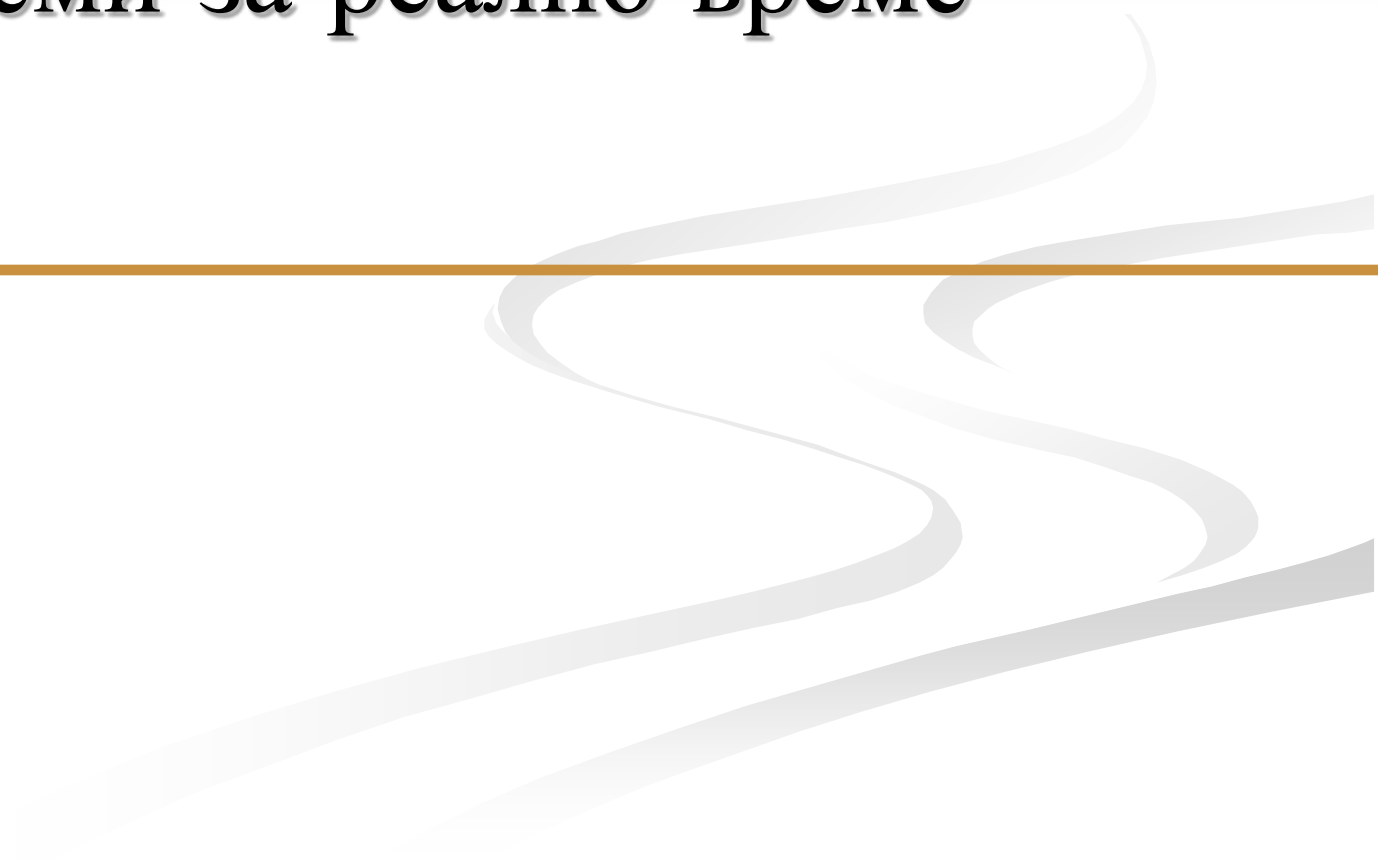
Архитектура 'Process-Control'

☞ Системи с отворен контур



Questions ?

Системи за реално време



Видове програмни модули (Програмиране от високо ниво)

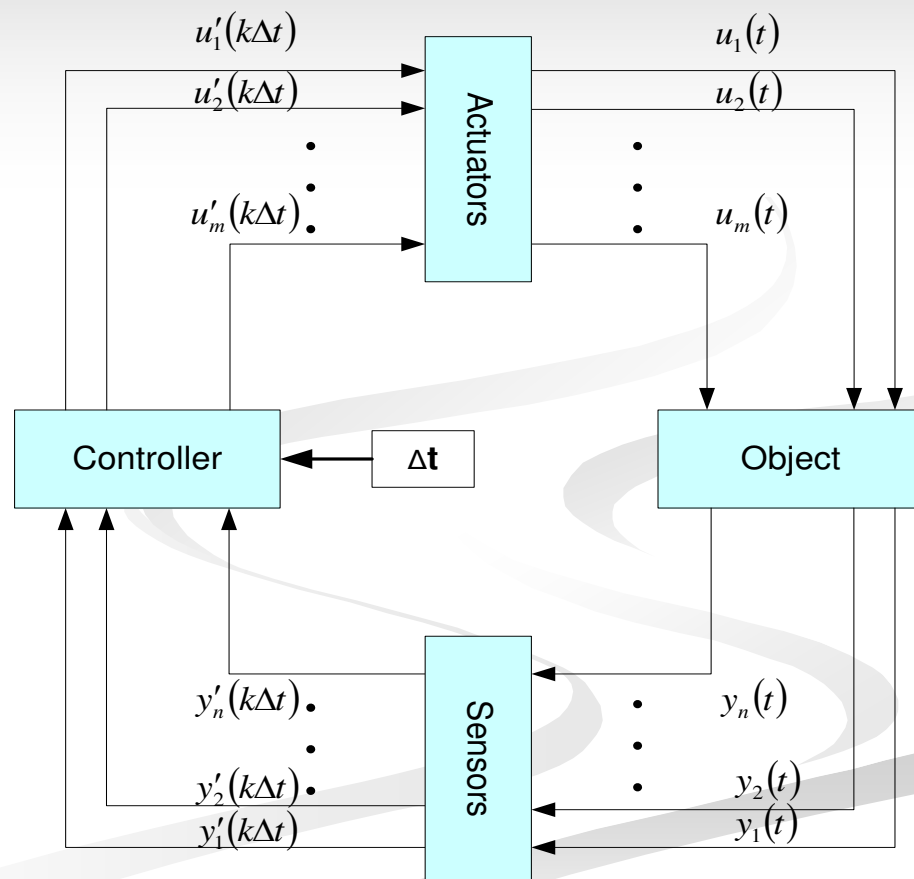
- ➡ Операционна среда
- ➡ Ядро
- ➡ Задачи (tasks)
- ➡ Задачи за обслужване на прекъсванията (ISR-tasks)

Базова структура на СРВ

Управляваща част

Управлявана част

Операционна среда



Операционна система за реално време (ОСРВ)

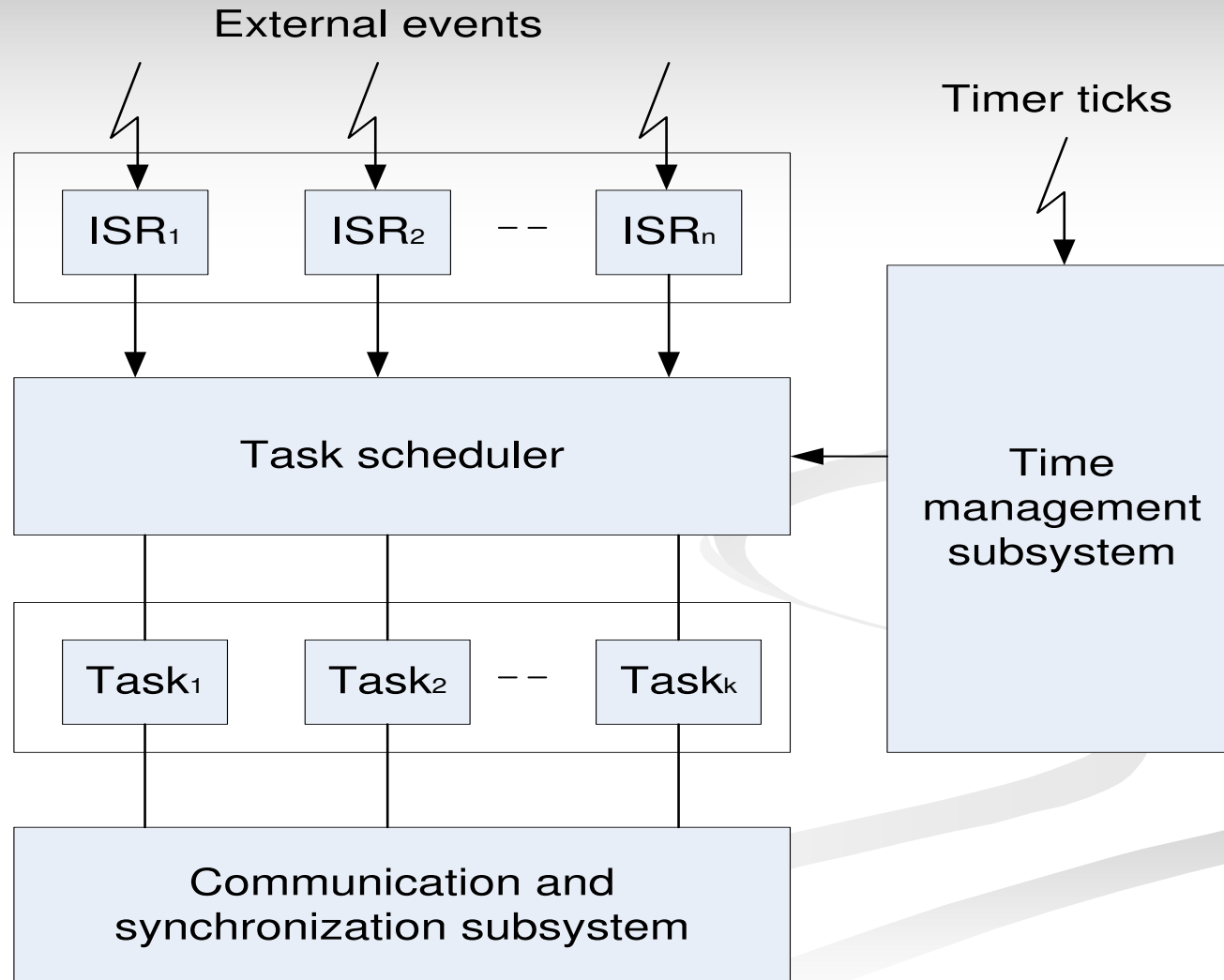
Дефиниция:

- ✓ *“Една компютърна система е система за реално време когато коректността на резултатите зависи не само от логиката, поради която са получени, но и от момента на тяхното създаване.”*
- ✓ *Коректен изход = Коректен резултат + Коректно време*

Характеристики на ОСРВ

- ☞ Главна задача на диспечера на ОСРВ - да следи за крайния срок (deadline).
- ☞ Система трябва да реагира своевременно и по предсказуем начин на външни стимули, появяващи се в непредвидим момент.
 - ✓ Тя трябва да има предсказуемо поведение при всички възможни сценарии.
 - ✓ Тя е само един модул и сама не може да гарантира коректността на цялата система.
- ☞ ОСРВ не е транзакционна система.
- ☞ Главна задача на диспечера на ОСРВ:
 - ✓ Да следи за крайния срок (deadline)

Структура на ОСРВ



Видове ОСРВ

- ☞ С твърди времеви ограничения (Hard Real-Time)
 - ✓ Неспазването на крайните интервали има катастрофален резултат за системата.
- ☞ Firm Real-Time
 - ✓ Неспазването на крайните интервали води до непредсказуема редукция на качеството
- ☞ С меки времеви ограничения (Soft Real-Time)
 - ✓ Крайните срокове могат да се пропускат или могат да се възстановяват
 - ✓ Редукцията на качеството е приемлива.
- ☞ Без времеви ограничения (Non Real-Time)
 - ✓ Няма крайни срокове.

Ограничения, влияещи върху задачите в една ОСРВ

- ☞ Ограничения от крайни срокове (Deadlines)
- ☞ Ресурсни ограничения
 - ✓ За да се постигне по-висока надеждност за изпълнение на задачата
 - ✓ Необходим е излишък за правилно функциониране.
- ☞ Приоритетни ограничения
 - ✓ $T_1 \rightarrow T_2$: Задачата T_2 само след като T_1 завършила изпълнението си.
- ☞ Отказоустойчивост
 - ✓ За да се постигне по-висока надеждност за изпълнение на задачата.
 - ✓ Необходим е излишък за правилно функциониране.

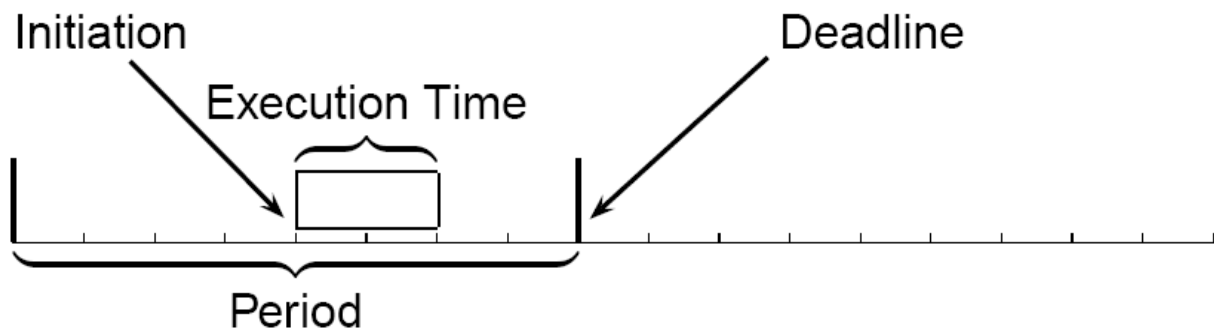
Видове реално-времеви задачи

☞ Според ограничение по време:

- ✓ Твърди (Hard real-time) задачи
- ✓ Меки (Soft real-time) задачи

☞ Според момент на активация

- ✓ Периодични
- ✓ Аperiодични
- ✓ Спорадични



Реално-времеви задачи

	Hard Real-Time	Soft Real-Time
Време за реакция	твърди граници	меки граници
При върхово натоварване	предсказуемо	деградиране
Управление на пространството	от средата	от компютъра
Надеждност	критична	не-критична
Излишък	активен	възстановим
Интегритет на данните	кратковременен	дълговременен
Откриване на грешки	автоматичен	външно управляемо

Реално-времеви задачи

👉 Периодични

- ✓ Управлявани от време (Time-driven).
- ✓ Характеристиките *са априорно* зададени
 - ✗ Задачата T_i се характеризира чрез периода си, най-лошо време за изпълнение и краен период

👉 Аперииодични

- ✓ Управлявани от събития (Event-driven).
- ✓ Характеристиките *не са априорно* зададени
 - ✗ Задачата T_i се характеризира чрез време за получаване, време за изпълнение, най-лошо време за изпълнение и краен период.

👉 Спорадични – аперииодични, за които се знае минималното време за появяване

Ядра за ОСРВ



Монолитни

- ✓ Един общ кодов фрагмент
- ✓ Малък размер и проста вътрешна структура
- ✓ Трудни за промяна и настройка



Многослойни

- ✓ Разделени на подсистеми – най-често се използва ОО дизайн
- ✓ По-добра изолация от потребителските задачи
- ✓ По сложни, с по-голямо време на реакция, но стабилни



Разпределени

- ✓ Разпределената система се представя като *virtual unicomputer*

Диспечирание на задачите



Основна цел

- ✓ Управление на времето на процесора



Възможни елементи за управление:

- ✓ Процеси (Processes)
- ✓ Нишки (Threads)
- ✓ Фибри(Fibers)

Те са леки (lightweight) процеси.

Много леки елемент, управлявани от приложението.

Състоят се от:

- Само код.

Не се използват в повечето ОСРВ.

За какво: Предварителен дизайн и симулация на реално-времеви нишки.

Граф на състоянието на задачите



Активна задача (active)

- ✓ Тя работи с процесора



Пасивна задача (passive)

- ✓ Съществува, но не е избрана за изпълнение



Прекратена задача (suspended)

- ✓ Била е стартирана, но не може да се изпълнява по някаква причина



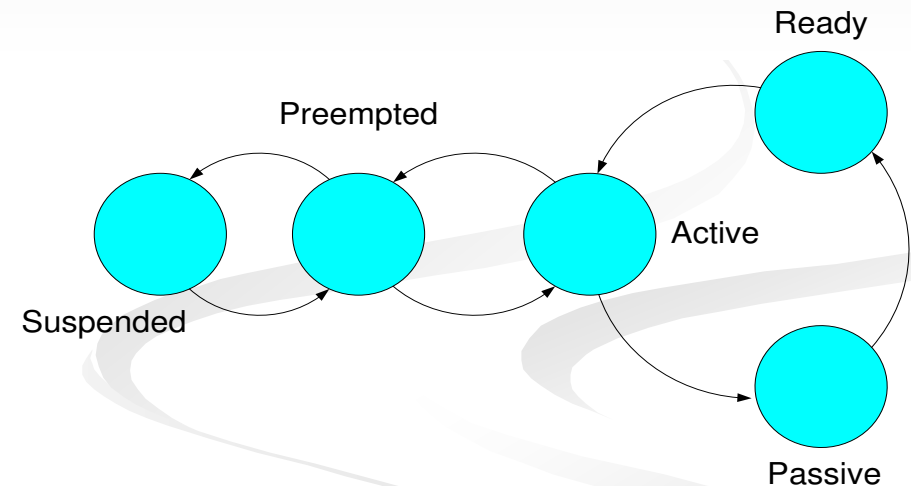
Изместена задача (preempted)

- ✓ Била е стартирана, е изместена по някаква причина



Готова задача (ready)

- ✓ Изчаква да бъде активизирана.



Промяна на състоянието на задачите



Active -> Passive

- ✓ Задачата завършва (експлицитно или поради скрито системен извикване: 'schedule', 'quit', др.)



Passive -> Ready

- ✓ Системно API-извикване ('execute', 'invoke', др.)



Ready -> Active

- ✓ по диспечерно решение



Active -> Preempted

- ✓ поява на диспечерируемо събитие преди края на задачата



Preempted -> Active

- ✓ по диспечерно решение



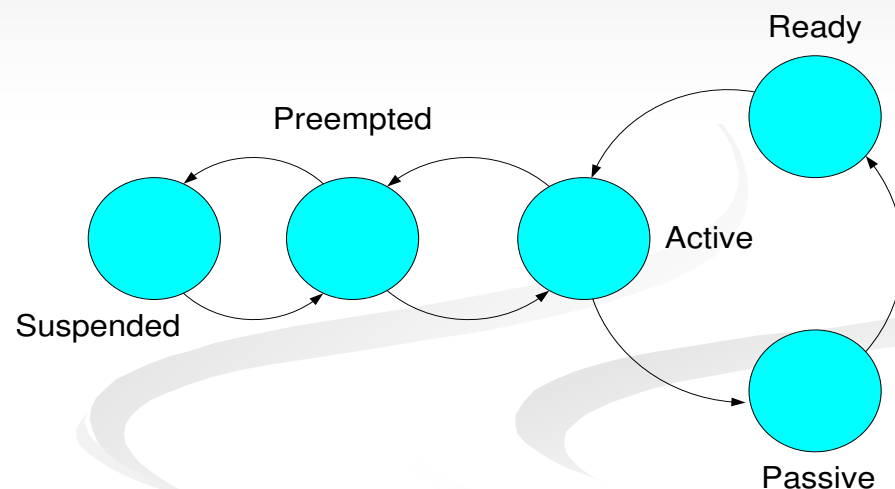
Active -> Suspended (via Preempted)

- ✓ Най-често нещо необходимо е заето.



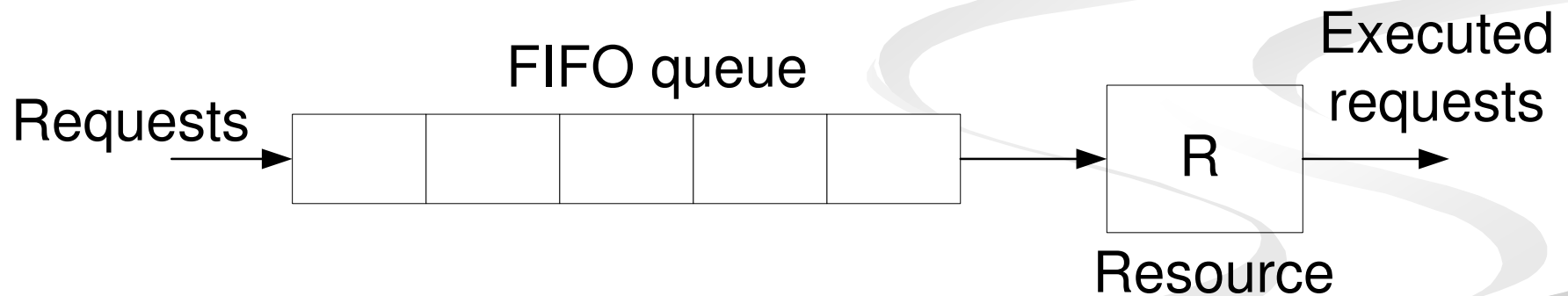
Suspended -> Preempted

- ✓ След освобождаване на нужен ресурс или подобно.



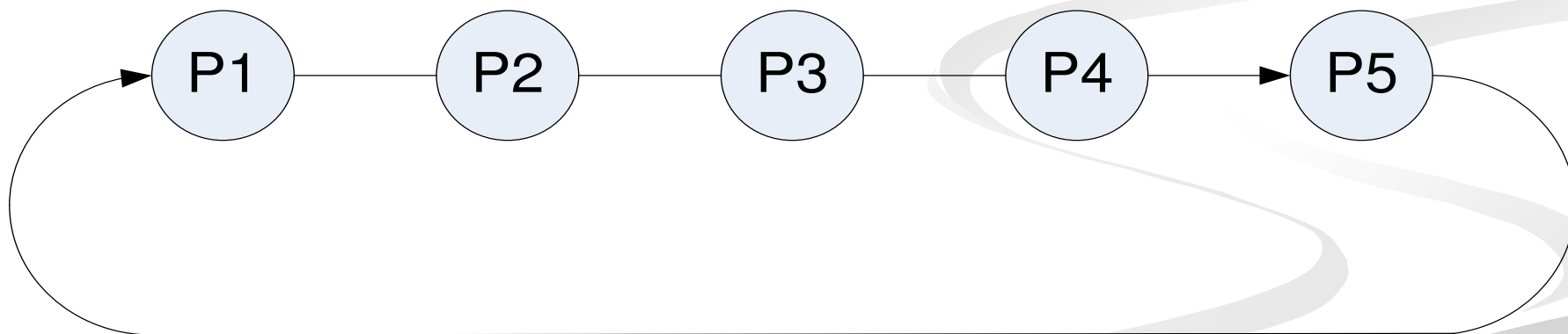
Управление на задачите

👉 Опашка (FIFO)



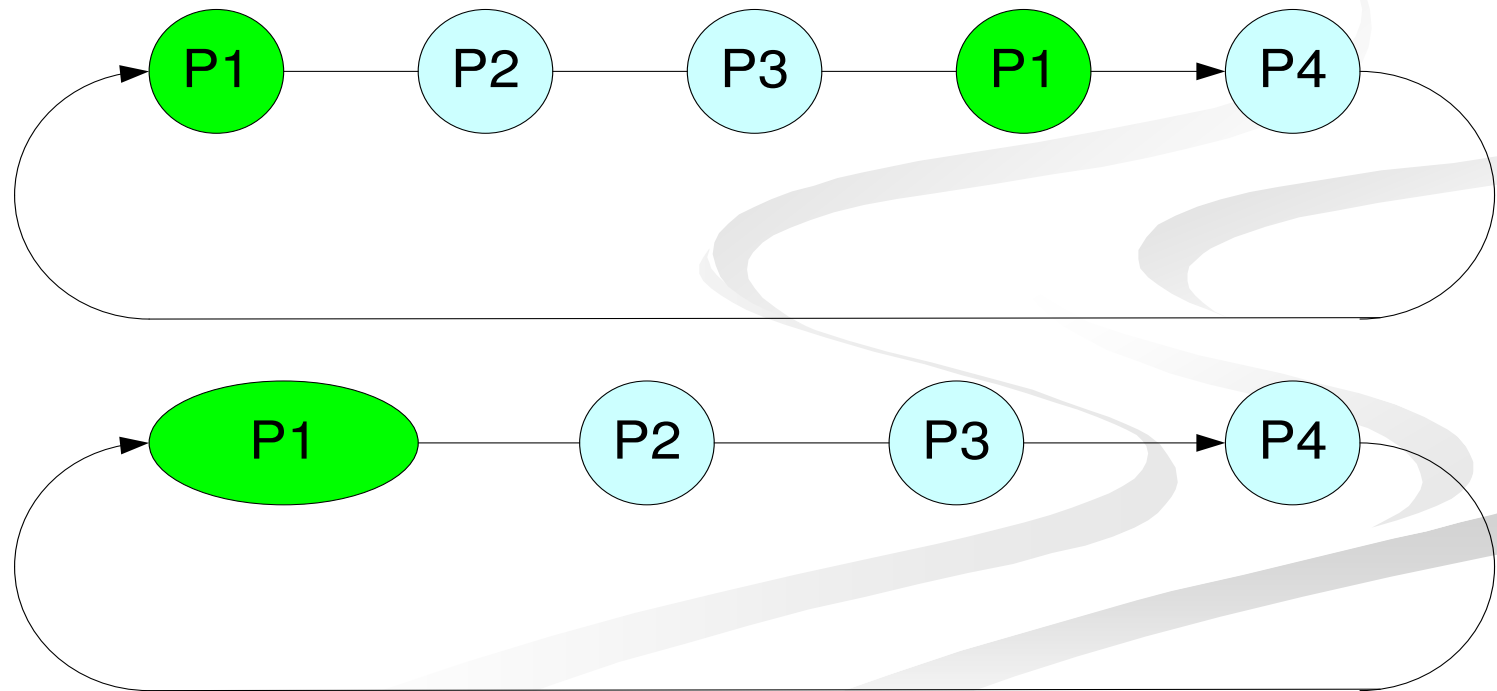
Управление на задачите

- ❏ Циклично заемане на процесора (Round-robin – Simple)



Управление на задачите

- Тегловно базирано циклично заемане на процесора (Round-robin – Weighted)

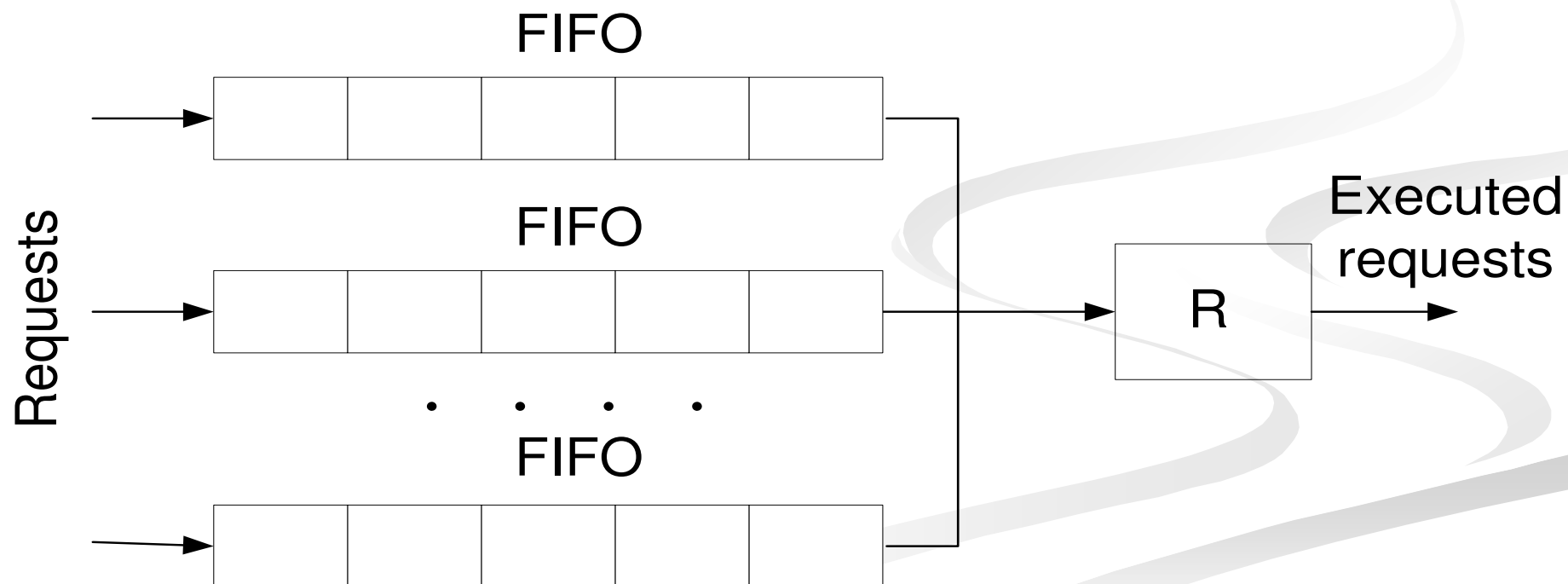


Управление на задачите

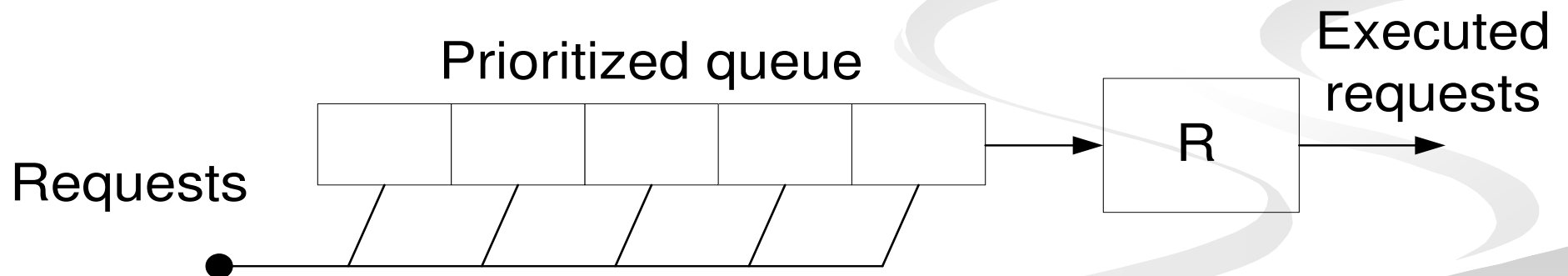
☞ Приоритетна диспечеризация

- ✓ Приоритетни опашки (Priority queues)
- ✓ Приоритизирани опашки (Prioritized queues)
- ✓ Смесена дисциплина на диспечирание (cyclic priority scheme)
- ✓ Динамична дисциплина на диспечирание

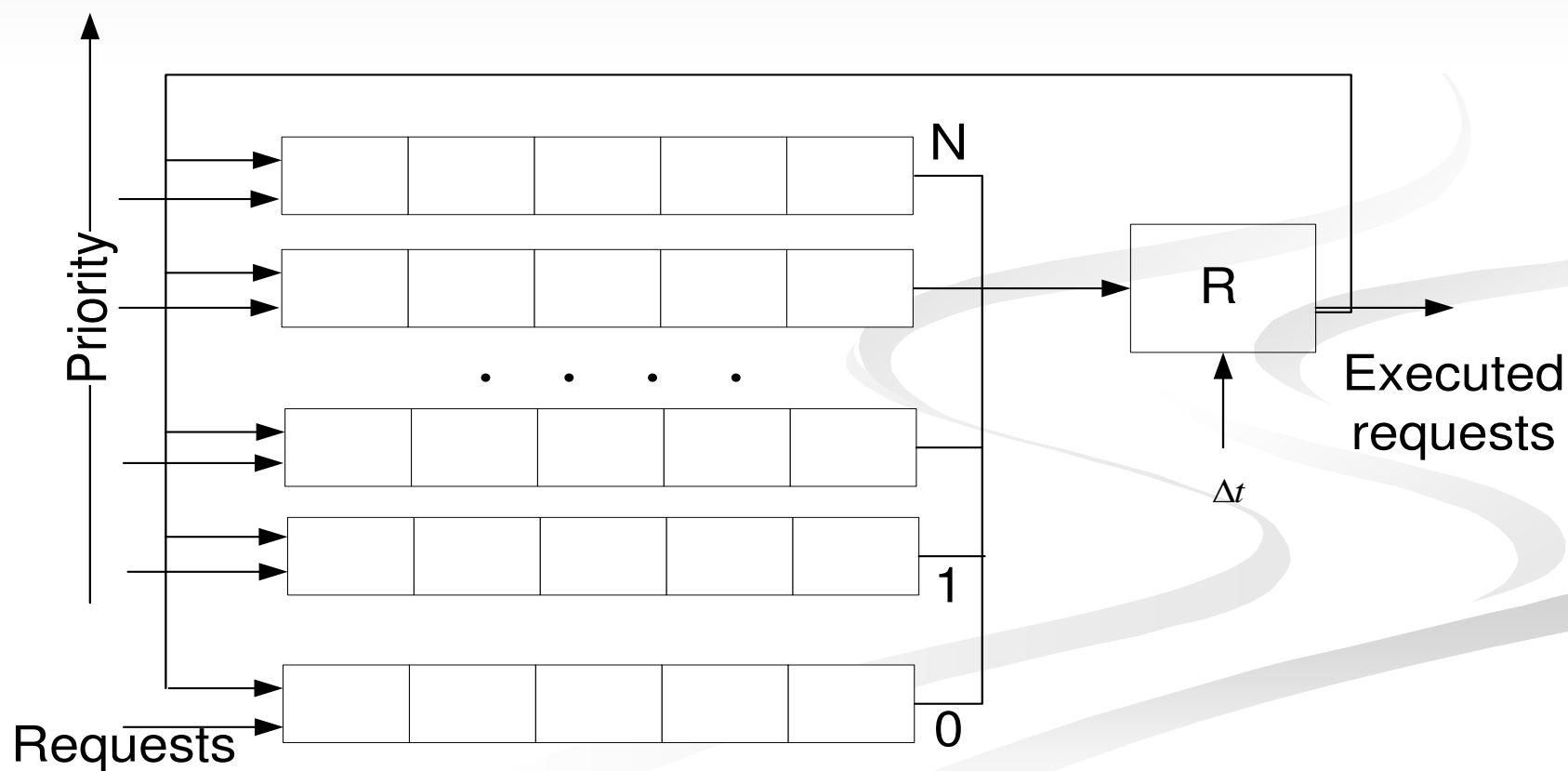
Приоритетни опашки



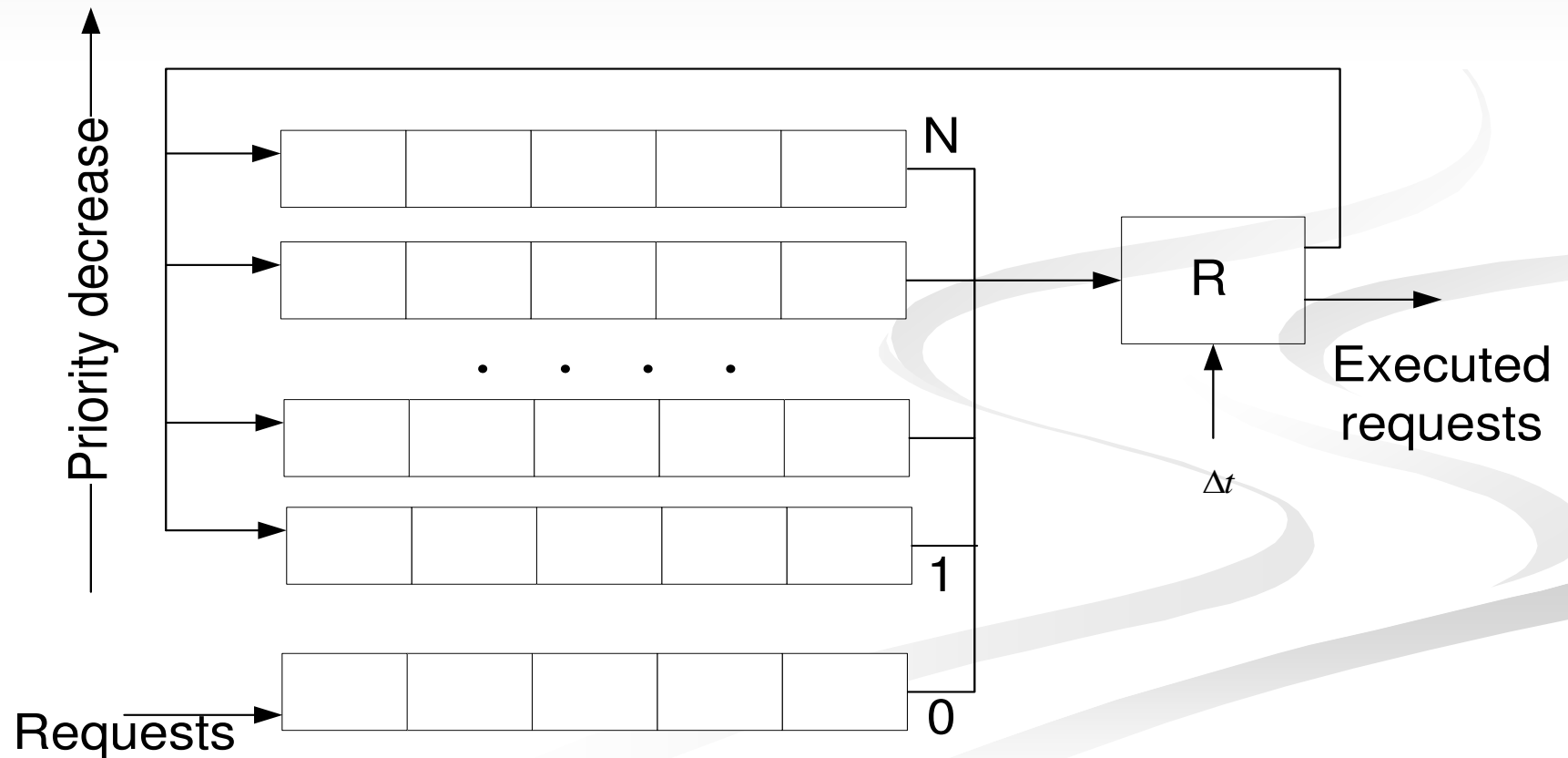
Приоритизирани опашки



Смесена дисциплина на диспечирание

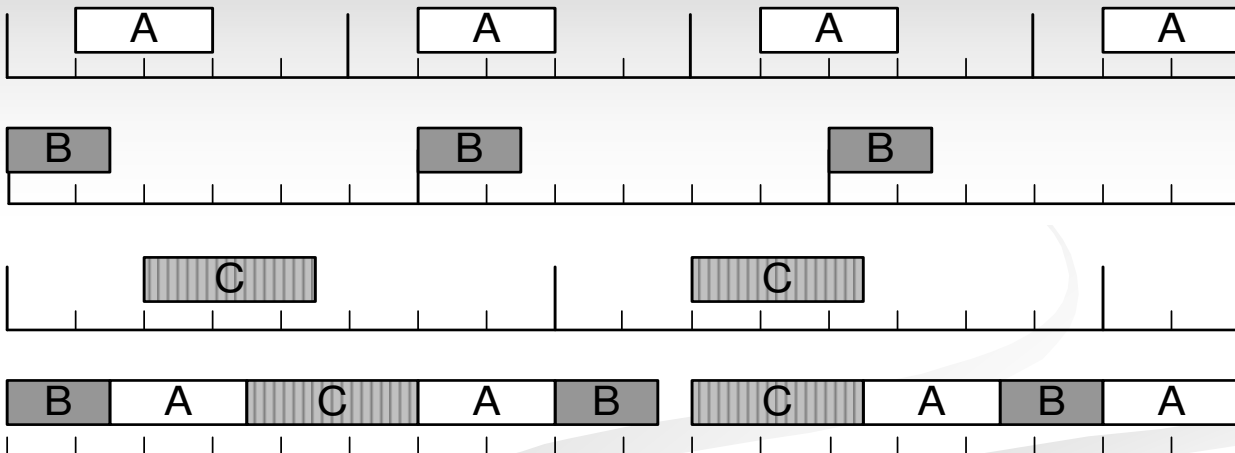


Динамична дисциплина на диспечирание

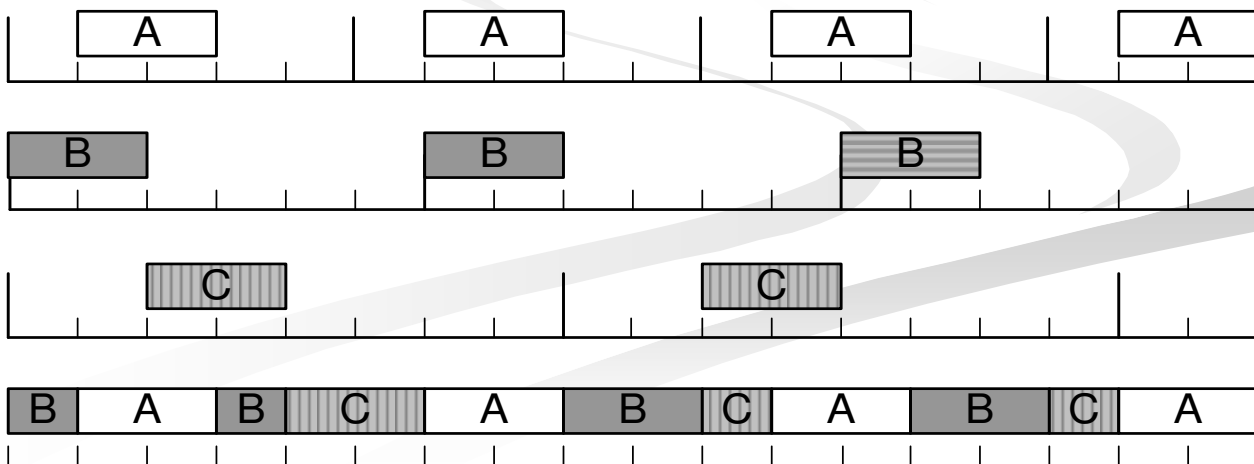


Дисциплина на диспечирание

Относителна



Абсолютна



Priority scheduling disciplines



Non-preemptive (relative) discipline

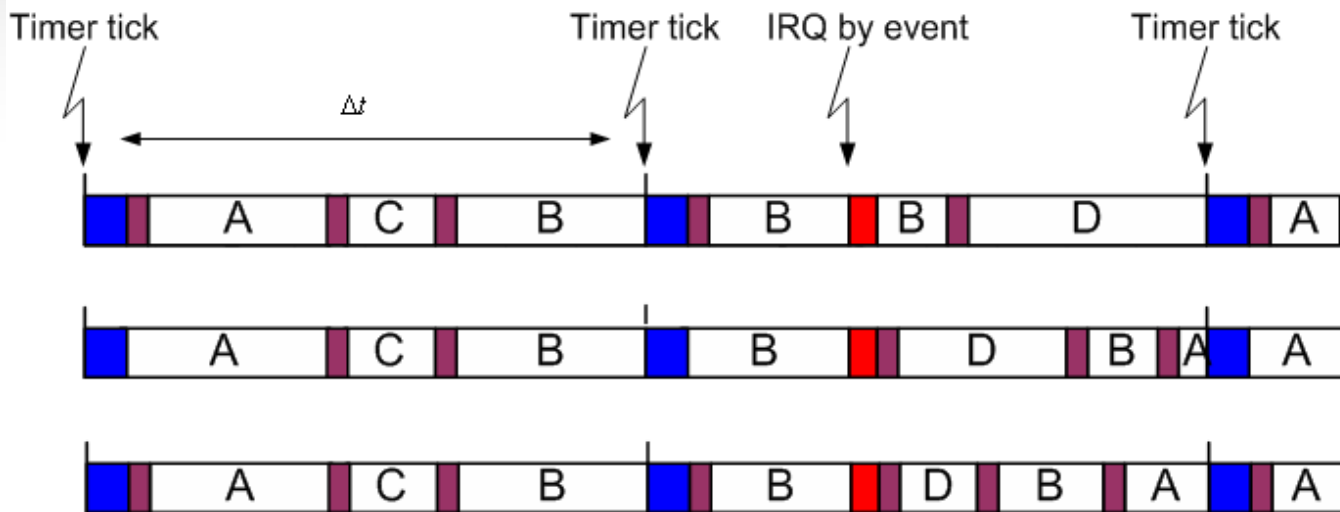
- ✓ Advantages
 - × No conflicts on resources
 - × Simpler kernel structure
- ✓ Disadvantages
 - × Less reaction time.



Preemptive (relative) discipline

- ✓ Advantages
 - × Predictability
 - × Simpler kernel structure
- ✓ Disadvantages
 - × More complicated kernel structure.
 - × Possible conflicts on resources.

Дисциплини за управление на времето



Synchronous

Asynchronous

Mixed

-  Timer subsystem
-  Task scheduler
-  ISR

Подсистема за измерване на времето

- ☞ Тя е самостоятелен системен обект.
- ☞ Две основни функции:
 - ✓ Часовник за реално време Real-time clock
 - ✗ Периодична активация от импулсен генератор.
 - ✗ Точност - Тактов генератор (между 10^5 наносекунди до милисекунди).
 - ✗ Структура (системен часовник, брояч за сек., мин., ч., дни, месеци, години)
 - ✗ Брояч на абсолютното време (48 ÷ 64 бита брояч, управляван от системния тактов генератор)
 - ✓ Измерване на интервали – интервални и календарни функции.

Управление на външни събития

☞ Същност:

- ✓ Реализира се при апаратно прекъсване.
- ✓ Управлява се Диспечер на събитията (Event Manager)
- ✓ Диспечер на събитията се състои от функции за управление на събития (Event Handlers)
- ✓ Функциите за управление на събития се активизират от функции за управление на прекъсванията (Interrupt Service Routines - ISR)

☞ Реализация:

- ✓ Сигнализиран се една или повече задачи за случилото се *или*
- ✓ Разпространение (Broadcasting) на съобщение една или повече задачи

Взаимодействие на задачите

Основни проблеми:

- ✓ Синхронизация – изисква за задачите и нишките да се гарантира:
 - ✗ коректно изпълнение на критичен код
 - ✗ гарантиране на достъп до ресурс по определен ред
 - ✗ сигнализация при настъпило събитие
- ✓ Комуникация – да се реализира високо и ниско ниво на междупроцесна комуникация.

Методи за синхронизация

- ☞ Основен проблем: ‘Състезание на сигнали’.
- ☞ Основни механизми:
 - ✓ Събитийни флагове (Event flags) – разпространяват събитията
 - ✓ Семафори – синхронизация и изключване
 - ✓ Мутекси (Mutex) – управление на достъпа до ресурс
 - ✓ Забрана на прекъсванията (Interrupt disabling) - управление на достъпа до ресурс
 - ✓ Променливи на състоянието (Conditional variables) - управление на достъпа до ресурс

Събитийни флагове

☞ Същност

- ✓ Бълева променлива, достъпна от всички процеси /нишки, които трябва да се сигнализират/синхронизират.
- ✓ Използвана за сигнализация 'един-към-много'.

☞ Операции:

- ✓ *set_flag* , *read_flag*, *reset_flag*, *read_and_reset_flag*

☞ Проблеми:

- ✓ Обща памет или системни ресурси, достъпни от всички.
- ✓ Кой начално инициализира флага при 'един-към-много' ?
- ✓ Състезание на сигнали

Семафори

```
int counter;  
ProcessIDqueue[ ];
```

```
up()  
{  
    counter++;  
    if ( counter > 0 ) return;  
    get first processID from ProcessIDqueue and wakeup the process.  
    return;  
}
```

```
down()  
{  
    counter--;  
    if ( counter >= 0 ) return;  
    put in ProcessIDqueue ID of calling process.  
    sleep_the_process();  
}
```

Двоични (сигнални) семафори

☞ Брояч: $-1, 0, 1$.

- ✓ Начално състояние:

- ✗ $counter = 0$

- ✓ Сигнализирано състояние:

- ✗ If $counter = 1$: включен

- ✗ If $counter = 0$: не е включен и никой не чака за него

- ✗ If $counter = -1$: не е включен и някой чака за него

- ✓ Операции:

- ✗ $down() \rightarrow signal()$

- ✗ $up() \rightarrow wait()$

Ресурсни семафори

👉 Еднослотни:

- ✓ $counter = \{ 1, 0, -1, \dots, -n \}$, $n \rightarrow$ макс. брой на чакащите
- ✓ Начална стойност: $counter = 1$

👉 Многослотни

- ✓ $counter = \{ K, 0, -1, \dots, -n \}$, $K \rightarrow$ брой на слотове, $n \rightarrow$ макс. брой на чакащите
- ✓ Начална стойност: $counter = 1$

👉 Операции:

- ✓ $down() \rightarrow lock()$
- ✓ $up() \rightarrow unlock()$

Взаимни изключвания (Mutual exclusions)

Същност

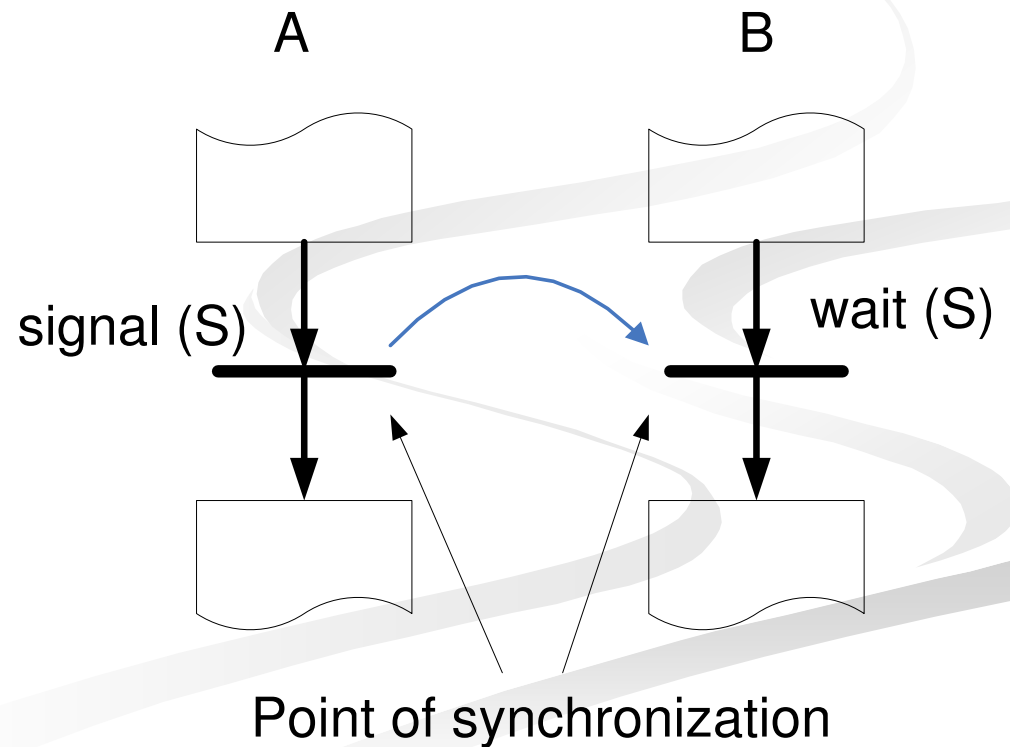
- ✓ Неконкурентен достъп до ресурсите.
- ✓ Основната задача е да изключи състезанието на сигналите.

Видове:

- ✓ Забрана на прекъсванията (Interrupt disabling)
- ✓ Инструкции от вида 'Read_Modify_Write'
- ✓ Ресурсни семафори
- ✓ Мутекси (Mutex)
- ✓ Монитори (Monitors)

Сигнализационни схеми

👉 Един-към-един : използват се двоични семафори

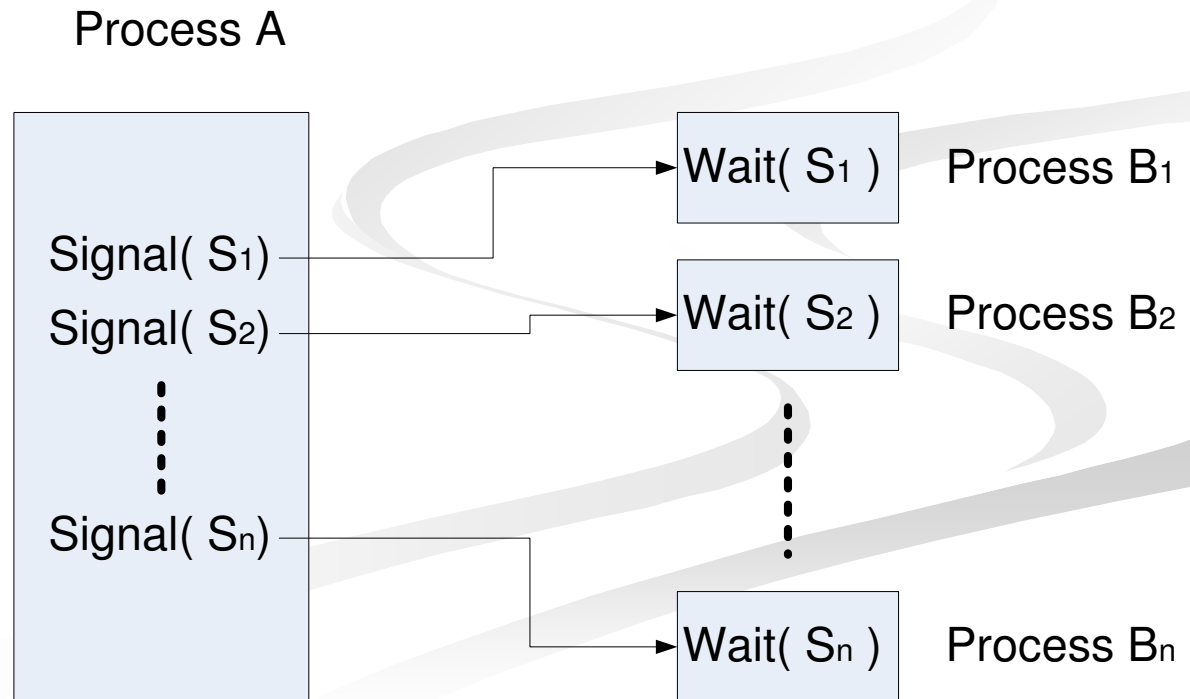


Сигнализационни схеми



Един-към-много

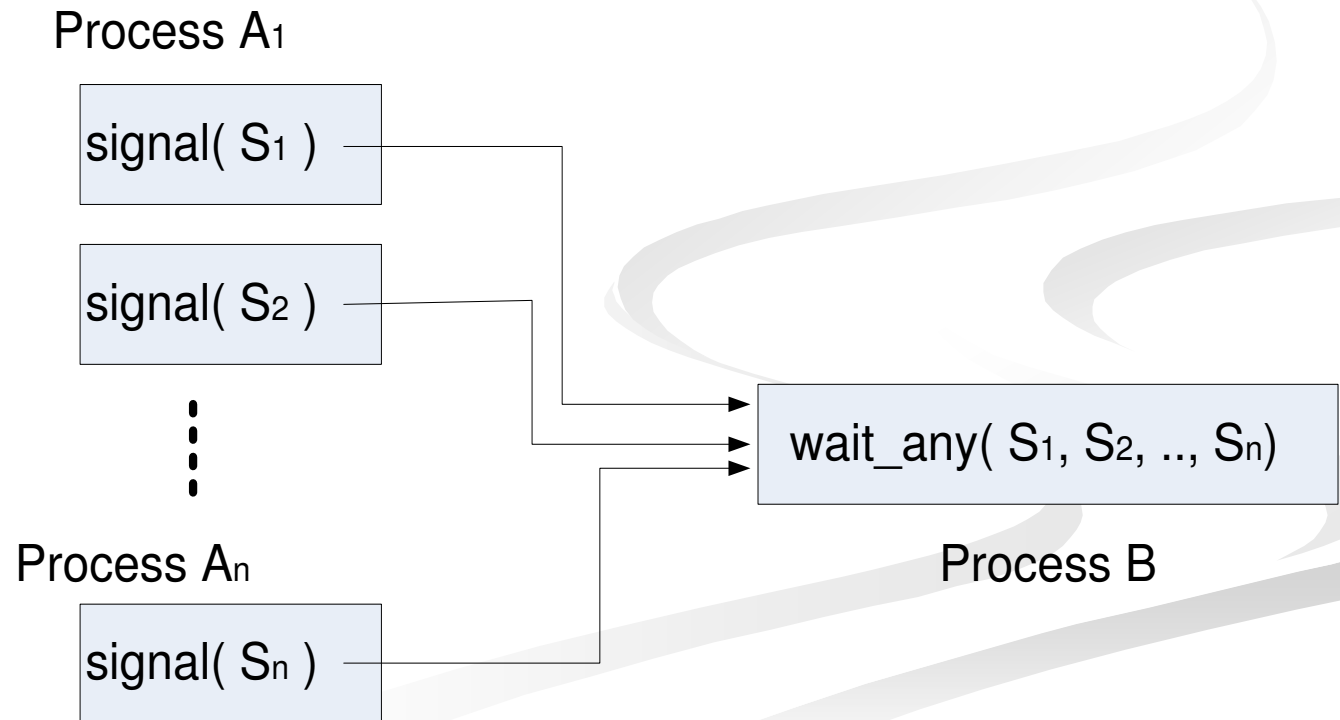
- ✓ Изчакващите процеси не се сигнализират едновременно, т.е. Не е много полезен начин за сигнализация



Сигнализационни схеми



Много-към-един



Сигнализационни схеми

☞ Неблокираща сигнализация

- ✓ Нова семафорна операция:

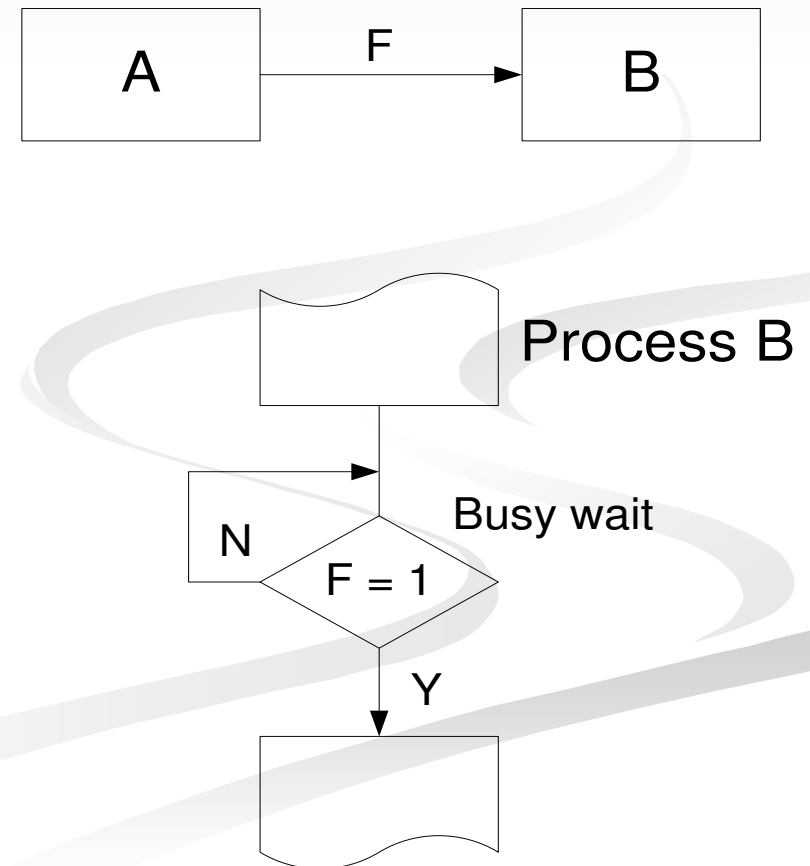
- ✗ `int test_and_reset()` – връща текущата стойност и го инициализира с 0.

☞ Изпълнение на синхронизационните схеми

- ✓ Сигнализация за събитие
- ✓ Двупосочна синхронизация
- ✓ Ресурсна синхронизация

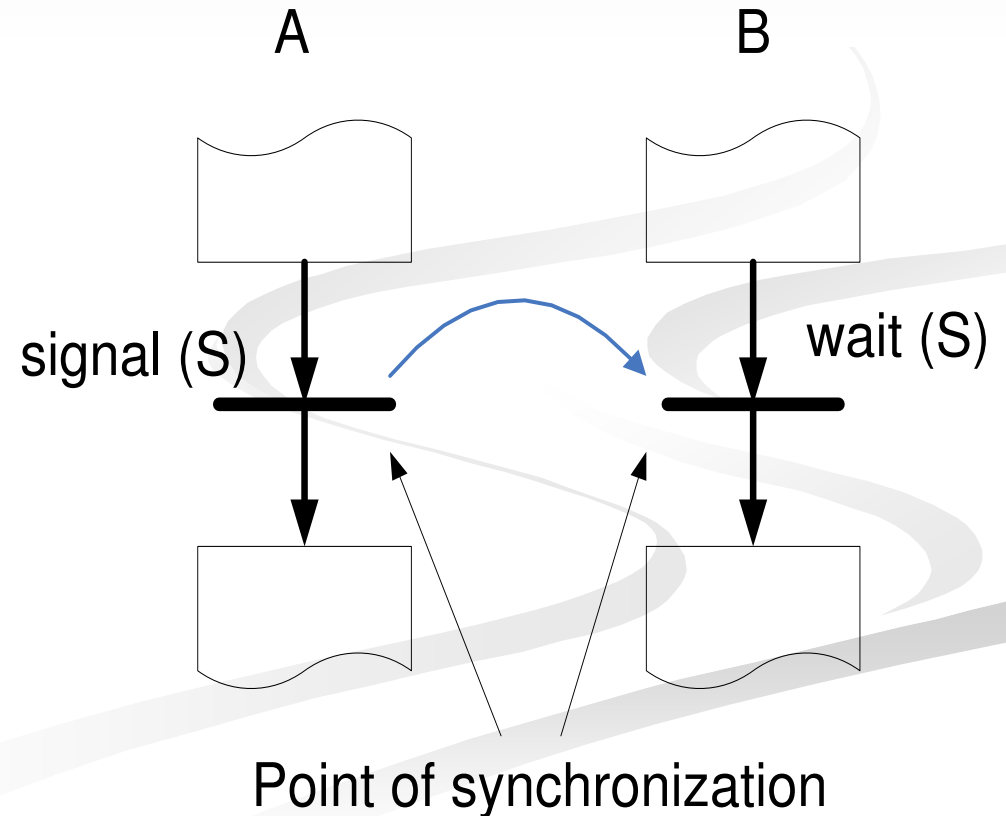
Сигнализация за събитие

☞ Чрез събитийни флагове



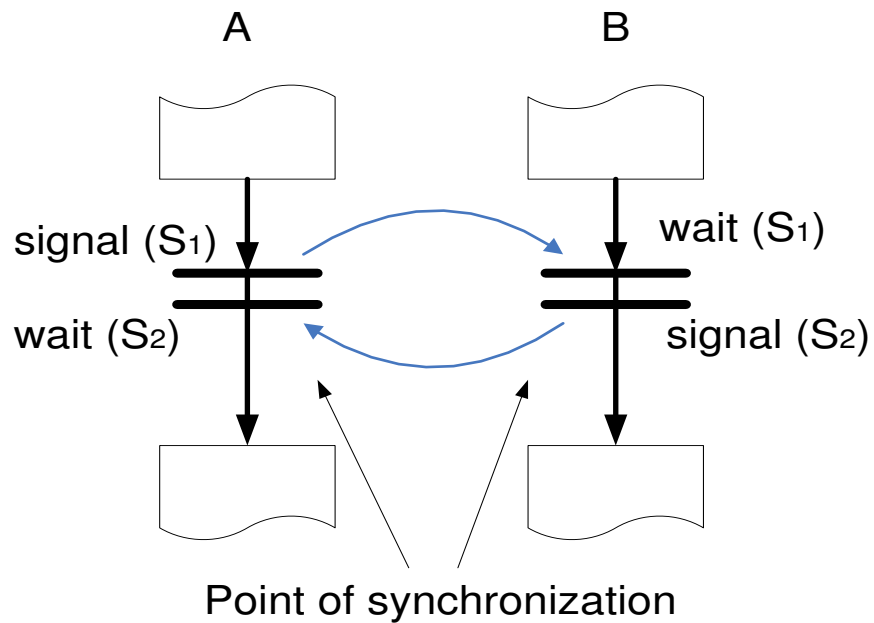
Сигнализация за събитие

☞ Чрез сигнални семафори

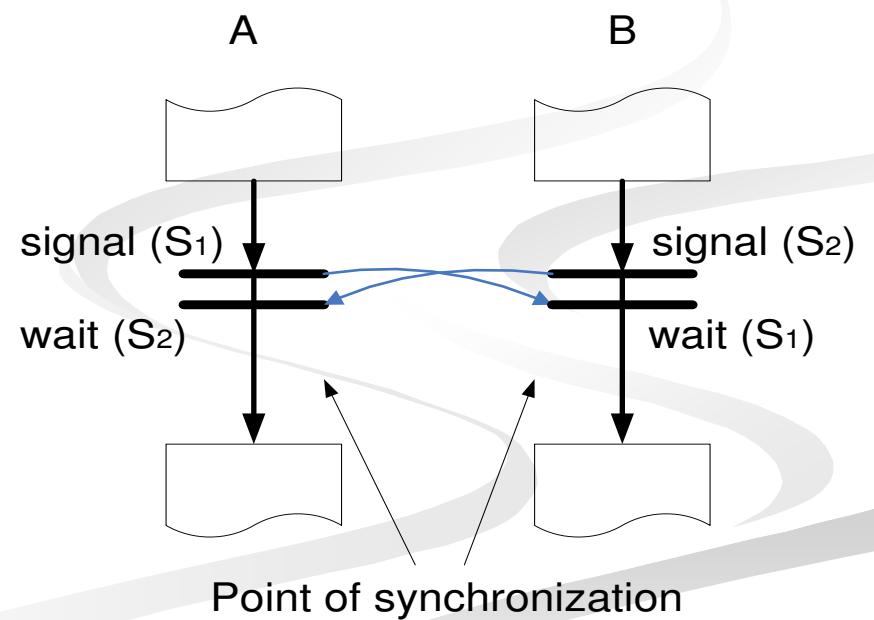


Двупосочни синхронизации

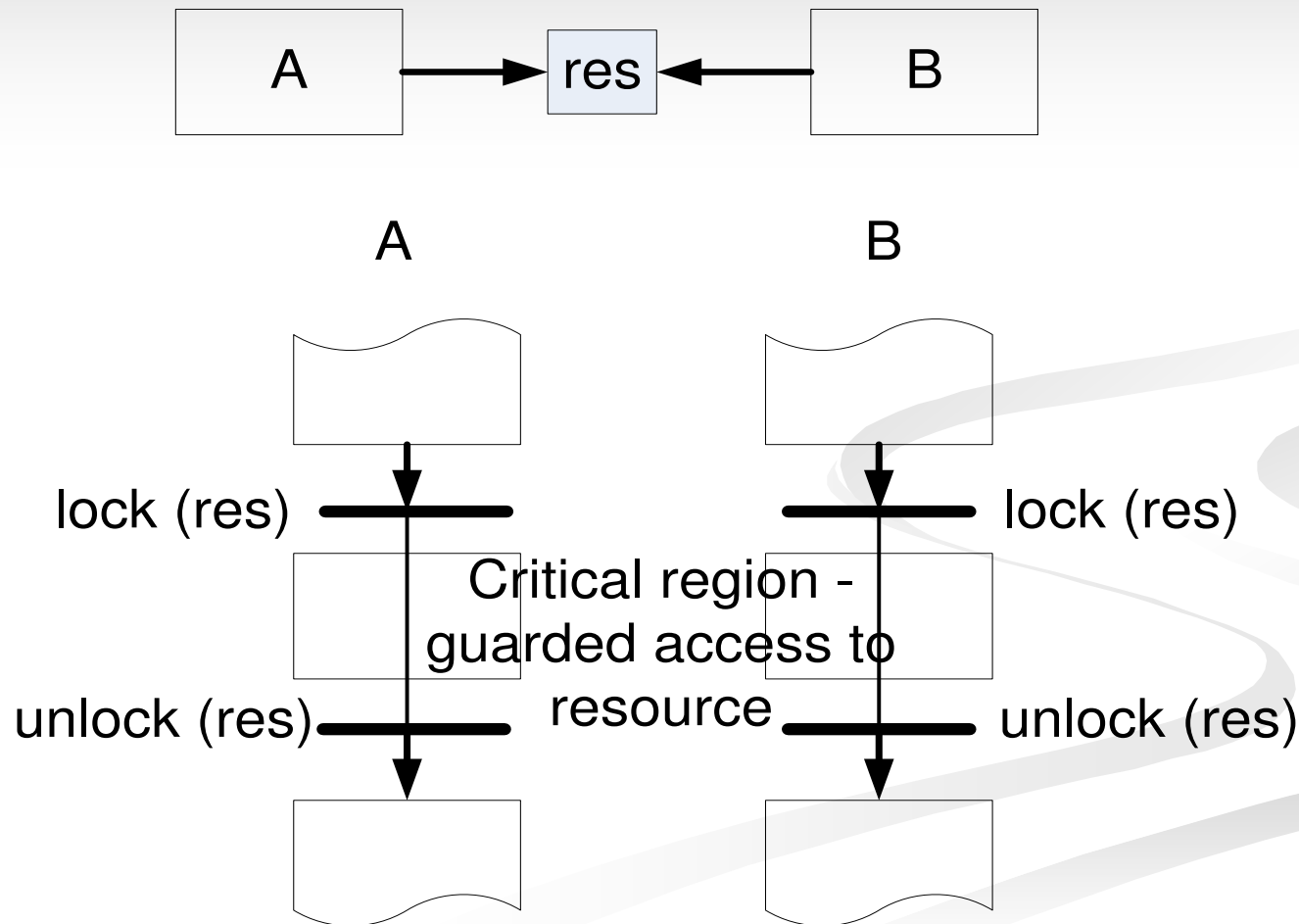
Hand-shaking



Symmetrical Rendezvous



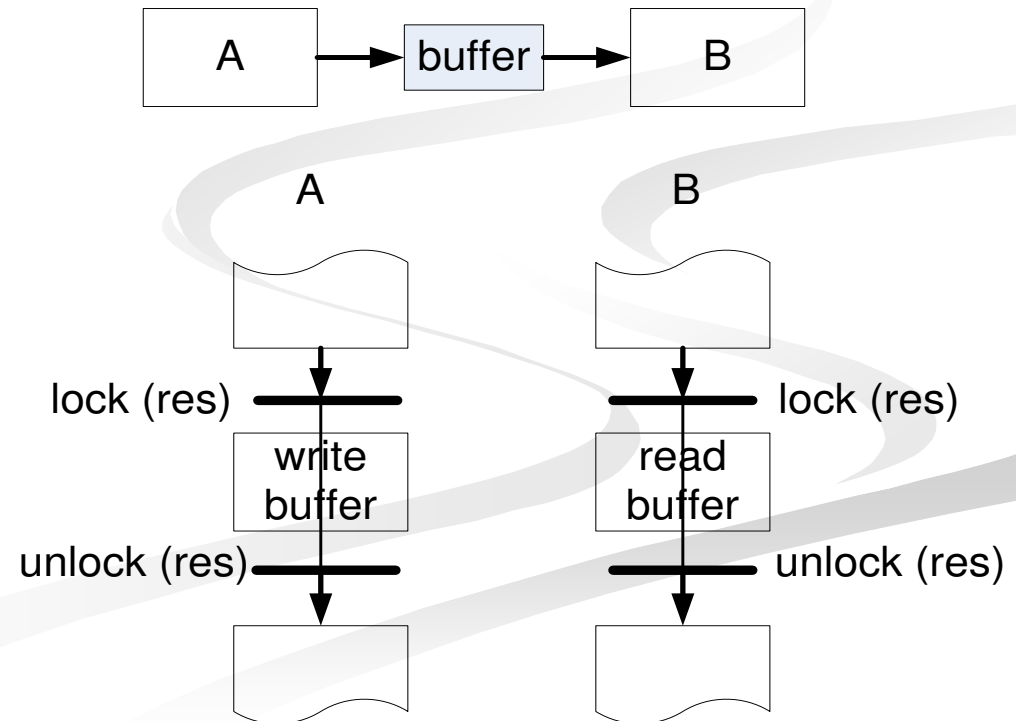
Синхронизация в ресурсни семафори



Междузадачна комуникация

✎ Асинхронна комуникация:

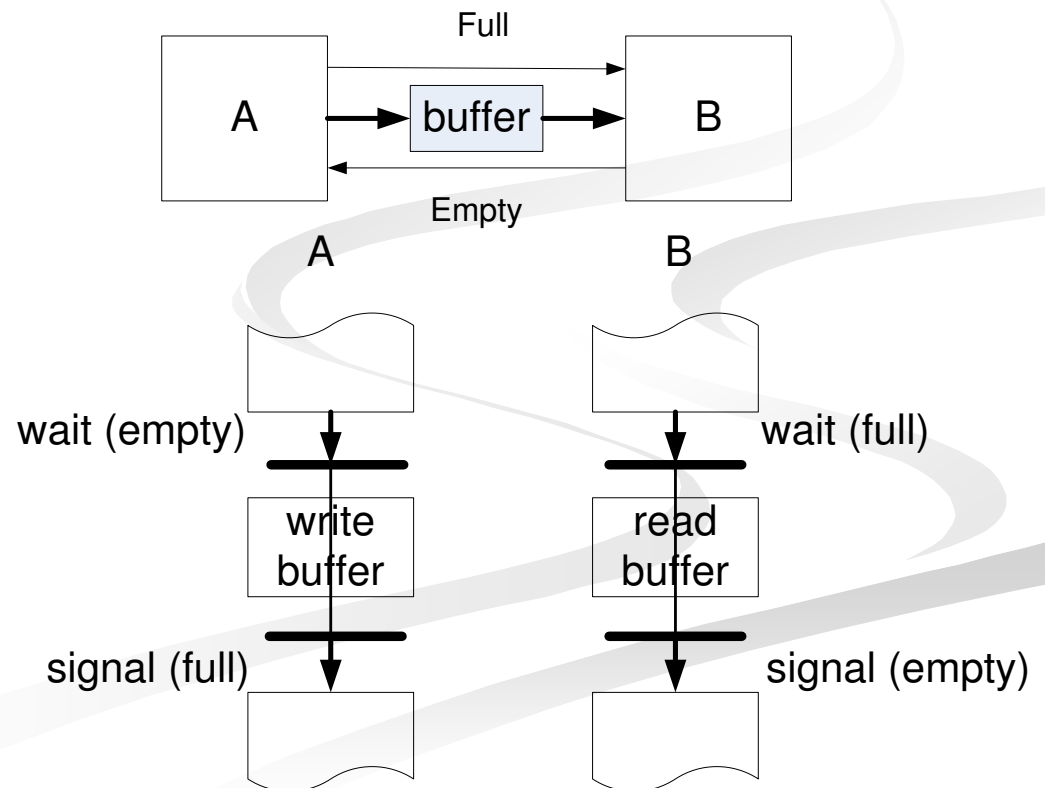
- ✓ Основен проблем: 'Производител-потребител' ('Producer-Consumer').



Task Interaction: Communications

👉 Синхронна комуникация:

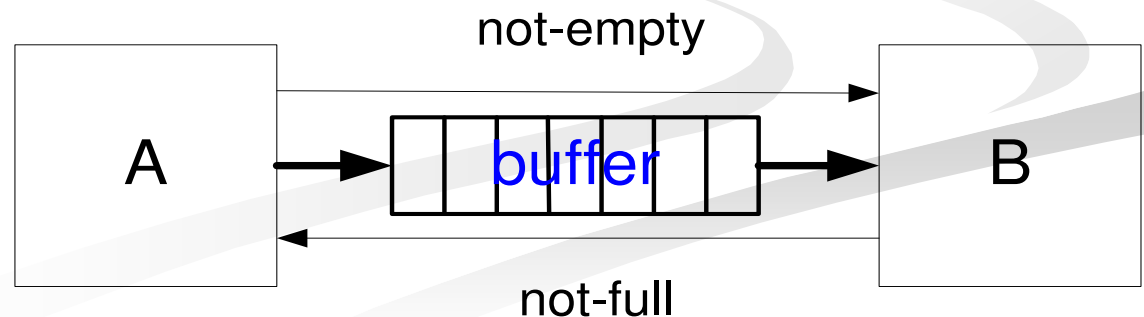
✓ Подобрен вариант на 'Производитель-потребитель'.



Междузадачна комуникация

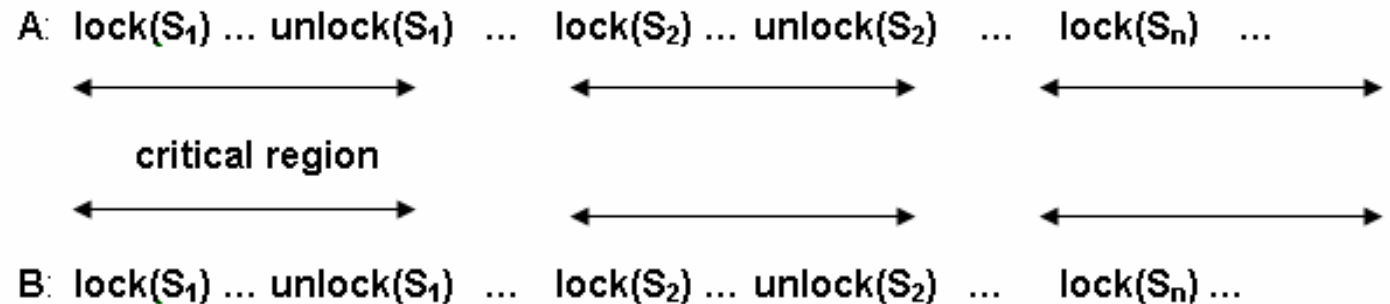
✎ Асинхронна комуникация с многопозиционен (многослотен) буфер

- ✓ Работата с буфера се управлява с ресурсен семафор.
- ✓ Състоянието се сигнализира с два двоични семафора:
 - ✗ 'not-full'
 - ✗ not-empty

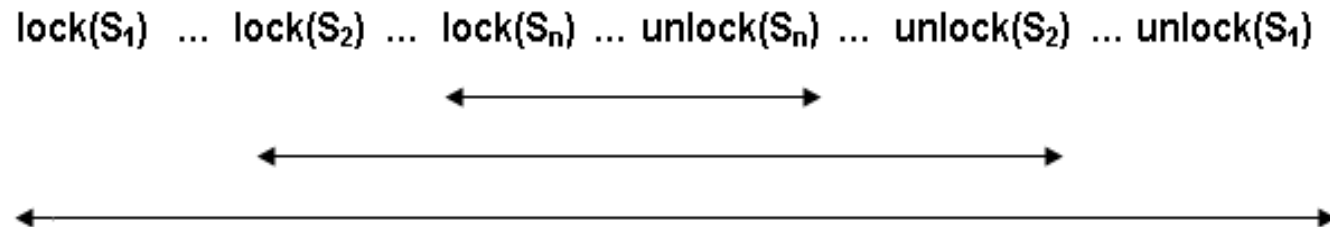


Достъп до ресурсите

👉 Последователен достъп



👉 Вложени критични секции



Взаимна блокировка (Deadlocks)

☞ Тип 'Един-към-един'

Задачите A and B се нуждаят от ресурсите R_1 и R_2 .

A: ... lock (R_1) . * . lock (R_2) .. unlock (R_2) .. unlock (R_1)

B: ... lock (R_2) ... lock (R_1) .. unlock (R_1).. unlock (R_2)

Ако $P(B) > P(A)$ и **B** прекъсна **A** в маркирания със ' * ' момент, задачите ще се блокират завинаги: **A** чака R_2 , **B** чака R_1 .

Взаимна блокировка (Deadlocks)

☞ Транзитивна блокировка

Задачите A, B, C използват ресурсите R_1 , R_2 , R_3 .

A: ... *lock* (R_1) . * . *lock* (R_2) .. *unlock* (R_2) .. *unlock* (R_1)

B: ... *lock* (R_2) . * . *lock* (R_3) .. *unlock* (R_3) .. *unlock* (R_2)

C: ... *lock* (R_3) . * . *lock* (R_1) .. *unlock* (R_1) .. *unlock* (R_3)

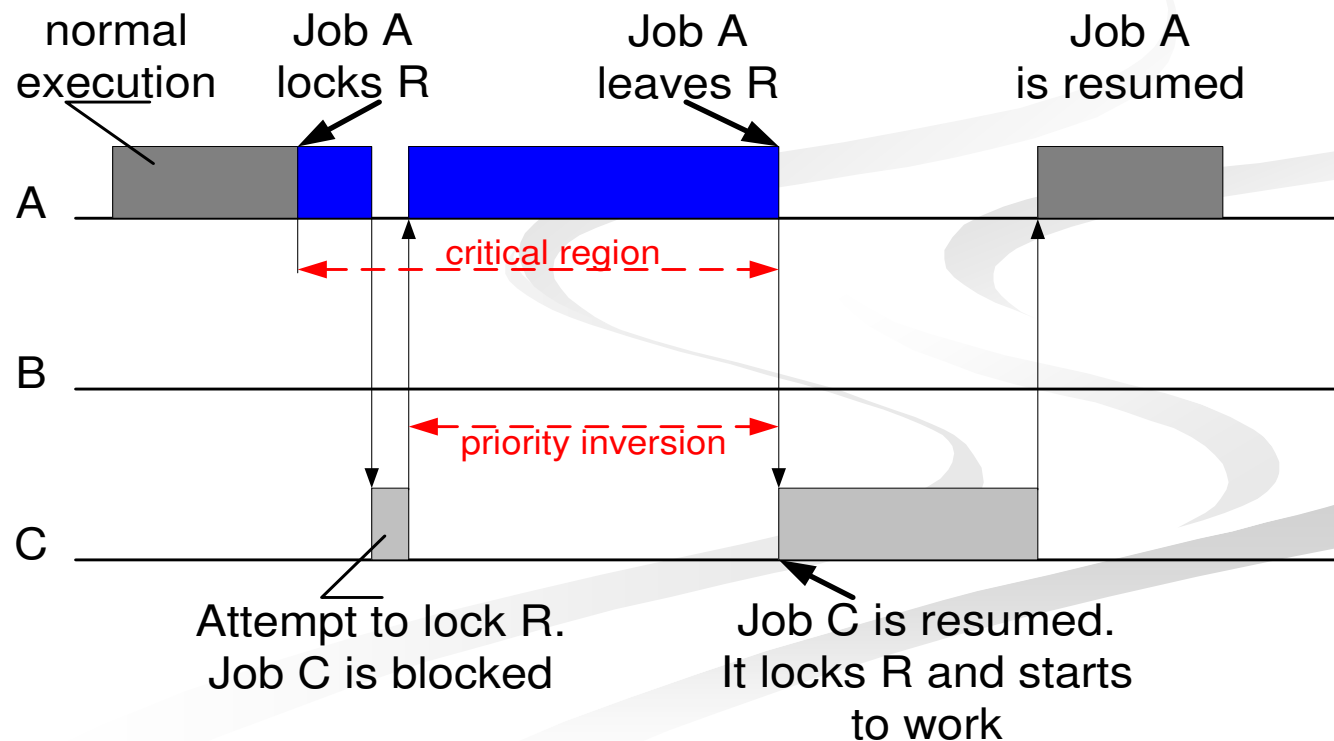
$P(C) > P(B) > P(A)$, **B** прекъсва **A**, **C** прекъсва **B**.

Блокировката е постоянна, защото **A** чака **R_2** , **B** чака **R_3** , **C** чака **R_1** .

Приоритетна инверсия

👉 Ограничена (Bounded) инверсия

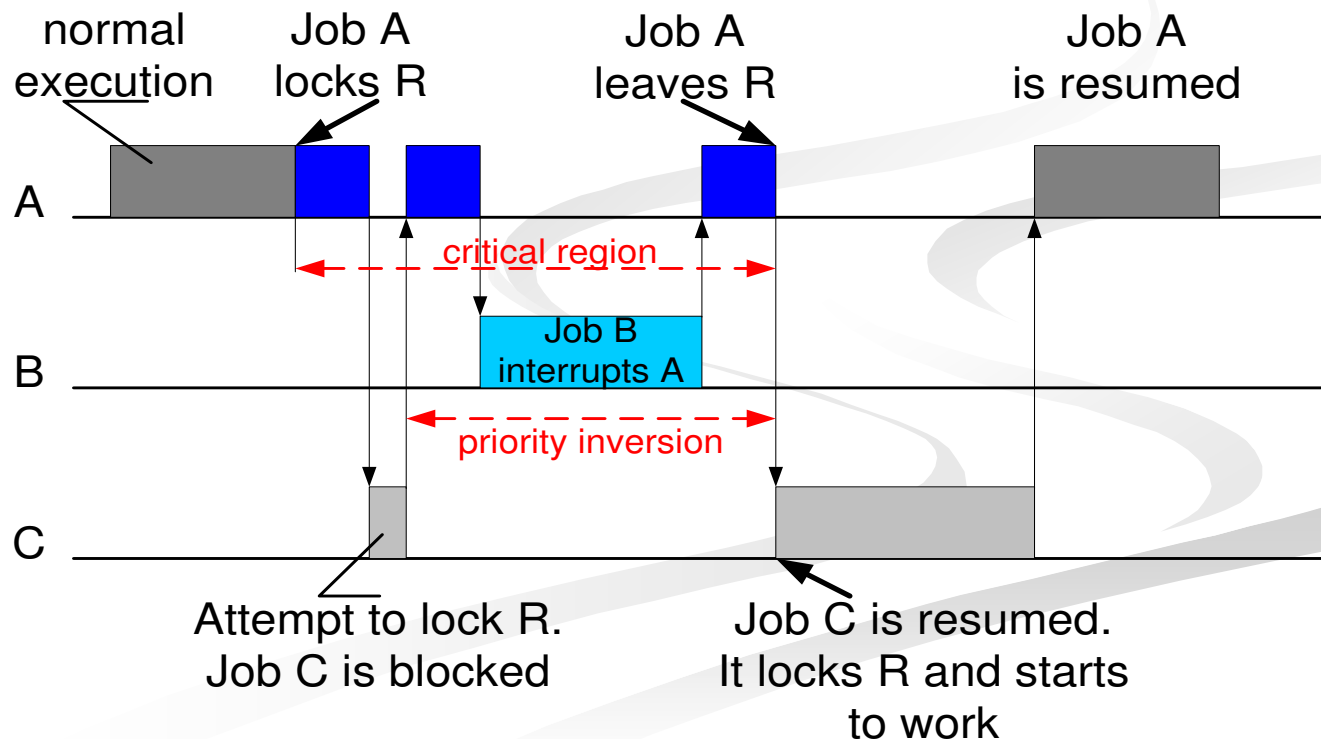
- ✓ $P(A) < P(C)$, ресурсът е **R**, C чака A да напусне критичната си секция и така спата приоритета.



Приоритетна инверсия

Неограничена инверсия

- ✓ $P(A) < P(B) < P$, A и C искат R, но B прекъсва A.



Анализ на диспечерируемостта



Основни решения

- ✓ Периодични задачи
- ✓ Спорадични задачи
- ✓ Разширен анализ



Диспечирание на периодични задачи

- ✓ Rate-monotonic scheduling (RMS)
- ✓ Earliest-deadline first (EDF)



Диспечирание на спорадични задачи

- ✓ RMS (aperiodic-servers).
- ✓ Dual-priority scheduling

Разширен анализ

☞ Включва:

- ✓ Влияние от ОС
- ✓ Влияние от ISR
- ✓ Процедури за стартиране на задачи
- ✓ Процедури за завършване на задачи
- ✓ Взаимоотношения между задачи (изчакване на критични секции)
- ✓ Други

Въпроси ?